Pyrus: Designing A Collaborative Programming Game to Support Problem-Solving Behaviors

Joshua Shi*
Northwestern University
joshshi@u.northwestern.edu

Armaan Shah*
Northwestern University
armaanshah96@gmail.com

Garrett Hedman
Northwestern University
ghedman@u.northwestern.edu

Eleanor O'Rourke
Northwestern University
erourke@northwestern.edu

ABSTRACT

While problem solving is a crucial aspect of programming, few learning opportunities in computer science focus on teaching problem-solving skills like planning. In this paper, we present Pyrus, a collaborative game designed to encourage novices to plan in advance while programming. Through Pyrus, we explore a new approach to designing educational games we call *behavior-centered game design*, in which designers first identify behaviors that learners should practice to reach desired learning goals and then select game mechanics that incentivize those behaviors. Pyrus leverages game mechanics like a failure condition, distributed resources, and enforced turn-taking to encourage players to plan and collaborate. In a within-subjects user study, we found that pairs of novices spent more time planning and collaborated more equally when solving problems in Pyrus than in pair programming. These findings show that game mechanics can be used to promote desirable learning behaviors like planning in advance, and suggest that our behavior-centered approach to educational game design warrants further study.

CCS CONCEPTS

• **Human-centered computing → Collaborative and social computing systems and tools**;

KEYWORDS

Educational Games, Collaborative Learning, CS Education, Problem Solving, Behavior-Centered Game Design

*The first and second authors contributed equally to this paper

CHI 2019, May 4–9, 2019, Glasgow, Scotland Uk

ACM ISBN 978-1-4503-5970-2/19/05…$15.00
https://doi.org/10.1145/3290605.3300886

ACM Reference Format:
Joshua Shi, Armaan Shah, Garrett Hedman, and Eleanor O'Rourke. 2019. Pyrus: Designing A Collaborative Programming Game to Support Problem-Solving Behaviors. In *CHI Conference on Human Factors in Computing Systems Proceedings (CHI 2019), May 4–9, 2019, Glasgow, Scotland Uk*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3290605.3300886

1 INTRODUCTION

Programming has historically been, and continues to be, notoriously difficult for novices to learn and master [32]. Despite growing interest in computer science due to the industry's demand for programmers, drop-out rates in CS courses are still high [5, 7, 51, 59]. One reason novices struggle with programming is that they often lack the required problem-solving skills to formulate and plan complex programs [43]. Yet while many researchers have argued that problem-solving is a crucial aspect of programming [19, 24, 57], few learning opportunities in computer science focus on teaching problem-solving skills like planning [40].

To remedy this, researchers have begun to explore approaches for scaffolding and teaching novices problem-solving skills in the context of computer science. Some have explored giving students explicit guidance on effective problem-solving approaches [40] and designing curricula around problem-based learning to emphasize problem-solving skills [34]. Others have proposed using collaboration as a way to help novice programmers solve challenging problems more effectively, most commonly in the form of pair programming [45, 62]. Despite these efforts, problem-solving remains largely ignored in introductory CS courses, and we have more to learn about effective approaches for teaching problem-solving skills in computer science.

One potentially effective approach for teaching problem-solving is through serious games, or games designed for a primary purpose other than entertainment. Video games are known for their ability to motivate players to develop complex problem-solving skills, and previous analyses of games have shown their potential to achieve educational goals [14, 25, 42, 52]. However, empirical studies evaluating the effectiveness of educational games have yielded mixed results [16, 42, 52, 64]. Researchers have noted that successful games are typically designed around pedagogical principles

[42, 52], but we still lack a design process for effectively embedding pedagogical principles into games.

In this paper, we present Pyrus, a collaborative programming game that encourages novices to plan before implementing code. Pyrus uses collaboration to encourage novices to discuss their ideas, and leverages game mechanics like distributed resources, a failure condition, and enforced turn-taking to incentivize advance planning and effective collaboration. Through Pyrus, we explore a new approach to designing educational games we call *behavior-centered game design.* Using this approach, game designers first identify specific behaviors they want learners to practice to reach desired learning goals, and then choose game mechanics that promote those behaviors. The aim of behavior-centered game design is to ensure that the core game-playing strategies align with the targeted learning goals.

We evaluated Pyrus through a within-subjects study (n = 18) with two conditions, in which pairs of novice undergraduate students worked on programming challenges in both Pyrus and a pair programming control condition. Through a mixed-methods analysis of conversation, behavioral data, and interviews, we found that pairs spent nearly twice as much time planning their solutions in Pyrus than they did in pair programming, and that Pyrus encourages them to participate more equally. While some participants found Pyrus to be fun and enjoyable, others were frustrated by the mechanics and felt that the game's constraints caused them to work slowly and inefficiently, suggesting a need for further iteration. These findings show that an educational game can encourage novice programmers to practice problem-solving skills, and provides initial evidence that educational games designed using a behavior-centered approach can effectively incentivize desired learning behaviors.

2 RELATED WORK

Researchers have studied problem-solving in the domain of programming extensively, and have explored a variety of approaches for supporting novices as they develop problem-solving skills. We first review related work on problem-solving and collaborative learning, then discuss educational games in general and programming games specifically.

Problem Solving in Programming

Problem solving is a crucial skill for programmers [57]. Researchers have developed a variety of models of the programming problem-solving process, most of which include the following stages: (1) understanding the problem, (2) planning a solution, (3) implementing the solution, (4) and testing and debugging the solution [19, 40, 43, 63]. However, novices often struggle to develop the skills needed to execute this problem-solving process effectively. Research on novice programmers has repeatedly concluded that, unlike experts, novices do not practice problem-solving behaviors such as planning in advance [37, 54, 60, 63]. Given that novices struggle to develop problem-solving skills, researchers have argued that introductory courses and educational resources must go beyond teaching syntax and semantics [40, 57].

A variety of approaches have been explored to help novices develop problem-solving skills, including resources that scaffold problem solving [39, 40, 50], a new CS curriculum that focuses on problem-solving skills [34], and interventions that aim to help novices when they get stuck [12, 28, 49]. For example, Loksa et. al. found that students have higher self-efficacy and metacognitive awareness when provided a list of problem-solving stages as guidance [40]. Linn et. al. found that expert commentaries on the process of solving complex programming problems can help students learn problem-solving skills [39]. Cao et. al. provide context-sensitive suggestions to help stuck users through an "Idea Garden" [12].

We contribute to research by exploring a new approach for encouraging novices to practice problem solving. Rather than directly teaching or scaffolding the problem-solving process, Pyrus uses game mechanics to incentivize behaviors like planning in advance, deepening our understanding of different approaches for teaching problem solving.

Collaborative Learning

Collaborative learning has been studied extensively in the learning sciences [30], and research shows that the active discussion of ideas in small groups can promote both engagement and critical thinking [26]. However, empirical studies show that not all collaborative learning experiences are equally effective [20]. Dillenbourg and Schneider identified a set of mechanisms that make collaborations effective, including explaining reasoning, considering and comparing different approaches, and regulating mutual understanding of a problem [21]. Researchers have explored a variety of methods for structuring collaborative activities to foster effective collaborations, often referred to as *scripts* [20]. One common scripting approach, known as *Jigsaw*, gives each member of a group exclusive access to a necessary part of the problem solution to prevent any individual from being able to solve the problem alone [4]. Jigsaw has been used to foster collaboration in a variety of learning environments, including cognitive tutoring systems [46] and CS1 labs [56].

In computer science education, the most widely adopted method of introducing collaboration is through pair programming [44, 45, 62], a core tenant of the eXtreme Programming methodology [6]. In pair programming, two partners work together to solve programming problems: the *pilot* controls the keyboard and implements the solution, while the *co-pilot* checks the implemented code for errors and bugs. Research shows that pair programming helps students solve problems more quickly and effectively [62] and that novices who pair

program are more confident in their work than students who do not pair [45]. However, pair programming is not strongly scripted and the pilot and co-pilot roles can break down in practice [13]. We are not aware of any research that aims to re-design the pair programming experience to improve its collaborative effectiveness.

We contribute to this body of research by exploring a new method of structuring collaboration between pairs of programmers by using game mechanics to directly incentivize effective collaborative behaviors. Pyrus' design builds on scripting and Jigsaw by using mechanics like enforced turn-taking and distributed resources to promote collaboration.

Educational Games

An extensive body of prior work has investigated the potential of teaching programming through educational games. Researchers and practitioners have designed games that teach language constructs [3, 22], computational thinking and algorithm design [9, 29, 31, 33, 47], and program comprehension via testing and debugging [8, 38]. For example, in Wu's Castle, novice programmers learn language constructs by writing arrays and loops to accomplish game goals [22], and in Code Hunt, players practice debugging skills by solving puzzles given only clues and test cases for the target algorithm [8]. While games have been developed to teach various aspects of programming, we are not aware of any that specifically promote planning during problem solving.

Researchers agree that educational games have potential to support learning [14, 25, 42], but we still have a limited understanding of how to integrate learning theory into games to create effective educational experiences. As a result, researchers have created resources to help educational game designers. Aleven et. al. developed a framework that combines learning objectives and instructional design principles with game mechanics, dynamics, and aesthetics [1]. Culyba developed a framework for designing *transformational* games that change their players by identifying desired transformations (e.g. learning outcomes) and the barriers that inhibit them [17]. Wendel et. al. present a set of requirements for the design of collaborative learning games [61]. These approaches focus on identifying high-level instructional principles and game affordances that could support learning, rather than understanding how to choose low-level game mechanics based on a stated behavioral goal.

In this work, we explore a new *behavior-centered game design* approach that focuses on identifying game mechanics that will promote desirable low-level learning behaviors. We see this approach as one which could compliment higher-level educational game design frameworks. Some educational games have been developed around stated behavioral goals: "brain points" were added to the math game Refraction to promote persistence and strategy [53], and GrACE teaches abstraction and algorithms by encouraging players to practice stepwise thinking [31]. We build on this work by formalizing the behavior-centered approach to game design.

3 BEHAVIOR-CENTERED GAME DESIGN

Our goal in this work is to design a game that encourages novice programmers to practice more programming problem-solving behaviors, in particular planning. To achieve this goal, we explore a new approach for creating educational games that we call *behavior-centered game design*. The key insight of this approach is that the game designer identifies specific low-level behaviors they want players to practice in order to reach high-level learning objectives, rather than targeting a high-level learning objective outright.

Specifically, behavior-centered game design is a process through which game designers (1) identify the obstacles learners face in reaching a learning outcome, (2) identify the behaviors that learners need to practice to reach the learning outcome, and (3) select game mechanics that will directly incentivize those behaviors to overcome the obstacles. These game mechanics are then assembled into a playable game which can be evaluated for its effectiveness in encouraging the targeted behaviors. For Pyrus, our desired learning outcome was for novice programmers to develop programming problem-solving skills, in part through effective collaboration. We detail our behavior-centered approach below.

Problem-solving

Obstacle: novice programmers struggle with the problem-solving process. Novices commonly make errors during problem solving and omit early stages of the process such as planning [36, 43, 58]. Without planning in advance, novices may work toward a solution with a poor understanding of the problem, which leads to mistakes like writing unnecessary code or not accounting for edge cases [23]. These types of errors are notoriously difficult for novices to debug [2]. In addition, a lack of planning means that novices are decomposing problems and composing solutions simultaneously, two steps which are already independently challenging [36]. These difficulties are present even when novices work in pairs [27].

Behavior: plan solutions in advance. To address these difficulties, we focused on incentivizing planning. In the planning stage of problem-solving, programmers devise a potential solution and evaluate it through processes like mental simulation or writing pseudocode. While programmers should plan before implementing a solution, the programming problem-solving process is not strictly ordinal and programmers may revisit the planning stage after implementing a partial solution. Our goal was to encourage planning in general, and planning in advance of implementation in particular.

Mechanics: discrete actions, distributed resources, and a failure condition. To encourage novices to plan their solutions in advance, we included three game mechanics in Pyrus that make it difficult for players to write programs without planning: *discrete actions*, *distributed resources*, and a *failure condition*. In Pyrus, players make progress by executing *discrete actions*, which are the building blocks that players use to compose their solutions. We hypothesized that discretizing the act of writing code using actions would encourage novices to work more deliberately, resulting in more planning. In addition, some programming constructs that players need to solve problems are only available if they possess the corresponding *distributed resources*, which are non-transferable and can only be used by the player who holds the resource. We hypothesized that distributed resources would encourage players to plan while determining how to use their respective resources most effectively. Finally, the number of actions a pair can execute is limited by a *failure condition*; if players do not solve the problem in a given number of actions, they lose the game. We hypothesized that the failure condition would create urgency and encourage players to use their discrete actions and distributed resources deliberately to avoid losing.

Collaboration

Obstacle: novices fail to collaborate effectively. While collaboration could encourage novices to discuss their ideas and plan more effectively, research on collaborative learning shows that people often struggle to collaborate without support. According to Dillenbourg et al., collaborative tasks must be structured to encourage effective interactions if they are to succeed [20, 21]. While pair programming is commonly employed in introductory computer science courses, this practice does not provide enough structure to guarantee effective collaboration. Partners are encouraged to take on the *pilot* and *co-pilot* roles, but these roles are not structured and switching between roles is often unenforced. In practice, research shows that the roles sometimes break down entirely, with little or no distinction between partners [13]. Furthermore, since partners take on the roles ad hoc, there is nothing to stop the pilot from dominating the conversation and implementation, leaving the co-pilot with nothing to do. Pairs are free to practice pre-established bad programming habits, even if they do so collaboratively.

Behavior: participate equally in problem solving. In the context of programming, we define an "effective collaboration" as one in which both partners are actively involved in discussing, designing, and implementing the solution to a problem. Rather than allowing one partner dominate by spending more time in the pilot role, we would like to see both partners participate equally in the problem-solving process and in particular the construction of the solution.

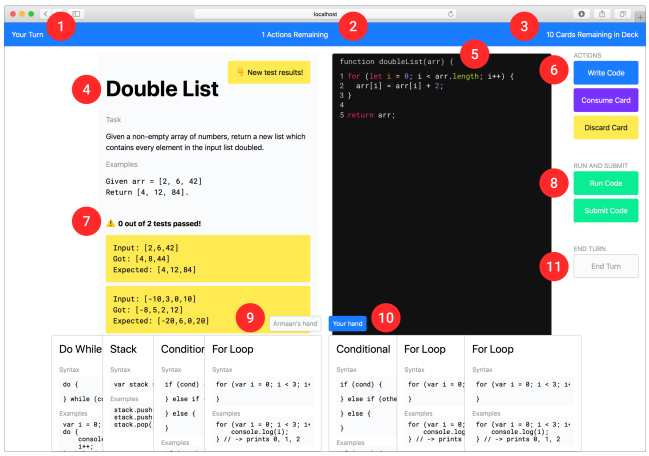

Figure 1: The Pyrus interface, which displays §1) the current player's turn, §2) the pilot's number of remaining actions, §3) the number of cards in the deck, §4) the problem prompt, §5) the editor, §6) available actions (i.e. write, consume, and discard), §7) test cases, §8) buttons to run or submit code, §9) the partner's hand, §10) the player's hand, and §11) a button to end the turn. §6 and §11 are omitted in the co-pilot's interface, since the co-pilot cannot perform actions.

Mechanics: enforced turn-taking and distributed resources. To encourage players to participate equally in problem solving, we included two game mechanics in Pyrus that foster positive interdependence, or the feeling that teammates are reliant on each other's success as part of their own [65]: *enforced turn-taking* and *distributed resources*. In Pyrus, *enforced turn-taking* forces players to switch roles at regular intervals. Furthermore, turns are designed to be short enough to keep both players actively engaged. We hypothesized that frequent turn-taking would encourage equal participation. In addition, *distributed resources* provide each player with only some of the programming constructs required to solve a problem. While turns enforce a division of labor, resources enforce a division of responsibilities. We hypothesized that making it impossible for one partner to succeed without the help of their teammate would encourage partners to discuss and coordinate, resulting in equal participation.

4 PYRUS SYSTEM

Pyrus (Figure 1) is a two-player game in which programmers collaborate in person to solve programming problems. As in traditional programming, players write and debug real code. Unlike traditional programming, players must work with special game mechanics and rules to solve their problem. In Pyrus, two programmers work together to write a program to solve a problem specified in a prompt (§4) with the goal of passing all provided test cases (§7). Each player works on their own computer, but they write code in a shared

editor (§5) toward a common goal (i.e. players win or lose together). During the game, players take turns writing code and observing (*pilot* and *co-pilot* roles, respectively).

Setup

Before play starts, each player is dealt four cards from a deck (§3). Players can see both their own (§10) and their partner's cards (§9), but may only use their own. Each card describes a programming construct (e.g. loop, conditional, data structure) and is non-transferable. These constructs can only be written by playing the appropriate card, and furthermore can only be written by the player who has that card in his or her hand. Once the hands are dealt, a player is selected by the system to be the pilot (§1) and play begins.

Play

On each turn, the pilot can perform up to four actions (§2). There are three types of actions (§6). The *write code* action allows the the pilot to write a single statement to declare a variable of a primitive data type or perform an operation on existing variable(s) (e.g. `var x = 12`). The *consume card* action allows the pilot to use a card in his or her hand to implement the corresponding construct in the editor. That card is then discarded. The *discard card* action allows the pilot to discard a card and draw a new one from the deck.

The pilot may elect to end his or her turn at any time (§11). At the end of each turn, the pilot draws two cards from the deck and the other player (the co-pilot) starts a new turn as the pilot. Players trade turns like this until they trigger either the win or failure condition. Players *win* if at any point during they game, they press the button to run their code against the test cases (§8), and all tests pass. Players *fail* if a player attempts to draw a card (either by using a *discard* action or automatically at the end of a turn) when the deck is empty. After failure the game is reset to the start state.

In Pyrus, the cards are *distributed resources*. The game uses *enforced turn-taking* to structure play, and players perform up to four *discrete actions* per turn. Play ends when the players complete a challenge or when the *failure condition* is triggered. These four game mechanics were designed to work in concert to encourage our two target behaviors: planning in advance and equal participation in problem solving.

Representative scenario

To illustrate a game of Pyrus, consider the following example game played by novice programmers JoJo and Arby. When JoJo and Arby join the game, each player is dealt four cards and JoJo is selected as pilot. JoJo is shown the interface in Figure 1. Arby's interface shows everything that JoJo's does except the three action buttons and the End Turn button.

The solution requires a loop and conditional, and while JoJo has a loop card, only Arby has a conditional. They decide that JoJo should use a Write Code action to declare a variable which will keep track of the loop's termination condition. JoJo clicks Write Code, declares a primitive variable (e.g. `var count = 0;`), and submits his action. Next, JoJo decides to implement his loop. He clicks Consume Card, selects his loop card, and writes `for (var i = 0; i < arr.length; i++) { }`. Now, both players agree that Arby should implement her conditional, so JoJo clicks End Turn. As JoJo's turn ends, he draws two cards, and his interface displays that Arby is now pilot. Arby's interface now gives her access to the actions, which she can use to write her conditional. After Arby consumes her conditional card to implement an if statement, JoJo and Arby decide to test their code, so JoJo clicks on the Run Code button. Their code is run against some test cases and the results are displayed below the prompt.

Play continues until JoJo and Arby deplete the deck, which triggers the failure condition and resets the game. Eventually, they pass all test cases and Arby clicks Submit Code to win.

5 STUDY DESIGN

To evaluate whether Pyrus encourages novices to plan and collaborate effectively, we conducted a within-subjects study in which pairs worked on programming challenges using both Pyrus and pair programming. The study was conducted as a half-day JavaScript bootcamp, meaning that everyone participated in the study on a single day. We were interested in evaluating (1) whether novices plan more in Pyrus, and (2) whether novices participate more equally in Pyrus.

Participants

Eighteen undergraduate students at a large private university (three female) participated in our study. They were recruited through a department mailing list. The study was presented as a bootcamp that would teach participants JavaScript fundamentals, give them an opportunity to practice solving programming challenges, and contribute to research. In order to maintain consistent level of experience, we invited students who had taken CS1 and CS2 but had no experience with JavaScript to participate. We chose this population because these students had enough experience to solve programming challenges that required problem-solving skills, and were unlikely to get stuck on syntax errors. While this population had more experience than the participants of most studies of novice programmers, we think they were appropriate for this study given our research goals. All participants provided informed consent for participation in the study, and were compensated with a $60 Amazon gift card for their time.

Procedure

First, participants were led through a 60-minute JavaScript tutorial by the authors. Next, they were split into pairs and worked on problems in one of two environments (Pyrus and

pair programming) for 45 minutes. Following this problem-solving session, they were given a 30-minute break, during which some students were interviewed and others filled out an online survey. Next, they worked on problems in the other environment for 45 minutes, followed by another 30-minute break for interviews and surveys.

JavaScript Tutorial. While each of our participants had programming experience, none was familiar with JavaScript. Therefore, to maintain a relatively even skill level and ensure that participants could work on JavaScript programming challenges, we first provided a 60-minute tutorial on JavaScript. This tutorial was designed by the authors and led by the first author. It covered the fundamentals of JavaScript syntax and programming constructs that participants would need to complete the challenges, including variables, built-in methods, and control-flow structures. Students practiced writing code in exercises during the tutorial, and all authors were available to answer questions.

Problem-Solving Sessions. We counterbalanced the problem-solving sessions to reduce order effects. After the tutorial, participants were randomly split into two groups and paired, such that one group had five pairs and the other had four. Participants in the first group solved problems in pair programming during the first session and Pyrus during the second, while participants in the second group did the opposite.

Before each programming session, one of the researchers described the environment that the participants would be working with. For pair programming, the researcher played a tutorial video explaining that the partners would work on a single computer, and that one would be the pilot while the other would be the co-pilot. The partners were told they should switch roles every 10 minutes, but this was not enforced. For Pyrus, the researcher first described the game and the game rules. Then, the researcher played a tutorial video showing an example of someone interacting with the Pyrus interface. During the problem-solving sessions, partners sat next to each other and were able to discuss freely. The researchers were available to answer any questions participants had about the two interfaces and the Pyrus game rules. We did not answer any questions about JavaScript or approaches for solving the challenges, but instructed participants to use the Internet as a resource.

To keep the two conditions as similar as possible, participants worked on the pair programming challenges in a web interface that was equivalent to Pyrus, but without any of the game mechanics (Figure 2). During pair programming, the pairs shared a single computer, rather than each typing on their own computer as in Pyrus, to emulate the traditional pair programming protocol that has been studied in educational contexts [45, 62].

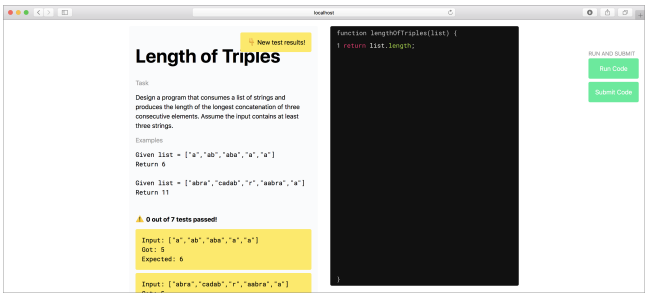

Figure 2: The pair programming interface resembles Pyrus to minimize differences between the two conditions.

We designed a sequence of four JavaScript programming problems for each of the programming sessions and participants worked on the problems in order. We sourced problems from the introductory programming website CodingBat[1] and also used simplified versions of problems found in *Cracking the Coding Interview*[2]. The problems were ranked in difficulty based on length, complexity of the solution, and variance in possible solution approaches, and then ordered such that they increased in difficulty. Our goal was to ensure that students with higher levels of incoming skill would not finish all problems within the given 45 minute time period.

Interviews and Surveys. During the 30 minutes following each programming session, a subset of participants were interviewed about their experience. We interviewed ten randomly selected participants after the first session, and eight of those same participants after the second session (fewer due to a limited availability of researchers). We followed a semi-structured interviewing protocol, asking questions about the pair programming experience (e.g. *"How did you behave when it was your partner's turn to write the code?"*) and the Pyrus experience (e.g. *"How did you decide what actions you would take on your turn?"*). At the end of the second interview, we also asked questions that encouraged participants to directly compare the two experiences (e.g. *"Compare and contrast your experience pair programming in the normal editor and in Pyrus. Did you approach solving challenges differently?"*). All participants who were not interviewed by a researcher completed online surveys that asked the same questions as the semi-structured interviews.

6 DATA ANALYSIS

We collected data from a variety of sources during our study, and apply both qualitative and quantitative approaches to analyze that data. Below we present each of our data sources, along with our measures and analysis process for each.

[1]https://codingbat.com/java
[2]http://www.crackingthecodinginterview.com/

Programming Session Transcripts

During programming sessions, we recorded audio of the dialog between pairs as they worked to solve problems, with the goal of learning about their problem-solving process and the quality of their collaborations. The resulting 13.5 hours of audio data were transcribed, with an associated timestamp for every line in the transcript. To address our research questions, we focused our analysis on understanding whether Pyrus encourages novices to plan in advance and participate equally in problem solving, as per our definition of effective collaboration. We developed a coding scheme that allowed us to quantify the amount of time pairs spent engaged in different stages of the problem-solving process, as well as the amount of time each partner spent in the pilot role.

To build our codebook for problem-solving stages, we took a deductive approach [48]. We first created codes for four problem-solving stages that have been identified as part of the programming process in the literature [19, 40]: (1) understanding the problem, (2) planning the solution, (3) implementing the solution, and (4) debugging the solution. We then amended and refined our codebook by following a data-driven inductive process [48]. We completed a comprehensive reading of the transcript files, refined our code definitions, and identified three additional problem-solving stages specific to the Pyrus environment: (5) planning around Pyrus, (6) interacting with Pyrus, and (7) implementing outside of IDE. Each line of the transcript was coded with at most one problem-solving stage, and the stages could occur in any order. Two authors worked together to code a training set of data, and met to resolve any conflicts and iteratively refine the codebook definitions. Then they independently coded 11% of the data (two transcripts) and achieved an inter-rater reliability Cohen's kappa 0.85 (values above 0.81 are considered "almost perfect" [15, 35]). The two authors then divided up and independently coded the rest of the transcripts.

In addition to coding each line in the transcript with a problem-solving stage, the authors also coded which partner was in the pilot role for each line. In pair programming, where the partners shared one computer, the pilot was the person who was typing. In Pyrus, where the partners each worked on their own laptop, the pilot was the person whose turn it was in the game. It was straightforward to determine who was the pilot from the audio data; participants talked about switching roles, and typing was audible.

Once the transcripts were coded, we used a quantitative approach to analyze the data. To capture the pairs' problem-solving process, we calculated the amount of time they spent in each problem-solving stage using the timestamps in the transcripts. To measure planning in advance, we also calculated the amount of time spent in the "planning the solution" stage before the first "implementing the solution" stage. To capture the equality of pairs' collaboration, we calculated the amount of time each partner spent in the pilot role. For all of these measures, we computed time using the timestamps associated with each line in the transcript. We chose to analyze time rather than counts of coded statements to ensure that our analysis would not inflate code counts for the pairs who were more talkative. We used a repeated measures ANOVA to analyze these within-subjects measures.

Programming Session Log Data

During the programming sessions we also collected log data from the Pyrus and pair programming interfaces. For each problem that the pair worked on we recorded their code in the editor every 30 seconds and the number of test cases the pair passed when they ran their code or submitted their final solution. Our goal in analyzing this data was to understand how quickly pairs wrote code and solved problems in the two conditions. We computed two measures with this data: (1) the amount of time it took for pairs to correctly pass all test cases for a problem, and (2) the number of problems solved (a problem is considered solved when all test cases have been passed). We used a repeated measures ANOVA to analyze these within-subjects measures.

Interview Transcripts and Survey Responses

Finally, we saved responses to the surveys and audio recorded interviews with participants. The resulting 3.6 hours of interview audio data were transcribed. We analyzed this data by conducting a thematic analysis [10]. All four authors began with a comprehensive reading of the transcripts to identify codes. Then we met and discussed our proposed codes to develop a codebook. Each interview was coded by one author, checked by another, and any disagreements between the two authors were discussed by all four. Our final codebook includes 73 codes, and is organized in a hierarchical structure with top-level codes such as "problem-solving process," "collaboration roles," and "Pyrus game mechanics."

Two authors independently coded four interviews (22.22% of data). We used a pooled PABAK Kappa to determine inter-rater reliability, which accounts for the prevalence of codes and potential bias between observers [11, 15, 18]. Our Kappa was 0.61 (0.61–0.8 is "substantial" strength of agreement) [35]). After coding the interview and survey data, related codes were clustered into themes that represent the high-level findings from our analysis.

7 RESULTS

Our results show that Pyrus encouraged novices to plan in advance and to participate more equally, the two behavioral outcomes we targeted during the game design process. However, we also found that some participants found Pyrus frustrating and inefficient. We unpack these findings below.

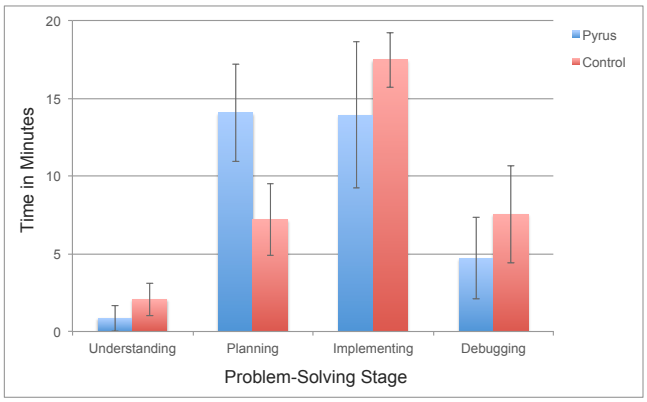

Figure 3: Amount of time pairs spent in each of the four main problem-solving stages while working on problems in Pyrus and pair programming.

Pyrus encouraged novices to plan in advance

Across all three data sources, we found evidence that novices planned their solutions in advance more in Pyrus than in pair programming. We consider planning in advance to include time spent in two problem-solving stages, the *planning the solution* stage and the *planning around Pyrus* stage. When working in Pyrus, plans for how to solve the problem were intertwined with plans for how to manage resources like cards and turns, and thus both are essential parts of planning in advance. We found that pairs in Pyrus planned for 14.07 minutes per session on average, compared to 7.41 minutes in the control ($F(1,8)=3.48$, $p<0.001$), as shown in Figure 3. Even when we only consider time spent in the *planning the solution* stage, during which pairs exclusively planned their solution to the problem, we saw that pairs planned for an average of 10.4 minutes per session in Pyrus compared to 7.41 in pair programming. However, given the size of this difference and our small sample size, this was not statistically significant ($F(1,8)=0.43$, $p=0.10$).

This finding was further supported by our interview and survey data. Six out of 18 participants (representing five out of nine total pairs) mentioned that they did not plan in advance while pair programming, compared to zero participants in Pyrus. In contrast, 16 out of 18 participants (representing all nine pairs) mentioned that they planned in advance in Pyrus, compared to five out of 18 participants (representing four out of nine pairs) in pair programming.

Overall, we found that participants approached problem solving differently in the two conditions. We analyzed the time when pairs first entered the implementation stage in the problem-solving session transcript, and found that pairs start implementing significantly earlier in pair programming ($F(1,8)=0.783$, $p<0.05$), at an average time of 2.87 minutes compared to 7.45 minutes in Pyrus. In interviews, multiple participants mentioned that they used a trial-and-error approach while pair programming. For example, PS39 stated *"We were able to tackle the code without having to spend too much time thinking of solutions. Instead we addressed issues as they came up."* This finding is consistent with prior work showing that novices do not plan solutions in advance [37, 60].

In contrast, pairs in Pyrus planned more of their solutions before beginning to implement. PS30 describes this difference: *"In Pyrus, I had to think much further ahead. This actually made it easier to think about the project as a whole, though. Instead of tackling one bit at a time, I started to look more into the big picture of it all."* When talking about their strategy in Pyrus, five participants (representing three pairs) mentioned that they wrote their solutions on paper before implementing them in the editor. An analysis of the problem-solving codes showed that pairs spent an average of 2.53 minutes planning before writing their first line of code in Pyrus, compared to 0.47 minutes in pair programming ($F(1,8)=0.68$, $p<0.05$).

We also found that these different approaches to problem-solving necessitated different approaches to debugging. Pairs spent more time debugging in pair programming, an average of 7.53 minutes compared to 4.70 minutes in Pyrus, although this difference was not significant ($F(1,8)=0.41$, $p=0.11$). When talking about their pair programming experience, six participants, (representing five distinct pairs) mentioned completely abandoning their partial solutions to start over from scratch. For example, PS41 said *"There were several times that we had to start over when we realized our approach was too complicated and there were easier ways. Starting over allowed us to have a fresh slate."* PS31 expounds further on the debugging process in pair programming: *"Even though we got more problems done, and we were doing them quicker, it was a lot more error. There was a lot of bugs and a lot of things we had to correct along the way."* This suggests that during pair programming, some participants had errors in their solution ideas or approaches that were not revealed until the debugging process. Participants also mentioned that they made fewer mistakes in Pyrus; PS45 described the difference in strategy in the two conditions, and how this impacted the number of mistakes he and his partner made:

> *"[In Pyrus] it was just a lot more organized. We would pretty much write down almost exactly what we were going to do, and then we would just put it in. Then, we'd have to kind of play the game where we figure out who has what things at their disposal. Then, for the pair programming it was like we could afford to make a lot more mistakes. We didn't have to write down our whole approach and we could just kind of start writing... But, I do think that our process was kind of a lot less organized and clean."*

These findings suggest that the problem-solving approaches that participants took as a result of the programming environment impacted the amount of debugging that was necessary to arrive at a correct solution.

When talking about their problem-solving approach in Pyrus, many participants mentioned that the game mechanics influenced their behavior. Ten participants mentioned that they prioritized plans based on the available cards, and five participants mentioned being more careful in their implementations because of the failure condition. PS44 described the strategy he and his partner used in Pyrus: *"we first looked at the cards in hand. We then wrote out our plans to solve the challenge on paper."* PS45 mentioned that the failure condition influenced his process, stating *"[in Pyrus] you had to be careful, because otherwise you'd lose if you did it wrong... You really had to have a plan to be using your resources properly."* These statements indicate that the Pyrus game mechanics influenced the way participants approached the problem-solving process, incentivizing them to plan their solutions in advance and be more deliberate while programming.

Pyrus encourages pairs to participate more equally

Through our data analysis, we found evidence that pairs participated more equally in problem solving in Pyrus than when pair programming. We analyzed the codes that captured when each partner was in the pilot role by calculating the difference in typing time between the two partners. We found that this difference was significantly smaller in Pyrus ($F(1,8)=1.17$, $p<0.05$); in Pyrus the difference in time spent typing between participants was 5.69 minutes, compared to a 17.78 minute difference in pair programming. This can be partially explained by the fact that for five pairs in the pair programming condition, the less-active typer spent under 10 of the 45 minutes in the pilot role.

Overall, we found that participants took on different roles in pair programming. PS31 stated simply, *"It was less of a contributing when you were not typing, and more contribution when you were typing."* Ten participants described the pilot as being the decision maker or leader. For example, PS46 states *"Our process, I guess, just like kind of lead person who just kind of ... did it, I guess? Unless they got like stuck or something."* PS35 said *"We each implemented our own solutions when it was our time to work because we thought it would be easier to do our own solutions than explain our solution to the other person."* In contrast, many described the role of the co-pilot as following along, pointing out small errors, and helping to debug. For example, PS41 states *"I let them code their solution to the problem because it was how we decided to work. I watched for errors and commented when I saw them."* Some participants also mentioned feeling lost when in the co-pilot role; PS32 states, *"It's really just like trying to interpret what they're doing, as opposed to interpreting what we're doing ."* These findings suggest that the pilot and co-pilot serve very different roles in pair programming, and that working in pairs does not inherently encourage collaborative planning.

In Pyrus, participants rarely mentioned differences between the roles of the typer and the observer. For example, when describing the roles in Pyrus, PS38 states *"We talked about what we needed to do first, and then whoever's turn it was just wrote it. There wasn't that much of a difference between whose turn it was."* Six participants explicitly mentioned that they felt they were on the same page with their partner when working in Pyrus. When discussing why they collaborated equally in Pyrus, participants mentioned the enforced turn-taking mechanic. PS33 stated:

> *"In other classes when you work in pairs, normally one person does all the typing, and I feel like that can really easily lead to an imbalance as far as learning goes. Forcing each person [in Pyrus] to type this way, I think, really does help make both people be aware of what's going on."*

PS45 summarized the difference between Pyrus and pair programming nicely:

> *"[In Pyrus] we were switching off who was writing what. It was just completely necessary to be really clear about what we were both doing from the outset. You really had to have a plan to be using your resources properly. And then, in [pair programming] it was just kind of like... as long as that person kind of had the framework in their mind we didn't need everyone to know everything completely."*

These findings show that the enforced turns mechanic in Pyrus changed the way that pairs collaborated while working on programming problems, resulting in more active participation from the non-dominant partner and a deeper shared understanding of the problem approach and solution.

Novices found Pyrus frustrating and inefficient

While we found evidence that Pyrus encourages more planning in advance and more equal participation, many participants found that Pyrus restricted their ability to implement code. The lack of flexibility led to frustration. For example, PS46 said *"It's just really annoying when you can see ... You know exactly what I need to type, but then it just won't let you do it"*. Additionally, PS43 stated *"I think a lot of coding is trying something, then failing. It's a system which doesn't allow you to fail, without failing completely, and restarting from scratch, which I think is really, really unhelpful"*.

However, not all participants found the experience frustrating; for example PS45 stated: *"I probably wouldn't write out my whole process beforehand. But working in Pyrus, we did, which I actually think it made it a lot more efficient. And I think we realized a lot of things that we would run into earlier than if we would have just started writing, so that was good."*

Participants also noted that writing code in Pyrus was slower than in pair programming, a point that came up in

the interviews and surveys of ten participants, even though the bootcamp and activities placed no emphasis on programming speed. In our analysis of the log data, we found that pairs did work more slowly in Pyrus, completing an average of 0.78 problems per session on average compared to 2.56 for pair programming ($F(1,8)=1.83$, $p<0.01$). This is not surprising, given that Pyrus required pairs to plan not only their solutions, but also how to implement them with the available cards and actions. Our analysis of the problem-solving stages showed that pairs spent an average of 4.02 minutes planning around Pyrus and 2.53 minutes interacting with Pyrus, a substantial amount of additional time. Interestingly, attitudes of frustration towards slower modes of programming have also been observed among novices in pair programming [13].

Four participants did note that they would like to incorporate some of the strategies they used in Pyrus into their own programming more often. For example, PS39 stated: *"When coding in Pyrus, I was forced to formulate a plan before beginning. We also had to come to an agreement on how to tackle the problem. This is something I rarely do when coding on my own, but something I wish I did more of."* When asked if there is any context in which they would like to use Pyrus, PS39 said *"I would use Pyrus if I was working on a code with someone to learn or for fun."* However, several participants said they would not like to use Pyrus in any contexts. For example, PS43 said *"I'm tempted to say that there is nothing that I would want my normal coding experience to resemble this coding experience."* This suggests that further iteration on the game design is needed to ensure that the experience is fun, particularly to make sure the constraints are not so restrictive as to cause frustration. However, it is also interesting to note that novices seem to value efficiency over equal participation and time spent planning their solutions, which could be a result of their prior programming experiences and their expectations around coding.

8 CONCLUSION

Our findings provide preliminary evidence that *behavior-centered game design* can be effective in guiding the design of a game that encourages players to practice a set of target behaviors, even when their natural inclination may be to do the opposite. In our within-subjects study, we found that Pyrus successfully encouraged novices to plan in advance and participate equally in problem solving, the behaviors we targeted in our design. Pairs in Pyrus spent twice the amount of time planning as they did in pair programming, a significant increase. There was also a significantly smaller difference in the amount of time each partner spent in the active pilot role. Most importantly, participants described how their problem-solving and collaborative behaviors were influenced by game mechanics like the failure condition, distributed resources, and enforced turns, showing how mechanics can be used to drive behavior in educational games.

While some participants enjoyed playing Pyrus, many found the game's constraints frustrating and complained that Pyrus was less efficient than pair programming. We believe some of these frustrations could be addressed through design iterations to improve game balance. For example, if we ensure that players have access to the cards they need to solve each problem in the first half of the deck, they may experience less frustration. Participants' complaints that Pyrus is "inefficient" and "slow" suggest that novices value solving problems quickly, and may prioritize this over developing a deep understanding of the problem and solution. Perhaps learners would be less likely to fixate on pace and performance if the game was explicitly framed around the goal of developing an effective problem-solving process.

Furthermore, while Pyrus successfully encouraged planning, we do not yet know whether practicing planning through Pyrus teaches novices transferable problem-solving skills. Even though some participants recognized that planning in advance was good for their understanding and problem-solving success, the pairs who played Pyrus first did not adopt those behaviors in the following pair programming session. Given that participants only interacted with Pyrus for 45 minutes, this lack of transfer is unsurprising. However, it is possible that Pyrus may over-scaffold planning and equal participation, which has been an issue in other popular educational games [41]. In the future, we envision integrating Pyrus into classroom lessons that teach problem-solving to help students practice behaviors like planning, and supporting transfer through activities like reflection [55]. Additional research in real-world contexts is needed to understand whether and how planning practice in Pyrus can be transferred outside of the game context.

This exploratory work contributes a new educational game Pyrus, which we see as a compelling proof-of-concept for *behavior-centered game design*. However, there is still much to learn about this methodology and how to apply it in service of higher-level learning goals. In future work, we plan to design additional games using a behavior-centered approach to better understand the strengths and weaknesses of this methodology. We also plan to analyze existing educational games through a behavior-centered lens to better understand the mechanics that lead to their success. However, this work takes an important initial step towards improving our understanding of how to design effective educational games.

ACKNOWLEDGMENTS

We thank the students and faculty in the Design, Technology, and Research program and the Delta Lab for their valuable feedback. This work was supported by Undergraduate Research Grants from Northwestern University.

REFERENCES

[1] Vincent Aleven, Eben Myers, Matthew Easterday, and Amy Ogan. 2010. Toward a framework for the analysis and design of educational games. In *Digital Game and Intelligent Toy Enhanced Learning (DIGITEL), 2010 Third IEEE International Conference on*. IEEE, 69–76.

[2] Basma S Alqadi and Jonathan I Maletic. 2017. An Empirical Study of Debugging Patterns Among Novices Programmers. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, 15–20.

[3] Ian Arawjo, Cheng-Yao Wang, Andrew C Myers, Erik Andersen, and François Guimbretière. 2017. Teaching programming with gamified semantics. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, 4911–4923.

[4] E Aronson, N Blaney, C Stephan, J Sikes, and M Snapp. 1978. *The Jigsaw Classroom*. Sage Publishing Company.

[5] Theresa Beaubouef and John Mason. 2005. Why the high attrition rate for computer science students: some thoughts and observations. *ACM SIGCSE Bulletin* 37, 2 (2005), 103–106.

[6] Kent Beck and Erich Gamma. 2000. *Extreme programming explained: embrace change*. addison-wesley professional.

[7] Jens Bennedsen and Michael E Caspersen. 2007. Failure rates in introductory programming. *ACM SIGcSE Bulletin* 39, 2 (2007), 32–36.

[8] Judith Bishop, R Nigel Horspool, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. 2015. Code Hunt: Experience with coding contests at scale. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 398–407.

[9] Acey Boyce and Tiffany Barnes. 2010. BeadLoom Game: using game elements to increase motivation and learning. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*. ACM, 25–31.

[10] Virginia Braun, Victoria Clarke, and Gareth Terry. 2014. Thematic analysis. *Qual Res Clin Health Psychol* 24 (2014), 95–114.

[11] Ted Byrt, Janet Bishop, and John B Carlin. 1993. Bias, prevalence and kappa. *Journal of clinical epidemiology* 46, 5 (1993), 423–429.

[12] Jill Cao, Irwin Kwan, Rachel White, Scott D Fleming, Margaret Burnett, and Christopher Scaffidi. 2012. From barriers to learning in the Idea Garden: An empirical study. In *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*. IEEE, 59–66.

[13] Edgar Acosta Chaparro, Aybala Yuksel, Pablo Romero, and Sallyann Bryant. 2005. Factors affecting the perceived effectiveness of pair programming in higher education. In *Proc. PPIG*. 5–18.

[14] Douglas B Clark, Emily E Tanner-Smith, and Stephen S Killingsworth. 2016. Digital games, design, and learning: A systematic review and meta-analysis. *Review of educational research* 86, 1 (2016), 79–122.

[15] Jacob Cohen. 1968. Weighted kappa: Nominal scale agreement provision for scaled disagreement or partial credit. *Psychological bulletin* 70, 4 (1968), 213.

[16] Thomas M Connolly, Elizabeth A Boyle, Ewan MacArthur, Thomas Hainey, and James M Boyle. 2012. A systematic literature review of empirical evidence on computer games and serious games. *Computers & Education* 59, 2 (2012), 661–686.

[17] Sabrina Haskell Culyba. 2018. *The Transformational Framework: A process tool for the development of Transformational games*. ETC Press.

[18] Han De Vries, Marc N Elliott, David E Kanouse, and Stephanie S Teleki. 2008. Using pooled kappa to summarize interrater agreement across many items. *Field Methods* 20, 3 (2008), 272–282.

[19] Fadi P Deek, Murray Turoff, and James A McHugh. 1999. A common model for problem solving and program development. *IEEE Transactions on Education* 42, 4 (1999), 331–336.

[20] Pierre Dillenbourg. 2002. Over-scripting CSCL: The risks of blending collaborative learning with instructional design.

[21] Pierre Dillenbourg and Daniel Schneider. 1995. Mediating the mechanisms which make collaborative learning sometimes effective. *International Journal of Educational Telecommunications* 1, 2 (1995), 131–146.

[22] Michael Eagle and Tiffany Barnes. 2008. Wu's castle: teaching arrays and loops in a game. In *ACM SIGCSE Bulletin*, Vol. 40. ACM, 245–249.

[23] Alireza Ebrahimi. 1994. Novice programmer errors: language constructs and plan composition. In *International Journal of Human-Computer Studies*, Vol. 41. 457–480.

[24] Alireza Ebrahimi and Christina Schweikert. 2006. Empirical study of novice programming with plans and objects. In *ACM SIGCSE Bulletin*, Vol. 38. ACM, 52–54.

[25] James Paul Gee. 2003. What video games have to teach us about learning and literacy. *Computers in Entertainment (CIE)* 1, 1 (2003), 20–20.

[26] Anuradha A Gokhale. 1995. Collaborative learning enhances critical thinking. *Volume 7 Issue 1 (fall 1995)* (1995).

[27] Brian Hanks and Matt Brandt. 2009. Successful and unsuccessful problem solving approaches of novice programmers. In *ACM SIGCSE Bulletin*, Vol. 41. ACM, 24–28.

[28] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R Klemmer. 2010. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1019–1028.

[29] Andrew Hicks, Barry Peddycord, and Tiffany Barnes. 2014. Building games to learn from their players: Generating hints in a serious game. In *International Conference on Intelligent Tutoring Systems*. Springer, 312–317.

[30] Cindy E Hmelo-Silver. 2013. *The international handbook of collaborative learning*. Routledge.

[31] Britton Horn, Christopher Clark, Oskar Strom, Hilery Chao, Amy J Stahl, Casper Harteveld, and Gillian Smith. 2016. Design insights into the creation and evaluation of a computer science educational game. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 576–581.

[32] Tony Jenkins. 2002. On the difficulty of learning to program. In *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, Vol. 4. Citeseer, 53–58.

[33] Dominic Kao and D Fox Harrell. 2015. Mazzy: A STEM Learning Game.. In *FDG*.

[34] Judy Kay, Michael Barg, Alan Fekete, Tony Greening, Owen Hollands, Jeffrey H Kingston, and Kate Crawford. 2000. Problem-based learning for foundation computer science courses. *Computer Science Education* 10, 2 (2000), 109–128.

[35] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics* (1977), 159–174.

[36] H Chad Lane and Kurt VanLehn. 2005. Teaching the tacit knowledge of programming to novices with natural language tutoring. *Computer Science Education* 15, 3 (2005), 183–201.

[37] Jari M Lavonen, Matti Lattu, and Veijo P Meisalo. 2001. Problem Solving with an Icon Oriented Programming Tool: A Case Study in Technology Education. *Journal of Technology Education* 12, 2 (2001), 21–34.

[38] Michael J Lee, Faezeh Bahmani, Irwin Kwan, Jilian LaFerte, Polina Charters, Amber Horvath, Fanny Luor, Jill Cao, Catherine Law, Michael Beswetherick, et al. 2014. Principles of a debugging-first puzzle game for computing education. In *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*. IEEE, 57–64.

[39] Marcia C Linn and Michael J Clancy. 1992. Can experts' explanations help students develop program design skills? *International Journal of Man-Machine Studies* 36, 4 (1992), 511–551.

[40] Dastyni Loksa, Andrew J Ko, Will Jernigan, Alannah Oleson, Christopher J Mendez, and Margaret M Burnett. 2016. Programming, problem

solving, and self-awareness: effects of explicit guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 1449–1461.

[41] Yanjin Long and Vincent Aleven. 2017. Educational game and intelligent tutoring system: A classroom study and comparative design analysis. *ACM Transactions on Computer-Human Interaction (TOCHI)* 24, 3 (2017), 20.

[42] Merrilea J Mayo. 2009. Video games: A route to large-scale STEM education? *Science* 323, 5910 (2009), 79–82.

[43] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *Working group reports from ITiCSE on Innovation and technology in computer science education*. ACM, 125–180.

[44] Charlie McDowell, Linda Werner, Heather Bullock, and Julian Fernald. 2002. The effects of pair-programming on performance in an introductory programming course. *ACM SIGCSE Bulletin* 34, 1 (2002), 38–42.

[45] Charlie McDowell, Linda Werner, Heather E Bullock, and Julian Fernald. 2003. The impact of pair programming on student performance, perception and persistence. In *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 602–607.

[46] Bruce M McLaren, Lars Bollen, Erin Walker, Andreas Harrer, and Jonathan Sewall. 2005. Cognitive tutoring of collaboration: Developmental and empirical steps towards realization. In *Proceedings of th 2005 conference on Computer support for collaborative learning: learning 2005: the next 10 years!* International Society of the Learning Sciences, 418–422.

[47] Edward F Melcer and Katherine Isbister. 2018. Bots & (Main) Frames: Exploring the Impact of Tangible Blocks and Collaborative Play in an Educational Programming Game. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 266.

[48] Matthew B Miles, A Michael Huberman, and Johnny Saldana. 2013. *Qualitative data analysis*. Sage.

[49] Mark L Miller. 1979. A structured planning and debugging environment for elementary programming. *International Journal of Man-Machine Studies* 11, 1 (1979), 79–95.

[50] Briana B Morrison, Lauren E Margulieux, and Mark Guzdial. 2015. Subgoals, context, and worked examples in learning computing problem solving. In *Proceedings of the eleventh annual international conference on international computing education research*. ACM, 21–29.

[51] Engineering National Academies of Sciences and Medicine. 2018. *Assessing and Responding to the Growth of Computer Science Undergraduate Enrollments*. The National Academies Press, Washington, DC, Chapter Front Matter.

[52] Harold F O'Neil, Richard Wainess, and Eva L Baker. 2005. Classification of learning outcomes: Evidence from the computer games literature. *The Curriculum Journal* 16, 4 (2005), 455–474.

[53] Eleanor O'Rourke, Kyla Haimovitz, Christy Ballweber, Carol Dweck, and Zoran Popović. 2014. Brain points: a growth mindset incentive structure boosts persistence in an educational game. In *Proceedings of the SIGCHI conference on human factors in computing systems*. ACM, 3339–3348.

[54] Roy D Pea. 1983. Logo Programming and Problem Solving.[Technical Report No. 12.]. (1983).

[55] David N Perkins, Gavriel Salomon, et al. 1992. Transfer of learning. *International encyclopedia of education* 2 (1992), 6452–6457.

[56] Leen-Kiat Soh. 2006. Implementing the jigsaw model in CS1 closed labs. In *ACM SIGCSE Bulletin*, Vol. 38. ACM, 163–167.

[57] Elliot Soloway. 1986. Learning to program= learning to construct mechanisms and explanations. *Commun. ACM* 29, 9 (1986), 850–858.

[58] Ian Utting, Allison Elliott Tew, Mike McCracken, Lynda Thomas, Dennis Bouvier, Roger Frye, James Paterson, Michael Caspersen, Yifat Ben-David Kolikant, Juha Sorva, et al. 2013. A fresh look at novice programmers' performance and their teachers' expectations. In *Proceedings of the ITiCSE working group reports conference on Innovation and technology in computer science education-working group reports*. ACM, 15–32.

[59] Christopher Watson and Frederick WB Li. 2014. Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*. ACM, 39–44.

[60] Noreen M Webb, Philip Ender, and Scott Lewis. 1986. Problem-solving strategies and group processes in small groups learning computer programming. *American Educational Research Journal* 23, 2 (1986), 243–261.

[61] Viktor Wendel, Michael Gutjahr, Stefan Göbel, and Ralf Steinmetz. 2012. Designing collaborative multiplayer serious games for collaborative learning. *Proceedings of the CSEDU* 2 (2012), 199–210.

[62] Laurie Williams and Richard L Upchurch. 2001. In support of student pair-programming. In *ACM SIGCSE Bulletin*, Vol. 33. ACM, 327–331.

[63] Leon E Winslow. 1996. Programming pedagogy–a psychological overview. *ACM Sigcse Bulletin* 28, 3 (1996), 17–22.

[64] Michael F Young, Stephen Slota, Andrew B Cutter, Gerard Jalette, Greg Mullin, Benedict Lai, Zeus Simeoni, Matthew Tran, and Mariya Yukhymenko. 2012. Our princess is in another castle: A review of trends in serious gaming for education. *Review of educational research* 82, 1 (2012), 61–89.

[65] Natalia Padilla Zea, José Luís González Sánchez, Francisco L Gutiérrez, Marcelino J Cabrera, and Patricia Paderewski. 2009. Design of educational multiplayer videogames: A vision from collaborative learning. *Advances in Engineering Software* 40, 12 (2009), 1251–1260.