

AN ABSTRACT OF THE THESIS OF

Faezeh Bahmani for the degree of Master of Science in Computer Science presented on May 29, 2014.

Title: Novel Approaches to Promoting End-user Programming

Abstract approved:

Margaret M. Burnett

End-user programming has become widespread. The increasing size of this population and the prevalence of barriers that they face has sparked the development of approaches that promote end-user programming by helping them overcome barriers and teaching them programming. Despite the fact that these approaches have done well in achieving those goals, there are still limitations. Specifically, these approaches place high expectations on the amount of prior knowledge that they should have and neglect to nurture their problem-solving skills. To fill in these gaps, our collaborators designed the approaches of *Idea Garden* and *debugging-first*. The Idea Garden approach attempts to provide problem-solving support by delivering problem-solving strategies and programming knowledge that help end-user programmers help themselves. In the debugging-first approach, which also makes use of the Idea Garden, users debug existing programs before creating their own.

In this thesis we study both approaches, finding that they fulfilled their goals in circumventing those limitations. Additionally, our results inform the design of the Idea Garden in a Debugging-first environment, shed lights on enhancing both approaches, and approaches with similar goals.

©Copyright by Faezeh Bahmani

May 29, 2014

All Rights Reserved

Novel Approaches to Promoting End-user Programming

by
Faezeh Bahmani

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented May 29, 2014
Commencement June 2014

Master of Science thesis of Faezeh Bahmani presented on May 29, 2014.

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 thesis 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.

Faezeh Bahmani, Author

ACKNOWLEDGEMENTS

I would like to thank my advisor Margaret Burnett for all the work she has put in to make this research possible.

I would also like to thank Carlos Jensen, Christopher Scaffidi, and Andrew Ko who I had the pleasure to work with. Their expertise and insights have helped me shape my work in many ways.

Also, I would like to give many thanks to Nicole Thompson, Shannon Thompson, Bella Bose, and Terri Fiez for making me feel welcome at OSU from the very beginning of the program.

I would like to thank my committee members Ron Metoyer, Carlos Jensen, and Peter Lachenbruch for taking the time to read my thesis and for attending my defense.

Irwin Kwan: Thank you for your constant feedback and guidance. I cannot appreciate enough all the lessons I learned from you.

To all of my coauthors and collaborators at OSU and UW, especially Jill Cao and Mike Lee. I learned so many lessons from you, it has been great working with you, and I hope we will keep collaborating for years to come.

I would like to thank all my Forms3 research associates, especially Chris Bogart and Todd Kulesza for helping me during my thesis writing process. I have also had the pleasure to work closely with Will Jernigan, Amber Horvath, Jilian Laferte, and Charles Hills. Their efforts and input have been wonderful.

I would like to express my gratitude towards my friends. Caius Brindescu whose continual support, feedback, and friendship was invaluable to me during my thesis writing process; Evgenia Chunikhina, whose emotional support, and friendship I cannot appreciate enough; Michael Hilton who provided invaluable insights into my writing; Mihai Codoban, Medha Jannat Mafruhatul, Mohammad Shahed Sorower, and Sergey Shmarkatyuk whose cheerful company kept me happy during the last year; and Jennifer Davidson who assisted me at various points of my studies.

Uncle Mohammad and Aimee, Uncle Reza and Maryam joon: Thank you for welcoming me into your homes during the times that I was most stressed. Aunt Tooran, thank you for listening to me and supporting me via warm hopeful words over the phone.

Thank you to my beloved brother, Amirreza, who always cheered me up over our video calls.

Most importantly, I am eternally grateful to my amazing and wise parents. Nothing would have been possible without your unconditional love during this entire process, and your kind words, gifts, and deeds. Chester, your energy and enthusiasm have been inspiring.

CONTRIBUTION OF AUTHORS

Jill Cao designed the approach introduced in the second manuscript and led the prototype, study, and writing the paper. Irwin Kwan, Scott D. Fleming, Josh Jordahl and I were the primary additional contributors to these three activities. Amber Horvath and Sherry Yang also contributed to these activities to a lesser extent, and Margaret Burnett provided guidance.

Michael J. Lee designed the approach introduced in the third manuscript and led the prototype, camps, and writing. My primary roles were to lead the think-aloud study and qualitative data analysis. In addition, I contributed to the prototype and to running the camps. The other authors also contributed to the development of the prototype, running the camps, analyzing the data and writing the paper. Andrew Ko and Margaret Burnett provided guidance.

TABLE OF CONTENTS

	<u>Page</u>
1 Introduction.....	1
2 End-user Programmers in Trouble: Can the Idea Garden help them to help themselves?	2
2.1 Abstract	3
2.2 Introduction	3
2.3 Background	4
2.3.1 <i>The Idea Garden's Host: CoScripter</i>	4
2.3.2 <i>Helping Users Learn to Do with the Idea Garden</i>	4
2.4 Experiment	6
2.4.1 Experiment Design.....	6
2.4.2 Participants	9
2.4.3 Procedure.....	9
2.4.4 Tasks.....	11
2.5 Analysis methodology.....	11
2.5.1 Task Performance.....	11
2.5.2 Ratings of Idea Garden's Helpfulness.....	12
2.6 Results	13
2.6.1 <i>RQ1: Does the Idea Garden help end users do programming tasks on their own?</i> 13	
2.6.2 <i>RQ2: Factors affecting success with the Idea Garden: Who and When?</i> .13	
2.6.3 <i>RQ2: Who alone? When alone?</i>	16
2.7 Our results in context	17

TABLE OF CONTENTS (Continued)

	<u>Page</u>
2.8 Open Questions for the IdeaGarden approach	18
2.9 Conclusion.....	21
3 Principles of a Debugging-First Puzzle Game for Computing Education	24
3.1 Abstract	25
3.2 Introduction	25
3.3 The Principles of Debugging Games	27
3.4 The Gidget prototype	29
3.5 Methods.....	30
3.5.1 Think-Aloud Study.....	31
3.5.2 Summer Camps	31
3.5.3 Coding and Analyses.....	32
3.6 Results	33
3.6.1 <i>Struggles with Programming Concepts</i>	33
3.6.2 <i>Counterproductive Problem-Solving Strategies</i>	36
3.6.3 <i>From Debugging-First to Programming: Level Creation</i>	38
3.6.4 <i>Overcoming Barriers: Practice Makes Perfect?</i>	39
3.7 Discussions and Implications	41
3.8 Conclusion.....	43
4 Conclusions.....	47
Bibliography.....	48

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
Figure 2.1. CoScripter's (a) script area, (b) table area, and (c) browsing area.	6
Figure 2.2. The Strategy treatment's Second web page feature (context-sensitive).....	8
Figure 2.3. The Programming treatment's Second web page feature (context-sensitive)...	8
Figure 2.4. The Combined treatment's Second web page feature (context-sensitive).	8
Figure 3.1. Gidget's level design mode (the Gidget character is circled).	26
Figure 3.2. The concepts in the most challenging levels during puzzle play.	34
Figure 3.3. Camp participants most often created puzzle levels to challenge other players	39
Figure 3.4. Percent improvement by comparing barriers before, then during level design.	41

LIST OF TABLES

<u>Table</u>	<u>Page</u>
Table 2.1. Each feature addresses a barrier with strategies and programming knowledge.....	7
Table 2.2. Problem-solving strategies in Idea garden features.	7
Table 2.3. Programming knowledge includes programming concepts and mini design patterns.....	7
Table 2.4. Summary of participants' performance scores	13
Table 2.5. Average participant ratings of the helpfulness of the 6 Idea Garden features.	14
Table 2.6. Scores at or above (colored slices) or below (white slices) the grand median for Idea Garden vs. Control, shown for all who/when combinations.....	16
Table 2.7 Participation ratings of Idea Garden feature helpfulness for participants with little knowledge, in the pet task.	16
Table 2.8. Participants who scored at or above (colored slices) the grand median separated by little or no knowledge.	17
Table 2.9. Participants who scored at or above (colored slices) the grand median separated by task.	17
Table 2.10. Summary statistic for Idea Garden participants with little knowledge who did the Pet task.	19
Table 2.11. Task performance of participants with little knowledge who did the Pet task.	20

LIST OF TABLES (Continued)

<u>Table</u>	<u>Page</u>
Table 2.12. Subjective ratings of participants with little knowledge who did the Pet task.	21
Table 2.13. Task performance of all males and females broken down by treatment.	21
Table 3.1. Barriers code sets (Cao et al. [7], Ko et al. [16]).	32
Table 3.2. Number of barriers per level and the percent improvement (%imp) in barriers from the puzzle play (PZ) to level design (LD) in the summer camps.	35
Table 3.3. Teams using the concept in at least one level they created are marked (✓).....	42

To mom and dad

1 Introduction

There are thousands of people today who are writing computer programs, either for work or as a hobby. Nardi describes these non-professional programmers as end-user programmers [Nardi 1993]. These include chemists, librarians, teachers, architects, and accountants [Nardi 1993]. A common example is an accountant creating a budget spreadsheet. End-user programmers differ from professional programmers in their level of software engineering training or motivation [Ko et al. 2011]. While most professional programmers have formal education in computer science and/or software engineering, many end-user programmers lack such training [Cao 2013]. Also, end-user programmers' motivation to program originates from their need to accomplish their tasks or pursue their hobbies [Dorn and Guzdial 2010], whereas professional programmers are driven by their motivation to program [Ko et al. 2011].

Research shows that end-user programmers struggle with programming barriers in different domains, such as Visual Basic [Ko et al. 2004], spreadsheets [Chambers and Scaffidi 2010; Kissinger et al. 2006], animation [Gross and Kelleher 2009] and mashups[Cao 2011; Zang and Rosson 2009].

This thesis evaluates two separate approaches to help end-user programmers and teach them programming, namely the Idea Garden by [Cao et al. 2012] and Debugging-first by [Lee et al. 2011]. We performed a summative evaluation of the Idea Garden approach in the Coscripter mashup environment. We also performed a formative evaluation of the Debugging-first approach and the Idea Garden in Gidget, a Debugging-first environment.

2 End-user Programmers in Trouble: Can the Idea Garden help them to help themselves?

Jill Cao¹, Irwin Kwan¹, Faezeh Bahmani¹, Margaret Burnett¹, Scott D. Fleming², Josh Jordahl¹, Amber Horvath¹, Sherry Yang^{1,3}

¹Oregon State University
Corvallis, OR, USA

² University of Memphis
Memphis, TN, USA

³Oregon Institute of Technology
Klamath Falls, OR, USA

Proceedings of the 2013 IEEE Symposium on *Visual Languages and Human-Centric Computing* (VL/HCC), San Jose, CA, USA

2.1 Abstract

End-user programmers often get stuck because they do not know how to overcome their barriers. We have previously presented an approach called the Idea Garden, which makes minimalist, on-demand problem-solving support available to end-user programmers in trouble. Its goal is to encourage end users to *help themselves* learn how to overcome programming difficulties as they encounter them. In this paper, we investigate whether the Idea Garden approach helps end-user programmers problem-solve their programs on their own. We ran a statistical experiment with 123 end-user programmers. The experiment's results showed that, even when the Idea Garden was no longer available, participants with little knowledge of programming who previously used the Idea Garden were able to produce higher-quality programs than those who had not used the Idea Garden.

2.2 Introduction

When doing a programming task, end users face many barriers such as decomposing design problems [4], using loops [5], and choosing and coordinating multiple modules [16]. To help users overcome such barriers on their own without the need for guided instruction, we have previously presented the Idea Garden approach [5], an add-on for end-user programming environments to help end-user programmers in trouble solve their own problems. The Idea Garden draws from Simon's problem-solving theory [21] and Minimalist Learning Theory [7], and delivers its help in the form of information snippets that, on demand, deliver problem-solving strategies and programming domain knowledge in the context of a user's own programming tasks. The core philosophy of the Idea Garden is not to automatically remove barriers for the user, but to rather enable the user to solve problems on their own with only minimal, self-guided assistance.

We previously performed an empirical study on Idea Garden's ability to help end-user programmers learn problem-solving strategies and programming knowledge during a programming task in which learning was not the primary goal [6]. This previous study revealed that after actively using the Idea Garden, users were able to demonstrate having learned the relevant problem-solving strategies and programming knowledge, as

evidenced by their ability to explain the relevant problem-solving strategies and programming knowledge.

In this paper, we move beyond learning to doing. We investigate whether end-user programmers who have used the Idea Garden can put their learning into practice in future programming tasks even when Idea Garden support is no longer available, via the following research questions:

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?

2.3 Background

2.3.1 The Idea Garden's Host: CoScripter

We implemented the Idea Garden prototype within CoScripter/Vegemite [18], an end-user programming-by-demonstration environment for web automation in Firefox. Using CoScripter, a user can demonstrate how to carry out a task by navigating to web pages, entering data in forms, and interacting with page elements. CoScripter translates the user's actions into a "web macro" script that the user can edit and execute (Fig. 2.1a). CoScripter also provides a table (Fig. 2.1b) that makes it possible to create mashups that combine data from multiple web pages. For example, a user can create a script to mash restaurant location with public transit by loading a web page of restaurants (Fig. 2.1c), copying its addresses to the table (Fig. 2.1b), then iterating to send each address to another web page (e.g., Google Maps) to compute travel time via transit. Thus, CoScripter requires understanding of programming concepts such as control flow and dataflow.

2.3.2 Helping Users Learn to Do with the Idea Garden

The goal of the Idea Garden is to help users form ideas to overcome programming barriers on their own. As we have described in previous work ([5, 6]), we leveraged Simon's problem-solving theory [21] to guide the design of the Idea Garden features. According to Simon's theory, two types of skills are necessary for solving problems in a

specific domain: domain-specific knowledge and general problem-solving strategies [21]. The Idea Garden features encourage both of these skills: they aim to encourage users to adopt new strategies and pick up new programming knowledge that is relevant to the problems they are currently trying to solve.

The Idea Garden features are designed to help users overcome barriers, such as those identified by Ko et al. [16] and in our prior work ([3, 4, 5]), by providing programming knowledge as well as strategies (Table 2.1). The Idea Garden prototype’s features target three programming barriers: “How-to-start”, where users had problems figuring out how to start their scripts; “Composition”, where users had problems combining multiple web page functions to come up with a result; and “More-than-once”, where users were unable to use iteration to repeat actions. The features that we developed were the *Getting started* feature, which addresses the “How-to-start” barrier by providing suggestions on what initial actions to take; the *Second web page* feature, which addresses the “Composition” barrier by suggesting that users can use the output from one page as input to a second page; and the *Generalize-with-repeat* feature, which addresses the “More-than-once” barrier by suggesting a process and commands that the user can use to repeat actions. Table 2.1 summarizes the relationships among the features, barriers, strategies, and programming knowledge. The associated strategies are described in Table 2.2 and the associated programming knowledge is described in Table 2.3.

Each feature has two versions, a context-sensitive version and a context-free version. The context-sensitive versions are available when the Idea Garden detects specific user action sequences suggesting barriers that the Idea Garden targets. The context-free versions are always accessible from a “Help” button at the top of the screen.

2.4 Experiment

2.4.1 Experiment Design

To answer our research questions, we conducted a between-subjects experiment with four treatments: one Control condition and three Idea Garden conditions: Strategy, Programming, and Combined. We asked non-Control participants to first work on the

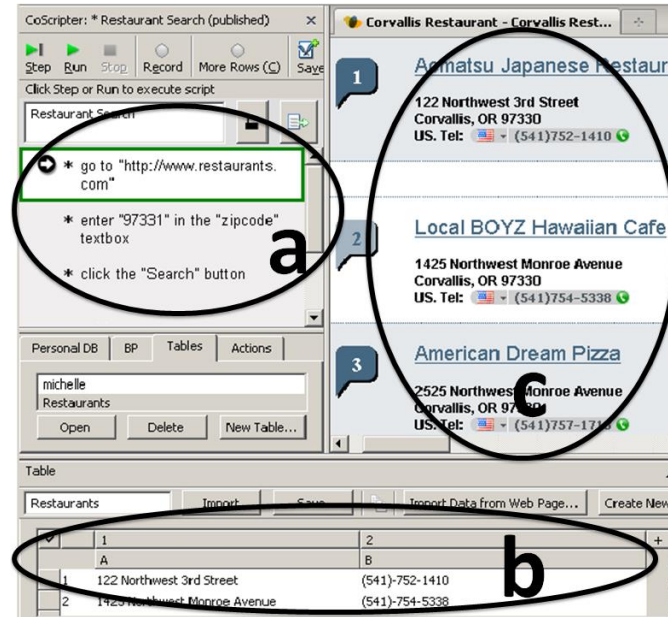


Figure 2.1. CoScripter's (a) script area, (b) table area, and (c) browsing area.

learning task, in which participants completed a programming task in CoScripter with the Idea Garden present. Then, we asked them to perform a learning transfer task [2] in which participants completed a programming task in CoScripter with the Idea Garden not present. 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 barriers (Table 2.1), but each treatment's features address the barriers differently. The *Strategy* treatment provides suggestions to apply a problem-solving strategy; the *Programming* treatment provides programming knowledge and the *Combined* treatment contains both strategy

Table 2.1. Each feature addresses a barrier with strategies and programming knowledge.

Feature	Barrier addressed	Strategy	Programming knowledge
<i>Getting Started</i>	How-to-start	Working backwards	Data extraction concept, Finder design pattern
<i>Second web page</i>	Composition	Divide-and-conquer (context-sensitive), Working backward (context-free)	Dataflow concept, Webpage-as-component design pattern
<i>Generalize-with-repeat</i>	More-than-once	Generalization	Iteration concept, Repeat-copy-paste design pattern

Table 2.2. Problem-solving strategies in Idea garden features.

Strategies: information that helps users problem-solve	
<i>Working backward</i>	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.
<i>Divide-and-conquer</i>	Break a problem into individual pieces, solve each piece, and join the individual solutions together.
<i>Generalization</i>	Solve one instance of a problem and generalize the solution to all instances in the problem.

Table 2.3. Programming knowledge includes programming concepts and mini design patterns.

Programming concepts: information that helps users build scripts	
<i>Data extraction</i>	The concept of selecting a slice of structured data from a web page and putting it into the table.
<i>Dataflow</i>	The concept of flowing data between web page and table, or between web pages.
<i>Iteration</i>	The concept of looping through rows of table to operate on each row.
Mini Design patterns: common ways that users structure their scripts	
<i>Finder</i>	Use a web page to find information (as opposed to computing information)
<i>Webpage-as-component</i>	Use a web page to compute information (as opposed to finding information).
<i>Repeat-copy-paste</i>	For each row in the scratchtable, copy-paste value from table to web page and submit.

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 2.2) whereas the *Programming* treatment contained the webpage-as-component design pattern and the dataflow concept (Figure 2.3). The Second

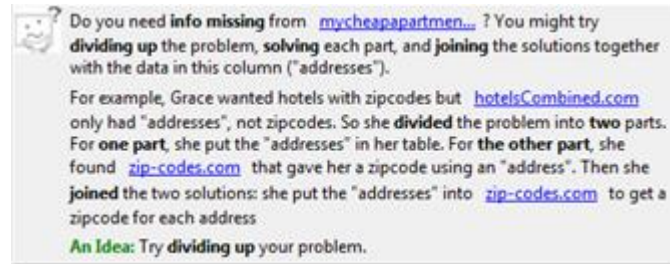


Figure 2.2. The Strategy treatment's Second web page feature (context-sensitive). This feature describes the "divide and conquer" strategy.

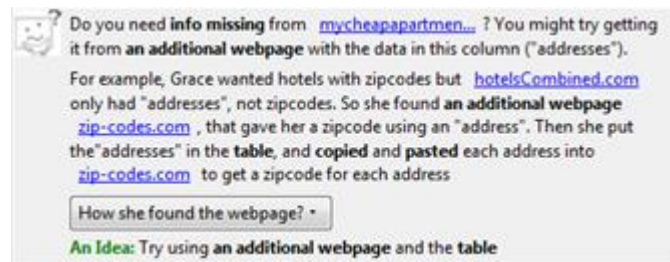


Figure 2.3. The Programming treatment's Second web page feature (context-sensitive). This feature presents the "dataflow" concept and the "webpage-as-component" pattern.



Figure 2.4. The Combined treatment's Second web page feature (context-sensitive). This feature describes both the "divide and conquer" strategy as well as the "dataflow" concept and the "webpage-as-component" pattern.

Webpage feature for the *Combined* treatment included both strategy information and programming knowledge (Figure 2.4).

Although Simon emphasized the importance of both domain knowledge—programming knowledge in our context—and problem-solving strategies, including both parts as in our Combined treatment has trade-offs. One trade-off is length versus effectiveness. As suggested by the Attention Investment model [1], the probability that a user would invest attention in a feature depends on the perceived cost of the investment. If a feature is too long, the user may perceive the cost of processing it as being too high and ignore it. In addition, too much information might lead to cognitive overload [22] which reduces the quality of information a user is able to get out of a feature. Thus, including both strategy and programming information as in the Combined treatment may potentially be less effective than just including one piece as in the Strategy and the Programming treatments.

For our study, we hypothesize that, even when the Idea Garden is no longer available, participants who previously had access to the Idea Garden, regardless of treatment, will be able to write a higher-quality program for a programming task compared to Control participants who had no previous access to the Idea Garden.

2.4.2 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), but excluding computer science and electrical engineering. We also 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.

2.4.3 Procedure

We assigned two 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, the first task was a learning task and the second task was a learning transfer task. Idea Garden participants were not informed that the Idea Garden would be unavailable during the second task.

Participants filled out a background questionnaire and then took 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 practice. We encouraged the participants to ask questions during this practice period. Participants then filled out a standard computer self-efficacy questionnaire [8] regarding CoScripter-related tasks.

Participants then had 25 minutes to work on the first task. Participants in the Idea Garden treatment had the Idea Garden enabled. To ensure that every Idea Garden 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 automatically saved every 15 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”. 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 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.

2.4.4 Tasks

Each participant worked on two tasks assigned in random order. 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. Both tasks had three implicit subtasks: (1) import a list of apartment addresses or shelter names from a webpage into the table; (2) iterate over the addresses and compute driving time, or iterate over the shelter names and look up shelter ratings; and (3) copy each driving time or shelter rating back to the table.

2.5 Analysis methodology

2.5.1 Task Performance

Because we were interested in learning-to-doing, we evaluated the transfer task’s performance only. Thus, whenever we mention “task performance”, we mean the second (transfer) task, in which 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. We graded all three because many users kept starting additional scripts, making it difficult for us to know which one had finally won out as the user's "intended" solution. This resulted in three scores per participant, from which we used the participant's highest score.

We graded the scripts and tables against a rubric based on the three subtasks listed in Section III.D. Each correct answer was defined precisely, so subjective interpretation was not needed to grade them. Specifically, each task's three subtasks were worth 5 points, for a total of 15 points possible. Within a subtask, each correct command or table column entry was worth 1 point. For example, the Apartment task's subtask 1 needed four commands (extract addresses, go to maps page, copy address from table, paste into maps page) and one table column (addresses), each worth 1 point. A participant with two correct commands and the correct table column would score 3 of 5 for this subtask.

Two researchers split up the scripts and the tables and graded them independently. Then, one researcher double-checked the grading. Since the rubrics did not involve subjective judgment, 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 (Kolmogorov-Smirnov $D=0.7361$, $p<2.2e-16$) nor did we use Kruskal-Wallis because of a large number of ties.

2.5.2 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.

2.6 Results

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

Evidence that the Idea Garden helped participants' task performance in the transfer task was not strong. Idea Garden participants averaged higher scores than Control participants (Table 2.4), but the difference was not significant at $p < .05$. Also, no one treatment had significantly higher scores than the others.

However, Idea Garden participants' reports of the Idea Garden's helpfulness from the

Table 2.4. 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

post-session questionnaire were significantly higher than neutral (one-sample $t=3.22$, $p=.00176$). (Neutral or below is what we might expect if the approach were not helpful. The one-sample t-test compares a sample value against an expected population mean). Table 2.5 summarizes.

Given that so many Idea Garden participants found the Idea Garden features helpful, what might this suggest? One possible explanation of our results might be that, as in our previous study [6], some participants who learned something from the Idea Garden were simply not able to transfer their learning to overcoming barriers on their own. However, another possibility, posed by RQ2, 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.

2.6.2 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., [10, 13, 17, 23]), 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 III.B, 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) $p=.0297$).

Table 2.5. 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***	

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 actually encounter a barrier and get stuck. To illustrate, the recorded log for one “little knowledge” participant, P11544 shows that she did not know to try to incorporate a second webpage—but when the Idea Garden suggested it, she followed the suggestion and succeeded. In contrast, a “no knowledge” participant, P22066, also saw the suggestion—but instead of trying to use two pages





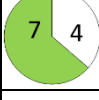
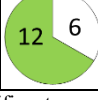


together, he switched to a different webpage altogether, which was not useful to his problem.

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 Apartment 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), $p=.001$). Task difficulty is a relevant issue here, because the Idea Garden is called upon only when a task is hard enough that a user runs into difficulties. One example of such difficulties with Pet came from Participant P12344:

P12344: "Couldn't find '# of reviews' for the shelter, then realized too late that I could find the info. on another web page."

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 Table 2.6. 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. Participants’ ratings confirmed this result: as Table 2.7 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 $t=2.46$, $df=20$, $p=.0231$). In essence, these results say 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 2.6. 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
<i>Pet</i>	 1 5 A1	 6 15 A2	 1 4 B1	 8 19 B2
	Fisher's exact test $p = .0265^*$		not significant	
<i>Apt</i>	 7 4 C1	 12 6 C2	 3 3 D1	 16 13 D2
	not significant		not significant	

2.6.3 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 result was that neither factor alone explained the results. Table 2.8 shows suggestive differences based on subpopulation alone, and Table 2.9 shows suggestive differences based on task difficulty alone, but these differences did not rise to significance.

Table 2.7 Participation 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
Sample mean = 3.29	
One-sample t-statistic = 2.46, DF = 20, $p = .0231^*$	

Table 2.8. 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.

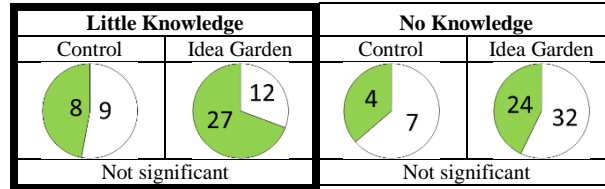
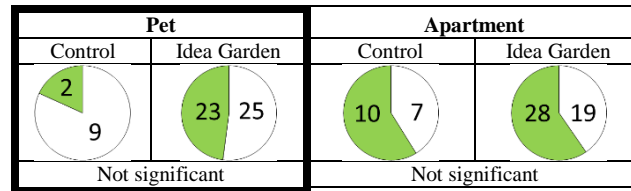


Table 2.9. Participants who scored at or above (colored slices) the grand median separated by task. In the more difficult Pet task, Idea Garden participants scored somewhat higher than control participants, and in the easier Apartment task, they scored almost identically. When subpopulation was not taken into account, the differences did not rise to significance.



The lack of significance for either factor alone could be due to the combination of the ceiling and floor effects in our data. Specifically, the Apartment task showed a “ceiling effect” in which *everyone* did pretty well, which diluted differences in the more difficult Pet task when the task data were combined. Likewise, no-knowledge participants’ floor effects (i.e., most gained little from the Idea Garden) diluted the differences the other participants showed. Investigating this possibility by isolating the factors for separate analysis did not resolve the question, because it left sample sizes so small that statistical differences would be unlikely. Thus, answering the impact of each factor alone will require follow-up empirical investigation.

2.7 Our results in context

Most empirical studies of systems supporting end-user programmers have not considered the difference between “little knowledge” and “no knowledge”. In fact, when Dorn’s study of a case-based informal learning system for learning Adobe Photoshop

scripting did not significantly increase participants' performance, Dorn hypothesized that a reason may have been the variety of his participants' prior programming experience [10]. Our results provide evidence to support Dorn's hypothesis.

Another approach with some similarities to the Idea Garden is Wrangler's proactive suggestions that recommend actions for users to take [14]. Guo et al. investigated Wrangler's suggestions in the context of a data-transformation tool but found that these suggestions did not improve task performance [14]. The Wrangler participants, unlike ours, were computer science students, and they generally ignored the suggestions. This result seems consistent with our result about the difficulty of the task: suggestions seem unlikely to make much difference when the participant does not need them.

Like the Idea Garden, the stencils-based tutorials investigated by Harms et al. aimed to facilitate learning of a UI in order to transfer the skills to a new context [15], but unlike the Idea Garden, that approach used scripted tutorials. With this approach, children were able to ask for step-by-step guidance when using a visual programming system. Results showed that children using stencils completed more transfer tasks. This result is consistent with our transfer task results.

In the context of these other studies, our statistical results are among the strongest that we have seen on learning-to-doing by end-user programmers. Learning-to-doing takes time, and producing significant effects after only a 25-minute learning task demands a very effective approach. For example, Dorn's results were able to support only learning, not learning-to-doing [10]. Harms et al. [15] succeeded at showing learning-to-doing, but in that study the learning support tools were still available during the transfer task, so additional learning was allowed to take place. The learning support tools were also present in the case of Dorn's study. In contrast, in our study, we isolated learning transfer from learning, by requiring Idea Garden participants to demonstrate learning transfer after the Idea Garden was no longer available to them.

2.8 Open Questions for the IdeaGarden approach

Our results raise a number of open questions regarding the Idea Garden approach.

One question that arises is whether there is a "best" Idea Garden variant. Although no treatment was significantly better than any other, the Programming treatment trended

better for the little-knowledge participants doing the difficult task. These participants scored on average 4.56 points higher than Control participants (Table 2.10), and had the largest percentage of participants who scored at or above the median (83.3%) (Table 2.11). Also, all of these Programming participants rated the Idea Garden as helpful (Table 2.12). These trends lead to this open question:

Open Question 1: Is the Programming variant of the Idea Garden more effective than the others? If so, why?

If Programming is the best variant, one attribute that may account for it may be that it was concrete enough for participants to act upon. The Programming content focused on programming concepts and mini design patterns in particularly concrete and actionable ways. For example, Programming’s Generalize-with-repeat feature, triggered by the participant’s own code, explained iteration in the context of that code. Participants’ favorable comments afterward suggest that they knew how this content applied to their current barrier:

P13411: “It was nice that [the Idea Garden] recognized when I would want to use the repeat command”.

P23344: “This was helpful because getting the script to work for all rows and columns was tricky for me at first”.

The Strategy features, on the other hand, were a little less situated, providing more general problem-solving guidance. Strategy content helped a number of participants (Table 2.12), but others could not figure out how to apply the strategy guidance:

P21055: “[It was] not clear enough on how to work backwards.”

P23255: “It[’]s an Ok suggestion, but it doesn’t say how to ‘join the solutions together’.”

Table 2.10. Summary statistic for Idea Garden participants with little knowledge who did the Pet task.

Treatment	N	Mean	Median	StdDev
Control	6	3.51	2.25	4.74
Strategy	8	5.23	5.13	4.58
Programming	6	8.06	7.9	4.17
Combined	7	5.34	4	4.04

Table 2.11. 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%)
<i>Programming</i>	<i>1</i>	<i>5 (83.3%)</i>
Combined	2	5 (71.4%)

Idea Garden content length may also be implicated. The Programming and Strategy contents were shorter than the Combined variant's content, and the Attention Investment model [1] predicts that users may therefore find the Combined variant less cost-effective. This prediction is consistent with the Combined variant's lower ratings than the other two variants in Table 2.12.

However, at odds with shorter length is the notion of comprehensiveness. This trait was one of the goals of the Combined variant—to provide both relevant problem-solving guidance and relevant programming knowledge all in one place. Because comprehensiveness of information has been positively associated with users' trust in a system [9], a decision to reduce comprehensiveness in favor of brevity should not be made lightly.

Further, the issue of trust is not a matter of comprehensiveness alone [9]. People form impressions of trust quickly, and there are many factors involved. Further, a lack of trust in a system has been linked to disuse of the system. Thus, the issue of end users' trust in a system's advice seems important:

Open Question 2: What factors influence an end user's trust in advice offered by systems like the Idea Garden, and how do these factors influence ways users process and act upon the offered advice?

Comprehensiveness raises another issue as well. Research has shown that, in the aggregate, males and females process information differently, with males preferring to selectively follow and act upon salient cues and females preferring to process information comprehensively before acting upon it [19]. This phenomenon may in part explain why male and female end-user programmers make use of different features when

Table 2.12. Subjective ratings of participants with little knowledge who did the Pet task. The Programming treatment had 100% of its participants finding the Idea Garden helpful.

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

Table 2.13. Task performance of all males and females broken down by treatment. Females performed best with combined whereas males performed best with programming.

Treatment	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%)

programming and debugging [3, 11]. Our data are consistent with these results, with females trending better with the Combined treatment than with other treatments, but males trending better with the Programming treatment (Table 2.13). This leads to our third open question:

Open Question 3: How can we design Idea Garden features to support both the comprehensive information processing style that is statistically associated with females and the selective information processing statistically associated with males [20]?

We plan to investigate these and similar questions to better determine how to improve the effectiveness of Idea Gardens on busy end users when they encounter barriers to getting their tasks done.

2.9 Conclusion

In this paper, we have presented a learning-to-doing (learning transfer) study of the Idea Garden's ability to help end-user programmers help themselves.

The results were that the Idea Garden helped end users with little knowledge of programming write significantly higher-quality programs in the difficult programming task, as compared to participants who had not previously used the Idea Garden.

This finding is somewhat remarkable in that learning transfer occurred after only 25 minutes exposure to Idea Garden support. In addition, this result is the first learning transfer investigation of end-user programming that we have been able to locate in which participants did not have access to the learning supports during the transfer task itself. Thus, it showed both that participants retained the learned information and that they were able to apply it to new contexts later without help.

Finally, this study is also the first we have seen in end-user programming that investigates the difference between end users with little knowledge of programming (e.g., prior experience with html or with statistical scripts) and those with none at all. Prior studies have combined these two subpopulations, and our results suggest that, at least in some situations, the distinction is important.

In summary, the Idea Garden helped make a little programming knowledge go a long way in helping these end-user programmers in trouble to help themselves. As “active users” with no particular motivation to learn programming, these end users were able to synthesize the knowledge presented by Idea Garden and apply that knowledge without guidance or assistance. Thus, with the Idea Garden’s help, they not only learned—they learned to do.

Acknowledgments

We thank our study participants and Romina Rodriguez for her assistance with the study. This work was supported in part by NSF grants 0917366 and 1240786.

References

- [1] A. Blackwell, First steps in programming: A rationale for attention investment models, IEEE HCC, pp. 2–10, 2002.
- [2] J. Bransford, A. Brown, and R. Cocking, How People Learn: Brain, Mind, Experience, and School, Expanded ed., National Academy Press, 2000.
- [3] J. Cao, K. Rector, T. Park, S. D. Fleming, M. Burnett, and S. Wiedenbeck, A debugging perspective on end-user mashup programming, IEEE VL/HCC, pp. 149–156, 2010.
- [4] J. Cao, Y. Riche, S. Wiedenbeck, M. Burnett, and V. Grigoreanu, End-user mashup programming: Through the design lens, ACM CHI, pp. 1009-1018, 2010.
- [5] J. Cao, S. D. Fleming, and M. Burnett, An exploration of design opportunities for ‘gardening’ end-user programmers’ ideas, IEEE VL/HCC, pp. 35-42, 2011.

- [6] J. Cao, I. Kwan, R. White, S. D. Fleming, M. Burnett, and C. Scaffidi, From barriers to learning in the Idea Garden: An empirical study, *IEEE VL/HCC*, pp. 59-66, 2012.
- [7] J. Carroll. *Minimalism Beyond the Nurnberg Funnel*. MIT Press, 1998.
- [8] D. Compeau and C. Higgins, Computer self-efficacy: Development of a measure and initial test, *MIS Quarterly* 19(2), pp. 189–211, May 1995.
- [9] C. L. Corritore, B., Kracher, B., and S. Wiedenbeck, On-line trust: Concepts, evolving themes, a model. *International Journal of Human-Computer Studies*, 58(6), pp. 737 – 758, 2003.
- [10] B. Dorn, ScriptABLE: Supporting informal learning with cases, *ICER*, pp. 69-76, 2011.
- [11] V. Grigoreanu, J. Brundage, E. Bahna, M. Burnett, P. ElRif, and J. Snover. Males’ and females’ script debugging strategies. *Second International Symposium on End-User Development*, Siegen, Germany, March 2-4, 2009.
- [12] V. Grigoreanu, M. Burnett, and G. Robertson. A strategy-centric approach to the design of end- user debugging tools, *ACM CHI*, 713-722, 2010.
- [13] P. Gross, J. Yang, and C. Kelleher, Dinah: An interface to assist non-programmers with selecting program code causing graphical output, *ACM CHI*, pp. 3397-3400, 2011.
- [14] P. J. Guo, S. Kandel, J. M. Hellerstein, and J. Heer. Proactive wrangling: Mixed-initiative end-user programming of data transformation scripts. *ACM UIST*, pp. 65-74, 2011.
- [15] K. J. Harms, C. H. Kerr, and C. L. Kelleher, Improving learning transfer from stencils-based tutorials, *ACM IDC*, pp. 157-160, 2011.
- [16] A. Ko, B. Myers, and H. Aung, Six learning barriers in end-user programming systems, *IEEE VL/HCC*, pp. 199–206, 2004.
- [17] S. Kuttal, A. Sarma, and G. Rothermel, History repeats itself more easily when you log it: Versioning for mashups, *IEEE VL/HCC*, pp. 69–72, 2011.
- [18] J. Lin, J. Wong, J. Nichols, A. Cypher, and T. Lau, End-user programming of mashups with Vegemite, *ACM IUI*, pp. 97–106, 2009.
- [19] J. Myers-Levy, Gender differences in information processing: A selectivity interpretation, in *Cognitive and Affective Responses to Advertising*, P. Cafferata and A. Tybout (eds.) Lexington Books, 1989.
- [20] O’Donnell, E. and Johnson, E. N. Gender effects on processing effort during analytical procedures. *International Journal of Auditing* 5, pp.91-105, 2001.
- [21] H. Simon, Problem solving and education, in *Problem Solving and Education: Issues in Teaching and Research*, D. Tuma and F. Reif (eds.) Lawrence Erlbaum, 1980.
- [22] J. Sweller, Cognitive load during problem solving: Effects on learning, in *Cognitive Science* 12, pp. 257-285, 1988.
- [23] N. Zang and M. B. Rosson, What’s in a mashup? And why? Studying the perceptions of web-active end users, *IEEE VL/HCC*, pp. 31-38, 2009.

3 Principles of a Debugging-First Puzzle Game for Computing Education

Michael J. Lee¹, Faezeh Bahmani², Irwin Kwan², Jilian LaFerte², Polina Charters¹,
Amber Horvath², Fanny Luor¹, Jill Cao², Catherine Law², Michael Beswetherick¹,
Sheridan Long², Margaret Burnett², Andrew J. Ko¹

¹University of Washington
Seattle, Washington, USA

²Oregon State University
Corvallis, Oregon, USA

IEEE Symposium on *Visual Languages and Human-Centric Computing* (VL/HCC),
Melbourne, Australia, To appear

3.1 Abstract

Although there are many systems designed to engage people in programming, few explicitly teach the subject, expecting learners to acquire the necessary skills on their own as they create programs from scratch. We present a principled approach to teach programming using a debugging game called Gidget, which was created using a unique set of seven design principles. A total of 44 teens played it via a lab study and two summer camps. Principle by principle, the results revealed strengths, problems, and open questions for the seven principles. Taken together, the results were very encouraging: learners were able to program with conditionals, loops, and other programming concepts after using the game for just 5 hours.

3.2 Introduction

In recent years, computer programming has been proposed to be a skill that everyone can and should have. Sites like `code.org` popularize it as a path to jobs and prosperity, and government agencies such as the UK Department of Education have introduced plans to teach "rigorous computer science" to all children from 5 to 14 [35]. Programming languages and tools appear to be moving mainstream in a way that aspires to provide everyone with opportunities to learn programming at their own pace, without needing a teacher or classroom.

There are many well-known tools to help people acquire programming skills independently. For example, Scratch [20] and Alice [14], now widely used, enable people to tell interactive stories, and sites like `Codecademy.org` allow users to follow simple tutorials to learn widely used languages such as JavaScript and Python. Unfortunately, these learning technologies have limitations that interfere with teaching at scale without instructors. Scratch and Alice, while quite effective at engaging learners in telling stories, require learners to somehow learn a language and a development environment before they can begin to write their programs. Thus, these environments require teachers and other instructional resources to help learners succeed. At the other end of the continuum are tutorial tools such as `Codecademy` and games such as `RubyWarrior`, which ask learners to follow instructions typing-in and running commands in a virtual terminal. Although these environments do present programming to learners, they provide little

instruction about what is happening or why, and leave learners little room to explore. Moreover, they do not follow best practices for intelligent tutoring systems (e.g., providing detailed, immediate feedback [33]).

We have also noticed that these kinds of environments communicate in ways that can be discouraging, often framing the computers as powerful and infallible entities, rather than as the efficient but unintelligent machines that they are. This framing can contribute to learners' sense of failure if they do not initially succeed [17]. For females (or males) who view the culture of computing as elitist and view themselves as not good enough [13,21], feedback such as their programs being called “invalid” can be discouraging.

In this paper, we present an alternative approach for learning technologies that teach computing, which we call a *debugging game*, instantiated in our online game, Gidget. This type of game requires players to *debug existing programs* before going on to create their own programs in the form of puzzle levels (as in Figure 3.1). We define our new approach through seven principles, which we present next.

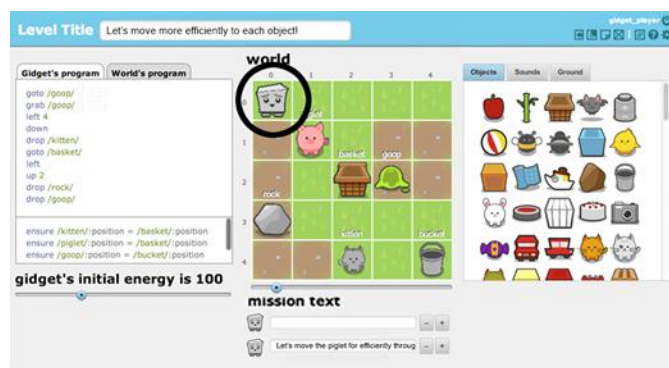


Figure 3.1. Gidget's level design mode (the Gidget character is circled). In this mode, learners design their own levels for others to solve. Players write code (left) that can include graphics (right), and see animated results (middle), and graphics for the level are on the right.

3.3 The Principles of Debugging Games

The contribution of this paper is a principled definition of the debugging game approach embodied by Gidget. We derived seven principles by drawing from best practices from game design, educational technologies, learning sciences, help systems, and by observing our players interact with earlier iterations of our evolving game, Gidget.

P1-debug. Debugging first: Encourage learners to learn programming concepts by debugging existing programs before creating new programs. Unlike many other educational technologies where creation occurs immediately [14,20], our approach provides nearly complete, but broken programs for learners to debug and fix before moving onto the more demanding task of creating new puzzles from scratch.

P2-game. Game-oriented: To make the environment be engaging to those who want to be *entertained* by solving puzzles [8,17,18,19], not just engaging to those who want to learn programming, it should feel like a game, drawing upon games’ combination of interactivity, story, and objectives to benefit learning [12].

P3-fallible. Computers as helpful but fallible: Frame computers as helpful but fallible collaborators. This is in contrast to other educational environments, which often frame the compiler, development environment, and other programming tools as all-knowing, authoritative figures, which can be discouraging for novice programmers [17].

P4-goals. Embedded goals: Give learners an explicit goal as scaffolding [28]. Provide one specific game goal – debugging faulty code – so that learners are focused and not distracted by additional objectives that can be distracting and negatively affect performance [2].

P5-instruction. Embedded instructions: Provide embedded instruction, with specific learning objectives, a planned curriculum, and an explicit, sequenced set of instructional materials and tasks [10,19]. This contrasts with open, creative environments, where learners are left free to explore at will [14,20,24].

P6-help. Scaffolded help: Deliver, on request, in-game help, including “Idea Garden” [7,8] help that provides incomplete examples, problem-solving strategies, and higher-level programming concepts to enable learners to help themselves.

P7-gender. Gender inclusiveness: Females represent 42% of all video game players in the USA [11], but are seriously underrepresented in computing fields [25]. We aim at this problem by building on best practices for reaching both males and females (e.g., [6,34,36]), such as avoiding competitive objectives and using a gender-neutral protagonist.

We call any learning environment that follows all of these principles a *debugging game*, which translates the task of debugging into game mechanics where players diagnose and fix defective programs. Given our definition and our debugging game principles, this paper investigates the following overarching research question: *How do these seven principles influence the ways novice programmers learn programming concepts and solve programming problems?*

The specific aspects of this research question we investigate in this paper are:

RQ1: What programming concepts did players struggle with when playing the game and when creating their own puzzle levels (programs)? This question aims to shed light on several of the above principles: how debugging (*P1-debug*) programs to achieve game-oriented (*P2-game*) goals (*P4-goals*) affected how participants of both genders (*P7-gender*) struggled with programming concepts, the challenges they encountered later in creating puzzle levels, and how we present both embedded instructions (*P5-instruction*) and scaffolded help (*P6-help*).

RQ2: What counterproductive problem-solving strategies did players try while playing the debugging game? This question targets how debugging game works for players solving problems on their own in the game (*P1-debug*), which includes presentation of the problems (*P3-fallible*, *P4-goals*), the instructions (*P5-instruction*) and scaffolded help (*P6-help*).

RQ3: What kinds of puzzle levels did players create after playing the debugging game, and what programming concepts did they apply? This question targets the “from debugging to creating” aspect, which rests particularly on whether the earlier instruction, help, and debugging practice was sufficient for participants to then create interesting, complex new programs (*P5-instruction*, *P6-help*, *P1-debug*).

3.4 The Gidget prototype

To investigate our research questions, we created a new version of the debugging game Gidget (Figure 3.1) that embodies the seven principles. Descriptions of earlier versions of Gidget have been reported elsewhere [17,18,19], so here we focus only on the details needed for this paper.

A story motivates the game’s objectives: a chemical spill is endangering animals and a robot named Gidget has been deployed to clean up the area (*P2-game*). Unfortunately, Gidget was damaged and is only able to provide faulty code (*P3-fallible*). It is the player’s job to help the robot by diagnosing and fixing the faulty code (*P1-debug*) to satisfy each level’s mission goals (*P4-goals*) in the form of assertions about the game’s world state.

The game has four “controls” to aid debugging: *one step*, *one line*, *to end*, and *stop* (*P1-debug*). These controls function similarly to conventional breakpoint debuggers, allowing players to run parts of the program or all of it, halt the program, and edit code at any time. When the learner uses *one step* or *one line*, Gidget provides a detailed explanation of each statement in the program, highlighting changes in the runtime environment.

The game uses an imperative, Python-like language to teach a specific set of programming concepts (*P5-instruction*) across 7 units of 34 levels. Each level starts with Gidget briefly explaining the level’s objective and providing hints about which concepts to use. The presentation order of the concepts was designed iteratively based on curricula found in CS1 textbooks, pilot testing with novices, and the authors’ cumulative experience teaching CS1 courses, following recent advice in educational game design [1]. Prior work [19] validated the curriculum as engaging to online adult participants (*P2-game*) that positively affected their attitudes towards programming, regardless of gender or level of education [9]. The units cover 1) game-specific constructs, 2) lists, 3) variables, 4) functions and objects, 5) Booleans and conditionals, 6) while and for each loops, with the final set 7) reviewing all of the concepts. Each unit ends with two assessment levels testing concepts covered in that unit [19].

Once the learner completes the curriculum (puzzle levels), they can use the level designer to create, save, modify, and share new levels. The level designer (Figure 3.1) is an interface that allows the player to write code for new levels' behavior, add introductory text to the level, change the size of the world, set the goals and original code for the level, and view the usable graphics and sounds in the game. It also introduces the concept of event handling (i.e., having objects in the game wait for a condition before running a code block), which was not covered in the game curriculum.

The game has four forms of scaffolded help (*P6-help*). First-time users see a 9-slide tutorial to learn the user interface for the game. The game has an in-game reference guide (available as a standalone help guide or as a tooltip on certain game elements), providing explanations and examples of each command in the language. The game's editor also provides keystroke-level feedback about syntax and semantics errors, highlighting erroneous code in red and explaining the problem in Gidget's speech bubble. Finally, on-demand ideas, examples, and strategies in the Idea Garden [8] style are prototyped as a combination of in-game tooltips and paper-prototyped suggestions.

Gidget's graphics, text, and game goals were all designed to be gender-inclusive (*P7-gender*). The game's story integrates socially relevant themes (i.e., cleaning a chemical spill and saving animals), helping a partner, and provides challenge through puzzles—all of which have been shown to appeal to both genders [29]. Gidget avoids game mechanics, like achievements or competition, that would possibly disengage females [37]. Following the premise that language impacts culture, it eschews violence-oriented terminology (e.g., players “remove” a game object instead of “destroying” it; players “run” or “stop” a program instead of “executing” or “killing” it) [23]. Finally, its collection of scaffolded help offers information in the “selective” and “comprehensive” style statistically favored by males and females, respectively [22].

3.5 Methods

We conducted two formative studies: a laboratory think-aloud study to record in-depth interactions with Gidget, and two summer camps to observe participants play puzzles and create levels over five days. We varied the levels, but not the concepts, between the two studies to 1) cover more concepts in one sitting during the think-aloud study, and 2)

verify with think-aloud data that it was *concepts* that participants struggled with and not the way the information was conveyed. Both studies' recruitment material avoided the word "programming" to prevent participants from self-selecting out. This paper focuses mainly on the summer camps since they included both puzzle play and level design, and triangulates against the think-aloud study's data where appropriate.

3.5.1 Think-Aloud Study

We recruited 10 college-aged teens (5 males and 5 females) for the one-on-one think-aloud laboratory study. Each was compensated \$20. None had taken programming classes beyond an introductory course required of most majors. We recorded participants playing the game on their own, completing as many levels as possible from a condensed set of 24 levels, for 81 to 97 minutes (median: 89.7). They did not use the level designer. The experimenter helped participants if they struggled for more than 3 minutes, so as to allow participants to proceed and provide data on more concepts.

3.5.2 Summer Camps

The two summer camps (which were identical, except as noted) took place on college campuses in Corvallis, Oregon and in Seattle, Washington. Each camp ran 3 hours/day for 5 days, for 15 hours total. About 5 hours were devoted to the Gidget puzzle curriculum; 5 hours to other activities such as icebreakers, guest speakers, and breaks; and 5 hours to creating new levels with the level designer and sharing them.

We recruited 34 teens aged 13-19. The Oregon camp had 10 males and 8 females with a median age of 13.5 years, and the Washington camp had 16 females with a median age of 14 years. Participants were divided into same-gender pairs of similar age and were instructed to follow pair programming practices, which are known to benefit both males and females [36]. One male participant from the Oregon camp and one female participant from the Washington camp had attended an introductory programming camp in the past. All other participants reported having no prior programming experience.

Camps used identical staff: a lead (male graduate student) led the activities and kept the camp on schedule; a researcher (female graduate student) recorded observations from a distance, and four helpers (all undergraduate females) answered questions, approached

struggling participants, and recorded observations. The staff provided no formal instruction about Gidget or programming. Helpers recorded, using pre-designed observation forms, instances when campers had problems, noting what the problem was, what steps they tried prior to asking for help, and what assistance resolved the issue.

3.5.3 Coding and Analyses

To categorize barriers participants encountered in both studies, we used two code sets from prior work (see Table 3.1). The *algorithm design barriers* are barriers that novice programmers encountered in end-user programming environments while designing algorithms [7]. The *learning phase barriers* are a sequence of barriers that novice programmers encountered when learning to program [16].

We coded each minute of the think-aloud transcripts and every observation instance from the camp forms using these code sets. Multiple codes were allowed. Two coders reached 80% agreement on 20% of the data (Jaccard index), after which one coder finished coding. Though we had 1014 minutes of video, we excluded 146 minutes of tutorial and incomplete level footage, resulting in 868 minutes of video with 878 barriers. From the camps, we recorded 793 observation notes, with 300 of these including at least one barrier.

We identified problematic concepts during puzzle play by examining the levels with the highest number of barriers (see Figure 3.2). We also identified additional concepts participants struggled with during level design. We excluded *understanding barriers*

Table 3.1. Barriers code sets (Cao et al. [7], Ko et al. [16]).

Algorithm Design Barriers	
Composition	Did not know how to combine the functionality of existing commands
More than once	Did not know how to generalize one set of commands for one object onto multiple objects
Learning Phase Barriers	
Design	Did not know what they wanted Gidget to do
Selection	Thought they knew what they wanted Gidget to do but did not know what to use to make that happen
Use	Thought they knew what to use, but did not know how to use it
Coordination	Thought they knew what specific things to use, but did not know how to use them together
Information	Thought they knew why it did not do what they expected, but did not know how to check
Understanding	Thought they knew how to use things together, but the things did not do what was expected

from both analyses because unlike other barrier types that were related to one specific part or concept in the code, these were caused by misunderstandings of several concepts that could not be mapped exclusively to one programming concept.

3.6 Results

3.6.1 *Struggles with Programming Concepts*

3.6.1.1 *Programming Concepts During Puzzle Play*

During puzzle play, participants struggled primarily with string equality, functions, and objects.

The first conceptual difficulty that affected a number of camp participants was string equality, which was introduced in Level 14. The concept caused 8 out of 29 barriers (Figure 3.2). The goal of this level required participants to change the string argument of the “set” command so that it matched “Please help me Dog!”. Participants often struggled because they had a more relaxed perception of string equality than programming requires, often setting capitalization differently or omitting the exclamation point. This was corroborated with evidence from the think-aloud study, where 3 of the 10 participants also struggled with string equality and received help from the experimenter—one participant exclaimed, “Are you kidding me?” after receiving help. Since several participants appeared unable to recognize string equality issues, it should be explicitly taught in embedded instructions (*P5-instruction*) and supported with clear examples in scaffolded help (*P6-help*).

Camp participants also had difficulties with functions, which were introduced in Level 20. They caused 22 out of 28 barriers recorded for this level (Figure 3.2). The most common issue participants faced was understanding the difference between a function *call* and a function *definition*, and many omitted function calls, assuming that the function definition would actually run the function. Furthermore, participants from both studies had trouble matching function calls with their definitions (function names were either not defined, or spelled incorrectly) or passing the wrong type or number of parameters. Participants continued to struggle with these concepts in all subsequent levels dealing with functions, particularly in levels 23 and 34 (Figure 3.2).

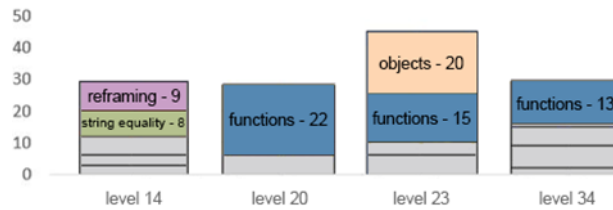


Figure 3.2. The concepts in the most challenging levels during puzzle play. Concepts accounting for fewer than 25% of the barriers in each level are shown unlabeled in gray.

The third problematic concept was defining new objects, which caused 20 out of 45 barriers in Level 23 (Figure 3.2), making it the most difficult level in the game (Table 3.2). In the camps, participants often omitted the object definition or struggled with the constructor. In addition, think-aloud participants had difficulties working with functions encapsulated within an object, often omitting or erroneously deleting the object name before the function call:

```

C14, minute 84: ... Maybe I will just put transport.
[Deletes /battery/ from /battery/:transport(Gidget,/battery/)]

```

These particular conceptual barriers have been reported in other studies as well [31], but they raise interesting design challenges for the debugging-first approach because of the tension between fun challenges versus instruction. Puzzles require intellectual engagement, but if the game provides too much instruction, the game is no longer a game, but just another tutorial—violating *P2-game*. Therefore, we must carefully balance the elements that are intellectually engaging versus the elements that can be frustrating work. Furthermore, some challenges may be trivially easy for some, and an insurmountable barrier to others, just as our data showed. Therefore, we should consider how the game can personalize the challenge, perhaps by providing context-sensitive *P6-help*, to balance engagement and instruction for a particular player.

3.6.1.2 Programming Concepts During Level Design

Once they started level design, participants encountered two new concepts that caused new barriers in addition to the ones from puzzle play: event handling (the “when” statement) and assertions (the “ensure” statement). Barriers regarding the event-handling concept were particularly high, contributing to 65 out of 238 barriers (27%) during level design.

The `when` statement, which is used for event-handling, runs a block of code when a condition is true. Participants had not seen any `when` statements during the puzzles and had a difficult time understanding how they differed from “if” (a selection barrier) and how to write a condition for a “when” (a use barrier). This appeared to stem from the small difference between the English words *if* and *when*, but the large semantic difference between the words in the game. Participants showed better understanding after helpers explained that if statements run code in sequence, and that `when` statements takes over control *whenever* its condition is satisfied, independent of where it is in the code.

In addition, the assertions concept caused 16 out of 238 barriers (6%). Assertions, implemented via the `ensure` statement, described the level goals. For example, `/gidget/:position = /button/:position` means that Gidget needs to end up on the button to “win” the level. Participants saw `ensure` statements throughout the game as they played each level, but did not have to write one until they designed their own levels. Interestingly, although participants did not encounter many barriers in the “conditionals” unit (Table 3.2, Unit 5), and did not have many problems *reading* the goals in the form of `ensure` statements (there were only 15 design barriers in Table 3.2), they struggled *writing* their own `ensure` statements, as seen in previous work [26].

Table 3.2. Number of barriers per level and the percent improvement (%imp) in barriers from the puzzle play (PZ) to level design (LD) in the summer camps. Each column contains the number of barriers in the level. Algorithm design barriers are shown in orange (top two rows), and learning phase barriers are shown in blue (five middle rows and the second-last row). Assessment levels were not coded and marked with hyphens. darker colors indicate higher counts.

	Unit 1 move/grab							Unit 2 goto/list					Unit 3 variables						Unit 4 functions/objects						Unit 5 Bool/conditionals						Unit 6 loops						Unit 7 overview			PZ	LD	Total	%imp
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39				
Composition	0	0	3	0	0	-	-	1	1	0	0	-	-	0	0	2	0	-	-	8	3	4	9	-	-	2	2	3	5	-	-	1	4	8	-	-	1	-	-	57	53	110	7
More-than-once	0	0	0	0	0	-	-	0	0	9	2	-	-	1	0	2	0	-	-	0	0	3	4	-	-	0	0	0	0	-	-	1	4	2	-	-	4	-	-	32	19	51	41
Design	0	0	0	0	0	-	-	0	4	0	1	-	-	3	0	1	0	-	-	1	0	0	2	-	-	0	0	0	0	-	-	1	1	0	-	-	1	-	-	15	2	17	87
Selection	9	0	3	0	1	-	-	1	5	9	2	-	-	12	7	2	2	-	-	10	2	3	16	-	-	1	0	2	3	-	-	3	3	6	-	-	4	-	-	106	65	171	39
Use	3	0	2	0	0	-	-	1	2	7	3	-	-	13	1	5	3	-	-	6	4	3	12	-	-	2	1	0	6	-	-	1	5	7	-	-	3	-	-	90	74	164	18
Coordination	0	0	0	0	0	-	-	1	0	0	0	-	-	0	0	1	0	-	-	3	3	3	2	-	-	2	2	2	5	-	-	1	2	6	-	-	1	-	-	34	25	59	26
Information	0	0	0	0	0	-	-	0	0	0	0	-	-	0	0	0	0	-	-	0	0	0	0	-	-	0	0	0	0	-	-	0	0	0	-	-	0	-	-	0	0	0	0
Subtotal	12	0	8	0	1	-	-	4	12	25	8	-	-	29	8	13	5	-	-	28	12	16	45	-	-	7	5	7	19	-	-	8	19	29	-	-	14	-	-	334	238	572	29
Understanding	6	1	3	0	4	-	-	3	7	5	4	-	-	15	7	3	1	-	-	13	8	4	9	-	-	0	3	4	9	-	-	2	5	10	-	-	5	-	-	131	20	151	85
Grand total	18	1	11	0	5	-	-	7	19	30	12	-	-	44	15	16	6	-	-	41	20	20	54	-	-	7	8	11	28	-	-	10	24	39	-	-	19	-	-	465	258	723	45

The barriers participants encountered with event-handling and assertions suggest that mere exposure to a programming construct in a program understanding task is not

necessarily sufficient to teach a participant how to use these constructs independently to author new behaviors. Therefore, educational technologies that require any amount of authoring have to recognize and teach code reading and writing tasks as distinct skills. In Gidget, this might be accomplished by: (1) including units that effectively combine both program understanding tasks and program writing tasks in a unit to gradually make players comfortable with writing each construct as they are introduced (*P5-instruction*), and (2) providing clearer examples and hints that relate back to previously covered and related concepts such as conditionals when trying to teach assertions (*P6-help*).

Finally, we compared the number of barriers males and females encountered within the Oregon camp and think-aloud studies, which had both genders represented (*P7-gender*). In the camp, females experienced many more barriers than males: four female teams experienced an average of *44.5 barriers/team* (126 barriers in puzzle play and 52 barriers in level design), and five male teams experienced an average of *28 barriers/team* (103 barriers in puzzle play and 37 barriers in level design). However, the think-aloud study showed no difference: both the 5 females and the 5 males averaged *6 barriers/level*. These contradictory results leave open the question of the approach’s gender-inclusiveness (*P7-gender*).

3.6.2 Counterproductive Problem-Solving Strategies.

Participants used a variety of strategies in an attempt to overcome these barriers, many of which were counterproductive. We identified their problem-solving “antipatterns” using the “Rule of Three” in accordance with the patterns research convention [30]. Five problem-solving antipatterns emerged from our data.

The “All-knowing computer” antipattern refers to a player’s failure to scrutinize the original code, even though they were told that it was filled with errors (as in *P3-fallible*). Instead, they largely trust that the original code is correct. The belief that the computer was always correct—observed also by Beckwith et al. [3] in a context that did *not* explicitly inform their participants that the code was incorrect—eventually led to many other barriers. In the think-aloud study, the original code from one level properly used a function call that was encapsulated within an object, and no one struggled with

encapsulation. But, in a later level, 3 out of 4 participants who skimmed over the original code that used a function call with the wrong object struggled with the concept.

In the “Reinvent the wheel” antipattern, a player deletes the original code without reading it and misses out on clues the code provides. Participants who used this antipattern could not benefit from one of the potential merits of the debugging-game approach, which is getting ideas from the original code. We observed that Team Heat from the camp used this antipattern in Level 23 and subsequently missed a clue indicating that there should be an object definition for every object, resulting in a selection barrier. When learners asked for help, the helper suggested that they restore the original code and read it.

The “When all you have is a hammer, everything looks like a nail” antipattern is where a player persists in using programming constructs that worked for earlier levels but are no longer applicable. The reflection-in-action model [32] points to the importance of reframing when devising solutions. This rigidity was problematic for several participants: for example, recall in Level 14 that the explicit goal was to ask the dog “Please help me Dog!” for help with the task. It was necessary to use the set command to set a variable to that string, but 8 teams ignored the set command and tried to use previously learned commands (such as goto), leading to 9 out of 29 barriers in Level 14 (Figure 3.2).

In the “I don’t want to try it” antipattern, participants avoid trying ideas. For example, We observed that Team Asian asked a helper whether multiple conditions could be used in an if statement and Team Heat asked if Gidget could grab multiple items. In both cases, the helper suggested they try it in the game to see what happens.

Finally, in the “I’ll use it as it is” antipattern, a player fails to adapt an existing example (e.g., from a tooltip or help sheet) to its particular context. This demonstrates a lack of analogical reasoning [27] in contrast to experienced programmers who are comfortable adapting examples [5]. In one instance from the think-aloud study, a participant looked up an example:

C10, minute 28: I am looking at nickname. Should I nickname the kitten?
[Reads the “nickname” tooltip and clicked on “name” in the tooltip]
I want to see an example. There is an example here ... Ok. So, I found an example saying “say /gidget/:name”. So, I’m going to try it again with kitten.

This prompted him to change the set command, which was correct, to the incorrect say from the example.

One possible explanation of the problem-solving antipatterns “All-knowing computer” and “Reinvent the wheel” may be that our participants wanted to avoid reading code, which could interfere with their learning how to understand programs. A debugging-game approach may need to incentivize program understanding. But doing so may be difficult: the attention investment model [4] predicts that understanding the program would need to seem (to learners) to have lower perceived costs, higher perceived benefit and/or lower perceived risk than writing code from scratch.

3.6.3 From Debugging-First to Programming: Level Creation

After only about 5 hours of self-directed instruction with our debugging game, participant teams from our two camps created 101 Gidget levels, with every team applying programming concepts in this creation process. We examined these participant-created levels, focusing particularly on the programming concepts used in the levels and the level’s story, as storytelling elements in these environments are known to affect engagement [14,15].

Each team created between 2 and 10 levels (median: 5). The majority (66/101) of the levels created were Gidget puzzles (e.g., Figure 3.3) or mazes meant to challenge other players, but some participants also had partially-completed or proof-of-concept levels (21/101). Some participants repurposed the level designer for unintended functionality. For example, team mustache built three levels to hold solutions to their other levels, Epsilon made 2 story-related levels without any puzzle-solving elements, and three teams from the Oregon camp used the level designer to draw pixel art. Overall, teams faced very few design barriers (2/258, Table 3.2), suggesting that they had many ideas for levels after playing through the game.

Every team designed two or more complete levels that used at least one of the taught programming constructs (see Table 3.3). The minimum knowledge to create a Gidget level is a Boolean expression to indicate a goal. Non-trivial Gidget levels (such as Figure 3) require knowledge of variables, Booleans, objects, and events. Thirteen teams

designed levels that required programming concepts such as conditionals, loops, or the event-

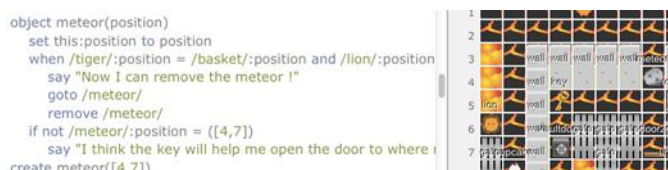


Figure 3.3. Camp participants most often created puzzle levels to challenge other players. Team Mustache developed this level where Gidget had to rescue animals, remove a meteor, and more. The code fragment contains objects, the event-driven when statement, and a conditional statement.

driven “when” statement and 6 used every concept in Gidget. All teams used at least one Boolean expression in their levels since it was mandatory to have a goal (written as a test case). Additionally, many teams (76%) used events in their levels so that an automatic event would occur as part of their stories. Some teams demonstrated their knowledge by writing their own incomplete puzzle code containing functions and loops for other players to debug.

Most teams motivated their levels using stories in Gidget’s mission text: 14 of 17 teams motivated at least one level with story text. Four teams each created multiple levels with a continuous story thread. The Gidget character was popular as a domestic figure (having a house or partner) or as an altruistic hero (often rescuing animals in outer space). None of our participants developed stories focused on popular culture as observed in other camp studies [20]; this may have been due to participants treating Gidget as a character upon which they could build their own ideas.

In the relatively short 5 hours allocated to level design, participants were able to try out many ideas and share results with their peers at every stage of their progress. Despite the fact that the level designer had the constraints of a 2D world and Gidget rules, our participants used it to not only program challenging puzzles, but to also tell imaginative stories.

3.6.4 Overcoming Barriers: Practice Makes Perfect?

One measure of whether participants in the camps learned from playing Gidget is to see if they encountered fewer barriers in puzzle play compared to level design. Using team-by-

team barrier data (similar to those calculated in the right-most columns of Table 3.2), we calculated each team's percent improvement per barrier type (Figure 3.4). We saw improvements in 15 out of 17 camp teams and an overall improvement of 45% from puzzle play to level design (see Table 3.2, lower-right corner). One explanation for the improvements is that teams used only basic programming concepts in their level designs, but Table 3.3 shows that only 4 teams constructed levels requiring 2 or fewer programming constructs, so this explanation does not hold for the other 11 teams. For these 11 teams, the best explanation for their improvement is that they did improve their programming skills while playing through the debugging puzzles.

We believe that two teams, Team Asian and Team Mustache (see Figure 3.4), did not improve on the number of barriers because they devised and implemented ambitious levels. Both teams used all six programming constructs in their levels. Team Mustache encountered 500% more (12 barriers in level design vs. 2 in the puzzle portion) learning phase barriers during level design than in puzzle play, but created multiple levels (such as Figure 3.3) incorporating complex concepts such as a when statement with multiple Boolean expressions to verify players completed objectives sequentially.

There was a noticeable difference in the amount of improvement in algorithm design barriers (19%) vs. learning phase barriers (51%). Table 3.2 shows the improvements for each barrier type (rightmost column). Two of the learning phase barriers improved by nearly 90%, compared to the best algorithm design barrier improvement of 41%. Furthermore, as Figure 3.4 shows, 15 teams improved on learning phase barriers, whereas only 10 teams improved on algorithm design barriers (Figure 4). Teams especially struggled with composition barriers, encountering them frequently but demonstrating only 7% improvement—the least amount of improvement out of all barrier types (Table 3.2). The fact that the algorithm design barriers did not greatly improve with instruction and practice from the game suggests that algorithm design concepts may require more thorough explanations and help (*P5-instruction*, *P6-help*) than what is currently provided.

In addition, despite the contradictory results in Section A.2 regarding the number of barriers per person in both genders, both genders had similar improvements with 58% and 64% for females and males, respectively. This positive evidence of gender

inclusiveness (*P7-gender*) is encouraging as to both females' and males' learning through this approach.

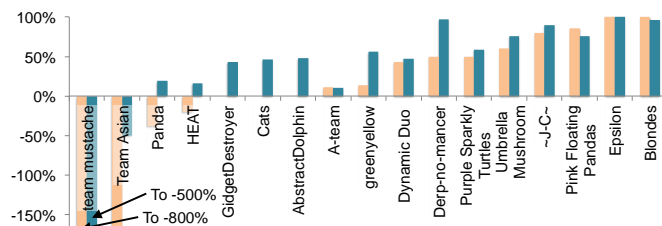


Figure 3.4. Percent improvement by comparing barriers before, then during level design. 15 of 17 teams showed improvements, with greater improvement on learning phase barriers (dark) than on algorithm design barriers (light).

3.7 Discussions and Implications

The results from our two studies suggest several strengths and weaknesses about the seven design principles.

First, taken together, the seven principles in Gidget succeeded in teaching enough programming for participants to successfully write their own programs. Everyone finished the game in under 5 hours. Along the way, participants gradually learned to overcome many of their earlier learning phase barriers (51% improvement), although their ability to overcome their algorithm design barriers was less impressive (19% improvement). Still, the complexity and breadth of the levels the participants were able to create was impressive given their short learning time. For example, half the teams decided to use loops and functions in their custom levels and succeeded at doing so (Table 3.3), which are often major difficulties for novices in other programming languages.

Principle *P3-fallible* has previously been shown [17] to be important in helping learners focus on their progress rather than on their failures/mistakes, and it seemed to promote engagement among our camp participants. However, we also identified problem-solving antipatterns that suggest that participants trusted the original code too much and did not scrutinizing it thoroughly. This suggests that even stronger messaging that the

computer (and original code) is fallible is needed—while at the same time not further dissuading learners from reading and understanding the code.

Table 3.3. Teams using the concept in at least one level they created are marked (✓). 6 teams demonstrated usage of all 6 concepts and 13 of 17 teams demonstrated 3 or more concepts.

Team	Bool.	Var.	Cond.	Loops	Func.	Event	Total
Purple Sparkly Turtles	✓	×	×	×	×	×	1
Derp-no-mancer	✓	✓	×	×	×	×	2
Blondes	✓	✓	×	×	×	×	2
GidgetDestroyer	✓	✓	×	×	×	×	2
Epsilon	✓	✓	×	×	×	✓	3
Umbrella Mushroom	✓	✓	×	×	×	✓	3
~J-C~	✓	✓	×	×	×	✓	3
greenyellow	✓	✓	×	×	×	✓	3
Pink Floating Pandas	✓	✓	×	×	×	✓	3
Cats	✓	✓	×	✓	×	✓	4
HEAT	✓	✓	×	✓	✓	✓	5
Team Asian	✓	✓	✓	✓	✓	✓	6
A-team	✓	✓	✓	✓	✓	✓	6
Panda	✓	✓	✓	✓	✓	✓	6
team mustache	✓	✓	✓	✓	✓	✓	6
AbstractDolphin	✓	✓	✓	✓	✓	✓	6
Dynamic Duo	✓	✓	✓	✓	✓	✓	6
Percent of Teams	100%	94%	35%	47%	41%	76%	-

Nuances regarding *P5-instruction* and *P6-help* have been discussed throughout this paper. Our instantiation of these principles in the current Gidget allowed participants to complete the curriculum largely independently, and the learning they achieved transferred beyond the puzzle-based curriculum to the level design phase. However, much of the participants' learning was limited to learning phase barriers: the algorithm design barrier improvement was much lower (Table 3.2). There were also recurring struggles with concepts such as string equality, functions, and objects (Table 3.2). These findings suggest that the type of static, contextual help in the current version of the game may be sufficient for teaching lower level concepts such as language syntax and semantics, but not for teaching algorithm design problem solving skills. Future work is necessary to identify appropriate ways of teaching these higher level skills in computing education learning technologies.

Finally, with respect to *P7-gender*—e.g., avoiding competitive orientation, gender-neutral protagonist, etc.—both genders were able to learn from Gidget’s debugging game approach. Though females in the Oregon camp encountered more barriers on average than males in the same camp, they improved at a similar rate. Nearly all participants showed a strong affinity to the Gidget character and were enthusiastic in their efforts to learn to communicate with it during both puzzle play and puzzle design. Nonetheless, these results suggest a need to further investigate how the other principles, especially scaffolded help, should be improved such that it adheres more to gender inclusiveness.

3.8 Conclusion

The debugging games approach avoids the problem where learners need a large amount of programming knowledge before they can begin creating their own programs. We found that the seven design principles used to create Gidget worked together in many different capacities to successfully teach programming concepts in just 5 hours to learners who did not necessarily want to learn programming. Debugging games and more broadly, educational technologies such as Alice, Scratch, Codecademy, and other creative environments and tutorials may benefit from adopting the design principles explored in this paper. For example, adopting a debugging-first approach may empower users to learn without requiring an instructor, teach them important program understanding and debugging skills, and can lead to more success at creating their own programs. Revising the communication and instruction that environments provide to frame computers as fallible entities may also play an important role in sustaining learners’ motivation.

Ultimately, if computer programming is ever to become mainstream, we must further explore the potential benefits of debugging games and other learner-centered approaches to teaching computing that can scale to millions of people. As our results indicate, the debugging game approach and its debugging-first, gender-inclusive, help-yourself puzzle game principles to computing education is not only a viable way forward, but one that learners can actually find captivating, engaging and fun:



Acknowledgment

We thank our participants. This work was supported in part by the National Science Foundation (NSF) under Grants CNS-1240786, CNS-1240957, CNS-1339131, CCF-0952733, CCF-1339131, IIS-1314356, IIS-1314384, and OISE-1210205. Any opinions, findings, conclusions or recommendations are those of the authors and do not necessarily reflect the views of NSF.

References

- [1] Aleven, V., Myers, E., Easterday, M., & Ogan, A. (2010). Toward a framework for the analysis and design of educational games. *IEEE DIGITEL*, 69-76.
- [2] Andersen, E., Liu, Y. E., Snider, R., Szeto, R., Cooper, S., & Popović, Z. (2011). On the harmfulness of secondary game objectives. *ACM FDG*, 30-37.
- [3] Beckwith, L., Burnett, M., Cook, C. (2002). Reasoning about many-to-many requirement relationships in spreadsheets. *IEEE VL/HCC*, 149-157.
- [4] Blackwell, A.F. (2002). First steps in programming: A rationale for attention investment models. *IEEE HCC*, 2-10.
- [5] Brandt, J., Guo, P.J., Lewenstein, J., Dontcheva, M., & Klemmer, S.R. (2009). Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. *ACM CHI*, 1589-1598.
- [6] Burnett, M., Beckwith, L., Wiedenbeck, S., Fleming, S.D., Cao, J., Park, T.H., Grigoreanu, V., Rector, K. (2011). Gender pluralism in problem-solving software, *Interacting with Computers*, 23, 450-460.
- [7] Cao, J., Kwan, I., White, R., Fleming, S., Burnett, M., & Scaffidi, C. (2012). From barriers to learning in the Idea Garden: An empirical study. *IEEE VL/HCC*, 59-66.
- [8] Cao, J., Kwan, I., Bahmani, F., Burnett, M., Fleming, S.D., Jordahl, J., Horvath, A., & Yang, S. (2013). End-user programmers in trouble: Can the Idea Garden help them to help themselves? *IEEE VL/HCC*.

- [9] Charters, P., Lee, M.J., Ko, A.J., & Loksa, D. (2013). Challenging Stereotypes and Changing Attitudes: The effect of a brief programming encounter on adults' attitudes toward programming. ACM SIGCSE.
- [10] Ellis, A. (2005). Research On Educational Innovations. Eye On Education, Inc., Larchmont, NY
- [11] ESA (2011). Essential facts about the computer and video game industry. Entertainment Software Association. Web. 21 Feb. 2012.
<http://www.theesa.com/facts/pdfs/ESA_EF_2011.pdf>
- [12] Gee, J.P. (2003). What video games have to teach us about learning and literacy. *Computers in Entertainment*, 1(1), 20.
- [13] Goode, J., Estrella, R., & Margolis, J. (2006). Lost in translation: Gender and high school computer science, In *Women and Information Technology: Research on Underrepresentation*, MIT Press, 89-114.
- [14] Kelleher, C., Pausch, R., & Kiesler, S. (2007). Storytelling Alice motivates middle school girls to learn computer programming. ACM CHI, 1455-1464.
- [15] Kerr, J., Kelleher, C., Ellis, R. & Chou, M (2013). Setting the scene: scaffolding stories to benefit middle school students learning to program. IEEE VL/HCC, 95-98.
- [16] Ko, A.J., Myers, B.A., & Aung, H. (2004). Six learning barriers in end-user programming systems. IEEE VL/HCC, 199-206.
- [17] Lee, M.J. & Ko, A.J. (2011). Personifying programming tool feedback improves novice programmers' learning. ACM ICER, 109-116.
- [18] Lee, M.J., Ko, A.J. (2012). Investigating the role of purposeful goals on novices' engagement in a programming game. IEEE VL/HCC, 163-166.
- [19] Lee, M.J., Ko, A.J., & Kwan, I. (2013). In-game assessments increase novice programmers' engagement and level completion speed. ACM ICER, 153-160.
- [20] Maloney, J.H., Peppler, K., Kafai, Y., Resnick, M., & Rusk, N. (2008). Programming by choice: Urban youth learning programming with scratch. ACM SIGCSE Bulletin, 40(1), 367-371.
- [21] Margolis, J. & Fisher, A. (2003). *Unlocking the Clubhouse: Women in Computing*, MIT Press.
- [22] Meyers-Levy, J. (1989). Gender differences in information processing: A selectivity interpretation, In *Cognitive and Affective Responses to Advertising*, Lexington Books, 219-260.
- [23] Misa, T. (2010). Gender codes: Defining the problem, in *Gender Codes: Why Women are Leaving Computing*, Wiley, 3-24.
- [24] Monroy-Hernández, A., & Resnick, M. (2008). Empowering kids to create and share programmable media. *Interactions*, 15(2), 50-53.

- [25] NCWIT (2010). NCWIT Scorecard: A report on the status of women in information technology. Nat'l Ctr. for Women & IT. Web. 30 Mar. 2013.
<<http://www.ncwit.org/pdf/Scorecard2010.pdf>>
- [26] Pane, J., & Myers, B. (2006). More natural programming languages and environments, In End User Development, Springer, 31-50.
- [27] Polya, G. (1971). How to Solve It: A New Aspect of Mathematical Method, Princeton Univ. Press.
- [28] Ram, A., & Leake, D.B. (1995). Goal-Driven Learning. MIT Press, Boston, MA.
- [29] Reinecke, L., Trepte, S., & Behr, K.M. (2008). Why Girls Play. Results of a Qualitative Interview Study with Female Video Game Players. Universitäts- und Landesbibliothek.
- [30] Rising, L. (1999). Patterns: A way to reuse expertise. IEEE Communications, 37(4), 34-36.
- [31] Scaffidi, C., & Chambers, C. (2012). Skill progression demonstrated by users in the Scratch animation environment. Int'l J. HCI, 28(6) 383-398.
- [32] Schön, D.A. (1983). The Reflective Practitioner: How Professionals Think in Action. Basic Books, NY.
- [33] Shute, V.J. (1993). A macroadaptive approach to tutoring. Journal of AI in Education, 4(1), 61-93.
- [34] Subrahmaniyan, N., Kissinger, C., Rector, K., Inman, D., Kaplan, J., Beckwith, L., & Burnett, M. (2007). Explaining debugging strategies to end-user programmers. IEEE VL/HCC, 127-136.
- [35] UK DFE (2013). National Curriculum in England: Computing Programmes of Study. (Dept. Education No. DFE-00171-2013). UK.
- [36] Werner, L.L., Hanks, B., & McDowell, C. (2004). Pair-programming helps female computer science students. ACM JERIC, 4(1).
- [37] Yee, N. (2006). Motivations for play in online games. Cyber Psychology & Behavior, 9(6), 772-775.

4 Conclusions

Overall, the two presented papers provided positive evidence for the success of the IdeaGarden and Debugging-first approach. The IdeaGarden approach helped end-user programmers with their barriers and the Debugging-first approach taught them programming while circumventing the main limitations of other similar approaches. Additionally, our results provided implications for the design of the Idea Garden in a Debugging-first environment, enhancing the Idea Garden approach, the Debugging-first approach and other similar approaches that aim to help end-user programmers with their barriers and teach them programming.

The first paper showed that the Idea Garden approach succeeded in creating a new approach for helping end-user programmers. This approach is different from other approaches in that it attempts to nurture the end-user programmers' problem-solving skills. Our results showed that end-user programmers with little programming knowledge (e.g., prior experience with html or with statistical scripts) could put their prior learning from the IG into practice during a difficult task, when the IG was no longer present. This is an important finding since it is the first to reveal that there is a distinction between end-user programmers with little knowledge of programming and those with no knowledge of programming. This distinction could be well considered by other approaches aimed at teaching programming to end-users.

The second paper provided positive evidence for the success of the Debugging-first approach in teaching programming to end users. This approach overcame the main limitation of other similar approaches, such as Alice, Scratch and Codecademy. For example, our results showed that unlike existing approaches, our learners were not required to acquire a large amount of programming knowledge before they could create their own programs. This might suggest that debugging games and more broadly, educational technologies could benefit from adopting the seven design principles of the Debugging-first approach.

Overall, the papers provided evidence that helping end-user programmers help themselves and debugging-first are viable ways forward for helping end users overcome barriers and teaching them programming.

Bibliography

- V. Aleven, E. Myers, M. Easterday, and A. Ogan., Toward a framework for the analysis and design of educational games, *IEEE DIGITEL*, 69-76, 2010.
- E. Andersen, Y. E. Liu, R. Snider, R. Szeto, S. Cooper, and Z. Popović, On the harmfulness of secondary game objectives, *ACM FDG*, 30-37, 2011.
- L. Beckwith, M. Burnett, and C. Cook, Reasoning about many-to-many requirement relationships in spreadsheets, *IEEE VL/HCC*, 149-157, 2002.
- A. Blackwell, First steps in programming: A rationale for attention investment models, *IEEE HCC*, 2-10, 2002.
- J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, Two studies of opportunistic programming: interleaving web foraging, learning, and writing code, *ACM CHI*, 1589-1598, 2009.
- J. Bransford, A. Brown, and R. Cocking, *How People Learn: Brain, Mind, Experience, and School*, Expanded ed., National Academy Press, 2000.
- M. Burnett, L. Beckwith, S. Wiedenbeck, S. D. Fleming, J. Cao, T. H. Park, V. Grigoreanu, and K. Rector, Gender pluralism in problem-solving software, *Interacting with Computers*, 23, 450-460, 2011.
- J. Cao, I. Kwan, R. White, S. D. Fleming, M. Burnett, and C. Scaffidi, From barriers to learning in the Idea Garden: An empirical study. *IEEE VL/HCC*, 59-66, 2012.
- J. Cao, I. Kwan, F. Bahmani, M. Burnett, S. D. Fleming, J. Jordahl, A. Horvath, and S. Yang, End-user programmers in trouble: Can the Idea Garden help them to help themselves?, *IEEE VL/HCC*, 151-158, 2013.
- J. Cao, K. Rector, T. Park, S. D. Fleming, M. Burnett, and S. Wiedenbeck, A debugging perspective on end-user mashup programming, *IEEE VL/HCC*, 149–156, 2010.
- J. Cao, Y. Riche, S. Wiedenbeck, M. Burnett, and V. Grigoreanu, End-user mashup programming: Through the design lens, *ACM CHI*, 1009-1018, 2010.
- J. Cao, S.D. Fleming, and M. Burnett, An exploration of design opportunities for ‘gardening’ end-user programmers’ ideas, *IEEE VL/HCC*, 35-42, 2011.
- J. Cao, *Helping End-User Programmers Help Themselves – The Idea Garden Approach*, Oregon State University, PhD Dissertation, 2013.
- J. Carroll, *Minimalism Beyond the Nurnberg Funnel*, MIT Press, 1998.
- P. Charters, M. J. Lee, A.J.Ko, and D. Loksa, Challenging Stereotypes and Changing Attitudes: The effect of a brief programming encounter on adults' attitudes toward programming, *ACM SIGCSE*, 653-658, 2013.
- C. Chambers and C. Scaffid, Struggling to excel: A field study of challenges faced by spreadsheet users, *IEEE VL/HCC*, 187-194, 2010.

- D. Compeau and C. Higgins, Computer self-efficacy: Development of a measure and initial test, *MIS Quarterly* 19(2), 189–211, 1995.
- C. L. Corritore, B. Kracher, and B. S. Wiedenbeck, On-line trust: Concepts, evolving themes, a model, *International Journal of Human-Computer Studies*, 58(6), 737 – 758, 2003.
- B. Dorn, ScriptABLE: Supporting informal learning with cases, *ICER*, 69-76, 2011.
- B. Dorn, and M. Guzdial, Learning on the job: Characterizing the programming knowledge and learning strategies of web designers, *ACM CHI*, 703-712, 2010.
- A. Ellis, *Research On Educational Innovations, Eye On Education*, 2005.
- ESA, Essential facts about the computer and video game industry, Entertainment Software Association, 2011.
- J. P. Gee, What video games have to teach us about learning and literacy, *Computers in Entertainment*, 1(1), 20, 2003.
- J. Goode, R. Estrella, and J. Margolis, Lost in translation: Gender and high school computer science, in *Women and Information Technology: Research on Underrepresentation*, MIT Press, 2006.
- V. Grigoreanu, J. Brundage, E. Bahna, M. Burnett, P. ElRif, and J. Snover. 2009. Males' and females' script debugging strategies. Second International Symposium on End-User Development, Siegen, Germany, March 2-4.
- V. Grigoreanu, M. Burnett, and G. Robertson, A strategy-centric approach to the design of end-user debugging tools, *ACM CHI*, 713-722, 2010.
- P. Gross and C. Kelleher, Non-programmers identifying functionality in unfamiliar code: strategies, *IEEE VL/HCC*, 75-82, 2009.
- P. Gross, J. Yang, and C Kelleher, Dinah: An interface to assist non-programmers with selecting program code causing graphical output, *ACM CHI*, 3397-3400, 2011.
- P. J. Guo, S. Kandel, J. M. Hellerstein, and J. Heer, Proactive wrangling: Mixed-initiative end-user programming of data transformation scripts, *ACM UIST*, 65-74, 2011.
- K. J. Harms, C. H. Kerr, and C. L. Kelleher, Improving learning transfer from stencils-based tutorials, *ACM IDC*, 157-160, 2011.
- C. Kelleher, R. Pausch, and S. Kiesler, Storytelling Alice motivates middle school girls to learn computer programming, *ACM CHI*, 1455-1464, 2007.
- J. Kerr, C. Kelleher, R. Ellis, and M. Chou, Setting the scene: scaffolding stories to benefit middle school students learning to program, *IEEE VL/HCC*, 95-98, 2013.
- C. Kissinger, M. Burnett, S. Stumpf, N. Subrahmaniyan, L. Y. Beckwith, S. Yang, and M. B. Rosson, Supporting end-user debugging: What do users want to know?, *AVI*, 135-142, 2006.

- A. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, M., ... and S. Wiedenbeck, The state of the art in end-user software engineering, *ACM Computing Survey*, 43(3), 21:1-21:44, 2011.
- A. Ko, B. Myers, and H. Aung, Six learning barriers in end-user programming systems, *IEEE VL/HCC*, 199–206, 2004.
- S. Kuttal, A. Sarma, and G. Rothermel, History repeats itself more easily when you log it: Versioning for mashups, *IEEE VL/HCC*, 69–72, 2011.
- M. J. Lee, F. Bahmani, I. Kwan, J. LaFerte, P. Charters, A. Horvath, F. Luor, J. Cao, C. Law, M. Beswetherick, S. Long, M. Burnett, and A. J. Ko, Principles of a Debugging-First Puzzle Game for Computing Education, *IEEE VL/HCC*, to appear, 2014.
- M. J. Lee, and A. J. Ko, Personifying programming tool feedback improves novice programmers' learning, *ACM ICER*, 109-116, 2011.
- M. J. Lee, A. J. Ko, Investigating the role of purposeful goals on novices' engagement in a programming game. *IEEE VL/HCC*, 163-166, 2012.
- M. J. Lee, A. J. Ko, and I. Kwan, In-game assessments increase novice programmers' engagement and level completion speed, *ACM ICER*, 153-160, 2013.
- J. Lin, J. Wong, J. Nichols, A. Cypher, and T. Lau, End-user programming of mashups with Vegemite, *ACM IUI*, 97–106, 2009.
- J. H. Maloney, K. Peppler, Y. Kafai, M. Resnick, and N. Rusk, Programming by choice: Urban youth learning programming with scratch, *ACM SIGCSE Bulletin*, 40(1), 367-371, 2008.
- J. Margolis, and A. Fisher, *Unlocking the Clubhouse: Women in Computing*, MIT Press, 2003.
- J. Meyers-Levy, Gender differences in information processing: A selectivity interpretation, in *Cognitive and Affective Responses to Advertising*, P. Cafferata and A. Tybout (eds.) Lexington Books, 1989.
- R. Miller, M. Bolin, L. Chilton, G. Little, M. Webber, and C. H. Yu, Rewriting the web with chickenfoot, in *No Code Required: Giving Users Tools to Transform the Web*, A. Cypher, M. Dontcheva, T. Lau, and J. Nichols Morgan Kaufmann, 2010.
- T. Misa, Gender codes: Defining the problem, in *Gender Codes: Why Women are Leaving Computing*, Wiley, 2010.
- A. Monroy-Hernández, and M. Resnick, Empowering kids to create and share programmable media, *Interactions*, 15(2), 50-53, 2008.
- B. Nardi, *A Small Matter of Programming: Perspectives on End-User Computing*, MIT Press, 1993.
- NCWIT, *NCWIT Scorecard: A report on the status of women in information technology*, Nat'l Ctr. for Women & IT, 2010.
- E. O'Donnell, and E. N. Johnson, Gender effects on processing effort during analytical procedures, *International Journal of Auditing*, 5(2), 91-105, 2000.

- J. Pane and B. Myers, More natural programming languages and environments, In End User Development, 31-50, 2006.
- G. Polya, How to Solve It: A New Aspect of Mathematical Method, Princeton Univ. Press, 1971.
- A. Ram and D. B. Leake, Goal-Driven Learning, MIT Press, 1995.
- L. Reinecke, S. Trepte, and K. M. Behr, Why Girls Play. Results of a Qualitative Interview Study with Female Video Game Players, Technical Report, Universität Hamburg, 2008.
- A. Repenning, and A. Ioannidou, Broadening participation through scalable game design, ICSE, 305–309, 2008.
- L. Rising, Patterns: A way to reuse expertise, IEEE Communications, 37(4), 34-36, 1999.
- C. Scaffidi, and C. Chambers, Skill progression demonstrated by users in the Scratch animation environment, Int'l J. HCI, 28(6), 383-398, 2012.
- D. A. Schön, The Reflective Practitioner: How Professionals Think in Action, Basic Books, NY, 1983.
- V. J. Shute, A macroadaptive approach to tutoring, Journal of AI in Education, 4(1), 61-93, 1993.
- H. Simon, Problem solving and education, in Problem Solving and Education: Issues in Teaching and Research, D. Tuma and F. Reif (eds.) Lawrence Erlbaum, 1980.
- N. Subrahmaniyan, C. Kissinger, K. Rector, D. Inman, J. Kaplan, L. Beckwith, and M. Burnett., Explaining debugging strategies to end-user programmers. IEEE VL/HCC, 127-136, 2007.
- J. Sweller, Cognitive load during problem solving: Effects on learning, in Cognitive Science 12, 257-285, 1988.
- UK DFE, National Curriculum in England: Computing Programmes of Study, Dept. Education No. DFE-00171-2013, 2013.
- L. L. Werner, B. Hanks, and C. McDowell, Pair-programming helps female computer science students, ACM JERIC, 4(1), 2004.
- N. Yee, Motivations for play in online games, Cyber Psychology & Behavior, 9(6), 772-775, 2006.
- N. Zang and M. B. Rosson, What's in a mashup? And why? Studying the perceptions of web-active end users, IEEE VL/HCC, 31-38, 2008.
- N. Zang, and M. B. Rosson, Playing with information: How end users think about and integrate dynamic data, IEEE VL/HCC, 85-92, 2009.