

A Principled Evaluation for a Principled Idea Garden

Will Jernigan¹, Amber Horvath¹, Michael Lee², Margaret Burnett¹, Taylor Cui¹,
Sandeep Kuttal¹, Anicia Peters¹, Irwin Kwan¹, Faezeh Bahmani¹, Amy J. Ko²

¹School of EECS, Oregon State University
Corvallis, Oregon, USA

²Information School, University of Washington
Seattle, Washington, USA

{jernigaw,horvatha,burnett,cui,tkuttals,peterani}@eecs.oregonstate.edu

{mjslee,ajko}@uw.edu

Abstract—Many systems are designed to help novices who want to learn programming, but few support those who are *not* interested in learning (more) programming. This paper targets the subset of end-user programmers (EUPs) in this category. 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.

Keywords—End-user programming; problem solving

I. 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 [30]. In this paper, 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 [10] 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 [6, 7, 8, 9]. 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 paper asks a principled “why”. What are the essential characteristics of systems like the Idea Garden? 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 Gadget EUP environment [26]. We then empirically investigate in two studies, 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”?*

II. 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) [10, 11]. 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” [10]. 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 [33]) to surprise EUPs as a curiosity-based enticement. To encourage learning while acting, the Idea Garden draws from constructivist theories surveyed in [4] 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 [35].

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 [15, 19, 20, 21, 34]. Stencils [22] presents translucent guides with tutorials to teach programming skills.

This work supported in part by NSF CNS-1240786, CNS-1240957, CNS-1339131, CCF-0952733, IIS-1314399, IIS-1314384, IIS-1528061, and OISE-1210205. Any opinions, findings, conclusions or recommendations are those of the authors and do not necessarily reflect the views of NSF.

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 programmers, 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 [29]. 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 [32]. Other programming environments such as Alice [21] 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., [13]). Some such environments (e.g., CoScripter/Koala [27]) 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 [1, 12].

Another way to reduce the amount of programming needed is by connecting the user with *examples* they can reuse as is. For example, FireCrystal [31] 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 [3], 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., [18]) 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. (Designers are somewhat related to EUPs in their frequent lack of experience or interest in programming per se.) For example, Bricolage [24] 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 [14]. That line of work partially inspired our research on helping EUPs generate new ideas in solving their programming problems.

III. THE IDEA GARDEN PRINCIPLES

Using MLT as a foundation, an earlier version of the Idea Garden was defined by Cao et al. as [6]:

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

(*Theory*) follow the principles from MLT [11] 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 paper 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 [25], 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 tend to gather information selectively [28], 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 [33], 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 I shows, *P4-Personality* and *P7-InterruptionStyle* have already been isolated for summative investigation in other end-user programming research [25, 33]. Thus, in this paper, we present our investigation of P1, P2, P3, P5, and P6.

IV. THE PRINCIPLES CONCRETELY: IDEA GARDEN PROTOTYPE

A. 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

TABLE I. CITATIONS SHOW EMPIRICAL EVIDENCE OF THE PRINCIPLES. AN UPDATED VERSION WITH THE CONTRIBUTIONS OF THIS PAPER APPEARS NEAR THE END. +: PRINCIPLE WAS HELPFUL, -: PRINCIPLE WAS PROBLEMATIC.

Principle	Formative Evidence	Summative Evidence
P1-Content		
P2-Relevance	-[9]	
P3-Actionable		
P4-Personality		+[25]
P5-InformationProcessing	+[8]	
P6-Availability		
P7-InterruptionStyle		+[33]



Fig. 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.)

puzzle game that centers on debugging (Fig. 1). Gidget has been used successfully by middle- and high-school teens [26] and by adults between the ages of 18 and 66 years old [25]. 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.

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.

B. 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 inline 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 Fig. 2).

P1-Content: The *Concept* portion is in the middle of Fig. 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 [9] 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 (Fig. 2), fulfilling P2.MyCode and P2.MyState. The antipatterns, explained in the next subsection, are what make these inclusions viable.

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

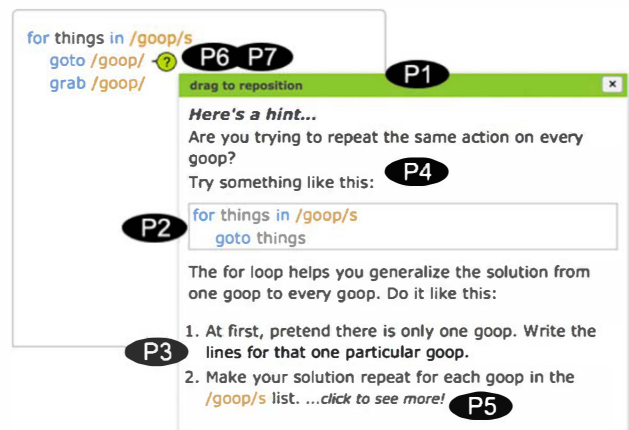


Fig. 2. Hovering over a 🔍 shows a hint. The superimposed Ps show where the 7 principles are instantiated in this hint.

in Fig. 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 (Fig. 2, P7) or within one of Gidget’s tooltips (Fig. 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 Fig. 1.

C. 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). For example: (1) *no-iterator*: not using an iterator variable within the body of a loop; (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; and (3) *partial implementation*: defining a function without calling it, calling an undefined function, or instantiating an undefined object.

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 (Fig. 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.

V. STUDY #1: PRINCIPLED FORMATIVE STUDY

Prior to implementing the 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.

The data for this study came from think-aloud data we obtained from a previous study [26]. The previous 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 II. The previous study’s objective was to investigate Gidget barriers

and successes [26]. Here we analyze that study’s 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.

Although the Idea Garden was not yet present, some UI elements in Gidget were consistent with some Idea Garden principles (Table III’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.

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

However, as Table III also shows, each principle helped with different barriers (defined in Table II). For example,

TABLE II. STUDY #1 AND #2 BARRIER CODES AND OUTCOME CODES.

Algorithm Design Barrier Codes [9, 26]	
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 [23, 26]	
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 Outcome 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.

TABLE III. STUDY #1: NUMBER OF INSTANCES IN WHICH PARTICIPANTS MADE PROGRESS FOR PRINCIPLES P2 AND P6. (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)

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 Section IV and conducted Study #2 to evaluate the results.

VI. 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 grad-

uate 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 II). 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 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.

VII. STUDY #2 RESULTS

A. 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 IV). Doing so enabled them to problem-solve their way past 77 of these 149 barriers (52%) without any guidance from the helper staff (Table V).



In fact, as Table V 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 IV 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.

B. 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 IV 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 VI enumerates the five ways the teams responded to the context-sensitive  (i.e., those triggered by the mini-antipatterns). The first way was the “ideal” way that we had


TABLE IV. 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%
P6-Available	Context Sensitive	6/19 32%	22/37 59%	10/14 71%	1/3 33%	9/21 43%	48/95 51%
	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 V. 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 VI: 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%

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 [9]. 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 VI), 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  as “to-do” list items. For example, Team Mouse, when asked about the  in the code in Fig. 3, 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 [16].

C. Teams' Behaviors with P3-Actionable

The two types of actionability that P3 includes, namely P3.ExplicitlyActionable (step-by-step actions as per Fig. 2's P3) and P3.ImplicitlyActionable (mental, e.g. “refer back...”)



Fig. 3. (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.

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 [2]. 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 [2].

D. 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 [17, 28]. 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.

VIII. 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. Fig. 4 shows part of one such level.

The teams created between 1 to 12 levels each (median: 6.5). As Fig. 4 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 help-

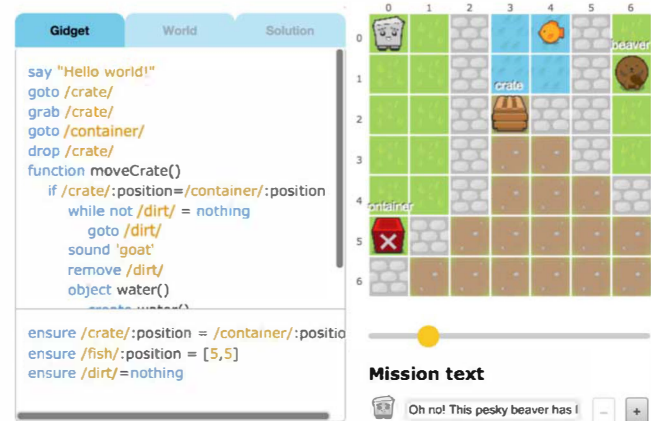


Fig. 4. Team Tiger's "River Dam" level's functions, conditionals, and loops.

ers 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 [26] (Table VII).

As Table VII 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 [26] 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 conserva-

TABLE VII. 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%
[26] camps	100%	94%	35%	47%	41%	76%	66%

TABLE VIII. INSTANCES OF BARRIERS AND % OF TOTAL 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 [26]:	437	56
Barriers (progress not available)	89%	11%

TABLE IX. SUMMARY OF PRINCIPLE-BY-PRINCIPLE EVALUATIONS. +: PRINCIPLE WAS HELPFUL, -: PRINCIPLE WAS PROBLEMATIC. *: TEAMS PROGRESSED IN THE MAJORITY (>=50%) OF THEIR BARRIERS WITH THESE IDEA GARDEN PRINCIPLES.

Principle	Ways	Formative Evidence	Summative Evidence
P1-Content		+Study1	+Study2
	P2-All	-[9]	
P2-Relevance	P2.1-MyCode		+Study2*
	P2.2-MyState	+Study1	+Study2*
	P2.3-MyRequirements	+Study1	
P3. Actionable	P3.1-ExplicitActionable		+Study2*
	P3.2-ImplicitActionable		+Study2*
P4-Personality			+ [25]
P5- InformProc		+ [8]	+Study2*
P6-A availability	P6.1-ContextFree	+,-Study1	+Study2*
	P6.2-ContextSensitive	+Study1	+Study2*
P7-Interrupt			+ [33]


tive 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 VIII. 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 VII), with significantly less need for in-person help (Table VIII, Fisher’s exact test, p=.0001).

IX. CONCLUDING REMARKS

Table IX summarizes Study 2’s evidence of each principle and its component parts. One way to view these results 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; mini-patterns 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*.

Finally, 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 [26]. However, unlike the prior study, they accomplished these gains with significantly less in-person help than in the previous study. These promising results suggest the effectiveness of the principles we have presented here in helping EUPs solve the programming problems that get them “stuck”—across a diversity of problems, information processing and problem-solving styles, cognitive stages, and people.

REFERENCES

- [1] Andersen, R. and Morch, A. Mutual development: A case study in customer-initiated software product development. *End-User Development*, (2009), 31-49.
- [2] 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)
- [3] 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.
- [4] Bransford, J., Brown, A., Cocking, R. (Eds), *How People Learn: Brain, Mind, Experience, and School*, National Academy Press, 1999.
- [5] Burnett, M., Beckwith, L., Wiedenbeck, S., Fleming, S., Cao, J., Park, T., Grigoreanu, V., Rector, K. Gender pluralism in problem-solving software. *Interact. Comput.* 23 (2011), 450-460.
- [6] 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)
- [7] 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.
- [8] 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.
- [9] 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.
- [10] Carroll, J. and Rosson, M. The paradox of the active user. *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*, MIT Press. 1987.
- [11] Carroll, J. *The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill*. 1990.
- [12] 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.
- [13] Cypher, A., Nichols, J., Dontcheva, M., and Lau, T. *No Code Required: Giving Users Tools To Transform the Web*, Morgan Kaufmann. 2010.
- [14] 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.
- [15] Dorn, B. ScriptABLE: Supporting informal learning with cases, In *Proc. ICER*, ACM, 2011. 69-76.
- [16] Grigoreanu, V., Burnett, M., Robertson, G. A strategy-centric approach to the design of end-user debugging tools. *ACM CHI*, (2010), 713-722.
- [17] 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.
- [18] 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.
- [19] Guzdial, M. Education: Paving the way for computational thinking. *Comm. ACM* 51, 8 (2008), 25-27.
- [20] 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.
- [21] 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.
- [22] Kelleher, C. and Pausch, R. Stencils-based tutorials: design and evaluation. *ACM CHI*, 2005, 541-550.
- [23] Ko, A., Myers, B., and Aung, H.. Six learning barriers in end-user programming systems. *IEEE VLHCC* 2004, 199-206.
- [24] Kumar, R., Talton, J., Ahmad, S., and Klemmer, S. Bricolage: Example-based retargeting for web design. *ACM CHI*, 2011. 2197-2206.
- [25] Lee, M. and Ko, A. Personifying programming tool feedback improves novice programmers' learning. In *Proc. ICER*, ACM Press (2011), 109-116.
- [26] 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.
- [27] 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.
- [28] 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.
- [29] Myers, B., Pane, J. and Ko, A. Natural programming languages and environments. *Comm. ACM* 47, 9 (2004), 47-52.
- [30] Nardi, B. *A Small Matter of Programming*, MIT Press (1993).
- [31] Oney, S. and Myers, B. FireCrystal: Understanding interactive behaviors in dynamic web pages. *IEEE VL/HCC* (2009), 105-108.
- [32] Pane, J. and Myers, B. More natural programming languages and environments. In *Proc. End User Development*, Springer (2006), 31-50.
- [33] 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.
- [34] 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.
- [35] Turkle, S. and Papert, S. Epistemological Pluralism. *Signs* 16(1), 1990.