Journal of Visual Languages and Computing 39 (2017) 51–65

Contents lists available at ScienceDirect

Journal of Visual Languages and Computing

journal homepage: www.elsevier.com/locate/jvlc

General principles for a Generalized Idea Garden

William Jernigan[a,*], Amber Horvath[a], Michael Lee[b], Margaret Burnett[a], Taylor Cuilty[a], Sandeep Kuttal[a], Anicia Peters[a], Irwin Kwan[a], Faezeh Bahmani[a], Amy J. Ko[c], Christopher J. Mendez[a], Alannah Oleson[a]

[a] School of EECS, Oregon State University, Corvallis, OR, USA
[b] Information Systems, New Jersey Institute of Technology, Newark, NJ, USA
[c] Information School, University of Washington, Seattle, WA, USA

ABSTRACT

Many systems are designed to help novices who want to learn programming, but few support those who are *not necessarily* interested in learning 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 them in three studies: a formative think-aloud study, a pair of summer camps attended by 42 teens, and a third summer camp study featuring a different environment attended by 48 teens. Finally, we present a generalized architecture to facilitate the inclusion of Idea Gardens into other systems, illustrating with examples from Idea Garden prototypes. Results have been very encouraging. For example, under our principles, Study #2's 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.

1. 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 [30,43]. 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 and learn just enough programming to get their tasks done.

We can describe these kinds of EUPs as being "indifferent" to learning programming (abbreviated "indifferent EUPs"), a subset of Minimalist Learning Theory's notion of "active users" [13]. Minimalist Learning Theory's active users are those 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 they are using and its features. According to the theory, active users such as our indifferent EUPs are willing to learn and do programming only if they expect it to help them complete their task.

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, 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 [8–12]. 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. Building upon prior studies that showed that early versions of the Idea Garden helped indifferent EUPs become unstuck [10–12], this paper investigates exactly *why* the Idea Garden helped them and *how* the underlying principles and structure of the Idea Garden contributed to that success.

Toward that end, this paper's first research contribution lies in asking a principled "why?": Why is the Idea Garden helpful to indifferent EUPs, and 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 both an Idea Garden prototype that sits on top of the Gidget EUP environment [33] and another prototype that extends the Cloud9 IDE environment [39]. We then empirically investigate in three studies, principle by principle, the following research question: *How do these principles influence the ways*

* Corresponding author.
E-mail addresses: jernigaw@eecs.oregonstate.edu (W. Jernigan), horvatha@eecs.oregonstate.edu (A. Horvath), mjlee@njit.edu (M. Lee), burnett@eecs.oregonstate.edu (M. Burnett), cuiltyt@eecs.oregonstate.edu (T. Cuilty), kuttals@eecs.oregonstate.edu (S. Kuttal), peterani@eecs.oregonstate.edu (A. Peters), ajko@uw.edu (A. Ko), mendezc@eecs.oregonstate.edu (C.J. Mendez), olesona@eecs.oregonstate.edu (A. Oleson).

http://dx.doi.org/10.1016/j.jvlc.2017.04.005
Received 30 December 2015; Received in revised form 16 November 2016; Accepted 10 April 2017
Available online 12 May 2017
1045-926X/

indifferent EUPs can solve the programming problems that get them "stuck"?

Our second contribution is generalization of the Idea Garden. Prior work on the Idea Garden [10–12] was in a single language environment (CoScripter). In this work, we use the new principles of the first contribution to build Idea Gardens in two additional languages and environments: Gidget, with its own imperative language, and Cloud9, with JavaScript. We also present a generalized architecture for the Idea Garden to enable others to instantiate Idea Garden systems and its principles in their own programming environments. This paper presents the architecture itself, how it was used to create multiple Idea Gardens, the motivations behind pieces of the architecture, and how it enables the seven Idea Garden principles. We illustrate with examples from the Gidget and Cloud9 Idea Garden prototypes.

2. Background and related work

One of the most relevant foundational bases for the Idea Garden's target population is Minimalist Learning Theory (MLT) [13,14]. MLT was designed to provide guidance on how to teach users who (mostly) do not want to be taught. More specifically, MLT's users are motivated 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" [13]. MLT aims to help those who face this paradox to learn—*despite* their indifference to learning.

Prior work has explored many ways of helping programmers by increasing access to information that may help a programmer find a solution to a problem. For example, systems have created stronger links to formal documentation (e.g., [47], used social question and answer sites to fill gaps in documentation (e.g., [40]), or brought relevant content from the web into the programming environment (e.g., [4]). Unlike these systems, which try to bring correct information to programmers, the Idea Garden does not try to give users complete or even entirely correct answers; rather, it tries to give users information about similar problems that may help them identify new approaches to solving their own problem.

Another class of prior work aims to explicitly teach problem-solving rather than informally suggest problem-solving strategies (as the Idea Garden does). For example, intelligent tutoring systems have long been studied in domains such as mathematics, physics, statistics, and even writing, finding that by breaking down problems into procedural steps, teaching these steps, and providing feedback when learners deviate from these steps, computer-based tutors can be as effective as human tutors [31,50]. Most of this work has not investigated the teaching of programming problem-solving, although there are some exceptions. The LISP tutor built models of the program solution space and monitored learners' traversals through this space, intervening with corrective feedback if learners encountered error states or made mistakes the tutor had previously observed [3]. Other more recent efforts to teach problem-solving to programmers have found that having teachers prompt novices about the strategies they are using and whether those strategies are appropriate and effective can greatly improve novices' abilities to problem-solve independently [39]. The Idea Garden also tries to increase users' awareness of their own and other possible problem-solving strategies, but as is needed in ordinary programming situations.

There is also research aimed specifically at populations of novice programmers who want to learn programming, characterized by new kinds of educational approaches or education-focused languages and tools [19,23,24,28,48]. For example, Stencils [27] presents translucent guides with tutorials to teach programming skills. While Stencils uses overlays to show users the only possible interface interactions and explains them with informative sticky notes, 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 and learn only enough programming to complete some *other* task.

EUP systems targeting novices who do not aspire to become professional programmers commonly attempt 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 [42]. 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 that are expressed in a language designed to match how children described games [45]. Other programming environments such as Alice [28] 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 the language/environment of its host.

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., [17]). Some of these types of environments (e.g., CoScripter/Koala) [38] 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 [1,16].

Another way to reduce the amount of programming needed by EUPs is to connect them with *examples* they can reuse as-is. For example, tools such as FireCrystal [44] and Scry [7] allow programmers to select user interface elements of a webpage and view the corresponding source code that implements it. Other systems are designed to simplify the task of choosing which existing programs to run or reuse (e.g., [22]) 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 aim to help EUPs *figure out for themselves* how to surmount those barriers. However, there is some work aimed at helping professional interface designers generate and develop ideas for their interface designs. For example, bricolage [32] allows designers to retarget design ideas by transferring designs and content between webpages, 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 [18]. This line of work partially inspired our research on helping EUPs generate new ideas in solving their programming problems.

3. The principles

This section presents seven principles that ground the content and presentation of the Idea Garden. It also presents the works that influenced the development of each principle.

Most of the principles used to create the Idea Garden draw from MLT's approach to serve active users. For example, P1-Content provides content that relates to what the active user is *already doing*; P2-Relevance shapes content for the active user in such a way that they feel it is *relevant to the task at hand*, to encourage the user to pick up just the content they need, just in time; P3-Actionable gives active users something to *do* with the information they have just collected; and P6-Relevance provides content to users within the context in which they are working so that they can keep their *focus on getting their task done* rather than searching for solutions from external sources.

The Idea Garden also draws foundations from the psychology of

Table 1
The seven Idea Garden principles and their explanations. The middle column contains the prior work that helped inform each Idea Garden principle. A hyphenated name signifies a principle (e.g., P1-Content), while a name with a dot signifies a subprinciple (e.g., P1. Concepts).

Principle	Based off work involving...	Explanation
P1-Content	MLT; constructivist learning	Content that makes up the hints need to contain at least one 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.Mini-patterns		*Design mini-patterns* 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	MLT; special-purpose programming languages and systems	For Idea Garden hints that are context-sensitive, the aim is that the user perceives them to be relevant. Thus, 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/post-conditions.
P3-Actionable	MLT [13,14]; constructivist learning; bricoleurs; special-purpose programming languages and systems	Because the Idea Garden targets MLT's "active users," hints must give them something to *do* [13,14]. 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. ExplictlyActionable		The hint prescribes actions that can be physically done, such as indenting or typing something.
P3. ImplicitlyActionable		The hint prescribes actions that are "in the head," such as "compare" or "recall".
P4-Personality	Constructivist learning; [34]	The personality and tone of Idea Garden entries must try to encourage constructive thinking. Toward this end, hints are expressed non-authoritatively and tentatively [34]. 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	Constructivist learning; [41]; special-purpose programming languages and systems	Because research has shown that (statistically) females tend to gather information comprehensively when problem-solving, whereas males gather information selectively [41], 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	MLT; Surprise-Explain-Reward [46]	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	Surprise-Explain-Reward [46]	Because research has shown the superiority of the negotiated style of interruptions in debugging situations [46], 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.

curiosity and constructivist learning. To deliver content to indifferent EUPs, the Idea Garden uses Surprise-Explain-Reward [46,52] to carefully surprise EUPs with a curiosity-based enticement that leads to constructivist-oriented explanations. This strategy informed our principles P6-Availability and P7-InterruptionStyle. To encourage learning while acting, the Idea Garden draws from constructivist theories surveyed in [5] to keep users active (informing P3-Actionable), make explanations not overly directive (P4-Personality), and motivate users to draw upon their prior knowledge (P1-Content and P5-InformationProcessing). 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 [49], encouraging users to act through P3-Actionable.

Table 1 provides a complete list each of the principles' foundations.

4. Study #1: principled formative study

During the development of our first Idea Garden in the programming-by-demonstration environment CoScripter, we hypothesized that some of above principles were key to the Idea Garden's success, but did not formally use them as a guide to implementation [12]. Therefore, as part of the process of generalizing with a second version of the Idea Garden, this time for the Gidget environment, we made explicit the above seven principles in the implementation of the system so that we could use them to construct the Idea Garden for Gidget.

To inform this work, prior to actually implementing the Idea Garden principles in the Gidget prototype, we conducted a small formative study we call Study #1. Our goal was to gather evidence about our proposed principles so that we could make an informed decision about which ones to focus on evaluating in a larger study we call Study #2 (presented in Section 6).

In the Gidget game (Fig. 1), 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 [36].

For Study #1, we reanalyzed think-aloud data that was presented in [37]. This study had 10 participants (5 females, 5 males) who were 18–19 years old, each with little to no programming experience. Each session was 2 h in length and fully video-recorded. The experimenter helped participants when they were stuck for more than 3 min. We re-analyzed the video recordings from this study using the code sets in Table 2. The objective of the previously reported study [37] was to investigate barriers and successes of Gidget players. 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.

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 user interactions with interface

Fig. 1. The Gidget puzzle game environment with superimposed callouts for readability. Dictionary entries appear in tooltips when players hover over keywords ("for" shown here). Hovering over an idea indicator (?) reveals an Idea Garden hint.

elements.

The Gidget UI elements' connections to Idea Garden principles primarily related to P2-Relevance and P6-Availability. Table 3 shows that when these principles were present, participants tended to make progress—usually without needing any help from the experimenter.

However, as Table 3 also shows, each principle helped with different barriers (defined in Table 2). For example, P2.MyGoal stood out in helping participants with Design barriers (i.e. did not know what they wanted Gidget to do). On the other hand, P6.ContextSensitive was strong with Coordination (knew what concepts to use, but not how to use them together), Composition (did not know how to combine functionalities of existing commands), and Selection (knew what they wanted Gidget to do, but not what to use to get that to happen) barriers.

These results revealed three useful insights for Study #2's principled evaluation and implementation of the Idea Garden prototype within Gidget: (1) We decided to evaluate our principles using a barrier-centric perspective based on our findings of complementary roles of different principles for different sections in "barrier space." (2) We designed several hints (described in Section 5) so that relevant ones would appear in context for the appropriate anti-patterns based on our promising results for P2-Relevance and P6. ContextSensitive. (3) We concentrated on creating hints for the concepts (P1.Concepts) that participants struggled with the most, since these were the ones users were most likely to encounter.

5. The principles concretely: The Idea Garden for Gidget

The Idea Garden works with a host EUP environment, and this paper shows the Idea Garden in two different hosts: Gidget, which we used in Study #2 (Fig. 1), and the Cloud9 browser-based IDE, which we used in Study #3 (Section 7). We begin with the Gidget-hosted Idea Garden prototype, to concretely illustrate ways in which the Idea Garden principles can be instantiated.

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

Algorithm Design Barrier Codes [10,37]	
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 [29,37]	
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, bud 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

Table 3
Study #1: Number of instances in which participants made progress for principles P2-Relevance and P6-Availability, as per the code definitions of Table 2. Maximum values are highlighted. When contributions of a principle to a barrier were much smaller than to the other barriers, we refer to that barrier(s) in the last column as "minor".

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)

*+: progress with no in-person help.
+☺: progress with help from experimenter.
-: no progress.

5.1. The Idea Garden Prototype for Gidget

Gidget has been used successfully by a wide range of age groups [34,35,37]. Indifferent EUPs are among the game's players: some users want to learn just enough programming to beat a level and no more. This makes the environment an ideal candidate for the Idea Garden.

The Idea Garden prototype for Gidget aims to help players who are unable to make progress even *after* they have used the host's existing forms of assistance. Before we added the Idea Garden to the game, 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 code keywords. The Idea Garden supplements these kinds of help by instantiating the seven principles as follows (illustrated in Fig. 2).

P1-Content: The content that users struggle with is presented in this prototype and derived from Study #1. The *Concept* portion is in the middle of Fig. 2 (i.e., the concept of iteration), the *Mini-pattern* is shown via the code example, and the *Strategy* portion is the numbered set of steps at the bottom.

P2-Relevance: Prior empirical studies [10] 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 (the "gist" of the hint), and then includes some of the user's own code and/or variable names (Fig. 2), fulfilling P2.MyCode and

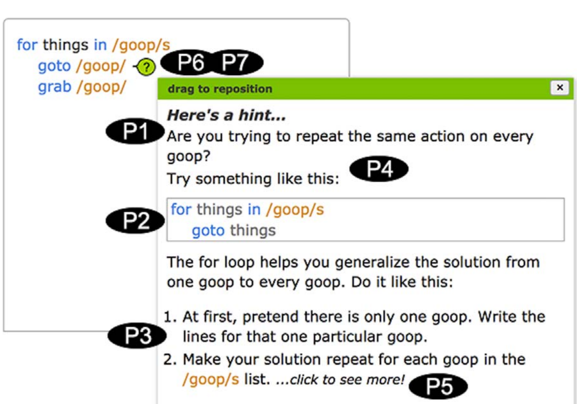

Fig. 2. Hovering over a ? shows a hint. The superimposed P's show how each of the 7 principles are instianted in this hint.

P2.MyState. The anti-patterns, 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 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 both comprehensive and selective information processing styles (P5).

P6-Availability and 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.

5.2. Anti-pattern support for the principles

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

Our prototype detects several anti-patterns that imply *conceptual* programming problems (as opposed to syntactical errors). When selecting which ones to support in this prototype, we chose anti-patterns that occurred in prior empirical data about Gidget at least three times (i.e., by at least three separate users). The following is a description of each programming anti-pattern 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 a *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 anti-patterns enables support for three of the Idea Garden principles. The anti-patterns define context (for P6.ContextSensitive), enabling the system to both derive and show a hint within the context of the problem and to decorate the screen with the ? symbol (P7-Interruption Style). For P2-Relevance, the hint communicates relevance (to the user's current problem) by generating itself based on the player's current code as they type it; this includes using players' own variable names within the hints. Fig. 2, which is the hint constructed in response to the no-iterator anti-pattern, illustrates these points. Fig. 3 shows additional examples of hints constructed in

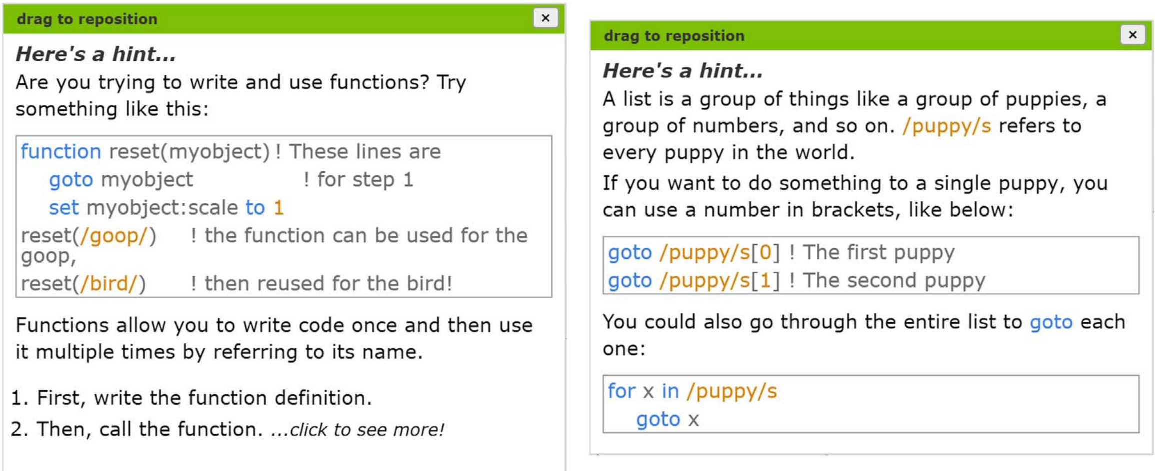

Fig. 3. Idea Garden hints in Gidget for (left) the two *function* anti-patterns and (right) for the *all-at-once* anti-pattern.

response to the above anti-patterns.

6. Study #2: the principles go to camp

Using the prototype discussed in the previous section, we conducted Study #2 (a summative study) to evaluate the usefulness of the Idea Garden principles to indifferent EUP teens. Our overall research question was: *How do the principles influence the ways indifferent EUPs can solve the programming problems that get them "stuck"?*

6.1. Study #2 methods

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 two summer camps took place on college campuses in Oregon and Washington. Each camp ran 3 h per day for 5 days, for 15 h total. Participants used desktop computers to play the game. Campers spent 5 h each in: Gidget puzzle play; in other activities such as icebreakers, guest speakers, and breaks; and in 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 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. One participant in the Washington camp opted out of our data collection for privacy reasons, so her team was excluded from analyses.

Recall that 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, with transportation provided. Consistent with this, in a survey of a 2013 camp recruited the same way, 25 of the 34 teens (74%) self-reported low confidence using computers and/or did not see themselves pursuing computing careers.

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, including code versions, Idea Garden ?s, hint interaction, and code execution. 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 the provided assistance (if any) 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 2). 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 anti-pattern-triggered Idea Garden hint marked by a ?. We considered these 39 instances evidence of "self-proclaimed" barriers, except if they were viewed by a team within 2 min of a visit from a camp helper (who may have pointed them to the hint). If teams somehow removed the fault, we coded the instance in two phases: for the barriers in Table 2 and the same barrier endings as for the observations. Two coders reached 80% and 93% agreement on 20% of the data sets 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.

We merged these sets of barriers with the Idea Garden hints that were involved in each and considered the principles involved in each hint. The results of this coding and analysis are presented next.

6.2. Study #2 results

This section discusses what Study #2 revealed about the principles of the Idea Garden. We did not explicitly investigate principles P4-Personality and P7-InterruptionStyle in Study #2, since each was investigated in prior work. However, both were found to be beneficial to EUPs in different ways: P4-Personality contributed to programming successes by helping users of an early Gidget game complete significantly more levels in the same amount of time than users without such a "personable" system [34]; and P7-InterruptionStyle's negotiated interruptions were shown to help EUPs debug programs more effectively [46].

Table 4
Percents of barrier instances in which progress occurred, categorized by barrier type, when Idea Garden principles P2, P3, and/or P6 were present. (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. Highlights point out cells discussed in the text.

		Barriers					
		Selection	Use	Coordi-nation	Under-standing	More Than Once	Total
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 5
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 anti-pattern-triggered was on the screen (top row). Maximums in each row are highlighted. (The 25 instances where teams did not make progress are not included in this table).

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%)

Teams did not always need the Idea Garden; they solved 53/64 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 4). Doing so enabled them to problem-solve their way past 77 of these 149 barriers (52%) without any guidance from the helper staff (Table 5).

In fact, as Table 5 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' barrier success rate with in-person help alone (59%) was only slightly higher than with the Idea Garden alone (52%).

Table 4 also breaks out the teams' success rates principle by principle (rows). Campers overcame 50% or more of their barriers when each of the reviewed principles was involved, showing they each made a contribution to campers' success. No particular difference in success rates with one principle versus another stands out in isolation,, likely due to the fact that the prototype uses most of them most of the time. However, viewing the table column-wise yields two particularly interesting barriers.

First, Selection barriers (where the camper knows what they want to do, but not what to use to do it; first column) were the most resistant to the principles. This surfaces a gap in the current Idea Garden version: A Selection barrier happens *before* use as the user tries to decide what to use, whereas the Idea Garden usually became active *after* a camper attempted to use some construct in code.

Second, Coordination barriers (where the camper knows what things to use, but not how to use them together; 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 mini-patterns (present in every hint), which show explicitly how to incorporate and coordinate combinations of program elements. For example, in Fig. 2, the Gidget iteration hint, the mini-pattern code example shows how to use iteration together with Gidget's *goto* keyword. This kind of concrete example, combined with the other subprinciples of P1 (specifically P1.Concepts and P1.Strategies, investigated more deeply in [9,10], and also Study #3 in this paper) may have contributed to campers' high success rates

Table 6
A summary of what Study #2 revealed about each of the principles.

Principle	Results from Study #2
P1-Content	● Coordination barriers showed the highest progress rates for all of the Idea Garden principles, but its success may be particularly due to P1's mini-patterns, which explicitly show how to *coordinate* combinations of program elements
P2-Relevance	● When campers picked up on the relevance of hints, they made progress the majority of the time. Still, it can be tricky to convey relevance to indifferent EUPs ● Hints should also attempt to convey *solution* relevance, not just *problem* relevance
P3-Actionable	● Explicitly actionable hints gave campers a single new action recipe to *try* ● Implicity actionable hints gave campers options on ways to *generate* multiple new action recipies on their own ● Hints helped campers *apply* new knowledge and *analyze* differences in action recipes, i.e., helped them at multiple stages of Bloom's taxonomy
P4-Personality	*N/A (already investigated in* [34]
P5-Information Processing	● Females tended to use comprehensive processing, whereas males tended to use selective processing (consistent with [21,41,6] ● Since the Idea Garden supports both information processing styles, campers were able to use whichever fit their problem-solving style, contributing to a more inclusive environment
P6-Availability	● Campers accessed context-sensitive hints about 5x more than context-free hints ● But campers sometimes revisited the context-free hints later, after the context had changed, to get a reminder of the hint's suggestions
P7-InterruptionStyle	*N/A (already investigated in* [46]

when overcoming coordination barriers.

Study #2 revealed a good deal of information about the principles and how campers could leverage them to help themselves progress through their problems. In Table 6, we list some of the results from Study #2.

Taken together, these results suggest that the principles of the Idea Garden can contribute to indifferent EUPs' successes across many diverse use cases and situations. P2-Relevance and P6-Availability worked together in a number of ways to provide campers with relevant hints that could be accessed in or out of context, supporting a wide variety of problem-solving styles. The complimentary roles of P3. ExplicitlyActionable and P3.ImplicitlyActionable helped campers either *apply* or *analyze* their knowledge at different stages in Bloom's taxonomy, leading to progress on problems. Finally, P5-InformationProcessing allowed teams to gather information in which-ever way fit their problem to them, promoting gender-inclusiveness by supporting different information processing styles. Below, we illustrate some examples that show the many ways campers used these principles.

6.2.1. Teams' behaviors with P2-relevance and P6-availability

In this section, we narrow our focus to observations of team's reactions to the (?) in relation to P2 and P6. We consider P2 and P6 together because the prototype supported P2-Relevance in a context-sensitive (P6) manner.

Teams appeared to be enticed by the context-sensitive hints. As the P6-Availability row in Table 4 shows, teams accessed context-sensitive hints about five times as often as the context-free 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. Therefore, these findings support the use of both context-sensitive *and* context-free availability of the Idea Garden hints.

Table 7 enumerates the five ways teams responded to the context-sensitive (?)s (i.e., those triggered by the mini-anti-patterns). 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

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%	

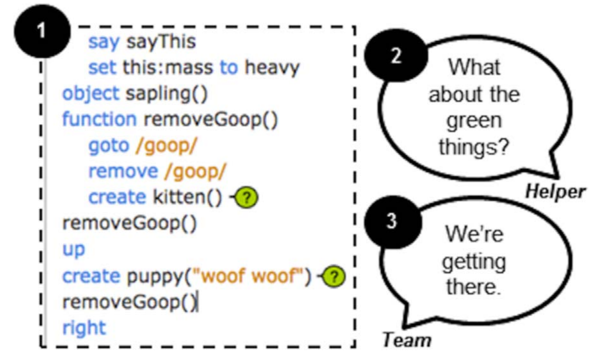

Fig. 4. (1) Team Mouse spent time working on code above the (?)s. When (2) a helper asked them about the (?)s in their code, they indicated (3) that the (?)s were action items to do later . Seven other teams also used this method.

research has reported challenges in convincing users of relevance [10]. 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 (40%), 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 Fig. 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 [20]. Many of the observations involving this technique did not include any indications of the teams being stuck.

In summary, campers used a variety of interaction styles when confronted with an Idea Garden (?) icon. When campers picked up on the relevance (P2) of the hint in their current context (P6), they often made progress (e.g., Team Turtle's "read-and-act-upon" approach to the icon). Although it can be tricky to convey the relevance of hints to indifferent EUPs, guiding campers toward a solution direction helps convey that relevance and get EUPs unstuck.

6.2.2. 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...") instructions, helped the teams in 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).

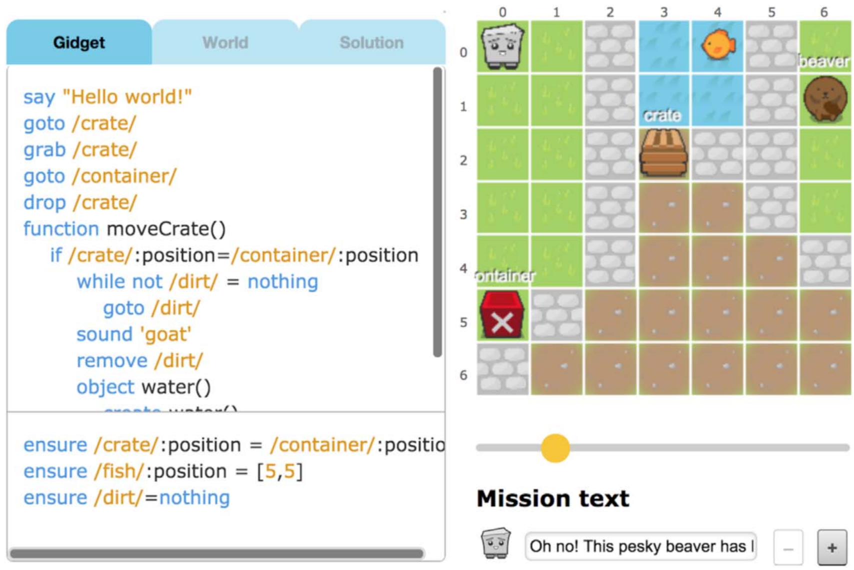

Fig. 5. Team Tiger's "River Dam" level with functions, conditionals, and loops.

> ***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 appear to have given teams *new options to consider*. In the following example, Team Owl ran out of ideas to try and did not know how to proceed. However, 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 observation #10, above) or to use as a checklist (Team Rabbit's observation #9, above). This behavior relates to the Bloom's taxonomy ability to *apply* learned material in new, concrete situations [2], where a person executes an idea (trying to use events). 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], where a person differentiates or organizes based on a certain idea (which whale to use).

6.2.3. Teams' Behaviors with P5-InformationProcessing

Recall that P5-InformationProcessing states that hints should support EUPs' information processing styles, 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, including the two above, were female. Their use of the comprehensive style is consistent with prior findings that females often use this style [21,41,6]. 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 teams' various working styles.

6.3. How much did they learn?

After about 5 h of debugging their way through the Gidget levels, teams reached the "level design" phase, where they were able to freely create their own levels. In contrast to the puzzle play activity where teams only fixed broken code to fulfill game goals, this "level design" portion of the camp required teams to author level goals, "world code," behavior of objects, and the starting code others would debug to pass the level. Fig. 4 shows part of one such level.

The teams created between 1 and 12 levels each (median: 6.5). As Fig. 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' uses of events were particularly telling. Although teams had seen Idea Garden hints for loops and functions throughout the

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. [37] camps	100%	94%	35%	47%	41%	76%	66%

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 explained 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 [37] (Table 8).

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 [37] 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 the second year only if no in-person help was involved, but give full credit to the Idea Garden the first year if one of our early Idea Garden sketches was used to supplement in-person helpers that year. Although this bias makes the Idea Garden improvement look lower than it should, it 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=0.0001).

7. Study #3: The Principles in Cloud9

To generalize our results so far, we chose a new host environment for our next Idea Garden prototype, namely Cloud9 [15]. As a web-based IDE, Cloud9 is a professional development environment, so this prototype of the Idea Garden needed to accommodate a much less restricted environment than Gidget.

Our target audience remained indifferent EUPs. Similarly to Gidget, in which some users want to learn just enough programming to beat a level, in Cloud9 some users want to learn just enough programming to personalize their website. We used the Cloud9 Idea Garden for two purposes in Study #3: both to evaluate the principles' generalizability to a different IDE and language, and to investigate the principles' generalizability to explicitly support problem-solving strategies (not just conceptual errors as in the Gidget-based prototype).

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
Second year's camps, Study #2 with Idea Garden: Barriers with progress	116 47%	130 53%
First year's camps [37]: Barriers (progress not available)	437 89%	56 11%

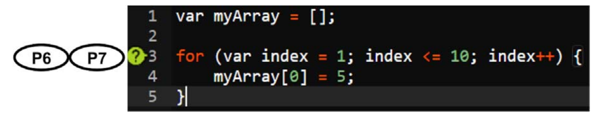

Fig. 6. An example of the Idea Garden decorating a Cloud9 user's code with a icon to indicate that it has detected an anti-pattern. Users can click on the icon to have the Idea Garden highlight the relevant hint.

7.1. The Idea Garden in Cloud9

As with the Gidget prototype, the seven principles informed the design and implementation of the Cloud9 prototype. Within those boundaries, we tailored the Idea Garden implementation to its Cloud9 host in several ways.

First, we housed Cloud9's 14 Idea Garden hints under headers in a side panel of the IDE, instead of tooltips. As in Gidget, if Cloud9 users triggered a programming anti-pattern when writing code, a decorated the screen next to the problematic code (Fig. 6) to support P6.ContextSensitive and P7-InterruptionStyle. Users could then click on the icon to have the Idea Garden highlight the titles of relevant hints in the side panel. For example, if a user wrote a *for* loop that contained a *no-iterator* anti-pattern (such as the one in Fig. 6), clicking the icon would highlight the title of the *Iteration with For* hint.

The appearances and structures of the hints were similar to those in Study #2's Idea Garden, in support of principles P1-Content, P3-Actionable, P4-Personality, and P5-InformationProcessing; Fig. 7 illustrates. (See [25] for a comprehensive comparison of the Idea Garden host-specific hints in Gidget vs. Cloud9.) The Idea Garden panel and all of its hints were always available in the IDE, to support P6-

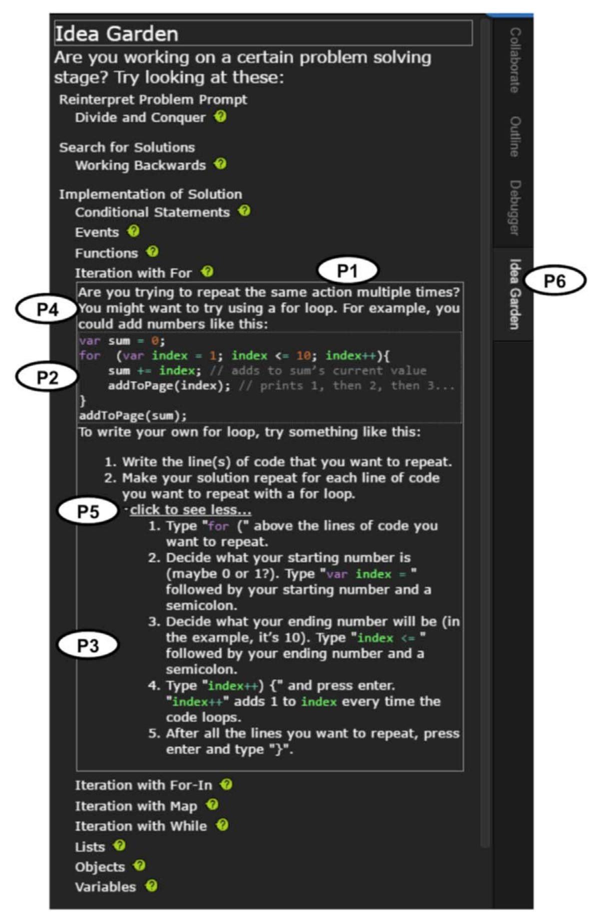

Fig. 7. The Idea Garden prototype instantiated in a side panel of the Cloud9 environment, with the *Iteration with For* hint expanded to show how different parts of the hint instantiated the principles.

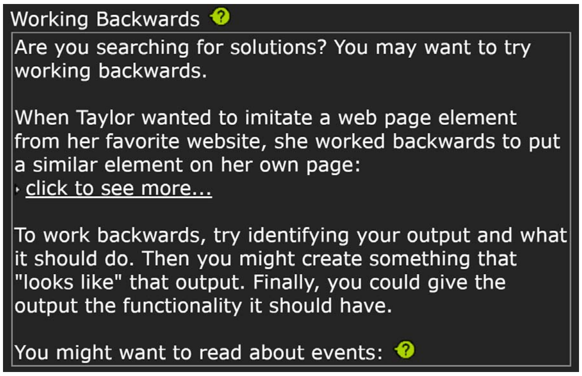

Fig. 8. The "Working Backwards" hint in the Cloud9 Idea Garden. This hint does not include a code example, and as such had to convey relevance through P2.MyState and P2.MyGoal, as seen in the "gist" of the hint ("Are you searching for solutions?").

ContextFree.

Because of this study's emphasis on problem-solving strategies, some hints in Cloud9 did not have Gidget counterparts. One example is the Working Backwards hint (Fig. 8), which supports the Working Backwards problem-solving strategy [51].

As Fig. 8 demonstrates, these problem-solving strategy hints did not include code examples, and this raised a challenge to the P2-Relevance principle: participants could not tell by looking at code (i.e., by using P2.MyCode) whether or not the hint was relevant. Thus, these problem-solving hints had to communicate relevance using only P2.MyState or P2.MyGoal. To resolve this, we designed the panel to follow the flow of the problem solving stages that had been explained to the campers, so that campers could determine relevance by the stage they were in, supporting P2.MyState. We also organized Cloud9 Idea Garden hints in the side panel so that campers could see and expand any hint they deemed relevant whenever the panel was open. This also supported P6.ContextFree: campers could have the Working Backwards hint onscreen while they wrote code and then, if they triggered an anti-pattern and opened another hint, they could still view the Working Backwards hint at the same time.

7.2. Study #3 methods, procedures, and participants

After porting the Idea Garden to Cloud9, we conducted Study #3, a two-week long day camp that taught 48 novice programmers web development [39]. Campers learned the basics of HTML, CSS, JavaScript, and a JavaScript library called React in order to make their own websites.

One aim of the camp was to empirically evaluate whether explicitly teaching problem solving to novice programmers could facilitate the development of programming skills.

Campers were divided into an experimental group (which attended the morning session) and a control group (which attended the afternoon session). Each session lasted 3 h, with a total of 10 sessions for each group. Both groups initially had 25 campers, but two students in the control group decided not to attend the camps, bringing the final count to 23 campers in the control group and 25 in the experimental group. Campers were not formally instructed to pair program or work in groups (unlike Study #2), but some still helped each other with programming tasks.

Both groups were identical in terms of the instructions given, prescribed programming tasks, and levels of assistance received from camp helpers. However, the experimental group received four interventions that the control group did not: (1) A short lecture about problem solving and metacognition in programming; (2) A set of prompts to be answered when campers asked helpers for assistance; (3) a physical, paper handout of a problem solving model; and (4) the Idea Garden prototype for Cloud9. As we explained above, the Idea Garden prototype supported anti-patterns observed in previous studies as well as problem-solving strategy hints consistent with interventions (1)–(3).

Throughout the camps, we measured the experimental group's Idea Garden usage through a Cloud9 event logging mechanism. We also collected data in both the control and experimental camps from camp helper observations and end-of-day surveys that campers filled out after each session, giving us multiple streams from which to triangulate results. These sources of data were combined with campers' code, which their workspace pushed to private BitBucket repositories every half hour during camp sessions, giving an impression of Idea Garden usage in Cloud9.

7.3. Study #3 results

Full results of this study are described in [39]. To summarize, the campers in the experimental group completed more self-initiated web development tasks (i.e., tasks that were not prescribed as part of camp instruction) than the control group. In addition, the experimental group did *not* have a significant association between in-person help requests and productivity (Pearson: $r(23)=0.278$, $p=0.179$), whereas the control group *did* have a significant association between the two (Pearson: $r(21)=0.467$, $p=0.025$). This suggests that the control group's productivity was significantly tied to help requests, whereas the experimental group was productive even *without* significant in-person help.

In this paper, we focus on what Study #3 revealed about the Idea Garden's principles and how well they generalized to Study 3's environment and goals. However, because the Idea Garden in Study #3 was a single element in a set of interventions, we cannot isolate its exact quantitative contributions to the results in [39]. We can, however, provide qualitative examples of the ways in which campers interacted with the Idea Garden though a principled lens and look for consistency or inconsistency between these examples with Study #2's findings.

7.3.1. Example: Successes with P1-Content, P3-Actionable, and P6.ContextFree

First, we consider a semi-ideal example, in which the Idea Garden was very helpful to highly productive campers who focused on JavaScript-related tasks. The top example of this type was a 12th grade male ("Bob"), who completed the second highest number of programming tasks of all the campers. Bob interacted frequently with the Idea Garden, reading and acting upon suggestions from the hints on iteration during day 3 of the two-week camp. Bob explicitly wrote about his use of the Idea Garden on day 5, when he said in his end-of-day survey that the Idea Garden "told me to try using a map function or a for-in loop and im [sic] trying to get them to work"— and on day 6, helpers' observation forms showed him successfully using iteration without further help.

Bob's Idea Garden usage parallels that of Study #2 teams who read a hint and then acted upon its suggestions – the "ideal" way we envision indifferent EUPs using the Idea Garden (e.g., Study #2's Team Turtle). In Study #2, 60% of teams who used this strategy made progress on their particular problem (Table 7, row 1). Study #3's Bob (who had no prior programming or web development experience) made progress on his iteration difficulties in the same way. Specifically, the logs showed that Bob used P6.ContextFree hints to compare different kinds of iteration, and explicitly used the hints' content (P1) and actionable suggestions (P3) to make progress on his loops.

7.3.2. Example: Issues with P2-Relevance

Not all cases looked like Bob's, of course. One issue that stood out was relevance to indifferent EUPs. In **Section** 7.1, we pointed out the challenge for showing relevance for the hints that are about problem-solving concepts rather than code concepts—but in Cloud9, the code concepts' hints also seemed to show relevance issues.

For example, a 9th grade male who we call "Bill" focused more on content creation with HTML and CSS than on JavaScript beyond its most basic bits, and rarely turned to the Idea Garden for these tasks—and with good reason, since the Idea Garden did not target those situations. Still, Bill did use JavaScript when he worked on implementing a map function that would go through an array of photos. Since Bill did not ask for much in-person help and did not collaborate much with other campers, we would have hoped that Bill would turn to the Idea Garden when he ran into difficulties at this stage of his JavaScript work.

However, his first interaction with the Idea Garden was not helpful. As he reported on one of his end-of-day surveys: "I tried looking at [the Idea Garden map hint] and it wasn't really useful." Logs of his Idea Garden usage on this day show that Bill opened the map hint once, but did not interact with it (i.e., did not try to paste in the code example and edit it, did not expand the hint's "click to see more" widget, etc.) and closed it shortly afterwards.

Bill's usage of the Idea Garden in this example, in which he read an Idea Garden hint but did not act on the hint's suggestions, is similar to the way some of Study #2's teams acted when they used the Idea Garden (e.g., Study #2's Team Beaver). In fact, Study #2, none of the teams who followed Bill's "read and ignore" strategy progressed through the problem they were trying to solve (Table 7, row 2). Similarly, although Bill did complete a few of the assigned tasks, he had one of the lowest numbers of tasks completed of all of the campers. Although the map hint was almost certainly relevant to Bill's attempts to iterate through his photo array, the hint seems to have failed to convey its relevance to Bill, since he decided it "wasn't really useful" without even trying to act upon it. This example highlights both the importance of P2-Relevance and some of the difficulties encountered in achieving it.

7.3.3. Study #3's principled results

Table 10 presents a principle-by-principle view of some of Study #3's results.

From a generalization perspective, we learned from Study #3 that the same principles used in Gidget-based Idea Garden generalized beyond Gidget to the Cloud9 IDE, to the JavaScript language, and to the new expanded scope of the hints. The fact that 21/25 (84%) of campers found and used the Idea Garden without any explicit prompting from instructors shows the effectiveness of at least P6-Availability and P7-InterruptionStyle in this environment—because if those principles had failed, campers would not have been able to interact with the Idea Garden. Of the campers who *did* interact with the Idea Garden, 12/21 (57%) used it to make progress with their problems, as measured by end-of-day survey responses and Cloud9 logging mechanisms. This number is very similar to the 53% who made progress using the Idea Garden in Study #2's Table 9. This suggests that the Idea Garden's effectiveness in Gidget generalized well to the Cloud9/JavaScript environment.

8. Generalized Idea Garden Architecture

To enable other researchers to implement Idea Gardens in their own programming environments, we developed a generalized architecture. Our architecture builds upon earlier work [12] that proposed an architecture for the Idea Garden in CoScripter, but did not address the generalizability question. In this section, we take on the generalizability question from an implementation perspective: Can Idea Gardens be ported from one environment to another relatively easily, or must each be implemented entirely from scratch?

To answer this question, we created the generalized architecture shown in Fig. 9. Fig. 9 shows both the generalized architecture and its interactions with a host programming environment. Both the Gidget and Cloud9 Idea Gardens shown in this paper were implemented using this architecture.

To see how the pieces fit together, consider an example situation in which the Idea Garden responds to a user typing in some code that contains an anti-pattern. The following sequence numbers correspond to those in Fig. 9:

1. Suppose the user types the following JavaScript code into the Cloud9 host programming environment: for (var x in arr) {f1(arr[0]);}
2. The Host (Cloud9) reports this user code to the Listener.
3. The Listener parses that code and finds a for loop that does not use its iterator variable (x in this case). The Listener recognizes this as an instance of the *no-iterator* anti-pattern, so it prepares an abstract event with type *no-iterator* to send off to the Controller. Along with this event, it sends the name of the unused variable, the name of the list from the *for* loop (arr) and the context in which it happened (such as: the line number, location on the screen, the main code window's contents, side preview windows, sets of menu buttons, etc.).
4. The Controller delegates further translations of the data that it needs to additional Information Processors plug-in's, and then …
5. uses the results to map the input abstract event (*no-iterator*) to its corresponding abstract action (*show_iteration_icon*).
6. The Controller then sends the abstract action (*show_iteration_icon*) and the user's code to the Actioner.
7. The Actioner delegates hint construction to the Hint Engine, which finds the relevant hint (the iteration hint), inserts the user's code and variable names into the iteration hint's code template, producing this customized example code for the hint: for (var x in arr) {console.log(arr[x]);}
8. The Actioner receives the hint, and…
9. tells the Host environment (Cloud9) to decorate the line of code with an Idea Garden icon indicator that links to the above hint.

The Listener plays a particularly important role in supporting many of the principles. First, it directly supports P1-Content by listening for anti-patterns and providing abstract events related to that content. The Listener also supports P2-Relevance by including the user's code when

Table 10
Study #3's results for each principle.

Principle	Study #3 Examples and Results
P1-Content	Having multiple kinds of hints that illustrate ways to use the same concept (e.g., iteration) helped participants like Bob decide to use a map function
P2-Relevance	Relevance can be difficult to convey to indifferent EUPs (e.g., Bill), which can lead to users not progressing past barriers. Further, relevance was even more challenging to portray for problem-solving strategy hints than for concept hints
P3-Actionable	Actionability was similarly important in Study #3 as it had been in Study #2. For example, Bob implemented a map function by following the Idea Garden's actionable suggestions, and made progress by doing so
P4-Personality	*N/A (not investigated in Study #3)*
P5-Information Processing	*(No results from Study #3's data)*
P6-Availability	The combination of P6.ContextSensitive and P6.ContextFree mattered in Study #3 (similarly to Study #2). For example, Bob used P6.ContextFree to compare different kinds of iteration hints to find the one he wanted to use
P7-InterruptionStyle	The negotiated interruption style used in Study #3's implementation was sufficient to attract participants' attention. 21/25 campers used the Idea Garden *without* any prompting from the helpers or instructors.

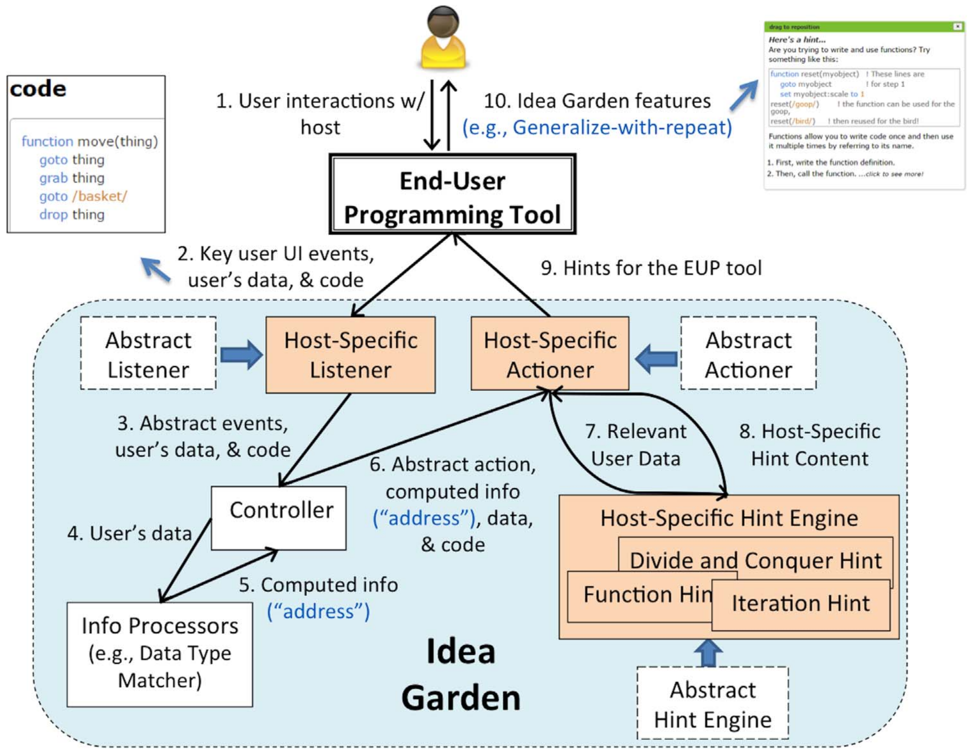

Fig. 9. Architecture of the Idea Garden. The black arrows represent the flow of data. User data (e.g., user's code) flows from the end-user programming tool to the host-specific listener. That data is passed along to the controller, information processors, and actioner, and finally used to build the 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, e.g., 2 points in blue to example code.

sending abstract events to the Controller, so that the code can be included in hints. Finally, the Listener supports P6-Availablity and P7-InterruptionStyle by observing user actions without interfering with the actions of the user or environment, then notifying the user of a hint in context (P6) with a negotiated interruption style (P7).

The Controller and Information Processors map abstract events to abstract actions (see Table 11 for example pairs of abstract events and abstract actions). By mapping abstract events (such as anti-patterns) to abstract actions (such as decorating the screen with the ?), the Controller notifies the Actioner to make Idea Garden hints available to the user in a certain context (P6). By passing along the context when the abstract event happens, the Idea Garden can include parts of that context to show users the relevance of hints (P2).

After the Controller matches an abstract event to an abstract action, the Host-Specific Actioner acts on the abstract action. The Actioner provides the input context to the Host-Specific Hint Engine. The Hint Engine customizes the hint (e.g. by replacing parts of the code example with the user's own variable and function names, supporting P2). The Hint Engine supports P5-InformationProcessing by containing parts of the hint within an expandable region. The Hint Engine supports P3-Actionable by requiring hints to include actionable instructions when implementers create the hints. As the user writes code, the Host-Specific Actioner updates the hints to include context-specific information (P2-Relevance). Finally, the Actioner finishes up the host-specific actions, decorating the screen with the ? icon to notify the user of a hint (supporting P7).

Table 11
Pairs of abstract events and abstract actions, which are matched to each other by the controller.

Abstract Events	Abstract Actions
user_needs_help_getting_started	show_getting_started_hint
no_iterator	show_iteration_icon
user_previewed_webpage	highlight_evaluation_hints

9. Concluding remarks

In this paper, we have investigated the generalizability of the Idea Garden. Our mechanisms for doing so were to (1) develop a set of *general principles* for the Idea Garden and evaluate them in multiple environments, (2) to develop a generalized architecture enabling Idea Gardens to at least conceptually "plug in" to environments willing to communicate user actions and receive communications for the interface, and evaluate its viability in multiple environments, and (3) to develop multiple types of support, covering both difficulties with *programming concepts* and difficulties with *problem-solving strategies*. Table 12 summarizes the results of our investigation from a

Table 12
Summary of principle-by-principle evaluations.

Principle	Ways	Formative Evidence	Summative Evidence
P1-Content		+Study1	+Study2*, +Study3
P2-Relevance	P2-All	-[10]	+,-Study3
	P2.MyCode		+Study2*
	P2.MyState	+Study1	+Study2*
	P2.MyRequirements	+Study1	
P3-Actionable	P3.ExplicitlyActionable		+Study2*, +Study3
	P3.ImplicitlyActionable		+Study2*, +Study3
P4-Personality			+[34]
P5-InformationProcessing		+[41]	+Study2*
P6-Availability	P6.ContextFree	+,-Study1	+Study2* +Study3
	P6.ContextSensitive	+Study1	+Study2*
P7-InterruptionStyle			+[46], +Study3

+: Principle was helpful, -: Principle was problematic. *: Teams progressed in the majority (> =50%) of their barriers with these Idea Garden principles principled perspective.

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 (e.g., Table 8) serve to validate the concept aspect of P1; mini-patterns were especially involved in teams' success rates with Coordination barriers. Together, these aspects enabled the teams to overcome, without any in-person help, 41–68% of the barriers they encountered across *diverse barrier types*. The content also generalized to the strategies aspect: Study #3's results showed that, unlike the control group, the experiment group (supported in part by Idea Garden strategy hints) did not need to rely on in-person help for their successes. This suggests that following P1-Content is helpful with a *diverse scope of problem-solving difficulties*, from conceptual barriers to strategies.

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 P2-Relevance and P6-Availability 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*.

P6-Availability and P7-InterruptionStyle: P6 requires making the Idea Garden available even when the context changes, and P7 requires supporting negotiated-style interruptions to allow users to initiate interactions with the Idea Garden on their own terms. The fact that almost all participants in both studies found and interacted with the Idea Garden in some way suggests that the pairing of P6-Availability and P7-InterruptionStyle succeeded in engaging EUPs in a *diversity of contexts*.

Taking the principles together, the studies presented in this paper show that Idea Gardens built according to these principles under our generalized architecture are very effective. For example, Study #2 in Gidget showed the teams learned enough programming in only about 5 h to begin building their own game levels comparable to those created in a prior study of Gidget [37]. However, unlike the prior study, they accomplished these gains with significantly less in-person help than they required in an earlier study that did not have the Idea Garden. Study #3 in Cloud9 showed that participants were able to complete more self-initiated tasks and to rely less on in-person helpers. In fact, Study #2's and Study #3's success rates without in-person help were remarkably similar.

Due to these gains in generalizability, the Idea Garden has now been implemented in multiple programming environments for multiple languages. The first Idea Garden, built using the predecessor of the generalized architecture, was in CoScripter, a programming-by-demonstration language and IDE for web automations. We used the generalized architecture to implement an Idea Garden for Gidget, an imperative, object-based language in its own IDE, and used it again to implement an Idea Garden for JavaScript in Cloud9.

These promising results 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, tasks, host IDEs, programming languages, and people.

Acknowledgments

This paper extends prior work presented in [26,39], and we thank our co-authors of those works for their insights and help during those phases of the project. We also thank the participants of our studies. This work was 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.

References

[1] R. Andersen, A. Mørch, Mutual development: a case study in customer-initiated software product development, End.-Use. Dev. (2009) 31–49.
[2] P. Airasian, K. Cruikshank, R. Mayer, P. Pintrich, J. Raths, M. Wittrock, A Revision of Bloom's Taxonomy of Educational Objectives (Complete edition), in: L. Anderson, D. Krathwohl (Eds.), A Taxonomy for Learning, Teaching, and Assessing, Longman, 2001.
[3] J.R. Anderson, F.G. Conrad, A.T. Corbett, Skill acquisition and the LISP tutor, Cogn. Sci.-ence 13 (4) (1989) 467–505.
[4] J. Brandt, M. Dontcheva, M. Weskamp, S. Klemmer, Example-Centric Programming: Integrating Web Search into the Development EnvironmentProc. CHI 2010, ACM, 2010, pp. 513–522.
[5] J. Bransford, A. Brown, R. (Eds) Cocking, How People Learn: Brain, Mind, Experience, and School, National Academy Press, 1999.
[6] M. Burnett, L. Beckwith, S. Wiedenbeck, S. Fleming, J. Cao, T. Park, V. Grigoreanu, K. Rector, Gender pluralism in problem-solving software, Interact. Comput. 23 (2011) 450–460.
[7] B. Burg, A.J. Ko, M.D. Ernst, (November). Explaining visual changes in web interfac-es, in: Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technol-ogy pp. 259–268. ACM, 2015.
[8] Jill Cao, Helping End-User Programmers Help Themselves - The Idea Garden Approach (Ph.D Dissertation), Oregon State University, Corvallis, OR, 2013.
[9] J. Cao, S.D. Fleming, M. Burnett, An exploration of design opportunities for 'gardening' end-user programmers' ideas, IEEE VL/HCC (2011) 35–42.
[10] J. Cao, I. Kwan, R. White, S. Fleming, M. Burnett, C. Scaffidi, From barriers to learning in the Idea Garden: an empirical study, IEEE VL/HCC (2012) 59–66.
[11] J. Cao, I. Kwan, F. Bahmani, M. Burnett, S. Fleming, J. Jordahl, A. Horvath, S. Yang, End-user programmers in trouble: can the Idea Garden help them to help themselves?, IEEE VL/HCC (2013) 151–158.
[12] J. Cao, S. Fleming, M. Burnett, C. Scaffidi, Idea Garden: situated support for problem solving by end-user programmers, Interact. Comput. (2014) (21 pages).
[13] J. Carroll, M. Rosson, The paradox of the active user. Interfacing Thought: Cognitive Aspects of Human-Computer Interaction, MIT Press, 1987.
[14] J. Carroll, The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill, 1990.
[15] Cloud9, - Your development environment in the cloud. ⟨http://www.c9.io⟩. Last (accessed 17.08,16), 2016.
[16] M. Costabile, P. Mussio, L. Provenza, A. Piccinno, Supporting End Users to be Co-designers of Their Tools. End-User Development, Springer, 2009, pp. 70–85.
[17] A. Cypher, J. Nichols, M. Dontcheva, T. Lau, No Code Required: Giving Users Tools To Transform the Web, Morgan Kaufmann, 2010.
[18] P. Diaz, I. Aedo, M. Rosson, J. Carroll, A Visual Tool for Using Design Patterns as Pattern Languages. In Proc. AVI, ACM Press, 2010, pp. 67–74.
[19] B. Dorn, ScriptABLE: Supporting Informal Learning with Cases, In Proc. ICER, ACM, 2011, pp. 69–76.
[20] V. Grigoreanu, M. Burnett, G. Robertson, A strategy-centric approach to the design of end-user debugging tools, ACM CHI (2010) 713–722.
[21] V. Grigoreanu, M. Burnett, S. Wiedenbeck, J. Cao, K. Rector, I. Kwan, End-user debugging strategies: a sensemaking perspective, ACM TOCHI 19 (1) (2012) 28.
[22] P. Gross, M. Herstand, J. Hodges, C. Kelleher, A code reuse interface for non-programmer middle school students, ACM IUI 2010 (2010) 219–228.
[23] M. Guzdial, Education: paving the way for computational thinking, Comm. ACM 51 (8) (2008) 25–27.
[24] C. Hundhausen, S. Farley, J. Brown, Can direct manipulation lower the barriers to computer programming and promote transfer of training? An experimental study, ACM TOCHI 16 (3) (2009), Article 13.
[25] W. Jernigan, Generalizing the Idea Garden: Principles and Contexts (M.S. Thesis), Oregon State University, 2015.
[26] W. Jernigan, A. Horvath, M. Lee, M. Burnett, T. Cuilty, S. Kuttal, A. Peters, I. Kwan, F. Bahmani, A. Ko, A principled evaluation for a principled Idea Garden, IEEE VL/HCC 2015 (2015) (235-24).
[27] C. Kelleher, R. Pausch, Stencils-based tutorials: design and evaluation, ACM CHI (2005) 541–550.
[28] C. Kelleher, R. Pausch, Lessons learned from designing a programming system to support middle school girls creating animated stories, IEEE VL/HCC (2006) 165–172.

[29] A. Ko, B. Myers, H. Aung, Six learning barriers in end-user programming systems, IEEE VLHCC (2004) 199–206.
[30] A. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, J. Lawrance, C. Scaffidi, H. Lieberman, B. Myers, M. Rosson, G. Rothermel, M. Shaw, S. Wiedenbeck, The state of the art in end-user software engineering, ACM Comput. Surv. 43 (3) (2011) 44, Article 21 (pages).
[31] J.A. Kulik, J.D. Fletcher, Effectiveness of intelligent tutoring systems a meta-analytic review, Rev. Educ. Res. (2015) (0034654315581420).
[32] R. Kumar, J. Talton, S. Ahmad, S. Klemmer, Bricolage: example-based retargeting for web design, ACM CHI (2011) 2197–2206.
[33] M.J. Lee, Teaching and Engaging With Debugging Puzzles (Dissertation)(UW), University of Washington, Seattle, WA, 2015.
[34] M. Lee, A. Ko, Personifying programming tool feedback improves novice programmers' learningProc. ICER, ACM Press, 2011, pp. 109–116.
[35] M.J. Lee, A.J. Ko, Investigating the role of purposeful goals on Novices' engagement in a programming game, IEEE VL/HCC (2012) 163–166.
[36] M.J. Lee, A.J. Ko Comparing the Effectiveness of Online Learning Approaches on CS1 Learning Outcomes. ACM ICER, 237-246.
[37] M. Lee, F. Bahmani, I. Kwan, J. Laferte, P. Charters, A. Horvath, F. Luor, J. Cao, C. Law, M. Beswetherick, S. Long, M. Burnett, A. Ko, Principles of a debugging-first puzzle game for computing education, IEEE VL/HCC (2014) 57–64.
[38] G. Little, T. Lau, A. Cypher, J. Lin, E. Haber, E. Kandogan, Koala: capture, share, automate, personalize business processes on the web, ACM CHI (2007) 943–946.
[39] D. Loksa, A.J. Ko, W. Jernigan, A. Oleson, C. Mendez, M.M. Burnett, Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance ACM Conference on Human Factors in Computing, 1449–1461, 2016.
[40] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, B. Hartmann, (May). Design lessons from the fastest q & a site in the west. In Proceedings of the SIGCHI conference on Human factors in computing systems, pp. 2857–2866. ACM, 2011.
[41] J. Meyers-Levy, Gender differences in information processing: a selectivity interpretation, in: P. Cafferata, A. Tubout (Eds.), Cognitive and Affective Responses to Advertising(eds.), Lexington Books, 1989.
[42] B. Myers, J. Pane, A. Ko, Natural programming languages and environments, Comm. ACM 47 (9) (2004) 47–52.
[43] B. Nardi, A Small Matter of Programming, MIT Press, 1993.
[44] S. Oney, B. Myers, FireCrystal: understanding interactive behaviors in dynamic web pages, IEEE VL/HCC (2009) 105–108.
[45] J. Pane, B. Myers, More Natural Programming Languages and EnvironmentsProc. End User Development, Springer, 2006, pp. 31–50.
[46] T. Robertson, S. Prabhakararao, M. Burnett, C. Cook, J. Ruthruff, L. Beckwith, A. Phalgune, Impact of interruption style on end-user debugging, ACM CHI (2004) 287–294.
[47] S. Subramanian, L. Inozemtseva, R. Holmes, (May). Live API documentation. ACM/IEEE International Conference on Software Engineering, pp. 643–652, 2014.
[48] N. Tillmann, J. De Halleux, T. Xie, S. Gulwani, J. Bishop, Teaching and learning programming and software engineering via interactive gaming, ACM/IEEE International Conference on Software Engineering, pp. 1117–1126, 2013.
[49] S. Turkle, S. Papert, Epistemological pluralism, Signs 16 (1) (1990).
[50] K. VanLehn, The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems, Educ. Psychol. 46 (2011) 197–221.
[51] W. Wickelgren, How to Solve Problems: Elements of a Theory of Problems and Problem Solving, W. H. Freeman & Company, 1974.
[52] A. Wilson, M. Burnett, L. Beckwith, O. Granatir, L. Casburn, C. Cook, M. Durham, G. Rothermel, Harnessing curiosity to increase correctness in end-user programming, ACM CHI (2003) 305–312.