

AN ABSTRACT OF THE THESIS OF

William Jernigan for the degree of Master of Science in Computer Science presented on October 26, 2015.

Title: Generalizing the Idea Garden: Principles and Contexts

Abstract approved:

Margaret M. Burnett

In previous work, the Idea Garden was created to help those relatively new to programming overcome their barriers in CoScripter. The goal of this thesis was to generalize the Idea Garden's success to other users and environments. We present a set of principles on how to help EUPs like this learn just a little when they need to overcome a barrier. We then instantiate the principles in a prototype and empirically investigate the principles in two studies: a formative think-aloud study and a pair of summer camps attended by 42 teens. Among the surprising results were the complementary roles of implicitly actionable hints versus explicitly actionable hints, and the importance of both context-free and context-sensitive availability. Under these principles, the camp participants required significantly less in-person help than in a previous camp to learn the same amount of material in the same amount of time. Furthermore, a third study including another pair of summer camps with 48 teens revealed that problem solving instruction coupled with Idea Garden helped the experimental condition advance to debugging more often and depend on helpers less than the control group.

©Copyright by William Jernigan

October 26, 2015

All Rights Reserved

Generalizing the Idea Garden: Principles and Contexts

by
William Jernigan

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented October 26, 2015

Commencement June 2016

Master of Science thesis of William Jernigan presented on October 26, 2015

APPROVED:

Major Professor, representing Computer Science

Head 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.

William Jernigan, Author

ACKNOWLEDGEMENTS

I want to thank Margaret Burnett for all the guidance she has given me. She has gathered this awesome group of researchers at Oregon State and built her network with great researchers abroad, helping her students succeed.

I thank Jill Cao for all her work on the Idea Garden. Her work gave me the opportunity to produce this thesis and inspired much of the work that we did.

I would like to acknowledge Irwin Kwan, Faezeh Bahmani, Michael Lee, Andrew Ko, Sandeep Kuttal, Anicia Peters, and Dastyni Loksa for setting an example, working hard on our research, and guiding me toward success.

Amber Horvath, Jilian LaFerte, Taylor Culty, Shannon Ernst, Alannah Oleson, and Chris Mendez did so much of the work that makes our research group look good, and they kept pushing me to do my best. Thanks to you.

I also thank Sheridan Long, Renuka Bhatt, Leah Hanen, Rory Moeller, and Claire Richards for giving everyone on the team new perspectives and motivation to do good work while helping propel our efforts.

I thank Andrew Faulring for his help with the technical aspects of Study #3.

Finally, I thank my wife Phaedra for being there for me every day and pushing me to be my very best.

TABLE OF CONTENTS

	<u>Page</u>
Introduction	1
My role in this research	2
Background and Related Work	3
The Idea Garden Principles	6
The Principles Concretely: Idea Garden Prototype in Gidget	9
The Idea Garden Prototype for Gidget	9
The prototype's support for the 7 principles	10
Antipattern support for the principles	11
Generalized Idea Garden Architecture	13
Example	15
Porting Idea Garden to new environments	16
Study #1: Principled Formative Study	18
Study #2 (Summative): The Principles go to Camp	22
Study #2 Results	24
Successes	24
Teams' Behaviors with P2-Relevance and P6-Availability	25
Teams' Behaviors with P3-Actionable	27
Teams' Behaviors with P5-Information Processing	28
How Much Did They Learn?	30
Study #3 Method	33

TABLE OF CONTENTS (CONTINUED)

	<u>Page</u>
Participants	34
The Camps	35
The Instruction	35
The Project	37
Data Collection	39
Study #3 Results	41
Summary of treatment results beyond the scope of this thesis	41
Differences between treatments in barriers and help requested	41
Camper experiences	43
Conclusion	45
Bibliography	48
Appendices.....	51
Appendix A: Idea Garden Hints in Gidget	52
Appendix B: Idea Garden Hints in Cloud9	63

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
1. Idea Garden icons in Gidget	9
2. Idea Garden hints in Gidget	10
3. Idea Garden Architecture	13
4. Team Mouse uses Idea Garden	27
5. Team Tiger's "River Dam" created level	30
6. Problem solving paper handout	33
7. Idea Garden in Cloud9	34
8. Camper E27's final project website	37

LIST OF TABLES

<u>Table</u>	<u>Page</u>
1. Prior evidence of Idea Garden Principles	8
2. Pairs of abstract events and abstract actions	15
3. Study #1 and #2 Barrier codes and outcomes	19
4. Study #1 participant progress with P2 and P6	20
5. Study #2 barrier-by-barrier and principle-by-principle progress	24
6. Study #2 barrier progress with/without in-person help	25
7. Study #2 team responses to Idea Garden icon	26
8. Percentage of teams using programming concepts in level design	31
9. Percentage of barriers with/without in-person help Study #2 vs. past study	32
10. Study #3 camp schedule	33
11. Study #3 tasks	36
12. Percentage of barriers by type for each condition	40
13. New evidence for Idea Garden principles	44

LIST OF APPENDIX FIGURES

<u>Figure</u>	<u>Page</u>
Appendix A: Idea Garden Hints in Gidget	52
Conditions	52
Events (collapsed)	53
Events (expanded)	54
Functions (collapsed)	55
Functions (expanded)	56
Conditional Statements (collapsed)	57
Conditional Statements (expanded)	58
Iteration (collapsed)	59
Iteration (expanded)	60
Lists	61
Objects (collapsed)	61
Objects (expanded)	62
Appendix B: Idea Garden Hints in Cloud9	63
The Idea Garden Panel	63
Reinterpret Problem Prompt: Divide and Conquer	63
Search for Solutions: Working Backwards (collapsed)	64
Search for Solutions: Working Backwards (expanded)	64
Implementation of Solution: Conditional Statements (collapsed)	65
Implementation of Solution: Conditional Statements (expanded)	66
Implementation of Solution: Events (collapsed)	65
Implementation of Solution: Events (expanded)	68
Implementation of Solution: Functions (collapsed)	69
Implementation of Solution: Functions (expanded)	70
Implementation of Solution: Iteration with For (collapsed)	71
Implementation of Solution: Iteration with For (expanded)	71

LIST OF APPENDIX FIGURES (CONTINUED)

<u>Figure</u>	<u>Page</u>
Implementation of Solution: Iteration with For-In (collapsed)	72
Implementation of Solution: Iteration with For-In (expanded)	72
Implementation of Solution: Iteration with Map (collapsed)	73
Implementation of Solution: Iteration with Map (expanded)	74
Implementation of Solution: Iteration with While (collapsed)	75
Implementation of Solution: Iteration with While (expanded)	76
Implementation of Solution: Lists (collapsed)	77
Implementation of Solution: Lists (expanded)	77
Implementation of Solution: Objects (collapsed)	78
Implementation of Solution: Objects (expanded)	78
Implementation of Solution: Variables	79
Evaluation of Implementation: Can it work better with Functions?	79
Evaluation of Implementation: Can it work better with Iteration? (Collapsed)	80
Evaluation of Implementation: Can it work better with Iteration? (Expanded)	80

INTRODUCTION

End-user programmers (EUPs) are defined in the literature as people who do some form of programming with the goal of achieving something other than programming itself [Nardi 1993]. In this thesis, we consider one portion of the spectrum of EUPs—those who are definitely *not* interested in learning programming per se, but are willing to do just enough programming to get their tasks done.

We can describe EUPs like this as being “indifferent” to learning programming (abbreviated “indifferent EUPs”). Indifferent EUPs are described well by Minimalist Learning Theory’s [Carroll and Rosson 1987] notion of “active users”. That theory describes users who are just interested in performing some kind of task—such as getting a budget correct or scripting a tedious web-based task so that they do not have to do it manually—not in learning about the tool and its features. According to the theory, active users like our indifferent EUPs are willing to do a bit of learning only if they expect it to help them get their task done.

We would like to help indifferent EUPs in the following situation: they have started a task that involves programming, and then have gotten “stuck” partway through the process. As we detail in the next section, indifferent EUPs in these situations have been largely overlooked in the literature.

We have been working toward filling this gap through an approach called the Idea Garden [Cao et al. 2011, Cao et al. 2012, Cao et al. 2013, Cao et al. 2014]. Our previous work has described the Idea Garden and its roots in Minimalist Learning Theory. In essence, the Idea Garden exists to entice indifferent EUPs who are stuck, to learn just enough to *help themselves* become unstuck. Empirical evaluations of the Idea Garden to date have been encouraging.

This thesis presents three research contributions: (1) a generalization of the Idea Garden through a set of principles analyzed in Study #1 and #2, (2) a generalization of the Idea Garden in an architecture to help port the system to new environments, and (3) the use of those principles and the architecture in a new context, analyzed in Study #3.

Our first contribution is answering this: how can the Idea Garden be generalized

into its essential characteristics? To answer this question, we present seven principles upon which (we hypothesize) the Idea Garden’s effectiveness rests, and instantiate them in a new Idea Garden prototype that sits on top of the Gidget EUP environment [Lee et al. 2014]. We then empirically investigate in studies #1 and #2, principle by principle, the following research question: *How do these principles influence the ways indifferent EUPs can solve the programming problems that get them “stuck”?*

Second, we present an architecture of the Idea Garden which helps implementors port the system to new environments. Third, we investigate the Idea Garden in Study #3 as part of a larger intervention to help new programmers when programming. We pose this research question: *Does the Idea Garden help EUPs in new settings with different programming problems that get them “stuck”?*

My role in this research

Many of these contributions involved collaboration with other researchers. My collaborators are mentioned in the acknowledgements, and this research could not have been completed without them. To clarify my own contributions to this work, they include the following:

- Lead developer for the last few months implementing the Idea Garden in Gidget,
- Lead developer for implementing Idea Garden in Cloud9
- Refined the Idea Garden architecture from [Cao 2013] and implemented it.
- First author for [Jernigan et al. 2015]
- Almost all of the qualitative thematic coding in Study #2 and #3 (with a partner)
- Lead analyst of the Gidget-based empirical work, including poring over data, visualizing data, writing scripts to transform data, interpreting data to write results, and running statistical tests.
- Lead researcher in determining the Idea Garden principles
- Helped to conduct Study #2 and #3

BACKGROUND AND RELATED WORK

As we have explained, the most relevant foundational basis for the Idea Garden’s target population is Minimalist Learning Theory (MLT) [Carroll and Rosson 1987, Carroll 1990]. MLT was designed to provide guidance on how to teach users who (mostly) don’t want to be taught. More specifically, MLT’s users are motivated simply by getting the task at hand accomplished. Thus, they are often unwilling to invest “extra” time to take tutorials, read documentation, or use other training materials—even if such an investment would save them time in the long term. This phenomenon is termed the “paradox of the active user” [Carroll and Rosson 1987]. MLT aims to help those who face this paradox to learn, *despite* their indifference to learning.

The Idea Garden also draws from foundations on curiosity and constructivist learning. To deliver content to indifferent EUPs, the Idea Garden uses Surprise-Explain-Reward (a strategy studied in [Robertson et al. 2004]) to surprise EUPs as a curiosity-based enticement. To encourage learning while acting, the Idea Garden draws from constructivist theories surveyed in [Bransford et al. 1999] to keep users active, make explanations not overly directive, and motivate users to draw upon their prior knowledge. Moreover, the Idea Garden encourages users to construct meaning from its explanations by arranging, modifying, rearranging, and repurposing concrete materials in the way bricoleurs do [Turkle and Papert 1990].

Our work is also related to research that aims to help naive users learn programming, often through the use of new kinds of educational approaches, or special-purpose programming languages and tools [Dorn 2011, Guzdial 2008, Hundhausen et al. 2009, Kelleher and Pausch 2006, Tillmann et al. 2013]. Stencils [Kelleher and Pausch 2005] presents translucent guides with tutorials to teach programming skills. The stencils overlaid upon the Alice interface show users the only possible interactions and explain them with informative sticky notes, but the Idea Garden aims to help users figure out the interactions themselves. Also, these approaches target users who aspire to learn some degree of programming, whereas the Idea Garden targets those whose motivations are to do only enough programming to complete some *other* task.

In EUP systems targeting novices who do not aspire to become professional pro-

grammers, a common thread has been to *simplify* programming via language design. For example, the Natural Programming project promotes designing programming languages to match users' natural vocabulary and expressions of computation [Myers et al. 2004]. One language in that project, the HANDS system for children, depicts computation as a friendly dog who manipulates a set of cards based on graphical rules, which are expressed in a language designed to match how children described games [Pane and Myers 2006]. Other programming environments such as Alice [Kelleher and Pausch 2006] incorporate visual languages and direct or tangible manipulation to make programming easier for EUPs. The Idea Garden approach is not about language design, but rather about providing conceptual and problem-solving assistance in whatever language/environment is hosting it.

A related approach is to reduce or eliminate the need for explicit programming. For example, programming by demonstration allows EUPs to demonstrate an activity from which the system automatically generates a program (e.g., [Cypher et al. 2010]). Some such environments (e.g., CoScripter/Koala [Little et al. 2007]) also provide a way for users to access the generated code. Another family of approaches seeks to *delegate* some programming responsibilities to other people. For example, meta-design aims at design and implementation of systems by professional programmers such that the systems are amenable to redesign through configuration and customization by EUPs [Andersen and Mørch 2009, Costabile et al. 2009].

Another way to reduce the amount of programming needed is by connecting the user with *examples* they can reuse as is. For example, FireCrystal [Oney and Myers 2009] is a Firefox plug-in that allows a programmer to select user interface elements of a webpage and view the corresponding source code. FireCrystal then eases creation of another web page by providing features to extract and reuse this code, especially code for user interface interactions. Another system, BluePrint [Brandt et al. 2010], is an Adobe Flex Builder plug-in that semi-automatically gleans task-specific example programs and related information from the web, then provides these for use by EUPs. Other systems are designed to simplify the task of choosing which existing programs to run or reuse (e.g., [Gross et al. 2010]) by emulating heuristics that users themselves

seem to use when looking for reusable code.

Although the above approaches help EUPs by simplifying, eliminating, or delegating the challenges of programming, none are aimed at nurturing EUPs' problem-solving ideas. In essence, these approaches help EUPs by lowering barriers, whereas the Idea Garden aims to help EUPs *figure out for themselves* how to surmount those barriers.

However, there is a little work aimed at helping professional interface designers generate and develop ideas for their interface designs. For example, Bricolage [Kumar et al. 2011] allows designers to retarget design ideas by transferring designs and content between webpages, thus enabling multiple design ideas to be tested quickly. Another example is a visual language that helps web designers develop their design ideas by suggesting potentially appropriate design patterns along with possible benefits and limitations of the suggested patterns [Diaz et al. 2010]. That line of work partially inspired our research on helping EUPs generate new ideas in solving their programming problems.

THE IDEA GARDEN PRINCIPLES

Using MLT as a foundation, an earlier version of the Idea Garden was defined in [Cao et al. 2014] as:

(Host) A subsystem that extends a “host” end-user programming environment to provide hints that...

(Theory) follow the principles from MLT [Carroll 1990] and...

(Content/Presentation) non-authoritatively give intentionally imperfect guidance about problem-solving strategies, programming concepts, and design patterns, via negotiated interruptions.

(Implementation) In addition, the hints are presented via host-independent templates that are informed by host-dependent information about the user’s task and progress.

This chapter presents seven principles to ground the Content/Presentation aspect above:

P1-Content. Hints must contain one or more of the following:

P1.Concepts = explains a programming *concept* such as iteration or functions. Can include programming constructs as needed to illustrate the concept.

P1.Minipatterns = *design minipatterns* show a usage of the concept that the user must adapt to their problem (minipattern should not solve the user’s problem).

P1.Strategies = a problem-solving strategy such as working through the problem backward.

P2-Relevance. For Idea Garden hints that are context-sensitive, the aim is that the user perceives them to be relevant. Thus, such hints use one or more of these types of relevance:

P2.MyCode = the hint includes some of the user’s code.

P2.MyState = the hint depends on the user’s code, such as by explaining a concept present in the user’s code.

P2.MyGoal = the hint depends on the requirements the user is working on, such as referring to associated test cases or pre/postconditions.

P3-Actionable. Because the Idea Garden targets MLT’s “active users”, hints must give them something to *do*. Thus, Idea Garden hints must imply an action that the user

can take to overcome a barrier or get ideas on how to meet their goals:

P3.ExplicitlyActionable = the hint prescribes actions that can be physically done, such as indenting something.

P3.ImplicitlyActionable = the hint prescribes actions that are “in the head”, such as “compare” or “recall.”

P4-Personality. The personality and tone of Idea Garden entries must try to encourage constructive thinking. Toward this end, hints are expressed non-authoritatively [Lee and Ko 2011], i.e., as a tentative suggestion rather than as an answer or command. For example, phrases like “try something like this” are intended to show that, while knowledgeable, the Idea Garden is not sure how to solve the user’s exact problem.

P5-InformationProcessing. Because research has shown that (statistically) females tend to gather information comprehensively when problem-solving, whereas males gather information selectively [Meyers-Levy 1989], the hints must support both styles. For example, when a hint is not small, a condensed version must be offered with expandable parts.

P6-Availability. Hints must be available in these ways:

P6.ContextSensitive = available in the context where the system deems the hint relevant.

P6.ContextFree = available in context-free form through an always-available widget (e.g., pull-down menu).

P7-InterruptionStyle. Because research has shown the superiority of the negotiated style of interruptions in debugging situations [Robertson et al. 2004], all hints must follow this style. In negotiated style, nothing ever pops up. Instead, a small indicator “decorates” the environment (like the incoming mail count on an email icon) to let the user know where the Idea Garden has relevant information. Users can then request to see the new information by hovering or clicking on the indicator.

As Table 1 shows, *P4-Personality* and *P7-InterruptionStyle* have already been isolated for summative investigation in other end-user programming research [Lee and Ko 2011, Robertson et al. 2004]. Thus, in this paper, we present our investigation of P1, P2, P3, P5, and P6.

Table 1: Citations show empirical evidence of the principles. An updated version with the contributions of this thesis appears near the end.

+: Principle was helpful, -: Principle was problematic.


Principle	Formative Evidence	Summative Evidence
P1-Content		
P2-Relevance	-[Cao et al. 2012]	
P3-Actionable		
P4-Personality		+[Lee and Ko 2011]
P5-InformationProcessing	+[Meyers-Levy 1989]	
P6-Availability		
P7-InterruptionStyle		+[Robertson et al. 2004]

THE PRINCIPLES CONCRETELY: IDEA GARDEN PROTOTYPE IN GIDGET

The Idea Garden Prototype for Gidget

The Idea Garden supplements a host EUP environment, and for this version of the Idea Garden, the host is Gidget, an online puzzle game that centers on debugging (Figure 1). Gidget has been used successfully by middle- and high-school teens [Lee et al. 2014] and by adults between the ages of 18 and 66 years old [Lee and Ko 2011]. Gidget has two target audiences: novices who wish to learn programming, and indifferent EUPs who have no interest in learning programming, but want to play Gidget’s puzzle games. The latter target audience made it a suitable host for the new version of the Idea Garden we present here.



Figure 1: Dictionary entries appear in tooltips when players hover over keywords (“for” shown here). Hovering over an idea indicator () then adds an Idea Garden hint. (The superimposed callouts are for readability.)

In the Gidget game, a robot named Gidget provides players with code to complete missions. According to the game’s backstory, Gidget was damaged, and the player must help Gidget diagnose and debug the faulty code. Missions (game levels) introduce or reinforce different programming concepts. After players complete all 37 levels of the “puzzle play” portion of the Gidget game, they can then move on to the “level

design” portion to create (program) new levels of their own.

The prototype’s support for the 7 principles

The Idea Garden prototype aims to help Gidget players who are unable to make progress even *after* they have used the host’s existing forms of help. Before we added the Idea Garden to it, Gidget had three built-in kinds of help: a tutorial slideshow, automatic highlighting of syntax errors, and an in-line reference manual (called a “dictionary” in Gidget) available through a menu and through tooltips over keywords in the code. The Idea Garden supplements these kinds of help by instantiating the seven principles as follows (illustrated in Figure 2).

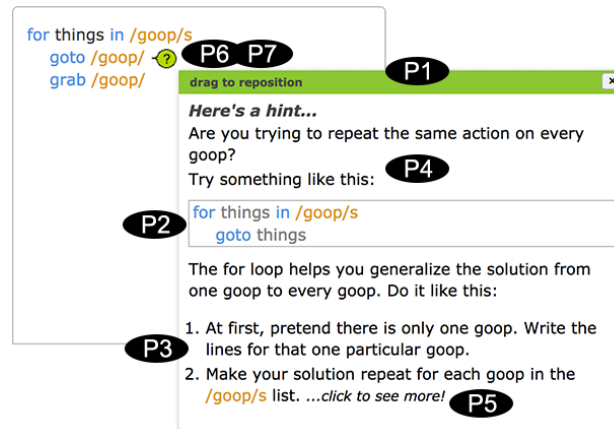



Figure 2: Hovering over a  shows a hint. The superimposed Ps show where the 7 principles are instantiated in this hint.

P1-Content: The *Concept* portion is in the middle of Figure 2, the *Minipattern* is shown via the code example, and the *Strategy* portion is the numbered set of steps at the bottom.

P2-Relevance: Prior empirical studies [Cao et al. 2012] showed that if Idea Garden users did not immediately see the relevance of a hint to their situation, they would ignore it. Thus, to help Gidget users quickly assess a hint’s relevance, the hint first says what goal the hint is targeting, and then includes some of the user’s own code and/or variable names (Figure 2), fulfilling P2.MyCode and P2.MyState. The antipatterns, explained in the next subsection, are what make these inclusions viable.

P3-Actionable, P4-Personality, and P5-InformationProcessing: Every hint suggests action(s) for the user to take. For example, in Figure 2, the hint gives numbered

actions (P3). However, whether the hint is the *right* suggestion for the user’s particular situation is still phrased tentatively (P4). Since hints can be relatively long, they are initially collapsed but can be expanded to see everything at once, supporting players with comprehensive and selective information processing styles (P5).

P6-Availability, P7-InterruptionStyle: Hints never interrupt the user directly; instead, a hint’s availability in context (P6.ContextSensitive) is indicated by a small green ⓘ beside the user’s code (Figure 2, P7) or within one of Gidget’s tooltips (Figure 1). The user can hover to see the hint, and can also “pin” a hint so that it stays on the screen. Context-free versions of all the hints are always available (P6.ContextFree) via the “Dictionary” button at the top right of Figure 1.

Antipattern support for the principles

Idea Garden’s support for several of the principles comes from its detection of *mini-antipatterns* in the user’s code. Antipatterns, a notion similar to “code smells”, are implementation patterns that suggest some kind of conceptual, problem-solving, or strategy difficulty. The prototype detects these antipatterns as soon as a player introduces one.

Our prototype detects several antipatterns that imply *conceptual* programming problems. In selecting which ones to support in this prototype, we selected antipatterns that occurred in prior empirical data about Gidget at least three times (i.e., by at least three users). The following is a description of each programming antipattern and the conceptual issue behind them:


(1) *no-iterator*: not using an iterator variable within the body of a loop. Users usually thought that loops would interact with every object in for loop’s list when using a reference to a single object instead of the iterator variable.

(2) *all-at-once*: trying to perform the same action on every element of the set/list all at once instead of iterating over the list. Users thought that functions built to work with objects as parameters would take lists as arguments.

(3) *function definition without call*: Users sometimes believed that the definition of a function would run once execution reached the function keyword; they did not realize they had to call the function.

(4) *function call without definition*: calling an undefined function. Sometimes, users did not realize that some function calls referred to definitions that they could not see (since they were defined in Gidget’s world code). They would try to call other functions that had no definition whatsoever.

(5) *instantiating an undefined object*: instantiating an undefined object. Similar to (4), objects could be defined in the world code and created in Gidget’s code. Some users thought they could create other objects they had seen in past levels despite the fact they were not defined in the current level.

Detecting antipatterns enables support for two of the Idea Garden principles. The antipatterns define context (P6.Context Sensitive), letting the hint to be derived from and shown in the context of the problem. For P2-Relevance, the hint communicates relevance (to the user’s current problem) by being derived from the player’s current code as soon as they enter it, such as using the same variable names (Figure 2, P2 and P6). The prototype brings these two principles together by constructing a context-sensitive hint whenever it detects a conceptual antipattern. It then displays the  beside the relevant code to show the hint’s availability.

GENERALIZED IDEA GARDEN ARCHITECTURE

Past research [Cao et al. 2011, Cao et al. 2012, Cao et al. 2013, Cao et al. 2014] has shown the Idea Garden’s success in CoScripter. Later chapters in this thesis show the Idea Garden’s success in Gidget. However, both implementations were specific to their hosts and not generalized. The architecture presented in this section extends and implements an earlier generalized architecture proposed in [Cao 2013] but never implemented beyond a small fraction until now. Figure 3 shows the components of the Idea Garden and which components interact with the host programming environment. The host calls the `startListening()` method of the Host-Specific Listener, which is described with its base class next.

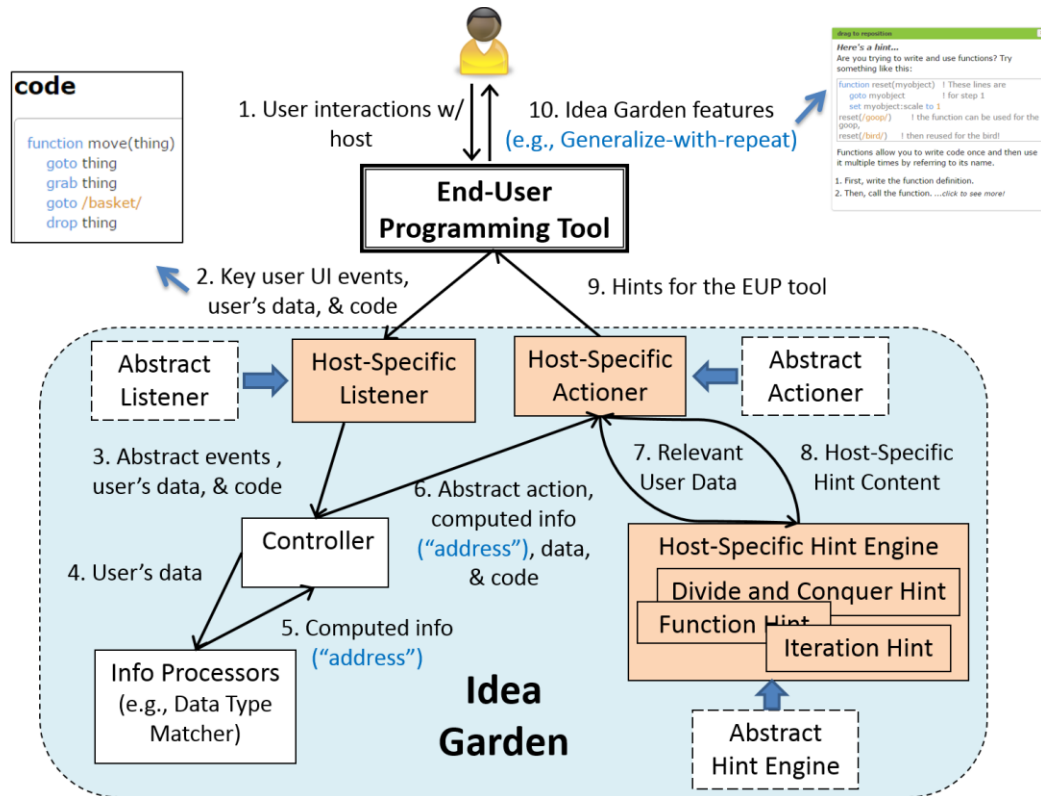





Figure 3: Architecture of Idea Garden. The black arrows represent the flow of data. User data (e.g., user’s code) flows from the end-user programming tool to the Host-Specific Listener. That data is passed along to the controller, information processors, actionner, and then used to build the hints that are sent back to the host environment. The thick blue arrows represent inheritance, so the Host-Specific Listener inherits from the Abstract Listener. The thin blue arrows point to examples, so the code mentioned near black arrow 2 is shown in the code image.


Listener: The Abstract Listener provides the abstract events that may be listened for (see Table 2) and also provides required interfaces to the Host-Specific Listener (through inheritance), which are included in this paragraph. In order to receive the relevant user events, code and data from the host, the `startListening()` method should call `igListener.processUserCode()` (and other similar functions like `processUserEditor()` and `processUserRequirements()` where applicable). The Host-Specific Listener then retrieves the code (or other data elements) from the host and starts listening for abstract events. The Host-Specific Listener finds abstract events by searching the user code for the antipatterns described in the Principles chapter or by listening for  clicks. Then, the Host-Specific Listener passes the abstract event along with relevant user code and data to the Controller (by calling `controller.onReceiveAbstractEvent()`).

Controllers and Information Processors: The Controller and Information Processors map abstract events to actions and process data like bug reports. These classes do not require host-specific versions. First, upon receiving an abstract event from the Host-Specific Listener, the Controller pairs the abstract event with an abstract action. Table 2 lists example pairs of abstract events and abstract actions. Second, the Controller then sends the matched abstract action to the Host-Specific Actioner, passing along the code that it received from the Host-Specific Listener by calling `actioner.onReceiveAbstractAction()`. If the Host-Specific Listener also sent additional data to the Controller, the Controller first sends the data to the relevant Information Processors, which calculates additional information based on the data and sends the calculated information back to the Controller. For example, the Information Processor used in CoScripter’s Idea Garden [Cao et al. 2012, Cao et al. 2013, Cao et al. 2014] is the *Data Type Matcher1* which determines a *data type* based on input data. The Controller then passes the calculated information on to the Host-Specific Actioner.

Abstract Events	Abstract Actions
user_needs_help_getting_started	show_getting_started_hint
for_loop_without_iterator_antipattern	show_iteration_icon
user_previewed_webpage	highlight_evaluation_hints

Table 2: Pairs of abstract events and abstract actions, matched by the Controller.

Actioner: Given an abstract event that requires an action, the Host-Specific Actioner produces the appropriate action. The Host-Specific Actioner inherits the Abstract Actioner, implementing the `actioner.onReceiveAbstractAction()` function mentioned above and the abstract actions it may execute, including `highlight_evaluation_hints`. First, the Host-Specific Actioner fills the hint with the relevant pieces of the user’s context, such as function names. Second, the Host-Specific Actioner decorates the host environment’s UI with a  linked to the relevant hint. For example, in Cloud9, if the  is clicked, the hint panel is opened and the linked hint is highlighted briefly.

Hint Engine: The Host-Specific Hint Engine provides context-free and context-sensitive access to hints through the hint panel. For example, in Cloud9, hint titles are shown initially and the hint contents can be shown by clicking the  next to the title. The panel starts with default versions of all the host-specific hints made by the implementer. The hints are updated by the Host-Specific Actioner to include context-specific information.

Example


Now, let’s consider an example situation where the Idea Garden responds to the user typing in some code that contains one of the antipatterns. The concrete items in the example use the Cloud9 implementation.

The following sequence occurs:

1. Suppose the user types in the following code:

```
for (var x in arr){f1(arr[0]);}
```
2. The host programming environment sends this user code to the listener.
3. The Listener parses that code and finds a *for* loop that does not use its iterator variable. This is an instance of the `for_loop_without_iterator` antipattern.

4. The Listener labels an object as an abstract event with type `for_loop_no_iterator`, then collects the name of the unused variable (in this case, `x`) and the name of the list from the *for* loop (`arr`), and sends it all together to the Controller.
5. The Controller determines the abstract action the Idea Garden should execute by mapping the input abstract event (`for_loop_no_iterator`) to its corresponding abstract action (`show_iteration_icon`).
6. The Controller sends this abstract action (`show_iteration_icon`) and the user's code to the Actioner.
7. The Actioner sends that user code on to the Hint Engine.
8. The Hint Engine finds the relevant hint, (the iteration hint), and inserts that user code into the code example in the iteration hint like this:

```
for (var x in arr){console.log(arr[x]);}
```
9. The Actioner puts an Idea Garden icon  in the user's IDE that links to the hint that the Hint Engine just updated.

Porting Idea Garden to new environments:

After acquiring a previous version of the Idea Garden (all versions so far have been in JavaScript), an implementer would follow these steps to adapt it for a new environment:

- Determine how users conceptually struggle in the new environment, perhaps with a study. Develop antipatterns that detect at least some of these struggles.
- For the antipatterns, create or modify host-specific antipattern code to detect when users exhibit these antipatterns in your environment in the Host-Specific Listener.
 - New code is needed for struggles you found typical (like antipatterns) that weren't present before.
 - You also may need new hints and design patterns within them. Some of the hints you'll set up to be triggered by antipatterns (as below), and some you'll simply make available other ways, such as

by linking them to other hints.

- All hints need to be made available in a context-free way, such as through your host environment's existing "help" button or reference manual link. (In Cloud9, a side panel provided access to all the hints.)
- Modify previous host-specific code to correspond to your new environment for struggles (antipatterns) that were implemented before, like `for_loop_no_iterator`,
- For those antipatterns, send them to the Controller as a new abstract event.
- In the Controller, map the new abstract events to new abstract actions. (Existing antipatterns are already mapped.) Send the abstract action to the Actioner.
- Make the Actioner tell the Hint Engine to update hints with user data/code.
- Make the Hint Engine display the hints on the screen. Use the same format for all the hints where possible.
- Make the Actioner display an icon on the screen when Idea Garden should respond to user actions, based on the abstract action sent to the Actioner.
- Delete code in the Host-Specific components that deal with user struggles that can't occur your environment (or set them aside for guidance).

About 85% of our code in Cloud9 was host-specific, but some of it is modifiable for new environments. About 15% was entirely host-independent, and the structure/framework is entirely reusable.

STUDY #1: PRINCIPLED FORMATIVE STUDY

Prior to implementing the Idea Garden principles in the prototype, we conducted Study #1, a small formative study. Our goal was to gather evidence about our proposed principles, helping us choose which ones to implement in the prototype that we would evaluate in Study #2.

We reanalyzed think-aloud data that we presented in [Lee et al. 2014]. This study had 10 participants (5 female, 5 male) 18-19 years old, with little to no programming experience. Each session was 2 hours, fully video recorded. The experimenter helped participants when they were stuck for more than 3 minutes. We re-analyzed the video recordings from this study using the code sets in Table 3. The objective of [Lee et al. 2014] was to investigate Gidget barriers and successes. Here we analyze the think-aloud data from a new perspective: to inform our research into how Idea Garden principles should target those issues. Thus, the Idea Garden was *not yet present* in Gidget for Study #1.

Table 3: Study #1 and #2 Barrier codes and Outcome codes.

Algorithm Design Barrier Codes [Cao et al. 2012, Lee et al. 2014]	
More than once	Did not know how to generalize one set of commands for one object onto multiple objects
Composition	Did not know how to combine the functionality of existing commands
Learning Phase Barrier Codes [Ko et al. 2004, Lee et al. 2014]	
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 things to use, but did not know how to use them together
Understanding	Thought they knew how to use something, but it did not do what they expected
Information	Thought they knew why it did not do what they expected, but did not know how to check
Barrier Outcomes Codes	
Progress	Participant overcame the barrier or partially overcame the barrier.
In-person help	Participant overcame the barrier, but with some help from the experimenter.
No Progress	Neither of the above.

Although the Idea Garden was not yet present, some UI elements in Gidget were consistent with some Idea Garden principles (Table 3's left column). We leveraged these connections to obtain formative evidence about the relative importance of the proposed principles. Toward this end, we analyzed 921 barriers and 6138 uses of user interfaces.

Table 4: Study #1: number of instances in which participants made progress for principles P2-Relevance and P6-Availability. (Max values highlighted.)

+ : progress with no in-person help.

+☺ : progress with additional help from experimenter.

- : no progress.

Principle (example UI elements)	Participants' progress			Which barriers
	+	+😊	-	
P2-Relevance				
P2.MyState (e.g., Error messages)	2128 44%	1378 28%	1368 28%	(Minor contribution to most)
P2.MyGoal (e.g., Mission/level goals)	767 42%	571 31%	487 27%	Design (& minor to most)
P6-Availability				
P6.Context-Sensitive Avail. (e.g., Tooltips over code)	1691 44%	1151 29%	1034 27%	Coord., Compos., Selection (& minor to most)
P6.Context-Free Avail. (e.g., Dictionary)	823 36%	845 37%	594 26%	(Minor to Design)

The Gidget UI elements' connection to Idea Garden principles primarily related to P2-Relevance and P6-Availability. Table 4 shows that, when these principles were present, participants tended to make progress—usually without needing any help from the experimenter.

However, as Table 4 also shows, each principle helped with different barriers (defined in Table 3). For example, P2.MyGoal stood out in helping participants with Design barriers, whereas P6.ContextSensitive was strong with Coordination, Composition, and Selection barriers.

These results revealed useful insights for Study #2's principled evaluation and the Idea Garden prototype: (1) The complementary roles that it revealed of different principles for different sections in “barrier space” caused us to design Study #2 to allow evaluation from a barrier perspective. (2) The promising results for P2-Relevance and P6.ContextSensitive motivated us to design several of the antipatterns described in Section IV, so as to trigger relevant hints in context. (3) The concepts (P1.Concepts) that participants struggled with the most were the ones we wrote the antipatterns and hints to target.

Informed by these insights, we implemented the principles in the form described in the Gidget Prototype chapter and conducted Study #2 to evaluate the results.

STUDY #2 (SUMMATIVE): THE PRINCIPLES GO TO CAMP

We conducted Study #2 as a (primarily) qualitative study, via two summer camps for teenagers playing the Gidget debugging game. The teens used the Idea Garden whenever they got stuck with the Gidget game. The study's goal was to evaluate the usefulness of the Idea Garden principles to these teens. Our overall research question was: *How do the principles influence the ways indifferent EUPs can solve the programming problems that get them "stuck"?*

The two summer camps took place on college campuses in Oregon and Washington. Each camp ran 3 hours/day for 5 days, for 15 hours total. Campers spent 5 hours each in Gidget puzzle play; other activities such as icebreakers, guest speakers, and breaks; and level design.


We recruited 34 teens aged 13-17. The Oregon camp had 7 males and 11 females; all 16 teens in the Washington camp were females. Both camps' median ages were 15 years. The participants were paired up into same-gender teams of similar age (with only one male/female pair) and were instructed to follow pair programming practices, with the "driver" and "navigator" switching places after every game level.

The Gidget game is intended for two audiences: those who want to learn programming *and* our population of indifferent EUPs. Since the Idea Garden targets the latter audience, we aimed to recruit camp participants with little interest in programming itself by inviting them to a "problem-solving" camp (without implying that the camp would teach programming).

The teens we attracted did seem to be largely made up of the "indifferent EUP" audience we sought. We interviewed the outreach director who spoke with most parents and kids of Study #2's Oregon camp, which targeted schools in economically-depressed rural towns, providing scholarships and transportation. She explained that a large percentage of campers came in spite of the computing aspect, not because of it: the primary draw for them was that they could come to the university, free of cost, transportation provided.

The same researchers ran both camps: a lead (male graduate student) led the activities and kept the camp on schedule; a researcher (female professor), and four helpers

(one male graduate student, three female undergraduates) answered questions and approached struggling participants. We provided no formal instruction about Gidget or programming. The Gidget system recorded logs of user actions, and the helpers observed and recorded instances when the campers had problems, noting if teams asked for help, what the problem was, what steps they tried prior to asking for help, and what (if any) assistance was given and if it resolved the issue.

We coded the 407 helper observations in three phases using the same code set as for Study #1: we first determined if a barrier occurred, then which types of barriers occurred, and finally what their outcomes were (Table 3). Two coders reached 85%, 90%, and 85% agreement (Jaccard Index), respectively, on 20% of the data during each phase, and then split up the rest of the coding. We then added in each additional log instance (not observed by a helper) in which a team viewed an antipattern-triggered Idea Garden hint marked by a . We considered these 39 instances evidence of “self-proclaimed” barriers. Two coders reached 80% and 93% on 20% of the data respectively, and one coder finished the remaining data. Finally, for purposes of analysis, we removed all Idea Garden instances in which the helper staff also gave assistance (except where explicitly stated otherwise), since we cannot know in such instances whether progress was due to the helpers or to the Idea Garden.

STUDY #2 RESULTS

Successes

Teams did not always need the Idea Garden; they solved 53 of their problems just by discussing them with each other, reading the reference manual, etc. However, when these measures did not suffice, they turned to the Idea Garden for more assistance 149 times (bottom right, Table 5). Doing so enabled them to problem-solve their way past 77 of these 149 barriers (52%) without any guidance from the helper staff (Table 6).

In fact, as Table 6 shows, when the Idea Garden hint or ? was on the screen, teams seldom needed in-person help: only 25 times (out of 149+25) = 14%. Finally, the teams' success rate with in-person help alone (59%) was only a little higher than with the Idea Garden alone (52%).

Table 5: Barrier-by-barrier progress when situation-based aspects of Idea Garden principles P2, P3, and/or P6 were on-screen. (P1, P5 not shown because all aspects were always present.) The total column (right) adds in the small numbers of Design, Composition, and Information barrier instances not detailed in other columns.

		Barriers					Total
		Selection	Use	Coordination	Understanding	More Than Once	
P2-Relevance	MyCode	8/20 40%	13/21 62%	1/1 100%	1/2 50%	12/24 50%	35/69 51%
	MyState	9/24 38%	28/54 52%	12/18 67%	2/4 50%	12/25 48%	64/128 50%
P3-Actionable	Explicitly Actionable	9/24 38%	28/54 52%	13/19 68%	2/4 50%	12/25 48%	66/130 51%
	Implicitly Actionable	10/23 43%	17/28 61%	1/1 100%	3/5 60%	14/29 48%	45/87 52%
	Context Sensitive	6/19 32%	22/37 59%	10/14 71%	1/3 33%	9/21 43%	48/95 51%
P6-Available	Context Free	2/5 40%	5/7 71%	2/2 100%	1/1 100%	2/5 40%	12/21 57%
Total (unique instances)		11/27 41%	33/62 53%	13/19 68%	4/7 57%	14/30 47%	77/149 52%

Table 6: Barrier instances and teams’ progress with/without getting in-person help. Teams did not usually need in-person help when an Idea Garden hint and/or antipattern-triggered 🟡 was on the screen (top row).

IG On-screen?	Progress without in-person help	Progress if team got in-person help
Yes (149+25 instances)	77/149 (52%)	25
No (155 instances)	53	91/155 (59%)

Table 5 also breaks out the teams’ success rates principle by principle (rows). No particular difference in success rates with one principle or aspect versus another stands out in isolation. However, viewing the table column-wise yields two particularly interesting barriers.

First, Selection barriers (first column) were the most resistant to the principles. This brings out a gap: the Selection barrier happens *before* use as the user tries to decide what to use, whereas the Idea Garden usually became active *after* a player attempted to use some construct in code. How the Idea Garden might close this gap is an opportunity we have barely begun to explore.

Second, Coordination barriers (third column) showed the highest progress rate consistently for all of the Idea Garden principles. We hypothesize that this relatively high success rate may be attributable to P1’s minipatterns (present in every hint), which show explicitly how to incorporate and coordinate combinations of program elements.

Teams’ Behaviors with P2-Relevance and P6-Availability

In this section, we narrow our focus to observations of how the teams reacted to the 🟡 from the lens of P2 and P6. We consider P2 and P6 together because the prototype supported P2-Relevance in a context-sensitive (P6) way.

Context-sensitivity seemed very enticing to the teams. As Table 5 shows, teams accessed P6.ContextSensitive hints about five times as often as the P6.ContextFree hints. Still, in some situations, teams accessed the context-free hints to revisit them out

of context. Despite more context-sensitive accesses, the progress rates for both were similar. Thus, this result supports providing for both of these situations, with both context-sensitive *and* context-free availability of the hints.

Table 7: Observed outcomes of responses to the 🐷. Teams made progress when they read a hint and acted on it (row 1, col 1), but never if they ignored what they read (row 2 col 1). (P2-Relevance’s mechanisms are active only within a hint.)

Response Type		Principles	Progress%	
Read hint and then...	...acted on it	P2+P6	25/42	60%
	...ignored it	P2+P6	0/4	0%
Didn’t read hint		P6	6/15	40%
Deleted code marked by 🐷		P6	4/19	21%
To-do listing		P6	3/4	75%

Table 7 enumerates the five ways teams responded to the context-sensitive 🐷s (i.e., those triggered by the mini-antipatterns). The first way was the “ideal” way that we had envisioned: reading and then acting on what they read. Teams responded in this way in about half of our observations, making progress 60% of the time. For example:

Team Turtle (Observation #8-A-2):

Observation notes: Navigator pointed at screen, prompting the driver to open the Idea Garden 🐷 on function. ... they still didn't call the function.

Action notes: ... After reading, she said "Oh!" and said "I think I get it now..." Changed function declaration from "/piglet/getpiglet" to "function getpiglet()". The 🐷 popped up again since they weren't calling it, so they added a call after rereading the IG and completed the level.

However, a second response to the 🐷 was when teams read the hint but did not act on it. For example:



Team Beaver (Observation #24-T-8):




Observation notes: ... "Gidget doesn't know what a sapling is", "Gidget's stupid". Looked at Idea Garden hint. ... "It didn't really give us anything useful" ...

This example helps illustrate a nuance of P2-Relevance. Previous research has reported challenges in convincing users of relevance [Cao et al. 2012]. In this example the team may have believed the hint was relevant to the *problem*, but not to a *solution* direction. This suggests that designing according to P2-Relevance should target solution relevance, not just problem relevance.

Third, some teams responded to the 🐷 by not reading the hint at all. This helped a

little in that it identified a problematic area for them, and they made progress fairly often (Table 7), but not as often as when they read the hint.

Fourth, some teams deleted code marked by the . They may have viewed the  as an error indicator and did not see the need to read why (perhaps they thought they already knew why). Teams rarely made progress this way (21%).

Fifth, teams used s as “to-do” list items. For example, Team Mouse, when asked about the  in the code in Figure 4, said “we’re getting there”. Using the  as something to come back to later is an example of the “to-do listing” strategy, which has been a very successful problem-solving device for EUPs if the strategy is explicitly supported [Grigoreanu et al. 2010].

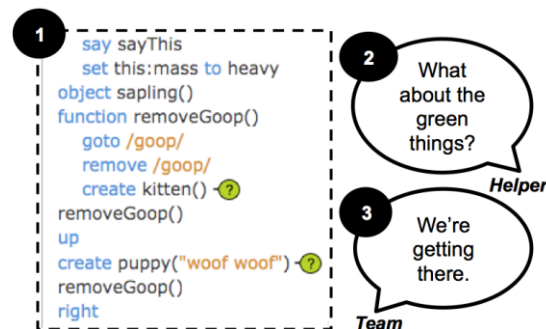





Figure 4: (1) Team Mouse spent time working on code above the s. When (2) a helper asked them about the s in their code, they indicated (3) that the s were action items to do later. Seven other teams also used this method.


Teams’ Behaviors with P3-Actionable

The two types of actionability that P3 includes, namely P3.ExplicitlyActionable (step-by-step actions as per Figure 2’s P3) and P3.ImplicitlyActionable (mental, e.g. “refer back...”) instructions, helped the teams in very different ways.

Explicitly actionable hints seemed to give teams new (prescriptive) *action recipes*. For example, Team Rabbit was trying to write and use a function. The hint’s explicitly actionable instructions revealed to them the steps they had omitted, which was the insight they needed to make their code work:

Team Rabbit (Observation #9-T-3)

Observation notes: They wrote a function... but do not call it.

Action notes: Pointed them to the  next to the function definition. They looked at the steps... then said, “Oh, but we didn’t call it!”

Explicitly actionable instructions helped them again later, in writing their very first

event handler (using the “when” statement). They succeeded simply by following the explicitly actionable instructions from the Idea Garden:

Team Rabbit (Observation #10-T-1)

Observation notes: They wanted to make the key object visible when[ever] Gidget asked the dragon for help. They used the Idea Garden hint for when to write a when statement inside the key object definition:

when /gidget/:sayThis = "Dragon, help!" ...

The when statement was correct.

In contrast to explicitly actionable instructions, implicitly actionable instructions seem to have given teams *new options to think over*. In the following example, Team Owl ran out of ideas to try and did not know how to proceed. But after viewing an Idea Garden hint, they started to experiment with new and different ideas with lists until they succeeded:

Team Owl (Observation #11-A-7):

Observation notes: They couldn't get Gidget to go to the [right] whale. They had written “right down grab first /whale/s.”

Action notes: Had them look at the Idea Garden hint about lists to see how to access individual elements ... Through [experimenting], they found that their desired whale was the last whale.

The key difference appears to be that the explicitly actionable successes came from giving teams a single new recipe to try themselves (Team Rabbit's second example) or to use as a checklist (Team Rabbit's first example). This behavior relates to the Bloom's taxonomy ability to *apply* learned material in new, concrete situations [Anderson et al. 2001]. In contrast, the implicitly actionable successes came from giving them ways to generate new recipe(s) of their own from component parts of learned material (Team Owl's example), as in Bloom's “analyze” stage [Anderson et al. 2001].

Teams' Behaviors with P5-Information Processing

Recall that P5-InformationProcessing states that hints should support EUPs information processing, whether comprehensive (process everything first) or selective (process only a little information before acting, find more later if needed). The prototype did so by condensing long hints into brief steps for selective EUPs, which could optionally be expanded for more detail for comprehensive EUPs. We also structured each hint the same way so that selective EUPs could immediately spot the type of information they wanted first.

Some teams including Team Monkey and Team Rabbit, followed a comprehensive

information processing style:

Team Monkey (Observation #27-S-6)

Observation notes: <Participant name> used the [IG hint] a LOT for step-by-step and read it to understand.


Team Rabbit (Observation #8-W-4)

Observation notes: They were reading the IG for functions, with the tooltip expanded. After closing it, they said "Oh you can reuse functions. That's pretty cool."

Many of the teams who preferred this style were female. Their use of the comprehensive style is consistent with prior findings that females often use this style [Grigoreanu et al. 2012, Meyers-Levy 1989]. As the same past research suggests, the four teams with males (but also at least one of the female teams) used the selective style.

Unfortunately, teams who followed the selective style seemed hindered by it. One male team, Team Frog, exemplifies a pattern we saw several times with this style: they were a bit *too* selective, and consistently selected very small portions of information from the hints, even with a helper trying to get them to consider additional pertinent information:

Team Frog (Observation #24-W-12 and #24-W-14):

Observation Notes: ... Pointed out  and even pointed to code, but they quickly selected one line of code in the IG help and tried it. ... They chose not to read information until I pointed to each line to read and read it...

In essence, the prototype's support for both information processing styles fit the ways a variety of teams worked.

HOW MUCH DID THEY LEARN?

After about 5 hours of debugging their way through the Gidget levels, teams reached the “level design” phase, in which teams were able to freely create whatever levels they wanted.

In contrast to the puzzle play activity, in which teams only fixed broken code to fulfill game goals, this “level design” part of the camp required teams to author level goals, “world code,” behavior of objects, and code that others would debug to pass the level. Figure 5 shows part of one such level.

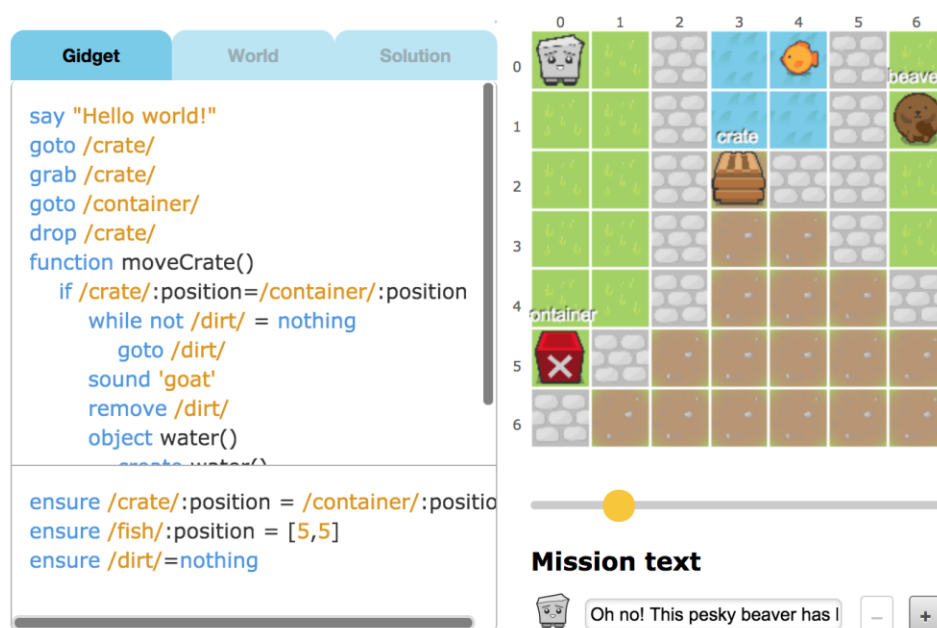


Figure 5: Team Tiger’s “River Dam” level’s functions, conditionals, and loops.

The teams created between 1 to 12 levels each (median: 6.5). As Figure 5 helps illustrate, the more complex the level a team devised, the more programming concepts the team needed to use to implement it. Among the concepts teams used were variables, conditionals (“if” statements), loops (“for” or “while”), functions, and events (“when” statements).

The teams’ use of events was particularly telling. Although teams had seen Idea Garden hints for loops and functions throughout the puzzle play portion of the game, they had never even seen event handlers. Even so, all 9 teams who asked helpers how to make event-driven objects were immediately referred to the Idea Garden hint that

explains it, and all eventually got it working with little or no help from the helpers.

The number of programming concepts a team chose to incorporate into their own levels can be used as a conservative measure of how many such concepts they really learned by the end of the camp. This measure is especially useful here, because the same data are available from the Gidget camps the year before, in which in-person help was the main form of assistance available to the campers [Lee et al. 2014] (Table 8).

Table 8: Percentage of teams using each programming concept during level design, for Study #2 versus Gidget camps held the year before. Note that the average is nearly the same.

Study	Bool	Var.	Cond.	Loops	Func.	Event	Avg.
Study #2 camps	100%	88%	25%	63%	44%	56%	63%
[Lee et al. 2014] camps	100%	94%	35%	47%	41%	76%	66%

As Table 8 shows, the teams from the two years learned about the same number of concepts on average. Thus, the amount of in-person help from the prior year [Lee et al. 2014] that we replaced by the Idea Garden’s automated help resulted in almost the same amount of learning.

As to how much in-person help was actually available, we do not have identical measures, but we can make a conservative comparison (biased against Idea Garden). We give full credit to Idea Garden this year only if no in-person help was involved, but give full credit to the Idea Garden last year if one of our early Idea Garden sketches was used to supplement in-person helpers that year. This bias makes the Idea Garden improvement look lower than it should, but is the closest basis of comparison possible given slight differences in data collection.

This comparison is shown in Table 9. As the two tables together show, Study #2’s teams learned about the same number of concepts as with the previous year’s camps (Table 8), with significantly less need for in-person help (Table 9, Fisher’s exact test, $p=.0001$).

Table 9: Instances of barriers and percentage of total barriers teams worked through with and without in-person help, this year under the principles described here, vs. last year. (Comparison biased against Idea Garden; see text.)

Study	Used in-person help	No in-person help
Study #2 camps with Idea Garden:	116	130
Barriers with progress	47%	53%
Prior year's camps [Lee et al. 2014]:	437	56
Barriers (progress not available)	89%	11%

STUDY #3 METHOD

The aim of the generalized architecture we presented in an earlier chapter was to generalize the Idea Garden to multiple contexts, i.e., to show that the Idea Garden generalizes from an implementation perspective. Study #3 then considers generalization from the language/environment, population, and tasks perspectives, aiming to show that the Idea Garden generalizes to another IDE and another programming language, helping users of that environment complete tasks different from previous studies.

The Idea Garden in Study #3 was one part of a four member intervention: (1) a problem solving lecture described in The Instruction section, (2) the handout in Figure 6, (3) the help request prompts described in The Project section, and (4) the Idea Garden shown in Figure 7, each of which are described throughout this section. Study #3 compared a traditional version of a web development camp (the control group) with an experimental version of the camp that was identical except for those four intervention elements. In this section, we describe our participants, the two camps, and the data we collected to measure the effects of our intervention.

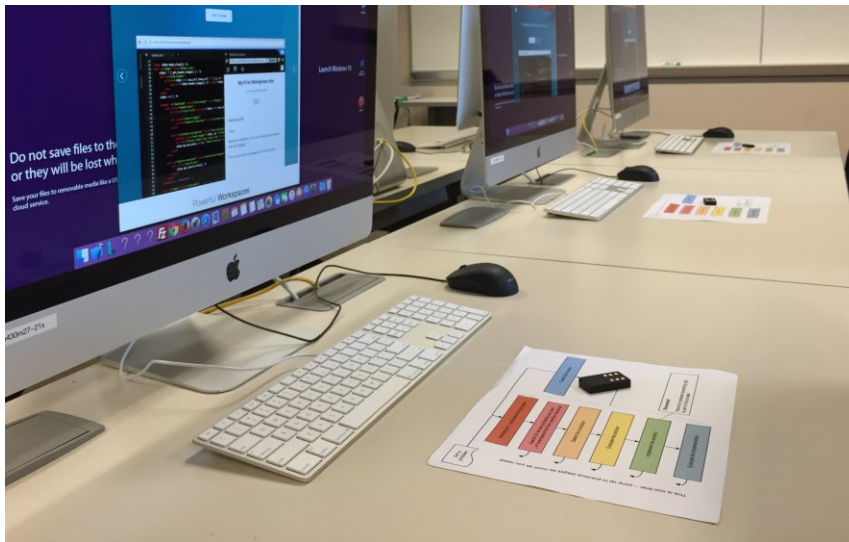


Figure 6: The paper handout and physical token we gave to campers to track their problem solving stage.

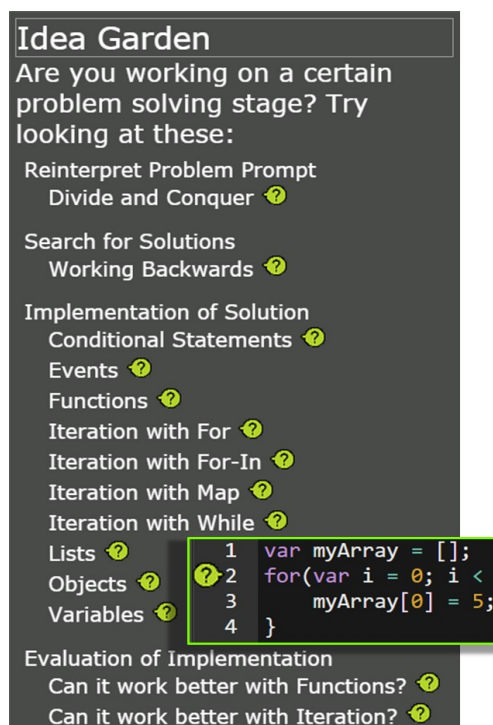


Figure 7: (Main) The Idea Garden panel in the Cloud9 IDE as campers see it when they opened the panel for the first time. (Callout) An example of the Idea Garden decorating the code with an icon. Here, the icon links to the *Iteration with For* hint.

Participants

Our participants were campers in a university-sponsored summer youth learning program. The program was based in a region with a large software industry, so many of the campers likely knew someone with coding skills. Campers in the youth program have historically been from upper-middle class families with college-educated parents, and have typically been only 20-30% female. Campers and parents were not aware of any difference between the two camps other than their scheduled time. The youth program managed registrations, recruiting 25 campers in the experimental group and 23 in the control. From this point forward, we refer to campers with a letter indicating their group followed a unique number (e.g. E27 is an experimental camper and C75 a control).

The experimental group included 8 females and 17 males. Two campers listed English as their non-primary language. The control group included 8 females and 15 males, and all listed English as their primary language. The two groups were largely

indistinguishable: they did not miss class at different rates (Kruskal-Wallis, $H=2.2$, $p=0.138$), they contained similar numbers of females ($X^2=0$, $df=1$, $p=1.000$), they had similar grade levels ($X^2=4.1829$, $df=3$, $p=0.242$), and similar self-reported programming and web development experience ($X^2 = 2.669$, $df=1$, $p=0.102$).

The Camps

Each camp consisted of ten 3-hour weekday sessions from 9am to 12pm (experimental) and from 1pm to 4pm (control). We placed the experimental group in the morning to bias any instructional improvements toward the control group (though this may have introduced other confounds, as we discuss later). Both camps took place in the same university computer lab. Campers worked in the Chrome web browser and *Cloud9*, a web-based IDE (*c9.io*).

The Instruction

We aimed to teach concepts, syntax, and semantics of HTML, CSS, and JavaScript with a focus on the *React* JavaScript framework (*facebook.github.io/react*). Our goal was for campers to feel capable of learning more about these technologies, but not necessarily capable of developing interactive web sites with them independently. We chose the *React* framework because it is based on a powerful but highly constrained *view* abstraction, which meant that there are only a small number of ways to implement any particular functionality. This made measuring task completion more straightforward, as we describe later in our results.

Day 1	HTML lecture and activity
Day 2	1-hour problem solving lecture (<i>experimental only</i>); Problem solving stages handout and prompts (<i>experimental only</i>); CSS lecture and activity
Day 3	JavaScript lecture and activity; Growth mindset development exercise
Day 4	React lecture and Interactive activity; Problem solving reminder (<i>experimental only</i>)
Days 5-9	Free development time
Day 10	Project presentations

Table 10: The camp schedule, with experimental camp’s additions as noted.

As Table 10 shows, the baseline camp included 4 days of lectures and practice, followed by 5 days of self-directed programming time on a course project. The lead in-

structor (a PhD student), presented HTML, JavaScript, and React lectures to both groups. Another instructor (a master’s student) presented a CSS lecture and a growth mindset exercise to both groups. Three additional undergrads also acted as helpers. All members of the instructional team had at least novice experience with web development and React. The lead instructor had no experience running camps or teaching programming.

The 1-hour problem solving lecture (the first part of our intervention, given only to the experimental group) taught campers six programming problem solving stages [Loksa et al. 2016]¹. The instructor began the lecture with a book sorting exercise. He asked the campers how to sort the books by size and followed their verbal instructions. Next, he asked the campers *how* they knew how to sort the books in that way and *why* they sorted the books that way. The campers discussed the *how* and *why* amongst themselves until they reported that they understood the problem. The instructor then prompted for more explanation until it became apparent to campers that the questions were not as simple as they initially seemed. The instructor used this realization to trigger a discussion of each of the six problem solving stages, starting with *reinterpreting the problem prompt*. Campers tried to identify the next stage of the process in groups at the instructor’s request. Once the campers identified the next stage (or the instructor identified it when campers ran out of ideas), he tied abstract concept of the stage to a concrete problem, such as the book sorting problem the lecture began with.

After the lecture, we provided the experimental group with a physical handout of the problem solving stages (shown in Figure 6) and a physical token so they could track their current state on the handout (the second part of our intervention). We instructed campers to track their progress through the stages as they worked on their website and to reflect on and adjust their strategies.

While the problem solving lecture detailed what programmers must *achieve* in the six stages, it did not prescribe *how* they achieve it. We did not mention any particular

¹ The six problem solving stages are out of the scope of this thesis. For reference, they are: (1) reinterpreting problem prompt, (2) search for analogous problems, (3) search for solutions, (4) evaluate a potential solution, (5) Implement a solution, and (6) evaluate implemented solution.

strategies or resources to use for each stage. The one exception to this is a mention of the development of sub-problems, which the instructor mentioned in the lecture and noted in the handout. The instructor also told the campers they could use the Idea Garden, which mentions some strategies such as *working backwards*.

The Project

After the four days of lecture and practice, campers in both groups spent the remaining five work days on a class project. The project was to build an interactive, React-based single-page web application that contained both static and interactive content about campers' interests. Figure 8 shows an example of a camper's final site. To scaffold the project, we provided a basic architecture for the application. We then provided a set of 20 progressively more difficult tasks for campers to complete at their own pace (see Table 11).

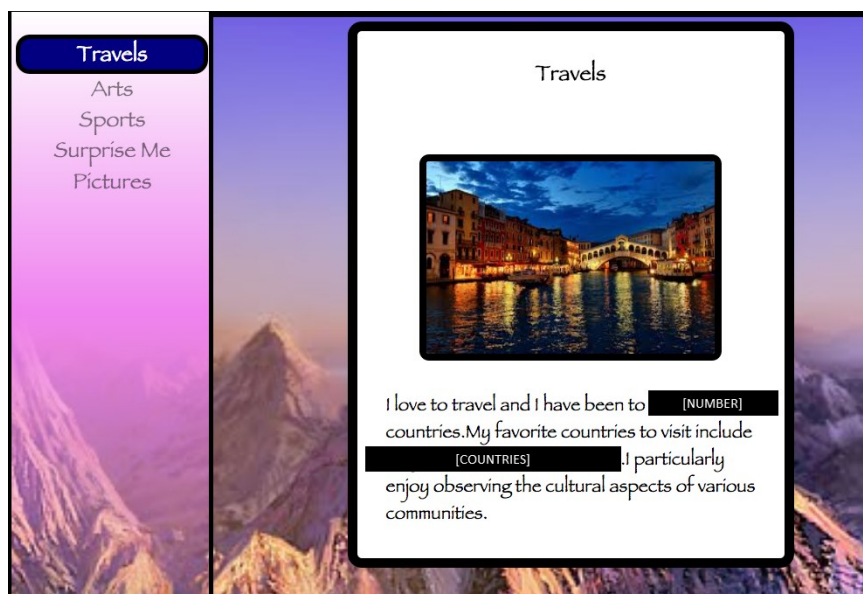


Figure 8: Camper E27's final project, showing buttons that link to different interests (left) and content and images (center). Details have been anonymized.

Task	Content	HTML	CSS	JS
Add a window title to the web page		✓		
Create objects to represent each of your interests	✓			✓
Change the background color and add a border to your page			✓	
Create a space for each of your interest's names		✓		✓
Add a component that displays a photo of your interest				✓
Display interest text paragraphs in their own <div> tags				✓
Give your page a background image			✓	
Give the content area a background color and rounded border			✓	
Use a component to display a page title stored in a variable				✓
Give each paragraph a unique style using .map()			✓	✓
Make a "Surprise Me" button that shows a random interest	✓			✓
Style your buttons with a border and transitions			✓	
Create a menu component with two buttons			✓	✓
Make the menu navigate between the interests and "about me" pages				✓
Fill your "about me" page with content about you	✓	✓		✓
Make the title match the currently selected page				✓
Add an image to "about me" page that changes when clicked				✓
Embed a video in your interest's content area		✓		
Link your images to an external page	✓			✓
Create a photo gallery that displays six images	✓	✓	✓	✓

Table 11: Condensed versions of the prescribed tasks given to the campers and the skills that each task required.

During both the after-lecture activities and project work time, campers in both groups had access to several types of help. We gave campers PDFs of the lectures along with HTML, CSS, and JavaScript "cheat sheets." We also encouraged campers to find online resources on their own. The two instructors and three helpers also offered help upon request. The helpers' goals were twofold: 1) to get the camper on a more productive path without giving them a solution and 2) to gather data about the camper's metacognitive awareness and problem solving strategies. To achieve these goals, helpers provided assistance only when asked to do so, and they never provided code.

When responding to a camper's help request, helpers first asked the camper two questions: 1) "*Describe the problem in as much detail as you can*" and 2) "*What have*

you tried so far?” Additionally, helpers asked the experimental group, “*What problem solving stage do you think you are in?*” (The help request prompts, the third part of our intervention). After these questions, the helpers provided assistance. Next, the helpers recorded detailed observations about the problem(s) the camper had encountered and the assistance provided. At the end of each day, helpers transcribed their notes, elaborating on details they did not capture previously. To practice, the helpers also trained in a 3-day pilot camp.

To provide context-sensitive problem solving prompts to the experimental group (the fourth part of our intervention), we implemented the Idea Garden [Cao et al. 2011, Cao et al. 2012, Cao et al. 2013, Cao et al. 2014, Jernigan et al. 2015] in a panel of the Cloud9 IDE (see Figure 7, main). The Idea Garden prototype for Cloud9 had many similarities to the Gidget version, but it differed in order to fulfill the principles and create an interaction design more cohesive with Cloud9. We reinforced the problem solving stages by housing the Idea Garden’s 14 hints under headers corresponding to the six stages.

The panel housing the hints helped fulfill P6.ContextSensitive and P6.ContextFree; we highlighted hints that were relevant in context after users clicked the ⓘ. For every hint where it made sense, we included identifiers from user code in the hint to fulfill P2-Relevance. Many hints had multiple sections that expanded, allowing users with different P5-InformationProcessing styles to customize what information they wanted to see. When campers triggered a programming anti-pattern, such as forgetting to use the iterator in a *for* loop, the Idea Garden placed an icon on the screen next to the problematic line of code (Figure 7, callout). If the camper then clicked on the icon, the titles of hints relevant to the problem became highlighted.

Data Collection

At the end of each camp day, campers completed an end-of-day survey. To learn about the campers’ metacognitive awareness during the camps, we adapted the techniques of [Whitebread et al. 2009, Sperling et al. 2002], asking campers to reflect on a difficult task and respond to the survey question “*How did you solve this problem? If you didn’t solve it, what did you try?*”

To measure campers’ programming self-efficacy, we adapted the scale by [Askar and Davenport 2009] to fit web development tasks. The eight survey prompts were on a 5-point Likert scale and featured statements such as “*I can write syntactically correct JavaScript statements*”, “*I can complete a programming project even if I only have the documentation for help.*”, and “*When I get stuck I can find ways of overcoming the problem.*”

To measure campers’ growth mindset, we used previous programming aptitude mindset measures of [Scott and Ghinea 2014]. The three survey prompts were also on a 5-point Likert scale and included the statements “*I do not think I can really change my aptitude for programming.*”, “*I have a fixed level of programming aptitude, and not much can be done to change it.*”, and “*I can learn new things about software development, but I cannot change my basic aptitude for programming.*”

To measure productivity, helpers saved the campers’ source code at the end of each camp session. We also captured the experimental group’s use of the Idea Garden, modifying a Cloud9 event logging mechanism to report Idea Garden interactions like opening a hint. The experimental group’s end-of-day surveys included three questions about how campers used the Idea Garden as a resource.

STUDY #3 RESULTS

We begin by briefly summarizing the effects of the treatment on metacognitive awareness, productivity, self-efficacy, and growth mindset. These measures are outside the scope of this thesis because the Idea Garden does not explicitly target helping users in these ways, but we present them here to explain how the campers were affected by the intervention. Next, we present the effects of the treatment on campers' barriers and help requests. Finally, we give a qualitative description of the campers' experiences and outcomes to give context to our results. We make no attempt to tease apart the effects of individual intervention elements (like the Idea Garden) because the intervention is intended to include all those elements at once, so the results presented represent the effect of all intervention elements combined.

Summary of treatment results beyond the scope of this thesis

Briefly, Study #3's overall results were: (1) Metacognitive awareness: campers in the experimental group were significantly more likely to write a description of a problem solving strategy than the control group in their end-of-day survey responses. (2) Productivity: the two groups completed similar amounts of project tasks, but the experimental group completed significantly more self-initiated tasks than the control group. (3) Self-efficacy: the experimental group had a significantly higher increase in self-efficacy than the control group over the course of the camp. Also, *all* female campers in the experimental group reported positive self-efficacy, whereas males and control group females did not all report positive self-efficacy. (4) Growth mindset: the control group had a significantly larger erosion of growth mindset than the experimental group over the course of the camp. These results suggest that the combination of the four interventions made a positive impact on the campers.

Differences between treatments in barriers and help requested

Our instruction aimed to help campers be more aware of their current problem solving state, and therefore more capable of evaluating their strategies. Therefore, we predicted that the experimental group would be more independent and make more progress before requiring help than the control group. For example, if a camper in the *implementing a solution* stage struggled with getting some JavaScript to work, the re-

peated exposure to the paper handout, the help request prompts, and the Idea Garden might remind them to search for an alternative solution, think of other similar problems they had solved before, or re-evaluate their understanding of the problem.

To detect this possible change in help requests, we classified the notes on each help request using a previously reported coding scheme on programming learning barriers [Ko et al. 2004]. We list the six barriers in Table 12, showing examples from campers.

Barrier	Definition from [Ko et al. 2004]	Representative Quote from Camper	Control	Experimental
<i>Design</i>	Did not know how to approach solving a problem.	<i>“I’m incredibly lost. I think I’m on task 4?” – camper C92</i>	9%	6.7%
<i>Selection</i>	Had an approach, but did not know what language or API features to use.	<i>“How can I get the title a different color?” – camper C95</i>	27.8%	21.3%
<i>Use</i>	Had a language or API feature, but did not know how to use it.	<i>“I’m kind of confused on how to write an if statement to display the pictures...if the tab is PhotoGallery” – camper E42</i>	34.4%	37.3%
<i>Coordination</i>	Did not know how to use two or more language or API features together.	<i>“This is no longer working. They were separately, but I tried combining them and it doesn’t” – camper C89</i>	4.2%	3.2%
<i>Understanding</i>	Observed a failure and did not have guesses about why it was failing.	<i>“I added this photo code to my webpage and now my buttons don’t work” – camper E37</i>	23.8%	28.8%
<i>Information</i>	Had a guess about why a failure occurred, but could not get information to confirm it.	<i>“I’m using getElementById here in the HTML, but it keeps evaluating to this ‘else’ so I know it’s not working” – camper E50</i>	0.8%	2.7%

Table 12: Each row defines the barrier and gives an example from a help request, along with the percent of each type of barrier reported by each condition in their help requests. Highlighted red cells indicate the control group had the higher of the two proportions, and green cells indicate the experimental group had the higher proportion.

Two researchers coded the helper observations from camper help requests. They

reached 88.75% agreement on 20% of the data and then coded the rest separately. The helper to camper ratio (1:5) in each camp constrained the amount of requests (289 requests in the control, and 309 in the experimental), so we focused on analyzing the relative proportion of different types of requests.

As shown in the two rightmost columns of Table 12, the proportion of help request types varied significantly by condition ($X^2=11.087$, $df=5$, $p=0.049$). Campers in the control group requested assistance with *design* and *selection* barriers more often (devising a solution to a problem and identifying programming language and API constructs to implement it). In contrast, the experimental group requested more help with *understanding* and *information* barriers (how to debug their implementations). Though the difference in proportions of help request types was not large, it appears that campers in the experimental group were more likely to select a solution and implement it independently, allowing them to progress all the way to evaluation before requiring debugging help.

Furthermore, we investigated if one group relied more heavily on the instructor and helpers by checking the correlations between campers' help requests and total productivity scores. We found that the experimental group showed no significant association between help requests and productivity (Pearson: $r(23)=0.278$, $p=0.179$), whereas the control group *did* have a significant association (Pearson: $r(21)=0.467$, $p=0.025$). This suggests that the control group not only encountered early stage barriers more often, but also relied more on the helpers to complete their work.

Camper experiences

Those results make it clear that campers had differing experiences between the two groups. Next, we discuss campers from both treatments, our observations of them, and their camp data to look at those differences. Recall that we refer to campers with a letter indicating their group followed a unique number (e.g. E27 is an experimental camper and C75 a control).

Camper C87 (an 11th grade male) earned a productivity score of 16, at about the 20th percentile in the control group. He completely avoided tasks requiring JavaScript, instead focusing on content. He never created a React component and only requested

assistance from helpers with CSS and HTML. Despite this, he showed enthusiasm for the content he created resulting in a distinctive, personal, and expressive site.

Camper E50 (a 9th grade male) earned a productivity score of 40, the 20th percentile in the experimental group. He focused primarily on simple content changes, but he at least tried the most challenging task (the photo gallery in Table 11). He worked independently and tried to use the Idea Garden, but reported: *“I tried looking at [the map hint] and it wasn't really useful”*. He encountered many early stage learning barriers as well, saying things like *“I don't know where to start. I did display a photo, but I don't know how to create a component.”* Throughout the camp, he demonstrated persistence, but avoided many tasks.

The control group also contained productive campers, such as campers C91 (10th grade male, 206 productivity) and C92 (11th grade male, 214 productivity), who earned the highest scores in the control group. These two campers worked together many times to make progress, but bailed on problems when they couldn't figure it out together, instead turning to helpers with a defeated attitude. For example, camper C91 said, *“Tell me what's wrong here because I'm not going to bother figuring out what's going on,”* showing how quickly he gave up on solving problems independently.

Camper E40 (a 12th grade male) earned a 321 productivity score, the second highest among all campers (we don't discuss camper E51's score of 347 because he had prior programming experience and didn't require much help). Camper E40 used all the resources at his disposal to proceed, discussing his problem solving activities with helpers and interacting frequently with the Idea Garden. On day 3, he read the iteration hints about *for*, *for-in*, and *map* and later asked for help iterating over his list of photos with a *map* function. On day 5, camper E40 said that the Idea Garden gave him new tactics: *“yeah, it told me to try using a map function or a for-in loop and im [sic] trying to get them to work.”* On day 6, helpers recorded two observations of him successfully using iteration without much help.

CONCLUSION

The results from Studies #1, #2, and #3 suggest the Idea Garden's generality in several ways.

First, Table 13 summarizes Study #2's evidence of each principle and its component parts. One way to view the results about these principles is in how they tease apart what each principle adds to supporting a *diversity* of EUPs' problem-solving situations.

P1-Content: Teams' successes across a variety of concepts (Table VII) serve to validate the concept aspect of P1; minipatterns were especially involved in teams' success rates with Coordination barriers; and strategies are discussed in P3 below. Together, these aspects enabled the teams to overcome, without any in-person help, 41%-68% of the barriers they encountered across *diverse barrier types*.

P2-Relevance and P6-Availability, in working together to make available relevant, just-in-time hints, afforded teams several different ways to use the 🧐 to make progress. This suggests that following Principles P2 and P6 can help support *diverse EUP problem-solving styles*.

P3-Actionable's explicit vs. implicit approaches had different strengths. Teams tended to use explicitly actionable instructions (e.g., "Indent...") to translate an idea into code, at the Bloom's taxonomy "apply" stage. In contrast, teams seem to follow implicitly actionable instructions more conceptually and strategically ("recall how you..."), as with Bloom's "analyze" stage. This suggests that the two aspects of P3-Actionable can help support EUPs' learning across *multiple cognitive process stages*.

P5-InformationProcessing: P5 requires supporting both the comprehensive and selective information processing styles, as per previous research on gender differences in information processing. The teams used both of these styles, mostly aligning by gender with the previous research. This suggests that following P5-InformationProcessing helps support *diverse EUP information processing styles*.

Table 13: Summary of principle-by-principle evaluations.

+: Principle was helpful, -: Principle was problematic.

*: Teams progressed in the majority ($\geq 50\%$) of their barriers with these Idea Garden principles.

Principle	Ways	Formative Evidence	Summative Evidence
P1-Content		+Study1	+Study2*
	P2-All	-[Cao et al. 2012]	
P2-Relevance	P2.1-MyCode		+Study2*
	P2.2-MyState	+Study1	+Study2*
	P2.3-MyRequirements	+Study1	
P3-Actionable	P3.1-ExplicitlyActionable		+Study2*
	P3.2-ImplicitlyActionable		+Study2*
P4-Personality			+ [Lee and Ko 2011]
P5-InformProc		+ [Meyers-Levy 1989]	+Study2*
P6-Availability	P6.1-ContextFree	+, -Study1	+Study2*
	P6.2-ContextSensitive	+Study1	+Study2*
P7-Interruption Style			+ [Robertson et al. 2004]

Second, Study #2 showed the teams learned enough programming in only about 5 hours to begin building their own game levels comparable to those created in a prior study of Gidget [Lee et al. 2014]. However, unlike the prior study, they accomplished these gains with significantly less in-person help than in the previous study.

Third, previous research [Cao et al. 2011, Cao et al. 2012, Cao et al. 2013, Cao et al. 2014, Cao 2013] together with this thesis and [Jernigan et al. 2015] show that the Idea Garden can be implemented in multiple environments. This thesis also presented the generalized architecture used in Study #3 that could help facilitate future implementations.

Finally, Study #3 showed that campers in the experimental group (who had access to the Idea Garden and problem solving instruction) did not depend on the helpers, but the control group did. The experimental group also advanced further in their problems

than the control group before requiring help. Campers from both Study #2 and Study #3 showed that the Idea Garden and its principles help support *independent work across diverse environments and tasks*.

These promising results from Studies #1, #2, and #3 suggest the effectiveness of the Idea Garden's principles and support for different contexts in helping EUPs solve the programming problems that get them "stuck"—across a diversity of problems, information processing and problem-solving styles, cognitive stages, environments, tasks, and people.

BIBLIOGRAPHY

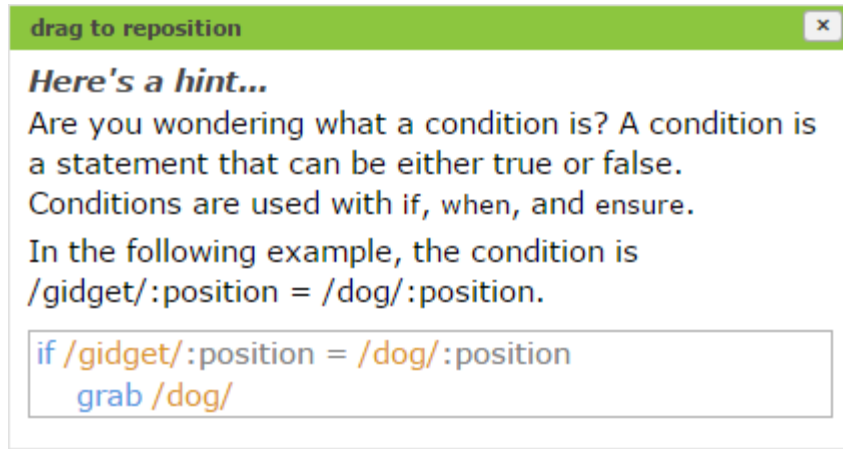
- [Andersen and Mørch 2009] Andersen, R. and Mørch, A. Mutual development: A case study in customer-initiated software product development. *End-User Development*, (2009), 31-49.
- [Anderson et al. 2001] Anderson, L. (Ed.), Krathwohl, D. (Ed.), Airasian, P., Cruikshank, K., Mayer, R., Pintrich, P., Raths, J., Wittrock, M. *A Taxonomy for Learning, Teaching, and Assessing: A revision of Bloom's Taxonomy of Educational Objectives (Complete edition)*. Longman. (2001)
- [Askar and Davenport 2009] Askar, P. and Davenport, D. 2009. An investigation of factors related to self-efficacy for Java programming among engineering students. *Online Submission* 8, 1. <http://eric.ed.gov/?id=ED503900>.
- [Brandt et al. 2010] Brandt, J., Dontcheva, M., Weskamp, M., and Klemmer, S. Example-centric programming: Integrating web search into the development environment. In *Proc. CHI 2010*, ACM (2010), 513-522.
- [Bransford et al. 1999] Bransford, J., Brown, A., Cocking, R. (Eds), *How People Learn: Brain, Mind, Experience, and School*, National Academy Press, 1999.
- [Burnett et al. 2011] Burnett, M., Beckwith, L., Wiedenbeck, S., Fleming, S., Cao, J., Park, T., Grigoreanu, V., Rector, K. Gender pluralism in problem-solving software. *Interacting with Computers*. 23 (2011), 450–460.
- [Cao 2013] Jill Cao. 2013. Helping End–User Programmers Help Themselves - The Idea Garden Approach. Ph.D Dissertation. Oregon State University, Corvallis, OR.
- [Cao et al. 2014] Cao, J., Fleming, S., Burnett, M., Scaffidi, C. Idea Garden: Situated support for problem solving by end-user programmers. *Interacting with Computers*, 2014. (21 pages)
- [Cao et al. 2011] Cao, J., Fleming, S. D., and Burnett, M., An exploration of design opportunities for ‘gardening’ end-user programmers’ ideas, *IEEE VL/HCC* (2011), 35-42.
- [Cao et al. 2013] Cao, J., Kwan, I., Bahmani, F., Burnett, M., Fleming, S., Jordahl, J., Horvath, A. and Yang, S. End-user programmers in trouble: Can the Idea Garden help them to help themselves? *IEEE VL/HCC*, 2013, 151-158.
- [Cao et al. 2012] Cao, J., Kwan, I., White, R., Fleming, S., Burnett, M., and Scaffidi, C. From barriers to learning in the Idea Garden: An empirical study. *IEEE VL/HCC*, 2012, 59-66.
- [Carroll and Rosson 1987] Carroll, J. and Rosson, M. The paradox of the active user. *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*, MIT Press. 1987.
- [Carroll 1990] Carroll, J. *The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill*. 1990.
- [Costabile et al. 2009] Costabile, M., Mussio, P., Provenza, L., and Piccinno, A. Supporting end users to be co-designers of their tools. *End-User Development*, Springer (2009), 70-85.
- [Cypher et al. 2010] Cypher, A., Nichols, J., Dontcheva, M., and Lau, T. *No Code Required: Giving Users Tools To Transform the Web*, Morgan Kaufmann. 2010.

- [Diaz et al. 2010] Diaz, P., Aedo, I., Rosson, M., Carroll, J. (2010) A visual tool for using design patterns as pattern languages. In *Proc. AVI*. ACM Press 2010. 67–74.
- [Dorn 2011] Dorn, B. ScriptABLE: Supporting informal learning with cases, In *Proc. ICER*, ACM, 2011. 69-76.
- [Grigoreanu et al. 2010] Grigoreanu, V., Burnett, M., Robertson, G. A strategy-centric approach to the design of end-user debugging tools. *ACM CHI*, (2010), 713-722.
- [Grigoreanu et al. 2012] Grigoreanu, V., Burnett, M., Wiedenbeck, S., Cao, J., Rector, K., Kwan, I. End-user debugging strategies: A sensemaking perspective. *ACM TOCHI* 19, 1 (2012), 5:1-5:28.
- [Gross et al. 2010] Gross, P., Herstand, M., Hodges, J., and Kelleher, C. A code reuse interface for non-programmer middle school students. *ACM IUI 2010*. 2010. 219-228.
- [Guzdial 2008] Guzdial, M. Education: Paving the way for computational thinking. *Comm. ACM* 51, 8 (2008), 25–27.
- [Hundhausen et al. 2009] Hundhausen, C., Farley, S., and Brown, J. Can direct manipulation lower the barriers to computer programming and promote transfer of training? An experimental study. *ACM TOCHI* 16, 3 (2009), Article 13.
- [Jernigan et al. 2015] Jernigan, W., Horvath, A., Lee, M., Burnett, M., Culty, T., Kuttal, S., Peters, A., Kwan, I., Bahmani, F., Ko, A. 2015. A principled evaluation for a principled Idea Garden. *IEEE VL/HCC* 2015, to appear.
- [Kelleher and Pausch 2006] Kelleher, C. and Pausch, R. Lessons learned from designing a programming system to support middle school girls creating animated stories. *IEEE VL/HCC* (2006). 165-172.
- [Kelleher and Pausch 2005] Kelleher, C. and Pausch, R. Stencils-based tutorials: design and evaluation. *ACM CHI*, 2005, 541-550.
- [Ko et al. 2004] Ko, A., Myers, B., and Aung, H.. Six learning barriers in end-user programming systems. *IEEE VLHCC* 2004, 199-206.
- [Kumar et al. 2011] Kumar, R., Talton, J., Ahmad, S., and Klemmer, S. Bricolage: Example-based retargeting for web design. *ACM CHI*, 2011. 2197-2206.
- [Lee and Ko 2011] Lee, M. and Ko, A. Personifying programming tool feedback improves novice programmers' learning. In *Proc. ICER*, ACM Press (2011), 109-116.
- [Lee et al. 2014] Lee, M., Bahmani, F., Kwan, I., Laferte, J., Charters, P., Horvath, A., Luor, F., Cao, J., Law, C., Beswetherick, M., Long, S., Burnett, M., and Ko, A. Principles of a debugging-first puzzle game for computing education. *IEEE. VL/HCC* 2014, 57-64.
- [Little et al. 2007] Little, G., Lau, T., Cypher, A., Lin, J., Haber, E., and Kandogan, E. Koala: Capture, share, automate, personalize business processes on the web. *ACM CHI* 2007, 943-946.
- [Loksa et al. 2016] Loksa, D., Ko, A., Jernigan, W., Oleson, A., Mendez, C., Burnett, M. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. *ACM CHI* 2016, under review.

- [Meyers-Levy 1989] Meyers-Levy, J., Gender differences in information processing: A selectivity interpretation, In P. Cafferata and A. Tubout (eds.), *Cognitive and Affective Responses to Advertising*, Lexington Books, 1989.
- [Myers et al. 2004] Myers, B., Pane, J. and Ko, A. Natural programming languages and environments. *Comm. ACM* 47, 9 (2004), 47-52.
- [Nardi 1993] Nardi, B. *A Small Matter of Programming*, MIT Press (1993).
- [Oney and Myers 2009] Oney, S. and Myers, B. FireCrystal: Understanding interactive behaviors in dynamic web pages. *IEEE VL/HCC* (2009), 105-108.
- [Pane and Myers 2006] Pane, J. and Myers, B. More natural programming languages and environments. In *Proc. End User Development*, Springer (2006), 31-50.
- [Sperling et al. 2002] Sperling, R., Howard, B., Miller, L., and Murphy, C. 2002. Measures of children's knowledge and regulation of cognition. *Contemporary educational psychology* 27, 1: 51-79.
- [Robertson et al. 2004] Robertson, T., Prabhakararao, S., Burnett, M., Cook, C., Ruthruff, J., Beckwith, L., and Phalgune, A. Impact of interruption style on end-user debugging. *ACM CHI* (2004), 287-294.
- [Scott and Ghinea 2014] Scott, M., and Ghinea, G. 2014. On the domain-specificity of mindsets: The relationship between aptitude beliefs and programming practice. *IEEE Transactions on Education*, 57, 3: 169-174.
- [Tillmann et al. 2013] Tillmann, N., De Halleux, J., Xie, T., Gulwani, S., and Bishop, J. Teaching and learning programming and software engineering via interactive gaming. *ACM/IEEE International Conference on Software Engineering*, 2013, 1117-1126.
- [Turkle and Papert 1990] Turkle, S. and Papert, S. Epistemological Pluralism. *Signs* 16(1), 1990.
- [Whitebread et al. 2009] Whitebread, D., Coltman, P., Deborah Pino Pasternak, Sangster, C., Grau, V., Bingham, S., Almeqdad, Q., and Demetriou, D. 2009. The development of two observational tools for assessing metacognition and self-regulated learning in young children. *Metacognition and Learning* 4, 1: 63-85.

APPENDICES

Appendix A: Idea Garden Hints in Gidget



Conditions

drag to reposition

Here's a hint...

Are you trying to make an object do something whenever a certain condition is met? Try something like this:

```

object cat(position)
  set this:position to position
  set this:flagged to false
  when /gidget/:position = this:position and
this:flagged = false ! for steps 1 and 2
    sound /'cat_Meow/' !
    set this:position[0] to this:position[0] + 1 !
This is for step 3
    set this:flagged to true ! This is for step 3
create cat([1,1]) ! This is for step 4

```

When is short for "whenever": it's a **rule** that tells an **object**, as soon as you create it, to **constantly monitor** the world for a particular condition. Do it like this:

1. Inside the object definition, start a when rule by indenting (tabbing) and typing "when".
2. On the same line as the "when", type the condition you want the cat to watch out for.

...click to see more!

Events (collapsed)

drag to reposition

Here's a hint...

Are you trying to make an object do something whenever a certain condition is met? Try something like this:

```

object cat(position)
  set this:position to position
  set this:flagged to false
  when /gidget/:position = this:position and
this:flagged = false ! for steps 1 and 2
    sound /'cat_Meow/' !
    set this:position[0] to this:position[0] + 1 !
This is for step 3
    set this:flagged to true ! This is for step 3
create cat([1,1]) ! This is for step 4

```

When is short for "whenever": it's a **rule** that tells an **object**, as soon as you create it, to **constantly monitor** the world for a particular condition. Do it like this:

1. Inside the object definition, start a when rule by indenting (tabbing) and typing "when".
2. On the same line as the "when", type the condition you want the cat to watch out for.
3. Under the "when" line, indent (tab) the lines that tell the cat what to do on this condition.
4. After you create a new cat, it will follow this rule from now on!

Events (expanded)

drag to reposition

×

Here's a hint...
Are you trying to write and use functions? Try something like this:

```
function reset(myobject) ! These lines are
  goto myobject          ! for step 1
  set myobject:scale to 1
reset(/goop/)            ! the function can be used for the
goop,
reset(/bird/)            ! then reused for the bird!
```

Functions allow you to write code once and then use it multiple times by referring to its name.

1. First, write the function definition.
2. Then, call the function. ...*click to see more!*

Functions (collapsed)

drag to reposition

Here's a hint...

Are you trying to write and use functions? Try something like this:

```
function reset(myobject) ! These lines are
  goto myobject          ! for step 1
  set myobject:scale to 1
reset(/goop/)            ! the function can be used for the
goop,
reset(/bird/)            ! then reused for the bird!
```

Functions allow you to write code once and then use it multiple times by referring to its name.

- First, write the function definition.
 - Type the keyword "function".
 - On the same line as "function", type the name you want to give to the function, followed by parentheses.
 - If you want to use parameters, type the parameter(s) in the parentheses.
 - Indent the commands that should run every time this function is called.
- Then, call the function.
 - Type the name of the function followed by the values you want to pass into the parameters wrapped in parentheses.

Functions (expanded)

drag to reposition ✕

Here's a hint...

Are you trying to do action(s) only when a condition such as `/puppy/:scale = 1` is true? Try something like this:

```
if /puppy/:scale = 1
  goto /puppy/
else
  goto /kitten/
```

If the condition is true, the if statement makes the action(s) occur. Do it like this:

1. Write the lines for those actions.
2. Make those actions run only when the condition is true with an if statement.
3. Should different actions happen if the condition is false? Type those actions in an `else` statement.

...click to see more!

Conditional Statements (collapsed)

drag to reposition ✕

Here's a hint...

Are you trying to do action(s) only when a condition such as `/puppy/:scale = 1` is true? Try something like this:

```
if /puppy/:scale = 1
  goto /puppy/
else
  goto /kitten/
```

If the condition is true, the if statement makes the action(s) occur. Do it like this:

1. Write the lines for those actions.
2. Make those actions run only when the condition is true with an if statement.
 - 2.1 **Type** "if /puppy/:scale = 1" before lines that you want to run when the condition is true.
 - 2.2 **Indent(tab)** the lines that should run.
3. Should different actions happen if the condition is false? Type those actions in an `else` statement.
 - 3.1 **Type** `else` after those lines and press enter.
 - 3.2 **Write** the code for the actions that should happen if the condition is false and indent that code.

Conditional Statements (expanded)

drag to reposition ×

Here's a hint...

Are you trying to repeat the same action on every dog?

Try something like this:

```
for x in /dog/s  
  goto x
```

The for loop helps you generalize the solution from one dog to every dog. Do it like this:

1. At first, pretend there is only one dog. Write the lines for that one particular dog.
2. Make your solution repeat for each dog in the `/dog/s` list. ...*click to see more!*

Iteration (collapsed)

drag to reposition ✕

Here's a hint...

Are you trying to repeat the same action on every dog?

Try something like this:

```
for x in /dog/s  
  goto x
```

The for loop helps you generalize the solution from one dog to every dog. Do it like this:

1. At first, pretend there is only one dog. Write the lines for that one particular dog.
2. Make your solution repeat for each dog in the `/dog/s` list.
 - 2.1 **Type** `"for x in /dog/s"` before lines you want to repeat.
 - 2.2 Under that, **indent (tab)** the lines you wrote in step 1, to show that they should repeat.
 - 2.3 **Replace** `"dog/"` with `"x"` wherever you have a line that should repeat for each dog in the `/dog/s` list.

Iteration (expanded)

drag to reposition ×

Here's a hint...

A list is a group of things like a group of puppies, a group of numbers, and so on. `/puppy/s` refers to every puppy in the world.

If you want to do something to a single puppy, you can use a number in brackets, like below:

```
goto /puppy/s[0] ! The first puppy
goto /puppy/s[1] ! The second puppy
```

You could also go through the entire list to `goto` each one:

```
for x in /puppy/s
  goto x
```

Lists

drag to reposition ×

Here's a hint...

Are you trying to add a new object to the game? Try something like this:

```
object bucket(message) ! These steps are
  say message ! for step 1.
create bucket("I am here") ! for step 2
create bucket("Me too") ! for step 2
```

The world already contains some objects that you have seen, like goop, bird, and dog. You can make new types of objects by doing the following.

1. First, you have to define the object with the `object` keyword.
2. Then, create the object with the `create` keyword.

...click to see more!

Objects (collapsed)

drag to reposition

Here's a hint...

Are you trying to add a new object to the game?
Try something like this:

```

object bucket(message)    ! These steps are
    say message           ! for step 1.
create bucket("I am here") ! for step 2
create bucket("Me too")    ! for step 2

```

The world already contains some objects that you have seen, like goop, bird, and dog. You can make new types of objects by doing the following.

1. First, you have to define the object with the `object` keyword. Do it like this:
 - 1.1 Type the keyword "object".
 - 1.2 On the same line as "object", type the name you want to give the object, followed by parentheses.
 - 1.3 If you want to use parameters, type the parameter(s) in the parentheses.
 - 1.4 If you have any commands you want to run when the object is created, indent those commands under the object line.
2. Then, create the object with the `create` keyword. Do it like this:
 - 2.1 Type the keyword "create".
 - 2.2 Type the name of the object followed by the values you want to pass into the parameters wrapped in parentheses.

Objects (expanded)

Appendix B: Idea Garden Hints in Cloud9

Idea Garden

Are you working on a certain problem solving stage? Try looking at these:

Reinterpret Problem Prompt

- Divide and Conquer ?

Search for Solutions

- Working Backwards ?

Implementation of Solution

- Conditional Statements ?
- Events ?
- Functions ?
- Iteration with For ?
- Iteration with For-In ?
- Iteration with Map ?
- Iteration with While ?
- Lists ?
- Objects ?
- Variables ?

Evaluation of Implementation

- Can it work better with Functions? ?
- Can it work better with Iteration? ?

The Idea Garden Panel

Divide and Conquer ?

Are you reinterpreting the problem, trying to eat an elephant? You might try eating it one slice at a time: divide the problem into parts, then solve each part by itself.

When Taylor got stuck on how to make a web-based card game, she thought about dividing the problem in these ways:

- Making players and cards with objects ?
- Referring to all the players at once to keep score, etc. with a list ?
- Referring to all the cards as a deck with a list ?
- Dealing cards then playing the game with functions ?

Based on how she divided up the problem, Taylor could then choose a particular section of the solution to work on first.

Reinterpret Problem Prompt: Divide and Conquer

Working Backwards ?

Are you searching for solutions? You may want to try working backwards.

When Taylor wanted to imitate a web page element from her favorite website, she worked backwards to put a similar element on her own page:

- [click to see more...](#)

To work backwards, try identifying your output and what it should do. Then you might create something that "looks like" that output. Finally, you could give the output the functionality it should have.

You might want to read about events: ?

Search for Solutions: Working Backwards (Collapsed)

Working Backwards ?

Are you searching for solutions? You may want to try working backwards.

When Taylor wanted to imitate a web page element from her favorite website, she worked backwards to put a similar element on her own page:

- [click to see less...](#)

1. First, Taylor decided she wanted a picture on her web page that would change its border color to green when clicked.
2. Second, Taylor placed a picture on her page with a green border, but it didn't change if clicked.
3. Finally, Taylor worked backwards; she gave the picture an onClick event that changed its border color to green.

To work backwards, try identifying your output and what it should do. Then you might create something that "looks like" that output. Finally, you could give the output the functionality it should have.

You might want to read about events: ?

Search for Solutions: Working Backwards (Expanded)

Conditional Statements ?

Are you trying to do something only when a condition is true? You may want to try using an if statement like this:

```
if (x === y) {  
    addToPage("x is equal to y");  
}  
else if (x === y + 1) {  
    addToPage("x is one more than y");  
}  
else if (x === y - 1) {  
    addToPage("x is one less than y");  
}  
else {  
    addToPage("x and y couldn't be compared");  
}
```

If the original condition is true, the if statement makes the action(s) occur. To write your own if statement, try something like this:

1. Write the lines for those actions.
2. Make those actions run only when a condition is true with an if statement.
· [click to see more...](#)
3. If different actions should happen when other conditions are true, type those conditions in an else if statement.
· [click to see more...](#)
4. Consider if different actions should happen if all the preceding conditions are false. If so, these actions should be typed in an else statement.
· [click to see more...](#)

Implementation of Solution: Conditional Statements (Collapsed)

Conditional Statements ?

Are you trying to do something only when a condition is true? You may want to try using an if statement like this:

```
if (x === y) {
    addToPage("x is equal to y");
}
else if (x === y + 1) {
    addToPage("x is one more than y");
}
else if (x === y - 1) {
    addToPage("x is one less than y");
}
else {
    addToPage("x and y couldn't be compared");
}
```

If the original condition is true, the if statement makes the action(s) occur. To write your own if statement, try something like this:

1. Write the lines for those actions.
2. Make those actions run only when a condition is true with an if statement.
 - ~ [click to see less...](#)
 - 1. Type "if (myCondition){ " before the lines that you want to run when the condition is true, replacing "myCondition" with your condition.
 - 2. Indent (tab) the lines that should run when the condition is met.
 - 3. Add a "}" at the end of the lines that should run.
3. If different actions should happen when other conditions are true, type those conditions in an else if statement.
 - ~ [click to see less...](#)
 - 1. Type "else if (", followed by a new condition, then "){ "
 - 2. Press enter and write the code for the actions that should happen if this condition is true.
 - 3. Indent (tab) the code you just wrote.
 - 4. After the code, press enter and type "}"
4. Consider if different actions should happen if all the preceding conditions are false. If so, these actions should be typed in an else statement.
 - ~ [click to see less...](#)
 - 1. Type "else { " after the if and else if statements and press enter.
 - 2. Write the code for the actions that should happen if all the conditions are false and indent that code.
 - 3. Add a "}" at the end of the lines that should run.

Implementation of Solution: Conditional Statements (Expanded)

Events

Are you trying to make your webpage respond to an event like clicking? You could try a React class like this:

```
var message = "Button not clicked."
var Button = React.createClass({
  handleClick: function() {
    message = "Button clicked!";
    this.forceUpdate();
  },
  render: function() {
    return (
      <div onClick={this.handleClick}>
{message}</div>
    )
  }
});
React.render(<Button/>,
document.getElementById("cool"));
```

Events allow you to watch for actions occurring on your website. To watch for events, try something like this:

1. Find what event your website will use and decide what should happen when the event occurs.
2. Decide which React element will respond to the event.
3. Inside your JavaScript file, write a function for what behavior should happen when the event occurs.
 - [click to see more...](#)
4. Create a component to render it.
 - [click to see more...](#)

Implementation of Solution: Events (Collapsed)

Events

Are you trying to make your webpage respond to an event like clicking? You could try a React class like this:

```
var message = "Button not clicked."
var Button = React.createClass({
  handleClick: function() {
    message = "Button clicked!";
    this.forceUpdate();
  },
  render: function() {
    return (
      <div onClick={this.handleClick}>
{message}</div>
    )
  }
});
React.render(<Button />,
document.getElementById("cool"));
```

Events allow you to watch for actions occurring on your website. To watch for events, try something like this:

1. Find what event your website will use and decide what should happen when the event occurs.
2. Decide which React element will respond to the event.
3. Inside your JavaScript file, write a function for what behavior should happen when the event occurs.

[click to see less...](#)

1. Type "myEventName: function() {" and press enter, replacing myEventName with the name of the event you're using.
2. Inside the function, add behavior for the event. For instance, you might want to change the border or background-color properties.
3. At the end of the function, use "this.forceUpdate()" to draw your element again
4. Press return and type "}" to close the event function.

4. Create a component to render it.

[click to see less...](#)

1. Type "React.render(<myElement />, ", replacing myElement with the name of your React class.
2. Type "document.getElementById("id"));", replacing "id" with the name of your element in the HTML page.

Implementation of Solution: Events (Expanded)

Functions ?

Are you trying to write and use functions? Try something like this:

```
function multiplyTwoNumbers (num1, num2) {  
  return num1 * num2;  
}  
x = multiplyTwoNumbers(5, 19); // The function can  
be used for these values  
y = multiplyTwoNumbers(3, 7); // ...and reused for  
these values!
```

Functions help generalize your solution by letting you to reuse the same code, process different inputs, and get an output with "return". To write your own functions, try something like this:

1. First, write the function definition.
 - [click to see more...](#)
2. Then, call the function.
 1. Type the name of the function followed by the values you want to use as parameters enclosed in parentheses.

Implementation of Solution: Functions (Collapsed)

Functions ?

Are you trying to write and use functions? Try something like this:

```
function multiplyTwoNumbers (num1, num2) {  
  return num1 * num2;  
}  
  
x = multiplyTwoNumbers(5, 19); // The function can  
be used for these values  
y = multiplyTwoNumbers(3, 7); // ...and reused for  
these values!
```

Functions help generalize your solution by letting you to reuse the same code, process different inputs, and get an output with "return". To write your own functions, try something like this:

1. First, write the function definition.
 - [click to see less...](#)
 - 1. Type the keyword "function", then the name you want to give to the function, followed by parentheses ().
 - 2. If you want to give your function input(s), you could use parameters. Type the parameter(s) inside the parentheses.
 - 3. Type {, then write the lines that should run every time this function is called.
 - 4. Return the output of the function with "return".
 - 5. Press enter and type } to complete the function definition.
2. Then, call the function.
 1. Type the name of the function followed by the values you want to use as parameters enclosed in parentheses.

Implementation of Solution: Functions (Expanded)

Iteration with For ?

Are you trying to repeat the same action multiple times? You might want to try using a for loop. For example, you could add numbers like this:

```
var sum = 0;
for (var index = 1; index <= 10; index++){
  sum += index; // adds to sum's current value
  addToPage(index); // prints 1, then 2, then
  3...
}
```

addToPage(sum);

To write your own for loop, try something like this:

1. Write the line(s) of code that you want to repeat.
2. Make your solution repeat for each line of code you want to repeat with a for loop.

[click to see more...](#)

Implementation of Solution: Iteration with For (Collapsed)

Iteration with For ?

Are you trying to repeat the same action multiple times? You might want to try using a for loop. For example, you could add numbers like this:

```
var sum = 0;
for (var index = 1; index <= 10; index++){
  sum += index; // adds to sum's current value
  addToPage(index); // prints 1, then 2, then
  3...
}
```

addToPage(sum);

To write your own for loop, try something like this:

1. Write the line(s) of code that you want to repeat.
2. Make your solution repeat for each line of code you want to repeat with a for loop.

[click to see less...](#)

1. Type "for (" above the lines of code you want to repeat.
2. Decide what your starting number is (maybe 0 or 1?). Type "var index = " followed by your starting number and a semicolon.
3. Decide what your ending number will be (in the example, it's 10). Type "index <= " followed by your ending number and a semicolon.
4. Type "index++" {" and press enter. "index++" adds 1 to index every time the code loops.
5. After all the lines you want to repeat, press enter and type "}".

Implementation of Solution: Iteration with For (Expanded)

Iteration with For-In ⓘ

Are you trying to repeat the same action on everything in a list? You may want to use a for-in loop:

```
var list = ["Alice", "Bob", "Charlie"];
for (var iterator in list) {
  addToPage(list[iterator]);
}
```

To write your own for-in loop, try something like this:

1. Pretend you only care about the first element in the list. Make your code work for this single element by using `list[0]`.
2. Make your solution repeat for each element in `list` with a for-in loop, replacing `list` with the name of the list.

· [click to see more...](#)

Implementation of Solution: Iteration with For-In (Collapsed)

Iteration with For-In ⓘ

Are you trying to repeat the same action on everything in a list? You may want to use a for-in loop:

```
var list = ["Alice", "Bob", "Charlie"];
for (var iterator in list) {
  addToPage(list[iterator]);
}
```

To write your own for-in loop, try something like this:

1. Pretend you only care about the first element in the list. Make your code work for this single element by using `list[0]`.
2. Make your solution repeat for each element in `list` with a for-in loop, replacing `list` with the name of the list.

· [click to see less...](#)

1. Type the line `for (var iterator in list)` {" above the lines you want to repeat.
2. Replace any 0's in your repeating code with `iterator`. For example, this might look something like `list[iterator]`.
3. End your loop by pressing return at the end of your repeating code and typing `}`.

Implementation of Solution: Iteration with For-In (Expanded)

Iteration with Map ?

Are you trying to repeat the same action multiple times? You might want to try using the map function with your list. Using map, you add up numbers 1 through 5 like this:

```
var myList = [1,2,3,4,5];
var sum = 0;
myList.map(function(e) {
  sum += e;
});
addToPage(sum);
```

To write your own map function, try something like this:

1. Decide what actions you want to do on each element in the list. Write the code that would do this for a single element in the list, such as "myList[0]".
2. Make your solution repeat for each element in myList with the map function.

· [click to see more...](#)

For more on lists, click the icon: ?

Implementation of Solution: Iteration with Map (Collapsed)

Iteration with Map ?

Are you trying to repeat the same action multiple times? You might want to try using the map function with your list. Using map, you add up numbers 1 through 5 like this:

```
var myList = [1,2,3,4,5];  
var sum = 0;  
myList.map(function(e) {  
    sum += e;  
});  
addToPage(sum);
```

To write your own map function, try something like this:

1. Decide what actions you want to do on each element in the list. Write the code that would do this for a single element in the list, such as "myList[0]".
2. Make your solution repeat for each element in myList with the map function.
↳ [click to see less...](#)
 1. Type the line "myList.map(function(e) {" above the lines you want to repeat.
 2. Replace any instances of "myList[0]" in your repeating code with "e". For instance, you might change "sum += myList[0]" to "sum += e".
 3. End your map function by pressing return at the end of your repeating code and typing "}".

For more on lists, click the icon: ?

Implementation of Solution: Iteration with Map (Expanded)

Iteration with While

Are you trying repeat a task multiple times until a condition is met? You might want to use a while loop. Try something like this:

```
var x = 0;
while (x < 5) {
  addToPage("repeating 5 times!");
  x++;
}
```

In this example, the lines inside the { and } will repeat while x is still less than 5. The variable x will be incremented each time the code inside the brackets runs. To make a while loop, try this:

1. Write the line(s) of code that you want to repeat.
2. Decide what your stopping condition will be.
3. Make your solution repeat until the stopping condition is met with a while loop.
[click to see more...](#)
4. Make sure that your stopping condition will be reached at some point! Otherwise, you'll have an infinite loop.

Implementation of Solution: Iteration with While (Collapsed)

Iteration with While

Are you trying repeat a task multiple times until a condition is met? You might want to use a while loop. Try something like this:

```
var x = 0;
while (x < 5) {
  addToPage("repeating 5 times!");
  x++;
}
```

In this example, the lines inside the { and } will repeat while x is still less than 5. The variable x will be incremented each time the code inside the brackets runs. To make a while loop, try this:

1. Write the line(s) of code that you want to repeat.
 2. Decide what your stopping condition will be.
 3. Make your solution repeat until the stopping condition is met with a while loop.
- [click to see less...](#)
1. Type "while (yourStoppingCondition) {", replacing "yourStoppingCondition" with whatever your condition is, above your code for doing the action once.
 2. Below your code for doing the action once, add a statement that changes the variable in the condition (in the example, this is "x++").
 3. Close your while loop by pressing return and typing "}"
 4. Make sure that your stopping condition will be reached at some point! Otherwise, you'll have an infinite loop.

Implementation of Solution: Iteration with While (Expanded)

Lists ?

Are you trying to use a list? Try something like this:

```
var myList = ["Amber" , "Alannah", "Charles" ];
addToPage(myList[0]);    // This prints "Amber" --
// lists start counting at 0, not 1!
myList[3] = "Will";      // Adds Will to the end
```

An list is like a group of variables in the order they were declared. You can refer to one variable in the list with brackets surrounding a number like `[0]`. You might try these steps:

1. Declare your list by typing `"var myList = "`, replacing `"myList"` with a name you choose.
2. Fill your list by typing the values you want inside square brackets. Separate the values by commas also. Try this: `"[1, 5, 29];"`

Some things to be cautious of when using lists:

· [click to see more...](#)

Implementation of Solution: Lists (Collapsed)

Lists ?

Are you trying to use a list? Try something like this:

```
var myList = ["Amber" , "Alannah", "Charles" ];
addToPage(myList[0]);    // This prints "Amber" --
// lists start counting at 0, not 1!
myList[3] = "Will";      // Adds Will to the end
```

An list is like a group of variables in the order they were declared. You can refer to one variable in the list with brackets surrounding a number like `[0]`. You might try these steps:

1. Declare your list by typing `"var myList = "`, replacing `"myList"` with a name you choose.
2. Fill your list by typing the values you want inside square brackets. Separate the values by commas also. Try this: `"[1, 5, 29];"`

Some things to be cautious of when using lists:

· [click to see less...](#)

1. You can overwrite variables in the list like this:


```
myList[2] = "Chris"; //This changes "Charles"
to "Chris"
```
2. You can overwrite an entire list like this:


```
myList = "Rory"; //Replaces the entire list
with "Rory"; everything else will be
overwritten!
```

Implementation of Solution: Lists (Expanded)

Objects 📌

Are you trying to use objects? To make your own object, try something like this:

```
var myObject = {
  name : "BobTheObject",
  color : "MellowYellow",
  dateCreated: "02/31/2017"
};

addToPage(myObject.name); //You can reference
properties like this! This statement prints
"BobTheObject"
```

An object is a container that holds a group of related properties. Like a variable, each property has a name and a value. You could try creating objects like this:

· [click to see more...](#)

Implementation of Solution: Objects (Collapsed)

Objects 📌

Are you trying to use objects? To make your own object, try something like this:

```
var myObject = {
  name : "BobTheObject",
  color : "MellowYellow",
  dateCreated: "02/31/2017"
};

addToPage(myObject.name); //You can reference
properties like this! This statement prints
"BobTheObject"
```

An object is a container that holds a group of related properties. Like a variable, each property has a name and a value. You could try creating objects like this:

· [click to see less...](#)

1. Type `var yourObjectName= {"` replacing `yourObjectName` with a name of your choosing.
2. Consider what properties you want your object to have and what values each property should contain.
3. For each of your properties, type `"yourPropertyName : yourPropertyValue, "` replacing `"yourPropertyName"` and `"yourPropertyValue"` with your names and values.
4. After your last property, type `"};"` instead of a comma.

Implementation of Solution: Objects (Expanded)

Variables ?

Are you trying to use variables? A variable is a container for a number, string, list, object, etc. Try something like this:

```
var myAge = 15;
var today = "My Birthday";
myAge = myAge + 1; //myAge is now 16
```

Variables store the information you need to make your program work. Your program can update variables and use them on multiple lines to make your page dynamic. To make a variable, you could try this:

1. Declare the variable by typing `var varName =` (replacing `varName` with your variable's name) and then the value you want to store.
2. Access or modify the variable by referring to it later in the program. Any changes you make will update the value stored in the variable.

Implementation of Solution: Variables

Can it work better with Functions? ?

Are you trying to evaluate your solution? You might want to improve your solution by generalizing it with a function. For example:

```
var result1 = 5 + 8;
var result2 = 12 + 3;
```

Could be rewritten as

```
function addTwoNumbers (a, b) {
  return a+b;
}
var result1 = addTwoNumbers(5, 8);
var result2 = addTwoNumbers(12, 3);
```

By generalizing your solution through functions, you can make your solution apply to more situations. The function above will set "result1" and "result2" depending on the input, but the first example where "result1" is set to "5 + 8" will not apply in situations beyond 5+8. For more on functions, click the icon: ?

Evaluation of Implementation: Can it work better with Functions?

Can it work better with Iteration? ?

Are you trying to evaluate your solution? If you notice your code has similar lines repeated for multiple variables, you might want to improve your solution by generalizing it through iteration. For example:

```
var yourListName = ["a", "b", "c"];
for (var x in yourListName) {
  yourListName[x] = x;
  addToPage(yourListName[x]);
}
```

The for loop helps you generalize your solution for every element in `yourListName`. Try something like this:

[click to see more...](#)

You might want to consider conditional statements to improve your iteration: ?

For more on for loops using lists, click the icon: ?

For more on lists, click the icon: ?

Evaluation of Implementation: Can it work better with Iteration? (Collapsed)

Can it work better with Iteration? ?

Are you trying to evaluate your solution? If you notice your code has similar lines repeated for multiple variables, you might want to improve your solution by generalizing it through iteration. For example:

```
var yourListName = ["a", "b", "c"];
for (var x in yourListName) {
  yourListName[x] = x;
  addToPage(yourListName[x]);
}
```

The for loop helps you generalize your solution for every element in `yourListName`. Try something like this:

[click to see less...](#)

1. Find code that you repeated multiple times.
2. Gather all the variables that you repeated the code for in a list.
3. Wrap the repeated code for one variable in a for loop.
4. Use the list from step 2 in the loop instead of one of the variables.

You might want to consider conditional statements to improve your iteration: ?

For more on for loops using lists, click the icon: ?

For more on lists, click the icon: ?

Evaluation of Implementation: Can it work better with Iteration? (Expanded)

