Personifying Programming Tool Feedback Improves Novice Programmers' Learning

Michael J. Lee and Andrew J. Ko
The Information School | DUB Group
University of Washington
{mjslee, ajko}@uw.edu

ABSTRACT

Many novice programmers view programming tools as all-knowing, infallible authorities about what is right and wrong about code. This misconception is particularly detrimental to beginners, who may view the cold, terse, and often judgmental errors from compilers as a sign of personal failure. It is possible, however, that attributing this failure to the computer, rather than the learner, may improve learners' motivation to program. To test this hypothesis, we present *Gidget*, a game where the eponymous robot protagonist is cast as a fallible character that blames itself for not being able to correctly write code to complete its missions. Players learn programming by working *with* Gidget to debug its problematic code. In a two-condition controlled experiment, we manipulated Gidget's level of *personification* in: communication style, sound effects, and image. We tested our game with 116 self-described novice programmers recruited on Amazon's Mechanical Turk and found that, when given the option to quit at any time, those in the experimental condition (with a personable Gidget) completed significantly more levels in a similar amount of time. Participants in the control and experimental groups played the game for an average time of 39.4 minutes (SD=34.3) and 50.1 minutes (SD=42.6) respectively. These finding suggest that how programming tool feedback is portrayed to learners can have a significant impact on motivation to program and learning success.

Categories and Subject Descriptors

K.3.2 Computer Science Education: Introductory Programming, D.2.5 Testing and Debugging.

General Terms

Design, Human Factors.

Keywords

Programming, Education, Personification, Motivation, Debugging

1. INTRODUCTION

For most beginners, the experience of writing computer programs is characterized by a distinct sense of failure. The first line of code beginners write often leads to unexpected behaviors, such as syntax errors, runtime errors, or program output that the learner did not intend. While all of these forms of feedback are essential to helping a beginner understand what programs are and how computers interpret them, the experience can be quite discouraging [28,29] and emotional [25].

These findings have significant implications for computing education. To many learners, error messages are not perceived as actionable facts, but as evidence that they are incompetent and that the computer is an all-knowing, infallible authority on what is right and wrong [6]. Even in programming environments designed for beginners such as Alice [24] and Scratch [35], where syntax errors are impossible and most runtime errors are avoided by having the runtime do something sensible rather than fail, the communication between the learner and the computer is framed as one-way: the computer does not express its interpretation of the code, it simply acts upon it without explanation. These relationships between learners and programming tools are more command-and-control than collaboration.

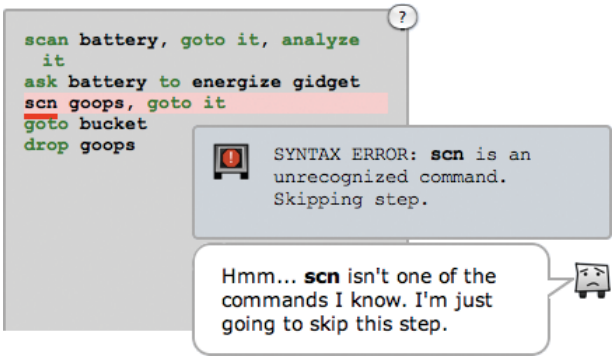

Figure 1. Runtime error highlighted in the instruction pane (*rear*), with corresponding error messages in the control

And yet, how people perceive their relationship to a computer is a critical determinant of not only their attitudes towards computers, but also their performance in using them to accomplish tasks [27]. Moreover, studies have shown that people expect computers to behave with the same social responses that people do [41]; for example, automated systems that blame users for errors negatively affect users' performance and their attitudes toward computers [17].

If negative feedback from computers affects people's performance on conventional computer tasks, does programming tool feedback also affect novice programmers motivation and learning success? To investigate this question, we designed Gidget, a web-based programming game in which the user helps a damaged robot correct its faulty code completing its missions (which are expressed as test cases). To investigate the role of feedback on learners' motivation, we designed two versions of the game, manipulating the robot's level of *personification*, changing communication style, sound effects, and appearance. As seen in Figure 1, the control version of the game used conventional, impersonal messages and appeared as a faceless terminal; the experimental version used personified language with personal pronouns, taking the blame for syntax and runtime errors, and had a face. In each condition, the information content conveyed through messages was the same. We then recruited a total of 250 individuals from all over the world using Amazon's Mechanical Turk [1], with 116 of them meeting our criteria as rank novice programmers. With this latter pool of participants, we found that those in the experimental group finished significantly more levels than those in the control group, meaning

ICER'11, August 8–9, 2011, Providence, RI, USA.

they successfully used more commands in the programming language. Participants in the control and experimental played the game for an average time of 39.4 minutes (SD=34.3) and 50.1 minutes (SD=42.6), respectively. However, there was no significant difference between conditions in the total time played, nor the number of times an individual executed a version of their code overall. Our findings also show that the experimental group completed more levels in fewer program executions than the control group, suggesting they were attending more to the steps of program execution explained by the robot.

In the rest of this paper, we discuss prior work on feedback in programming tools, detail our game and study design, and then discuss our results and their implications on computing education.

2. RELATED WORK

The role of feedback and critique in learning has long been studied in education [2,3,4,25,28,29,32,45]. For example, for some learners, negative feedback is more than discouraging: it is an explicit judgment of their abilities. Recent work in educational psychology has found that learners' sensitivity to critique have a strong relationship to self-reported motivation, self-reported performance levels in college courses, and avoidance of further opportunities to receive critical feedback [2,32]. Other work has found that females pay greater attention to the valence of critique (positive or negative) and that they are more likely to view negative critique as indicative of global ability on any task, rather than ability at a particular task [45]. Moreover, research on self-efficacy shows that building confidence at a skill requires not only success on tasks, but the sort of success learners believe is due to their own perseverance and creativity [3,4]. Dweck [14] has argued similarly that learners develop *self-theories* of themselves, appearing to have either a fixed mindset (where they believe intelligence is inborn) or a growth mindset (where they believe that intelligence can improve with hard work). All of these theories and findings appear to be at play in learners' first encounters with computer programming [29,39]. Our work builds on these ideas, investigating how redirecting negative feedback away from the learner and back to a personified computer entity affects learning.

Our research follows a long tradition of efforts to create programming environments for beginners [23]. Many of these technologies have focused on increasing learner motivation by incorporating new factors to entice learners to explore computational activities. For example, Logo [44] and more recently EToys [22] both created computational spaces for children to explore music, language, and mathematics; Light-bot [31] pushed players to take the robot's point-of-view of the environment to successfully navigate through levels; Playground [15] and LEGO Mindstorms [37] had similar goals, enticing children with the modeling and simulation of phenomena from the world or actually enabling them to write programs that sense the world. These approaches and others like them seek to entice learners with their intrinsic curiosity about the world and its processes.

Other approaches have motivated children with opportunities for self-expression. Play [48], My Make Believe Castle [34], Hands [42], ToonTalk [19], Klik & Play [33], Stagecast [47], Toque [49] and others all focus on enabling learners to create novel animations and games. Similar efforts have been made at the college level with projects such as Georgia Computes! [11] and Game2Learn [5], which encourages students to create and test their own games. Examples include Bug Bots [12] – a game where players attempt to repair robots by dropping tiles into a flowchart representing a computer program – and Virtual Bead Loom [8] – a game where students are encouraged to learn looping functions to create bead artwork instead of placing beads one at a time. Other systems that have added to these self-expression goals the ability to share the content one has created. For example, MOOSE Crossing invites learners to create characters and spaces in a virtual, multi-user text-based world [10]; more recently, Storytelling Alice [24] and Scratch [35] have focused on enabling learners to tell and share stories. Kelleher et al. [24] were one of the first to demonstrate that opportunities and affordances for storytelling can significantly improve learners' motivation to program. Our work follows these traditions, but provides learners with the story, allowing them to contribute to its progress by interacting with a character in a game.

While all of the systems discussed thus far aimed to increase motivation, several systems have aimed to lower demotivating factors in programming tools. Such approaches include simplifying the textual programming language syntax [10,43], designing languages that mimic how children describe program behavior [42], preventing syntax errors entirely by designing program construction interfaces that use drag and drop interactions (e.g., [7,22,35]) or form filling [33,34,47] rather than text. Others have attempted to simplify the debugging of programs by enabling learners to select "why" questions about program output [28,30]. Our research follows the same vein as these projects, aiming to mitigate factors inherent to programming that would diminish motivation by changing the programming environment. However, in contrast to prior work, our work will not add new capabilities to the programming environment, but rather changes how the existing capabilities of tools relate to the learner through the delivery and presentation of feedback and suggestions.

Given practice, novice programmers develop strategies to effectively understand unfamiliar code [16,18,21]. Working with a partner often affords the benefit of having working off each others' strengths and splitting up the work accordingly. Research exploring the effectiveness of pair-programming in introductory courses have shown that there are significant benefits for both teammates [9,36] and individuals [9]. This work has been extended to pair-debugging for novice programmers by Murphy et al., who report that interactive pairs often attempt more problems, and that critical pairs who reflected on their work often were more likely to successfully identify and resolve bugs [38]. Similarly, recent work has demonstrated that (cognitive) apprenticeship, where beginners are given regular feedback by experts, yields a higher retention rate of students in a beginner computer science course [50].

Our research builds on the ideas from these studies by having the learner and computer game character co-create the code that will accomplish the game goals. Previous studies have found that by simply telling participants that they were on the same team as a computer and representing this with armbands of the same color, participants showed greater affinity towards computers, being more willing to cooperate with it, conform to it suggestions, and assess the computer as more friendly and intelligent than computers on an opposing team [40]. Our work will shed new insight on how changing the role of the computer from an authoritative figure to a collaborator needing assistance will affect learner motivation.

3. METHOD

The goal of our study was to investigate the role of programming tool feedback on learners' motivation to program. To do this, we designed the programming game *Gidget*, shown in Figure 2, which asked learners to help a damaged robot fix its faulty programs, in order to accomplish its missions. Our study had two conditions – control and experimental – manipulating the *personification* of the robot protagonist, Gidget. By personifying Gidget, we aimed to increase the agency of the character, adding human-like qualities to an otherwise cold and emotionless entity. In the control condition, Gidget was represented as a faceless terminal screen that provided terse, impersonal feedback in response to commands and error messages (Figure 1). In contrast, the experimental condition

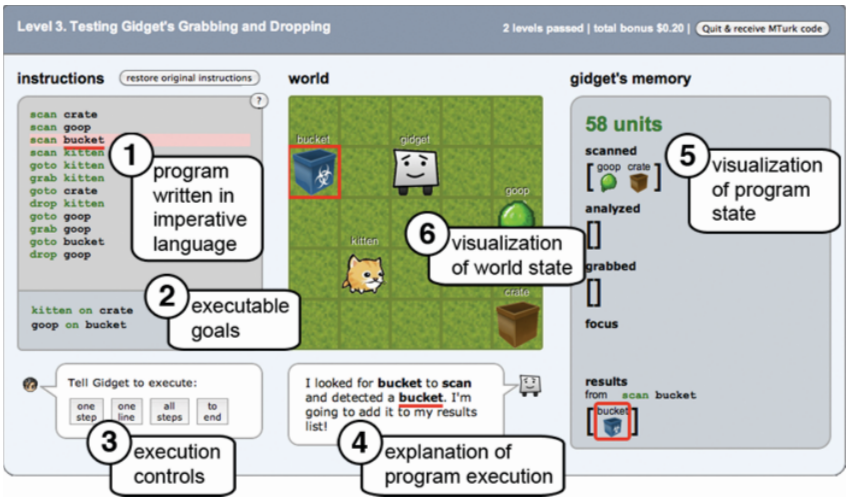

Figure 2. Gidget, shown here in its experimental condition, where learners help a damaged robot fix its programs.

represented Gidget as an emotive robot that included the use of personal pronouns such as "I" in the feedback, coupled with facial expressions corresponding to the runtime error state of the program. Participants were recruited on Amazon Mechanical Turk and offered 40¢ for completing the first level and 10¢ for each additional level completed. The total bonus and the levels completed were displayed in the upper right corner of the interface, along with a button giving the participants the option to quit at any time (Figure 2). The key dependent variable in our study was *levels completed* as a measure of learners' motivation to program.

Our null hypothesis was:

> H_0: There is no difference in *levels completed* between the control condition, using conventional, emotionless feedback and the experimental condition, using personified feedback.

In the rest of this section, we describe the game in more detail and discuss the experiment designed to test this hypothesis.

3.1 The Game

Our online game, called *Gidget* (shown in Figure 2), is an HTML5 and JavaScript application using jQuery. The game was tested for compatibility on MacOS X, Windows 7, and Ubuntu Linux 10 using Apple Safari 5, Mozilla Firefox 3.6 & 4.0, and Google Chrome 10 (we could not support Internet Explorer because it lacked the *contentEditable* attribute, which was used to implement the editor).

In the game, learners are guided through a sequence of levels that teach the design and analysis of basic algorithms in a simple imperative language designed specifically for the game. When players begin, they are told a story that motivates the game: there has been a chemical spill in a small town that has caused all the locals to evacuate and is threatening the local wildlife. The only thing that can safely protect the animals and clean the spill is a small robot capable of identifying and solving problems. Unfortunately, the robot was damaged during transportation, and now struggles to complete its missions, generating programs that *almost* solve the problems, but not quite. It is up to the learner to help the robot by figuring out and fixing the problematic code it generates. In this sense, the learner and the robot are a team, working together to save animals, clean up the spill, and ultimately shut down the hazardous chemical factory.

The primary activity in the game is to learn how to communicate with the robot via commands to help it accomplish a series of goals. The levels, goals, language, and user interface, however, were designed to teach specific aspects of algorithm design. The first 9 levels focus on teaching the 7 basic commands in the robot's syntax grammar and variations on how they can be written, each containing some invalid syntax that the learners must understand and correct. The subsequent 9 levels teach useful design patterns for composing these commands to achieve more powerful behaviors, each containing some semantic error in the ordering of the composite command sequences. In each level, one or more goals (Figure 2.2) are specified in terms of executable tests.

Table 1 explains Gidget's 7 commands. Learners were able to access a similar syntax reference as Table 1, but without the explanations, through the ? button at the top right of the editor. Each of the 7 commands could be followed by a ',' and subsequent command, allowing Gidget to iterate over a set of things with a given name. For example, if there were multiple kittens in Figure 2, the command "goto kitten, grab it" would iteratively go to each kitten, grab the kitten, and then go to the next kitten. The 'focus' stack in Figure 2.5 determines how the keyword 'it' is resolved; the 'results' stack in Figure 2.5 tracks matching names for each command.

In the game, Gidget programs are primarily capable of findings things in the 'world' (Figure 2.6), going to them, checking their properties, and moving them to other places on the grid. In some cases, objects have their own abilities, which Gidget can invoke like a function. After each execution step, the effect of these commands are shown in the 'memory' pane (Figure 2.5) and explained by Gidget (Figure 2.4) to reinforce the semantics of each command. Each step also costs Gidget 1 unit of '*energy*' (displayed at the top of Figure 2.5), forcing learners' to carefully consider how to write their code to complete each level within the allotted number of energy units.

In each condition, the robot is detailed in its interpretation of each command in its program. Not only does it explain what action it is taking in each step (Figure 3) and visualize these changes to the data structures it maintains in memory (Figure 2.5) to support its

Table 1. Gidget command syntax and semantics.

scan *thing*
Enables Gidget to **goto** all things with name *thing*. Scanned things are added to the set named *scanned* in Gidget's memory.

goto *thing1* [**avoid** *thing2*]
Moves Gidget to all of the things matching the name *thing1*, one square at a time If a thing to **avoid** is given, for each step that Gidget takes, he attempts to find a path that stays at least 1 square away from things with the name *thing2*.

analyze *thing*
Enables Gidget to **ask** all things with name *thing* to perform an action. Analyzed things are added to the set named *analyzed* in Gidget's memory.

ask *thing* **to** *action thing* *
Causes *thing* to perform *action*, if action is defined. Zero or more things are passed as arguments. Gidget's execution is suspended until the thing asked has completed requested action.

grab *thing*
Adds all things with name *thing* to the set named *grabbed* in Gidget's memory, removing them from the grid and constraining their location to Gidget's location.

drop *thing*
Removes all things with name *thing* in that were previously grabbed from the set *grabbed* set.

if *thing* **is[n't]** *aspect*, *command*
For each thing with name *thing* that has been analyzed, execute the specified command if that thing contains an aspect of name *aspect*.

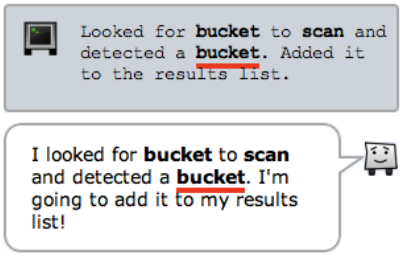

Figure 3. The two communication styles used to express either a positive or neutral affect. Positive affect is conveyed through the robot's facial expressions.

execution, but when it arrives at a command that it does not recognize or a command with missing information, it explicitly highlights this missing information and explains to the learner what interpretation it is going to make of the ambiguous command before proceeding (Figure 1). Moreover, in the case of parsing errors, the system opens up a syntax guide mentioned previously, highlighting the syntax rule that Gidget guessed was being used.

To aid the players with debugging, we implemented four execution controls for the code: *one step*, *one line*, *all steps*, and *to end* (Figure 5). The *one step* button evaluates one compiled instruction in the code, just like a breakpoint debugger does, but also displaying text describing the execution of the step (Figure 3). The *one line* button evaluates all steps contained on one line of the code, jumping the the final output of that line immediately. The *all steps* button evaluates the entire program and the goals in one button press, but animates each step. The *to end* button does the same as *all steps*, but does not animate anything.

3.1.2 Control vs. Experimental Condition

Personification of the robot's appearance was a key manipulation in our experiment. In the control condition, Gidget was designed to be a cold, emotionless computer terminal – something that the player would feel minimal emotional attachment towards. In contrast, in the experimental condition, Gidget was designed to be more human-like – a cute, unconfident robot with changing facial expressions based on the success of its execution. In the control condition, Gidget had two distinct states: an error/fail state that was shown during any syntax or runtime error, and a neutral state that was shown otherwise (Figure 4). The error state, with its large, jarring stop icon, attempts to capture the style common to compiler error messages. In contrast, the experimental condition had three distinct states for Gidget: an error/fail state that was shown during any kind of error, a success state that was displayed when a goal was completed, and a neutral state that was shown otherwise (Figure 4). These facial expressions were specifically designed to make Gidget more human-like and add affect to its messages throughout the game (Figure 3).

In both conditions, Gidget was designed to be verbose to help players know what was going on with the code during execution. The messages in the control condition were terse, actionable facts about the program state, presented in conventional fixed-width *Courier New* font. The text in the experimental condition contained the exact same information, using the softer, sans-serif *Verdana* font (Figure 3), but was personified in three specific ways. We started with the control text, then followed one or more of these rules: use a

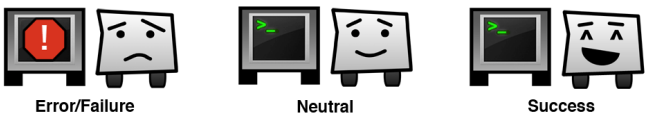

Figure 4. Representations of Gidget based on its game condition - control (*left*), and experimental (*right*) - and state.

personal pronoun (e.g. "I," "you"), admit failure (e.g. "I don't know this command"), and express affect (via exclamation points and emoticons). Examples include:

Control: "Unknown command, so skipping to next step."
Experimental: "I don't know what this is, so I'll just go on to the next step."

Control: "Dropped cat. Removing from memory banks."
Experimental: "I dropped the cat. I'll remove it from my memory."

Control: "ERROR: Nothing to *ask* by that name."
Experimental: "Hmm... I couldn't find anything to *ask* by that name."

The dialogue pane between Gidget and the player exhibit another major difference between the two conditions. In the control condition, the player is portrayed as a satellite dish (Figure 5) to signify that there is a large physical distance between the learner and robot, requiring radio communication. In the experimental condition, players are given the choice between three avatars (Figure 5) to represent themselves. This image is used in place of the satellite dish from the control condition, signifying that there is closeness and teamwork between Gidget and the player.

Next, the shape of the communication text boxes are different between the two conditions (as seen in Figure 3). The control condition was designed to look visually cold and direct. In contrast, the experimental condition used comic speech-bubbles for both Gidget and the player with the intention of having the exchange look like a conversation (Figure 3). These themes were extended to other parts of the interface, where the control condition's interface boxes have shaper curves than their experimental condition counterparts, which have larger, rounded corners.

Furthermore, there were labeling differences between conditions. First, level titles in the experimental condition were composed of the control conditions' level name with the addition of "Gidget" to add agency. For example, level 1 was titled "Testing Scanner" or "Testing Gidget's Scanner," and level 5 was titled "Utilizing Special Items" or "Using Special Items with Gidget." In the same manner, the memory pane was labeled "Memory banks" in the control condition, and "Gidget's memory" in the experimental condition.

Finally, sound effects were played in both conditions when Gidget performed an action or when a major event, such as Gidget running out of energy or Gidget not completing his goals, occurred. They were designed to supplement the text and provide additional depth to the world as Gidget moved through it (Figure 2.6). All sound effects were identical between conditions, except the general error and parser error sounds, which were manipulated to evoke different feelings. Errors in the control condition used sounds similar to those heard in operating systems when a critical error occurs. In contrast, errors in the experimental condition used sounds to attract players' attention without making it seem like the computer was "yelling." These sounds were deliberately chosen to add or subtract from the personification between the two conditions.

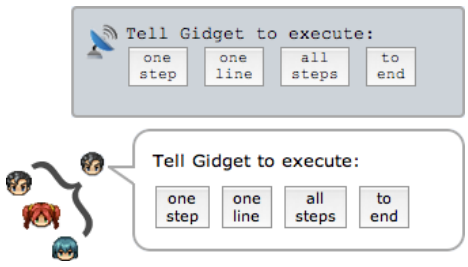

Figure 5. Communications pane representing the user in the control (*top*) and experimental (*bottom*) conditions. Players in the experimental condition are given the option to choose an avatar to represent themselves when they start the game.

3.2 Recruitment

Previous studies have shown effects due to giving computers personality traits in adult populations of varying ages [17,40,41]. We focused on replicating these studies in programming tools for adults of a similar age range. To recruit these individuals, we used Amazon.com's Mechanical Turk, an online marketplace where individuals can receive micro-payments for doing small tasks called Human Intelligence Tests (HITs). It is an attractive platform for researchers because it provides quick, easy access to a large workforce willing to receive a small monetary compensation for their time [46]. Since workers are sampled from all over the globe, Mechanical Turk studies have the benefit of generalizing to varied populations more than samples from limited geographic diversity that are more common in traditional recruiting methods [26]. However, due to the nature of the low monetary compensation and anonymity of the workers, careful consideration has to be taken to ensure that participants are not "gaming the system" [13,26]. To address this, we required that participants complete at least one level to receive credit for the HIT, ensuring that they actually had to interact with Gidget and the code before being allowed to quit.

3.3 Pricing & Validation

Since our game had a total of 18 levels, we decided that we would compensate our participants with a base rate and a nominal bonus payment for each level they completed. Previous studies have found that higher payment does not necessarily equate to better results [20], so we wanted to calibrate our payments to established market prices. To do this, we observed Mechanical Turk HITs tagged "game" for a period of 14 days. These HITs were further filtered to include only those that had an actual gameplay element as the main component as opposed to tasks such as writing reviews for third-party games. From these HIT descriptions, we constructed a list of 'reward' and 'time allotted' values, along with any explicit bonus payments mentioned. Our goal was to set a base reward that was high enough to attract participants, but also as low as possible to minimize participants' sense of obligation to spend time on our HIT. Likewise, we wanted our bonus payment per stage to have a minimal factor on participants' decision to continue the game.

Based on our data, we determined our optimal base reward as 30¢ for starting the HIT, and an additional 10¢ for each level completed. To ensure participants actually tried the game, we required that they complete at least one level to get paid. This meant the minimum compensation any participant received was 40¢. Participants were not informed of the total number of levels, eliminating this factor from their decisions to continue playing the game. Finally, we deliberately avoided mentioning anything about programming in the HIT description and tags to prevent people from self-selecting out of the HIT because of its association with programming. However, since the HIT description included the words "game" and "robot," we may have introduced some gender-biased self-selection effects.

To further validate our pricing model and detect defects and usability problems in the game, we conducted a pilot test on Mechanical Turk with 12 paid participants. In addition, an informal, 4-participant, lab study was conducted to gather information that we could not capture from Mechanical Turk. In this lab study, participants were asked to think-aloud while playing the game to test the clarity of the instructions and observe any problems they had with the interface. Observational study participants were volunteers and were not compensated for their time.

The pilot study results verified that participants were willing to complete levels and that the system functioned as-intended overall. Based on the data we received, we clarified some of the post-game survey questions and fixed several minor defects. We also set the ceiling for submission time to 3 hours to make the HIT less intimidating, as setting it too high could be misinterpreted by potential participants as the task being overly difficult. The observational study surfaced unclear instructions, confusing interface elements, defects, and usability problems in the game. Based on this information, we improved the text and interface elements, running another pilot to ensure that the usability and clarity of the game had improved.

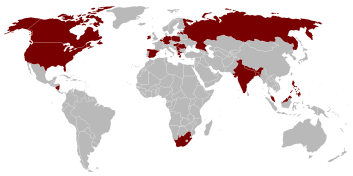

Figure 6. Geographical distribution of the 116 novice programmers in our study, spanning 24 countries.

3.4 The Participants

On game load, each participant was randomly assigned one of two conditions: control or experimental. This information, along with their current state in the game were logged on the client-side to ensure participants would not be exposed to the other condition, even if they refreshed their browser. Once a participant chose to quit, they were given a post-survey and a unique code to receive payment for their submission. The survey was designed to get demographic information (e.g. gender, age, education, country), identify prior programming experience, and solicit feedback and attitudes about the game. In addition to the survey responses, we automatically collected the following information from each participant upon quitting: the number of levels completed; time stamps for level start, level complete, quit, and any execution button invocations;, all character-level edits to each level's program, execution button presses, game condition, choice of user avatar (if in the experimental group), and payment code.

We defined "novice programmers" as participants who reported in the survey that they have never had: 1) "taken a programming course," 2) "written a computer program," or 3) "contributed code towards the development of a computer program." This information was cross-validated with an additional question later in the survey that asked them to rate their agreement with the statement, "I identify myself as a beginner/novice programmer."

Because we deliberately chose not to mention anything about programming in our HIT description, we were not able to control for a specific target audience. Therefore, we recruited a large sample of 250 participants from Mechanical Turk, with 116 meeting our criteria as being novice programmers.

Since the scope of this paper is how personification of the computer and its feedback affects novice programmers, these 116 participants are the primary focus of our analysis. This was a balanced, between-subjects design with 58 participants in each condition. Demographic data revealed that there that participants from the control and experimental conditions were well proportioned, with no significant differences between groups by gender, age, or education. There were a total of 50 females and 66 males with a mean age of 27.5 (SD=8), ranging from 18 to 59 years old. As shown in Figure 6, participants were spread across 24 countries, with most participants coming from the USA (27.6%) followed closely by India (22.4%). About 13.8% of participants were the lone representatives of their respective countries. Many did not provide geographical data (24.1%). Consistent with other Mechanical Turk study demographics, our sample of novice programmers were well-educated [13,26], answering that their highest level of education achieved was: less than high school (<1%), high school (13%), some college (23%), an associates degree (3%), a bachelor's degree (38%), a masters degree (14%), or a doctoral degree (6%).

4. RESULTS

In this section, we provide quantitative evidence for a number of patterns based primarily on the 116 logs and survey responses collected from the participants identified as novice programmers. Our dependent measures were not normally distributed so non-parametric tests were used for analyses. Our level of confidence was set at α=0.05.

4.1 Difference in Levels Completed

The minimum and maximum number of levels completed for both conditions were the same, at 1 and 15, respectively. The median number of levels completed for the control and experimental conditions were 2 and 5, respectively. There was a significant difference in the number of levels participants completed between the two conditions (Wilcoxon rank sums: W=3803, Z=2.3, N=116, p<.05) – meaning that we reject our null hypothesis.

The distribution of 'levels completed' (Figure 7) shows that a large number of participants from both groups quit the game after completing the first level. This was particularly true for those in the control group, who lost 41.3% of their members in contrast to the 29.3% lost by the experimental group. The large drop off in the sixth level for both conditions will be addressed in the discussion section, below. Since all participants were classified as novice programmers and there was no statistical difference in demographics, this suggests that our personification of Gidget in the experimental condition had a positive effect on participants' motivation to play.

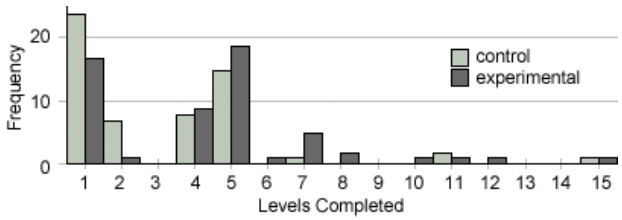

Figure 7. Histogram of *levels completed* for each condition.

4.2 No Difference in Play Time

The minimum time spent playing the game for the control and experimental condition was 5.4 minutes and 8.4 minutes, respectively. The maximum time spent playing the game was 2.81 hours and 2.97 hours respectively. The median overall play time for the control and experimental conditions were 27.1 minutes and 35 minutes, respectively. There was no significant difference in the length of time participants in either condition played the game overall (W=3689.5, Z=1.6, N=116, n.s.).

Since the previous result showed that the experimental group completed more levels than the control group, we checked to see if participants in either group were spending more time per individual level. To do this, we calculated the median time each participant took to complete the levels they attempted, and then compared the two resulting distributions of medians. We found that there was no significant difference in the median time to successfully *complete* levels between conditions (W=3407.5, Z=0.08, N=116, n.s.). Likewise, there was no significant difference in the time participants spent on the level they attempted, but did *not* complete (W=3387.5, Z=-0.03, N=116, n.s.).

The difference in levels completed, but the lack of significant difference in playing time suggests that those in the experimental condition learned commands (i.e., by completing more levels) more efficiently. This suggests that something in our manipulation caused the experimental condition participants to better understand and use the commands to fix Gidget's problematic code. We address possible explanations for this in our discussion.

4.3 No Difference in Execution

There were no significant differences in how frequently the participants used the four execution control buttons overall (*one step*: W=3693.5, Z=1.7, n.s., *one line*: W=3532, Z=0.8, n.s., *all steps*: W=3488, Z=0.5, n.s., *to end*: W=3740, Z=1.9, n.s.; N=116).

Since we found previously that the experimental group completed more levels than the control group, we checked to see if there was a difference between the conditions for the number of code executions used per individual level. To do this, we calculated the median number of code executions each participant used to complete the levels they attempted, and then compared the two resulting distributions of medians. This was repeated for each execution button. We found that there were no significant differences in the median number of code executions for *completed* levels by condition (*one step*: W=3293, Z=-0.5, n.s., *all steps*: W=3061.5, Z=-1.9, n.s., *to end*: W=3305.5, Z=-0.5, n.s.; N=116). However, we found that the use of *one line* was significantly different: W=2987.5, Z=-2.3, N=116, p<.05. On closer inspection of the data, we found that this difference was due to participants in the control condition using a higher (median) number of *one line* code executions. This means that participants in the control condition were running their code line-by-line, but skipping some of the finer details provided by the *one step* execution.

Finally, we checked both conditions to see if there was a difference in the raw number of code executions for levels the participants attempted but did not complete. We found that there were no significant differences between conditions in the number of code executions for levels that participants attempted but did *not* complete (*one step*: W=3339.5, Z=-0.3, n.s., *one line*: W=3310, Z=-0.5, n.s., *all steps*: W=3303.5, Z=-0.5, n.s., *to end*: W=3483, Z=0.5, n.s.; N=116). Since participants quit on different levels of varying difficulty, this suggests that those from both conditions put approximately the same amount of effort into testing and executing their code before deciding to give up, independent of the level they were playing.

4.4 Differences in Survey Feedback

There was no significant difference in participants' self-reported level of enjoyment playing the game between the two conditions (W=3117, Z=-1.1, N=116, n.s.). Likewise, there was no significant difference in participants' reporting whether they would recommend the game to a friend wanting to learn programming (W=3629.5, Z=1.4, N=116, n.s.). These results are consistent with reports by Nass et. al, who found that participants did not attribute success or enjoyment of an activity to changes in their performance [40].

There was, however, a significant difference in participants' reporting that they wanted to help Gidget succeed (W=3901, Z=3.1, N=116, p<.01). Participants in the experimental condition were significantly more likely than those in the control condition to agree to the statement, "I wanted to help Gidget succeed."

4.5 Comparison to Experienced Programmers

To contrast our reported findings, we briefly present data from participants who did not qualify as novice programmers. These were the participants who reported in the survey that they have: 1) taken a programming course, 2) written a computer program, or 3) contributed code towards the development of a computer program. After trimming data that was unusable due to data transmission errors, we had 120 participants with prior programming experience, 61 in the control condition, and 59 in the experimental condition.

This experienced group completed a wider range of levels in both conditions, with the median number of levels completed for the control and experimental conditions being 5 and 4, respectively. However, unlike our major finding from the novice group, there was not a significant difference in the number of levels completed between the two conditions in the experienced group (W=3392.5, Z=-0.9, N=120, n.s.). Likewise, there was no significant difference in the overall play time between conditions (W=3376, Z=-1.0, N=120, n.s.).

Like the novice programmers, there was no significant difference in experienced participants' self-reported level of enjoyment playing the game between the two conditions or whether they would recommend the game to a friend. In addition, unlike the novice programmers, the experienced participants did not show a significant difference in wanting to help Gidget succeed (W=3624, Z=0.3, N=120, n.s.).

Finally, there were no significant differences in how frequently the participants used three of the four execution control buttons overall (*one step*: W=3256.5, Z=-1.6, n.s., *one line*: W=3398.5, Z=-0.9, n.s., *all steps*: W=3751, Z=0.9, n.s.; N=120). However, there was a significant difference in the number of code executions for *to end*: W=3139, Z=-2.2, N=120, $p<.05$, where participants in the control condition used the button more frequently than those in the experimental condition. This suggests that the experienced programmers attempted to treat the game more as a traditional edit-compile-run cycle, rather than debugging the program step-by-step.

5. DISCUSSION

Our findings demonstrate that more personified programming tool feedback can increase novice programmers' motivation to program. More specifically, we have shown that casting the computer as a verbose but naïve and unconfident teammate that blames itself for errors has demonstrated to have a positive effect on novice learners' performance in learning a simple textual programming language. We also found that novice programmers exposed to this unconfident teammate were more likely to report that they wanted to help it.

These results, combined with the lack of a significant difference in median time spent on levels or execution of the program, suggests that the experimental group was likely making better use of the information provided by the robot than the control group. One possible explanation for this is that by personifying the feedback provided by the programming environment, experimental group participants were more likely to *attend* to the information content in the messages, and thus more likely to understand the program semantics. This is supported by our finding that the control group participants were significantly more likely to use the "one line" execution control, skipping over many (but not all) of the robot's messages. Another interpretation is that both groups attended to the messages similarly, but the phrasing led the experimental group participants to somehow process the information more deeply, by framing it as human rather than computer. Future studies should explore these possible interpretations, isolating the effect of personification on attention to feedback.

Although our results suggest that our manipulation increased success on learning, we did not find that participants were willing to spend more time playing the game. This may be due to the unconstrained nature of Mechanical Turk tasks, which provide no additional extrinsic incentives to continue; it may also be due to difficulties that learners encountered in particular levels of the game. This was particularly true of level 6, where there was a major drop off of participants in both conditions (Figure 7). This level introduced conditional statements, suggesting that it is an inherently difficult concept for novice programmers to comprehend. More work needs to be done to uncover how feedback tool personification affects other aspects of motivation such as wanting to continue to work on a problem after multiple failures on a single level.

Our analysis of the performance of experienced programmers also suggests that the motivation and learning effects due to personification may diminish with experience. It is likely that the experienced programmers playing the game quickly learned the semantics of the language and did not need to read the feedback provided by the programming tool in order to complete each level.

5.1 Threats to Validity

Our study has a number of limitations that limit its generalizability. First, Mechanical Turk allows participants to self-select into HITs given that they meet certain qualifications. Our HIT did not require any special qualifications and used the default setting from Amazon. Although we tried to account for factors that would affect the HITs listing on Amazon's HIT page, those who filtered for higher-paying HITs would be less likely to find our HIT, whereas those filtering for a tag labeled "game" would be more likely to find our HIT.

Also, the game was accessible by computer, connected to the Internet, listed on a website requiring login. Although not directly translatable to programming ability, gaining access to the game requires a fair amount of computer knowledge. As our demographic data indicated, our participants were well-educated, with 86% of them reporting that they had some college education or beyond.

Finally, though small, there was an economic incentive for participants to participate in the study. Moreover, they would receive a bonus payment for levels they completed. Since these economic incentives would not exist in a place like a classroom, it is unclear how or findings would generalize to other extrinsically motivated learning contexts. For instance, Mechanical Turk turk users have a choice of which tasks to engage in; students in a classroom often do not.

6. CONCLUSIONS & FUTURE WORK

We have presented Gidget, a game intended for novice programmers who are tasked with helping a damaged robot complete its missions by debugging its defective code. By personifying the robot – characterizing it as fallible, having it convey information about coding errors conversationally, and having it take the blame for mistakes – we have found that novice programmers complete more game levels than learners who received more conventional feedback, in a comparable amount of time. Given our results, we conclude that personifying the computer and making it less authoritative has many immediate motivational and learning benefits for novices wanting to learn how to program.

Our results also suggest several directions for future work. We want to further refine our gameplay elements to better understand exactly which specific manipulations accounted for the experimental group's players to complete more levels in the same amount of time. For example, were they were more attentive to Gidget's personified text, or was Gidget's face? Adding more instructional content or altering the economic incentive (from Mechanical Turk) may also yield additional insights. We also want to explore the effectiveness of Gidget in introductory programming courses to see if our results hold with students in a classroom setting. If these findings can be replicated in other learning contexts, they may have a significant effect on how feedback is provided to learners in a wide range of computing education contexts.

7. ACKNOWLEDGEMENTS

We would like to thank the second author's daughter, Ellen Ko, for her extensive input on game dynamics and level design. This material is based in part upon work supported by the National Science Foundation under Grant Number CCF-0952733.

8. REFERENCES

1. Amazon Mechanical Turk. http://www.mturk.com
2. Atlas, G.D., Taggart, T., & Goodell D.J. (2004). The effects of sensitivity to criticism on motivation and performance in music students, *British J. of Music Ed*., 21(1), 81-87.
3. Bandura, A. (1977). Self-efficacy: Toward a unifying theory of behavioral change. *Psychological Review*, 84: 191-215.
4. Bandura, A. (1986). Social foundations of thought and action: A social cognitive theory, *Englewood Cliffs*, NJ: Prentice-Hall.
5. Barnes, T., Richter, H., Powell, E., Chaffin, A., & Godwin, A. (2007). Game2Learn: building CS1 learning games for retention. *ITiCSE*, 121-225.
6. Beckwith, L., Burnett, M., & Cook, C., (2002). Reasoning about Many-to-Many Requirement Relationships in Spreadsheet Grids, *IEEE VL/HCC*, 149.
7. Begel, A. (1996). LogoBlocks: A Graphical Programming Language for Interacting with the World. *EECS*, *MIT*.
8. Boyce, A., & Barnes, T. (2010). BeadLoom Game: Using Game Elements to Increase Motivation and Learning. *FDG*, 25-31.
9. Braught, G., Eby, L.M., & Wahls, T. (2008). The effects of pair-programming on individual programming skill. *SIGCSE*, 200-204.
10. Bruckman, A. (1997). MOOSE Crossing: Construction, Community, and Learning in a Networked Virtual World for Kids. *MIT Media Lab*. Boston, MA.
11. Bruckman, A., Biggers, M., Ericson, B., McKlin, T., Dimond, J., DiSalvo, B., Hewner, M., Ni, L., & Yardi, S. (2009). Georgia Computes!: Improving the Computing Education Pipeline. SIGCSE, 86-89.
12. Chaffin, A., & Barnes. T (2010). Lessons from a course on serious games research and prototyping. *FDG*, 32-39.
13. Downs, JS., Holbrook, MB, Sheng, S., & Cranor, L.F. (2010). Are your participants gaming the system?: screening mechanical turk workers. *ACM CHI*, 2399-2402.
14. Dweck, C. S. (1999). Self-Theories: Their role in motivation, personality, and development. *The Psychology Press.*
15. Fenton, J. and Beck, K. (1989). Playground: An Object Oriented Simulation System with Agent Rules for Children of All Ages. *ACM OOPSLA*, 123-137.
16. Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L. and Zander, C., (2008) Debugging: finding, fixing, and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*. v18. 93-116
17. Fogg, B. J., & Nass, C. (1997). How users reciprocate to computers: an experiment that demonstrates behavior change. *ACM CHI*, 331-332.
18. Gross, P. and Kelleher, C., (2010). Non-programmers identifying functionality in unfamiliar code: strategies and barriers. *JVLC 21, 5*, 263-276.
19. Harel, I. Children Designers. (1991) *Ablex Publishing, N.J.*
20. Hsieh, G., Kraut, RE, & Hudson, SE. (2010). Why pay?: exploring how financial incentives are used for question & answer. *ACM CHI*, 305-314.
21. Jeffries, R. (1982). A comparison of the debugging behavior of expert and novice programmers. *AERA Annual Meeting*.
22. Kay, A., Etoys and Simstories. http://www.squeakland.org
23. Kelleher, C. and Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM CSUR*, 37(2),83-137.
24. Kelleher, C., Pausch, R., & Kiesler, S. (2007). Storytelling Alice Motivates Middle School Girls to Learn Computer Programming. *ACM CHI*, 1455-1464.
25. Kinnunen, P., & Simon, B. (2010). Experiencing programming assignments in CS1: the emotional toll. *ICER*, 77-86.
26. Kittur, A., Chi, E.H., & Suh, BW. (2008). Crowdsourcing user studies with Mechanical Turk. *ACM CHI*, 453-456.
27. Klein, J., Moon, Y., Picard, R.W. (1999). This computer responds to user frustration. *ACM CHI*, 242–243.
28. Ko, A. J., Myers, B. A., & Aung, H. (2004). Six Learning Barriers in End-User Programming Systems. *IEEE VL/HCC*, 199-206.
29. Ko, A. J. & Myers B.A. (2009). Attitudes and Self-Efficacy in Young Adults' Computing Autobiographies. *IEEE VL/HCC*, 67-74.
30. Kulesza, A. (2009). Approximate learning for structured prediction problems. *UPenn WPE-II Report*.
31. Light-Bot. http://armorgames.com/play/2205/light-bot
32. Linderbaum, B. (2006) The Development and Validation of the Feedback Orientation Scale. *J. of Management*, 1372-1405.
33. Lionet, F., & Lamoureux, Y., *Klik and Play*, Maxis, 1994.
34. Logo Computer Systems, Inc., *My Make Believe Castle*, 1995.
35. Maloney, J., Resnick, M., Rusk, N., Silverman, B., Eastmond, E. (2010). The Scratch Programming Language and Environment. *ACM TOCE*.
36. McDowell, C., Werner, L., Bullock, H., & Fernald, J. (2002). The effects of pair-programming on performance in an introductory programming course. *SIGCSE*, 38–42.
37. MindStorms. http://www.mindstorms.lego.com
38. Murphy, L., Fitzgerald, S., Hanks, B., & McCauley, R. (2010) Pair debugging: a transactive discourse analysis. *ICER*, 51-58.
39. Murphy, L. and Thomas, L. (2008). Dangers of a fixed mindset: Implications of self-theories research for computer science education. *ITiCSE*, 271-275.
40. Nass, C., Fogg, B.J., & Moon, Y. (1996). Can computers be teammates? *International J. of Human-Computer Studies*, 45, 669-678.
41. Nass, C. (2000). Machines and Mindlessness: Social Responses to Computers. *J. of Social Issues*, 56, 81-103.
42. Pane, J. Myers, B.A., & Miller, L.B. (2002). Using HCI Techniques to Design a More Usable Programming System. *IEEE VL/HCC*, 198-206.
43. Papert, S. Mindstorms: Children, Computers, and Powerful Ideas. *Basic Books New York, NY.*
44. Resnick, M., Martin, F., Sargent, R., & Silverman, B. (1996). Programmable Bricks: Toys to Think With. *IBM Systems J.*, vol. 35, no. 3-4, 443-452.
45. Roberts, T.A. (1991). Gender and the influence of evaluations on self-assessments in achievement settings. *Psychological Bulletin*, vol. 109(2), 297-308.
46. Ross, J., Irani, I., Silberman, M. Six, Zaldivar, A., & Tomlinson, B. (2010). Who are the Crowdworkers?: Shifting Demographics in Amazon Mechanical Turk. *ACM CHI*, 2863-2872.
47. Smith, D., Cypher, A., & Tesler, L. (2002). Programming by example: novice programming comes of age. *CACM*, 75-81.
48. Tanimoto, S., & Runyan, M. (1986). Play: an iconic programming system for children. *Visual Programming Environments*, 367-377.
49. Tarkan, S., Sazawal, V., Druin, A., Golub, E., Bonsignore, E.M., Walsh, G., & Atrash, Z. (2010). Toque: designing a cooking-based programming language for and with children. *ACM CHI*, 2417-2426.
50. Vihavainen, A., Paksula, M., & Luukkainen, M. (2011). Extreme apprenticeship method in teaching programming for beginners. *SIGCSE,* 93-98.