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

Michael J. Lee[1], Faezeh Bahmani[2], Irwin Kwan[2], Jilian LaFerte[2], Polina Charters[1], Amber Horvath[2], Fanny Luor[1], Jill Cao[2], Catherine Law[3], Michael Beswetherick[1], Sheridan Long[2], Margaret Burnett[2], Amy J. Ko[1]

University of Washington
The Information School[1]
Seattle, Washington, USA

Oregon State University
School of EECS[2] and STEM Academy[3]
Corvallis, Oregon, USA

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

***Keywords—Computer science education; debugging; summer camp; educational game; computational thinking; user study***

## I. INTRODUCTION

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

There are many well-known tools to help people acquire programming skills independently. For example, Scratch [20] and Alice [14], now widely used, enable people to tell interactive stories, and sites like Codecademy.org allow users to follow simple tutorials to learn widely used languages such as JavaScript and Python. Unfortunately, these learning technologies have limitations that interfere with teaching at scale without instructors. Scratch and Alice, while quite effective at engaging learners in telling stories, require learners to somehow learn a language and a development environment before they can write their own programs. Consequently, these environments require teachers and other instructional resources to help learners succeed. At the other end of the continuum are tutorial tools such as Codecademy and games such as RubyWarrior, which ask learners to follow instructions typing-in and running commands in a virtual terminal. Although these environments do present programming to learners, they provide little instruction about what is happening or why, and leave learners little room to explore. Moreover, they do not follow best practices for intelligent tutoring systems (e.g., providing detailed, immediate feedback [33]).


Figure 1. Gidget's level design mode (the Gidget character is circled). In this mode, learners design their own levels for others to solve. Players write code (left) that can include graphics (right), and see animated results (middle), and graphics for the level are on the right.

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

In this paper, we present an alternative approach for learning technologies that teach computing, which we call a debugging game, instantiated in our online game, Gidget. This type of game requires players to *debug existing programs* before going on to create their own programs in the form of puzzle levels (as in Figure 1). We define our new approach through seven principles, which we present next.

## II. THE PRINCIPLES OF DEBUGGING GAMES

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

**P1-debug. Debugging first**: Encourage learners to learn programming concepts by debugging existing programs before creating new programs. Unlike many other educational technologies where creation occurs immediately [14,20], our

978-1-4799-4035-6/14/$31.00 ©2014 IEEE

approach provides nearly complete, but broken programs for learners to debug and fix before moving onto the more demanding task of creating new puzzles from scratch.

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

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

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

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

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

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

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

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

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

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

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

## III. THE GIDGET PROTOTYPE

To investigate our research questions, we created a new version of the debugging game Gidget (Figure 1) that embodies the seven principles. Descriptions of earlier versions of Gidget have been reported elsewhere [17,18,19], so here we focus only on the details needed for this paper.

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

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

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

Once the learner completes the curriculum (puzzle levels), they can use the level designer to create, save, modify, and share new levels. The level designer (Figure 1) is an interface that allows the player to write code for new levels' behavior, add introductory text to the level, change the size of the world,

set the goals and original code for the level, and view the usable graphics and sounds in the game. It also introduces the concept of event handling (i.e., having objects in the game wait for a condition before running a code block), which was not covered in the game curriculum.

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

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

## IV. Methods

We conducted two formative studies: a laboratory think-aloud study to record in-depth interactions with Gidget, and two summer camps to observe participants play puzzles and create levels over five days. We varied the levels, but not the concepts, between the two studies to 1) cover more concepts in one sitting during the think-aloud study, and 2) verify with think-aloud data that it was *concepts* that participants struggled with and not the way the information was conveyed. Both studies' recruitment material avoided the word "programming" to prevent participants from self-selecting out. This paper focuses mainly on the summer camps since they included both puzzle play and level design, and triangulates against the think-aloud study's data where appropriate.

### A. Think-Aloud Study

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

### B. Summer Camps

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

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

Camps used identical staff: a lead (male graduate student) led the activities and kept the camp on schedule; a researcher (female graduate student) recorded observations from a distance, and four helpers (all undergraduate females) answered questions, approached struggling participants, and recorded observations. The staff provided no formal instruction about Gidget or programming. Helpers recorded, using pre-designed observation forms, instances when campers had problems, noting what the problem was, what steps they tried prior to asking for help, and what assistance resolved the issue.

### C. Coding and Analyses

To categorize barriers participants encountered in both studies, we used two code sets from prior work (see Table 1). The *algorithm design barriers* are barriers that novice programmers encountered in end-user programming environments while designing algorithms [7]. The *learning phase barriers* are a sequence of barriers that novice programmers encountered when learning to program [16].

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

We identified problematic concepts during puzzle play by examining the levels with the highest number of barriers (see Figure 2). We also identified additional concepts participants struggled with during level design. We excluded *understanding barriers* from both analyses because unlike other barrier types that were related to one specific part or concept in the code, these were caused by misunderstandings of several concepts that could not be mapped exclusively to one programming concept.

TABLE I. BARRIERS CODE SETS (CAO ET AL. [7], KO ET AL. [16]).

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

## V. RESULTS

### A. Struggles with Programming Concepts

#### A.1) Programming Concepts During Puzzle Play

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

The first conceptual difficulty that affected a number of camp participants was string equality, which was introduced in Level 14. The concept caused 8 out of 29 barriers (Figure 2). The goal of this level required participants to change the string argument of the "`set`" command so that it matched "`Please help me Dog!`". Participants often struggled because they had a more relaxed perception of string equality than programming requires, often setting capitalization differently or omitting the exclamation point. This was corroborated with evidence from the think-aloud study, where 3 of the 10 participants also struggled with string equality and received help from the experimenter—one participant exclaimed, "Are you kidding me?" after receiving help. Since several participants appeared unable to recognize string equality issues, it should be explicitly taught in embedded instructions (*P5-instruction*) and supported with clear examples in scaffolded help (*P6-help*).

Camp participants also had difficulties with functions, which were introduced in Level 20. They caused 22 out of 28 barriers recorded for this level (Figure 2). The most common issue participants faced was understanding the difference between a function *call* and a function *definition*, and many omitted function calls, assuming that the function definition would actually run the function. Furthermore, participants from both studies had trouble matching function calls with their definitions (function names were either not defined, or spelled incorrectly) or passing the wrong type or number of parameters. Participants continued to struggle with these concepts in all subsequent levels dealing with functions, particularly in levels 23 and 34 (Figure 2).

The third problematic concept was defining new objects, which caused 20 out of 45 barriers in Level 23 (Figure 2),

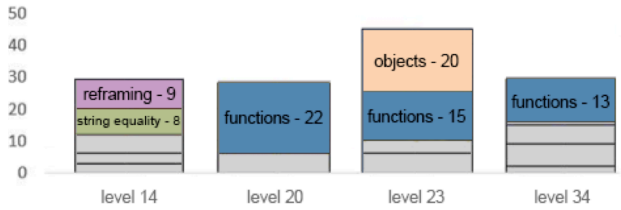


Figure 2. The concepts in the most challenging levels during puzzle play. Concepts accounting for fewer than 25% of the barriers in each level are shown unlabeled in gray.

making it the most difficult level in the game (Table 2). In the camps, participants often omitted the object definition or struggled with the constructor. In addition, think-aloud participants had difficulties working with functions encapsulated within an object, often omitting or erroneously deleting the object name before the function call:

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

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

#### A.2) Programming Concepts During Level Design

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

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

TABLE II. NUMBER OF BARRIERS PER LEVEL AND THE PERCENT IMPROVEMENT (%IMP) IN BARRIERS FROM THE PUZZLE PLAY (PZ) TO LEVEL DESIGN (LD) IN THE SUMMER CAMPS. EACH COLUMN CONTAINS THE NUMBER OF BARRIERS IN THE LEVEL. ALGORITHM DESIGN BARRIERS ARE SHOWN IN ORANGE (TOP TWO ROWS), AND LEARNING PHASE BARRIERS ARE SHOWN IN BLUE (FIVE MIDDLE ROWS AND THE SECOND-LAST ROW). ASSESSMENT LEVELS WERE NOT CODED AND MARKED WITH HYPHENS. DARKER COLORS INDICATE HIGHER COUNTS.

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

In addition, the assertions concept caused 16 out of 238 barriers (6%). Assertions, implemented via the `ensure` statement, described the level goals. For example, `ensure /gidget/:position = /button/:position` means that Gidget needs to end up on the button to "win" the level. Participants saw `ensure` statements throughout the game as they played each level, but did not have to write one until they designed their own levels. Interestingly, although participants did not encounter many barriers in the "conditionals" unit (Table 2, Unit 5), and did not have many problems *reading* the goals in the form of `ensure` statements (there were only 15 design barriers in Table 2), they struggled *writing* their own `ensure` statements, as seen in previous work [26].

The barriers participants encountered with event-handling and assertions suggest that mere exposure to a programming construct in a program understanding task is not necessarily sufficient to teach a participant how to use these constructs independently to author new behaviors. Therefore, educational technologies that require any amount of authoring have to recognize and teach code reading and writing tasks as distinct skills. In Gidget, this might be accomplished by: (1) including units that effectively combine both program understanding tasks and program writing tasks in a unit to gradually make players comfortable with writing each construct as they are introduced (*P5-instruction*), and (2) providing clearer examples and hints that relate back to previously covered and related concepts such as conditionals when trying to teach assertions (*P6-help*).

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

### B. Counterproductive Problem-Solving Strategies

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

The "All-knowing computer" antipattern refers to a player's failure to scrutinize the original code, even though they were told that it was filled with errors (as in *P3-fallible*). Instead, they largely trust that the original code is correct. The belief that the computer was always correct—observed also by Beckwith et al. [3] in a context that did *not* explicitly inform their participants that the code was incorrect—eventually led to many other barriers. In the think-aloud study, the original code from one level properly used a function call that was encapsulated within an object, and no one struggled with encapsulation. But, in a later level, 3 out of 4 participants who skimmed over the original code that used a function call with the wrong object struggled with the concept.

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

The "When all you have is a hammer, everything looks like a nail" antipattern is where a player persists in using programming constructs that worked for earlier levels but are no longer applicable. The reflection-in-action model [32] points to the importance of reframing when devising solutions. This rigidity was problematic for several participants: for example, recall in Level 14 that the explicit goal was to ask the dog "`Please help me Dog!`" for help with the task. It was necessary to use the `set` command to set a variable to that string, but 8 teams ignored the `set` command and tried to use previously learned commands (such as `goto`), leading to 9 out of 29 barriers in Level 14 (Figure 2).

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

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

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

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

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

### C. From Debugging-First to Programming: Level Creation

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

Each team created between 2 and 10 levels (median: 5). The majority (66/101) of the levels created were Gidget puzzles (e.g., Figure 3) or mazes meant to challenge other players, but some participants also had partially-completed or proof-of-concept levels (21/101). Some participants repurposed the level designer for unintended functionality. For example, team mustache built three levels to hold solutions to their other levels, Epsilon made 2 story-related levels without any puzzle-solving elements, and three teams from the Oregon camp used the level designer to draw pixel art. Overall, teams faced very few design barriers (2/258, Table 2), suggesting that they had many ideas for levels after playing through the game.

Every team designed two or more complete levels that used at least one of the taught programming constructs (see Table 3). The minimum knowledge to create a Gidget level is a Boolean expression to indicate a goal. Non-trivial Gidget levels (such as Figure 3) require knowledge of variables, Booleans, objects, and events. Thirteen teams designed levels that required programming concepts such as conditionals, loops, or the event-driven "`when`" statement and 6 used every concept in Gidget. All teams used at least one Boolean expression in their levels since it was mandatory to have a goal (written as an assertion). Additionally, many teams (76%) used events in their levels to so that an automatic event would occur as part of their stories. Some teams demonstrated their knowledge by

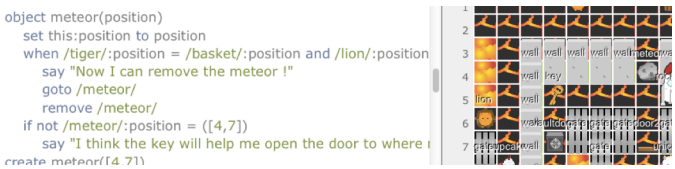


Figure 3. Camp participants most often created puzzle levels to challenge other players. Team Mustache developed this level where Gidget had to rescue animals, remove a meteor, and more. The code fragment contains objects, the event-driven `when` statement, and a conditional statement.

writing their own incomplete puzzle code containing functions and loops for other players to debug.

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

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

### D. Overcoming Barriers: Practice Makes Perfect?

One measure of whether participants in the camps learned from playing Gidget is to see if they encountered fewer barriers in puzzle play compared to level design. Using team-by-team barrier data (similar to those calculated in the right-most columns of Table 2), we calculated each team's percent improvement per barrier type (Figure 4). We saw improvements in 15 out of 17 camp teams and an overall improvement of 45% from puzzle play to level design (see Table 2, lower-right corner). One explanation for the

TABLE III. TEAMS USING THE CONCEPT IN AT LEAST ONE LEVEL THEY CREATED ARE MARKED (✓). 6 TEAMS DEMONSTRATED USAGE OF ALL 6 CONCEPTS AND 13 OF 17 TEAMS DEMONSTRATED 3 OR MORE CONCEPTS.

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

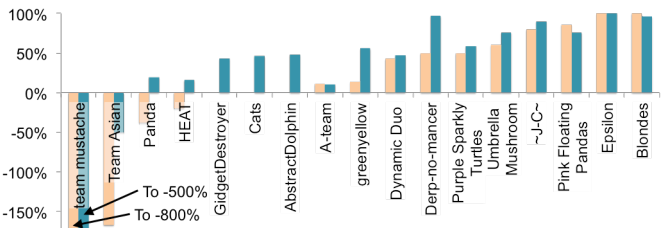


Figure 4. Percent improvement by comparing barriers before, then during level design. 15 of 17 teams showed improvements, with greater improvement on learning phase barriers (dark) than on algorithm design barriers (light).

improvements is that teams used only basic programming concepts in their level designs, but Table 3 shows that only 4 teams constructed levels requiring 2 or fewer programming constructs, so this explanation does not hold for the other 11 teams. For these 11 teams, the best explanation for their improvement is that they did improve their programming skills while playing through the debugging puzzles.

We believe that two teams, Team Asian and Team Mustache (see Figure 4), did not improve on the number of barriers because they devised and implemented ambitious levels. Both teams used all six programming constructs in their levels. Team Mustache encountered 500% more (12 barriers in level design vs. 2 in the puzzle portion) learning phase barriers during level design than in puzzle play, but created multiple levels (such as Figure 3) incorporating complex concepts such as a `when` statement with multiple Boolean expressions to verify players completed objectives sequentially.

There was a noticeable difference in the amount of improvement in algorithm design barriers (19%) vs. learning phase barriers (51%). Table 2 shows the improvements for each barrier type (rightmost column). Two of the learning phase barriers improved by nearly 90%, compared to the best algorithm design barrier improvement of 41%. Furthermore, as Figure 4 shows, 15 teams improved on learning phase barriers, whereas only 10 teams improved on algorithm design barriers (Figure 4). Teams especially struggled with composition barriers, encountering them frequently but demonstrating only 7% improvement—the least amount of improvement out of all barrier types (Table 2). The fact that the algorithm design barriers did not greatly improve with instruction and practice from the game suggests that algorithm design concepts may require more thorough explanations and help *(P5-instruction, P6-help)* than what is currently provided.

In addition, despite the contradictory results in Section A.2 regarding the number of barriers per person in both genders, both genders had similar improvements with 58% and 64% for females and males, respectively. This positive evidence of gender inclusiveness (*P7-gender*) is encouraging as to both females' and males' learning through this approach.

## VI. Discussions and Implications

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

First, taken together, the seven principles in Gidget succeeded in teaching enough programming for participants to successfully write their own programs. Everyone finished the game in under 5 hours. Along the way, participants gradually learned to overcome many of their earlier learning phase barriers (51% improvement), although their ability to overcome their algorithm design barriers was less impressive (19% improvement). Still, the complexity and breadth of the levels the participants were able to create was impressive given their short learning time. For example, half the teams decided to use loops and functions in their custom levels and succeeded at doing so (Table 3), which are often major difficulties for novices in other programming languages.

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

Nuances regarding *P5-instruction* and *P6-help* have been discussed throughout this paper. Our instantiation of these principles in the current Gidget allowed participants to complete the curriculum largely independently, and the learning they achieved transferred beyond the puzzle-based curriculum to the level design phase. However, much of the participants' learning was limited to learning phase barriers: the algorithm design barrier improvement was much lower (Table 2). There were also recurring struggles with concepts such as string equality, functions, and objects (Table 2). These findings suggest that the type of static, contextual help in the current version of the game may be sufficient for teaching lower level concepts such as language syntax and semantics, but not for teaching algorithm design problem solving skills. Future work is necessary to identify appropriate ways of teaching these higher level skills in computing education learning technologies.

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

## VII. Conclusion

The debugging games approach avoids the problem where learners need a large amount of programming knowledge before they can begin creating their own programs. We found that the seven design principles used to create Gidget worked together in many different capacities to successfully teach programming concepts in just 5 hours to learners who did not necessarily want to learn programming. Debugging games and more broadly, educational technologies such as Alice, Scratch, Codecademy, and other creative environments and tutorials

may benefit from adopting the design principles explored in this paper. For example, adopting a debugging-first approach may empower users to learn without requiring an instructor, teach them important program understanding and debugging skills, and can lead to more success at creating their own programs. Revising the communication and instruction that environments provide to frame computers as fallible entities may also play an important role in sustaining learners' motivation.

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


## ACKNOWLEDGMENT

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

## REFERENCES

[1] Aleven, V., Myers, E., Easterday, M., & Ogan, A. (2010). Toward a framework for the analysis and design of educational games. *IEEE DIGITEL*, 69-76.

[2] Andersen, E., Liu, Y. E., Snider, R., Szeto, R., Cooper, S., & Popović, Z. (2011). On the harmfulness of secondary game objectives. *ACM FDG*, 30-37.

[3] Beckwith, L., Burnett, M., Cook, C. (2002). Reasoning about many-to-many requirement relationships in spreadsheets. *IEEE VL/HCC*, 149-157.

[4] Blackwell, A.F. (2002). First steps in programming: A rationale for attention investment models. *IEEE HCC*, 2-10.

[5] Brandt, J., Guo, P.J., Lewenstein, J., Dontcheva, M., & Klemmer, S.R. (2009). Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. *ACM CHI,* 1589-1598.

[6] Burnett, M., Beckwith, L., Wiedenbeck, S., Fleming, S.D., Cao, J., Park, T.H., Grigoreanu, V., Rector, K. (2011). Gender pluralism in problem-solving software, *Interacting with Computers*, 23, 450-460.

[7] Cao, J., Kwan, I., White, R., Fleming, S., Burnett, M., & Scaffidi, C. (2012). From barriers to learning in the Idea Garden: An empirical study. *IEEE VL/HCC*, 59-66.

[8] Cao, J., Kwan, I., Bahmani, F., Burnett, M., Fleming, S.D., Jordahl, J., Horvath, A., & Yang, S. (2013). End-user programmers in trouble: Can the Idea Garden help them to help themselves? *IEEE VL/HCC*.

[9] Charters, P., Lee, M.J., Ko, A.J., & Loksa, D. (2013). Challenging Stereotypes and Changing Attitudes: The effect of a brief programming encounter on adults' attitudes toward programming. *ACM SIGCSE*.

[10] Ellis, A. (2005). *Research On Educational Innovations*. Eye On Education, Inc., Larchmont, NY

[11] ESA (2011). Essential facts about the computer and video game industry. *Entertainment Software Association*. Web. 21 Feb. 2012. <http://www.theesa.com/facts/pdfs/ESA_EF_2011.pdf>

[12] Gee, J.P. (2003). What video games have to teach us about learning and literacy. *Computers in Entertainment*, 1(1), 20.

[13] Goode, J., Estrella, R., & Margolis, J. (2006). Lost in translation: Gender and high school computer science, In *Women and Information Technology: Research on Underrepresentation*, MIT Press, 89-114.

[14] Kelleher, C., Pausch, R., & Kiesler, S. (2007). Storytelling Alice motivates middle school girls to learn computer programming. *ACM CHI*, 1455-1464.

[15] Kerr, J., Kelleher, C., Ellis, R. & Chou, M (2013). Setting the scene: scaffolding stories to benefit middle school students learning to program. *IEEE VL/HCC*, 95-98.

[16] Ko, A.J., Myers, B.A., & Aung, H. (2004). Six learning barriers in end-user programming systems. *IEEE VL/HCC*, 199-206.

[17] Lee, M.J. & Ko, A.J. (2011). Personifying programming tool feedback improves novice programmers' learning. *ACM ICER*, 109-116.

[18] Lee, M.J., Ko, A.J. (2012). Investigating the role of purposeful goals on novices' engagement in a programming game. *IEEE VL/HCC*, 163-166.

[19] Lee, M.J., Ko, A.J., & Kwan, I. (2013). In-game assessments increase novice programmers' engagement and level completion speed. *ACM ICER*, 153-160.

[20] Maloney, J.H., Peppler, K., Kafai, Y., Resnick, M., & Rusk, N. (2008). Programming by choice: Urban youth learning programming with scratch. *ACM SIGCSE Bulletin*, 40(1), 367-371.

[21] Margolis, J. & Fisher, A. (2003). *Unlocking the Clubhouse: Women in Computing,* MIT Press.

[22] Meyers-Levy, J. (1989). Gender differences in information processing: A selectivity interpretation, In *Cognitive and Affective Responses to Advertisin*g, Lexington Books, 219-260.

[23] Misa, T. (2010). Gender codes: Defining the problem, in *Gender Codes: Why Women are Leaving Computing*, Wiley, 3-24.

[24] Monroy-Hernández, A., & Resnick, M. (2008). Empowering kids to create and share programmable media. *Interactions*, *15*(2), 50-53.

[25] NCWIT (2010). NCWIT Scorecard: A report on the status of women in information technology. *Nat'l Ctr. for Women & IT*. Web. 30 Mar. 2013. <http://www.ncwit.org/pdf/Scorecard2010.pdf>

[26] Pane, J., & Myers, B. (2006). More natural programming languages and environments, In *End User Development*, Springer, 31-50.

[27] Polya, G. (1971). *How to Solve It: A New Aspect of Mathematical Method*, Princeton Univ. Press.

[28] Ram, A., & Leake, D.B. (1995). *Goal-Driven Learning*. MIT Press, Boston, MA.

[29] Reinecke, L., Trepte, S., & Behr, K.M. (2008). *Why Girls Play. Results of a Qualitative Interview Study with Female Video Game Players*. Universitäts- und Landesbibliothek.

[30] Rising, L. (1999). Patterns: A way to reuse expertise. *IEEE Communications*, 37(4), 34-36.

[31] Scaffidi, C., & Chambers, C. (2012). Skill progression demonstrated by users in the Scratch animation environment. *Int'l J. HCI*, 28(6) 383-398.

[32] Schön, D.A. (1983). *The Reflective Practitioner: How Professionals Think in Action*. Basic Books, NY.

[33] Shute, V.J. (1993). A macroadaptive approach to tutoring. *Journal of AI in Education*, 4(1), 61-93.

[34] Subrahmaniyan, N., Kissinger, C., Rector, K., Inman, D., Kaplan, J., Beckwith, L., & Burnett, M. (2007). Explaining debugging strategies to end-user programmers. *IEEE VL/HCC*, 127-136.

[35] UK DFE (2013). *National Curriculum in England: Computing Programmes of Study*. (Dept. Education No. DFE-00171-2013). UK.

[36] Werner, L.L., Hanks, B., & McDowell, C. (2004). Pair-programming helps female computer science students. *ACM JERIC*, 4(1).

[37] Yee, N. (2006). Motivations for play in online games. Cyber Psychology & Behavior, 9(6), 772-775.