# Pyrus: A Collaborative Programming Game to Support Problem-Solving

**Josh Shi**
**Armaan Shah**
Northwestern University
633 Clark St.
Evanston, IL 60208, USA
joshshi@u.northwestern.edu
armaanshah2018@u.northwestern.edu



## Abstract
While prior work has shown that pair programming is successful in helping students develop confidence and produce better programs, research in collaboration and computing education suggests that it could be designed more effectively as a pedagogical tool. In this paper we present Pyrus, a collaborative programming game that explicitly scaffolds for behaviors linked to developing programming problem-solving skills. We evaluated Pyrus by comparing the amount of planning and verbal explanation demonstrated by pairs of programmers who worked on programming challenges in Pyrus and traditional pair programming. Our results suggest that Pyrus may more reliably support planning than traditional pair programming.

## Author Keywords
Programming; Computer Science Education; Self-Regulation; Collaboration; Serious Games

## ACM Classification Keywords
H.5.0 Information interfaces and presentation: General

## Introduction
Despite the increasing demand for programmers, retention within computer science programs is poor [2]. Possible explanations for the high attrition rate among CS programs include the difficulty of the material, that students must often develop programming problem-

solving skills on their own, and low confidence among students in their abilities [1,4].

Pair programming, a method of programming in which two programmers work together on the same problem and take turns writing code, successfully combats some of these issues when used as a pedagogical tool. Pair programming has been found to improve retention among novice programmers [6]. Students who pair program also produce better programs and are more confident in their solutions than their peers who program alone [6,7].

While pair programming can help alleviate some of the challenges faced by novice programmers, it is not designed to explicitly support behaviors that have been identified as useful for developing problem-solving skills. Without strict guidelines, there's no guarantee that two paired programmers will actually work collaboratively [7]. Additionally, previous work in computing education has demonstrated the importance of strengthening self-regulation (the ability to be aware of one's thoughts and actions and evaluate how well they are moving one closer towards a goal) through practicing behaviors like planning for novices in developing their programming problem-solving skills [5]. Investigations into collaboration have similarly identified certain behaviors such as verbal explanation which make collaboration effective for learning [3]. While pair programming is collaborative, it does not provide explicit scaffolds for these desirable behaviors.

We suggest that pair programming could be more effective as an educational tool if explicitly scaffolded for both protocol and problem-solving behaviors. One method of creating engaging, educational resources is through serious games, which are effective in supporting collaboration and problem-solving [8].

The conceptual contribution of this paper is the idea of applying methods from game design and pair programming to create a collaborative coding game which scaffolds for behaviors that are vital to developing programming problem-solving skills among novices. We do this through Pyrus, a coding game in which pairs of programmers work collaboratively on computer science problems.

## Related Work

Prior investigations into pair programming have demonstrated its effectiveness as an educational tool and in improving retention among novice programmers. McDowell et. al. [6] demonstrated that students who paired on assignments in an introductory CS course produced better programs and completed the course at higher rates. In later research, McDowell et. al. [7] also found that among students who passed the course, those who paired were significantly more likely to be registered as computer science majors a year later. While this body of work provides a starting point for understanding the benefits of pair programming in computer science education, it focuses only on observing the effects of pair programming. Pyrus builds upon this body of work by investigating how pair programming experiences can be augmented to promote desired learning behaviors such as planning and collaboration.

Loksa et. al. [5] have identified the importance of self-regulation in developing programming problem-solving skills and contributed a framework for observing self-regulation behaviors. In Pyrus, we explicitly scaffold for

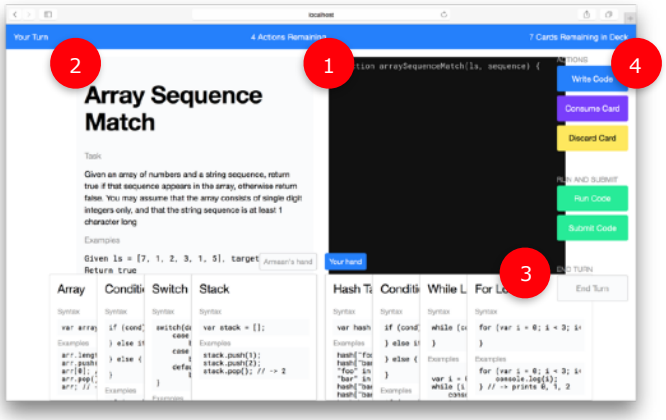


**Figure 1:** The Pyrus interface, showing 1) the editor, 2) the prompt, 3) the cards held by each player, and 4) actions available to the programmer.

```
1   let ls = [];
2   let total = 0;
3   for (let i = 0; i < 5; i++) {
4       ls.push(i);
5       total += i;
6   }
```

**Figure 2:** In Pyrus, every expression is written through an action. To write line 1, which declares a variable `ls` that holds an array, the programmer would use a Consume move along with an Array card, since an array is not a primitive data type. The second line, which declares a variable that holds a number (a primitive data type), may be written by a Write action and does not require a corresponding card. Similar to the first line, the for loop starting on the third line would be created using a Consume action with a For Loop card. The lines within the for block (lines 4-5) would both be written using a Write action, as they are operations on existing variables.

planning, one of the behaviors highlighted in the framework.

Dillenbourg et. al. [3] have identified mechanisms of collaboration, such as verbal explanations both inter- and intra-participant, which improve learning outcomes. In the design of Pyrus, we aim to explicitly scaffold for explanation.

Prior research in serious games has explored the use of game mechanics in creating greater planning and collaborative outcomes. In one study, eighth graders who participated in a class using Mobile Serious Games (MSGs) achieved a higher perception of their own collaboration skills and a higher score in planning than students in a class without MSGs [8]. Games like this demonstrate the potential of serious games as an educational aid and suggest that game design can be used to scaffold for collaborative and problem-solving behaviors. We took inspiration from this system to inform the design of Pyrus.

## System Description

Pyrus is a two-player game in which programmers collaborate to solve programming challenges. Pyrus is designed to scaffold for self-regulation and collaborative behaviors by making these inexorable while programming. Pyrus augments the coding process by introducing cards and actions which become the resources that programmers use to make progress in their challenge. Like in pair programming, players take turns being active (the pilot).

The two players in Pyrus work at separate computers, but sit next to each other. The Pyrus interface displays 1) an editor where the programmers write their code, 2) a prompt which contains a problem description and test cases, 3) cards which contain programming constructs that players use to complete the challenge and are initially held in a deck, and 4) actions which players can perform on their turn (Figure 1).

On each turn, the active player (pilot) may perform up to four actions. There are three distinct actions. 1) Write: the pilot writes code to either declare a variable holding a primitive data type or perform an operation on existing variable(s). Only one expression may be written. 2) Consume: the pilot uses a card in his or her hand to implement a corresponding expression in the editor. That card is then discarded. 3) Discard: the pilot discards a card and draws a new one from the deck.

Programmers write code using combinations of Write and Consume actions, along with cards (Figure 2). Each player is dealt four cards at the beginning of the challenge, and at the end of each turn the pilot draws two more. If a player draws from an empty deck, both partners lose and must restart. Cards contain programming constructs such as data structures and control flow expressions. An expression represented on a card may only be written as part of a Consume action with the corresponding card. In other words, most of the code required to solve these problems, like loops or arrays, are not freely available to the programmer as they are in traditional programming.

We designed Pyrus to incorporate a variety of game mechanics that we hypothesized would support planning and explanation more effectively than traditional pair programming.

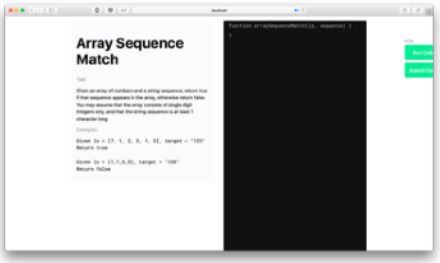


**Figure 3:** The interface used in the control group.

*Turn-taking*
Turns appear in pair programming as the pilot-copilot dynamic between partners. Unlike pair programming, this dynamic is enforced in Pyrus through limits on the number of actions a programmer can perform in a single turn. Turn-taking creates interdependence between programmers through a division of labor and makes the challenge unsolvable by any individual. We hypothesized that students would plan more because of this enforced division of labor, as players would have to discuss which actions to take to fully utilize their turns.

*Cards*
Cards are randomly distributed among players. We hypothesized that cards would create a distribution of resources that supports planning, as participants would have to coordinate in order to make use of their cards depending on what cards they needed and when. We also hypothesized that the same division of resources in addition to the division of labor from turns meant that students would have to discuss their thought processes with their partner more often, thus collaborating more.

*Constraints*
Participants are forced to program in discrete actions, some of which are backed by cards. A failure condition, in the form of a challenge reset when a user attempts to draw from an empty deck, forces the pair to fit their solution within the allotted actions. We hypothesized that limiting the number of actions a pair could perform in a given challenge (bounded by the number of cards in the deck) and discretizing the act of writing code meant students would be more deliberate in the steps they took to make progress, resulting in more planning.

## Methods
To understand whether programmers working under Pyrus would exhibit more planning and explanation than programmers in pair programming, we designed an experiment with two conditions: in the treatment, pairs worked under Pyrus using the interface detailed above (Figure 1); in the control, each pair was asked to pair program. To keep the experiences as similar as possible, we created a modified version of the Pyrus interface for the control condition which only included the prompt, the text editor, and buttons to run and submit code (Figure 3).

In both conditions, pairs were asked to work through the same series of seven programming problems. The first problem was taken from a similar study by Loksa et al. [5], and the rest were taken from introductory programing challenge website CodingBat[1]. All participants were told that they were free to communicate with their partners in any way they liked and that they could use resources like the Internet to look things up as they worked. They were given 31 minutes to work on as many problems as they could in the order the problems were presented.

In the treatment, participants viewed a video describing the Pyrus rules and interface. Each pair was then led through an example problem. The participants followed the rules through a combination of system enforcement and human intervention, which was necessary in some corner cases where the system was not able to detect a player performing an illegal action. In the control, participants were shown a video describing the interface (Figure 3) and pair programming. Participants

[1] http://codingbat.com

| **Code** |
| --- |
| Planning |
| **Definition** |
| Expressing intent to perform some task |
| **Example** |
| "Let's see. First what I would do is make a new array." |

| **Code** |
| --- |
| Explanation |
| **Definition** |
| Justifying a suggested or existing process |
| **Example** |
| "Oh, so *j* has to start at one after *i* I think, because you don't want to go to the ones before it." |

**Table 1:** The codes and definitions used for planning and explanation, as well as examples from transcripts.

| | **Planning** | **Explanation** |
| --- | --- | --- |
| **T1** | 48 | 11 |
| **T2** | 40 | 12 |
| **C1** | 46 | 11 |
| **C2** | 28 | 13 |

**Table 2:** Instances of planning and explanation displayed by programmers in treatment (T1, T2) and control groups (C1, C2).

in the control were not forced to switch between pilot/copilot roles and worked on the same machine.

Via email, we recruited undergraduate students who had taken at least CS1, CS2 and an introductory data structures course but no more than three additional CS courses. We recruited eight students who formed four pairs, two in the control and two in the treatment.

We recorded audio of participants talking during the sessions and transcribed and coded for examples of planning and explanation. The schemas for identifying these behaviors were adapted from Loksa et. al.'s [5] and Dillenbourg et. al.'s [3] respective frameworks for self-regulation and collaboration (Table 1). We developed this combined schema together, and one author completed all of the coding to maximize consistency. Participants filled out pre-study surveys providing information on what classes they had taken and how long they had been programming, and we also interviewed participants after each session to collect information on how they worked.

## Results

*Planning*

We observed 46 and 28 instances of planning in the control and 48 and 40 instances of planning in the treatment groups (Table 2). These results suggest that Pyrus may more consistently support planning than pair programming. One possible explanation for the 46 instances of planning observed in C1 could be because one or both of the participants had already developed their problem-solving skills. The two participants in C1 had three and four years of programming experience, the most out of any group (all other participants had three years or fewer of experience).

Anecdotal data collected from follow-up interviews indicate that the experience of using Pyrus changed how participants thought about programming. "Normally we wouldn't write [our solution] out," one participant said. "We would try it out and get the output and move on. Since we had such a limited amount of cards we would plan everything out perfectly and go from there." One also expressed how Pyrus's constraints forced him to plan more carefully. "Normally if I'm just writing my first draft of code I don't really care about how many lines I'm using or how many variables I initialize… But in [Pyrus], I'm limited by the number of actions and the cards that I have, so I have to really think through, what do I want to do and is this a good use of one of my actions."

*Explanation*

Among the control groups, we observed 11 and 13 instances of explanation. Among the treatment groups, we observed 11 and 12 instances (Table 2).

Although we hypothesized that there would be more explanation in the treatment, we found that all participants exhibited a near equivalent level of the behavior. Because of the similar outcomes, Pyrus maintains the existing explanation dynamics in traditional pair programming.

## Conclusion

This paper presents Pyrus, a collaborative programming game that explicitly scaffolds for behaviors linked to developing programming problem-solving skills. Pyrus demonstrates the feasibility and potential benefits of augmenting pair programming for pedagogical use by supporting collaboration and self-regulation via game design.

One limitation of this work is our small sample size. With only two pairs in each condition, we are unable to draw generalizable conclusions from our results. Additionally, our participants were highly variable in their programming knowledge and experience, further confounding our findings. We hope to run additional studies with larger sample sizes to validate our results.

It is also difficult to draw conclusions on our results because of the limited breadth of data we've collected. In the future, we are interested in supplementing our analysis with more data, e.g. around the skill or experience differences between participants, to give us a better understanding of how those factors might affect how players work in Pyrus.

We designed Pyrus by drawing upon previous work in collaboration, self-regulation, and serious games. After conducting a study, we found this initial exploration into Pyrus's impacts on student behavior encouraging.


## Acknowledgements

We thank Eleanor O'Rourke, Garrett Hedman, and the members of the Design, Technology, and Research Community for their mentorship and guidance. This work was assisted by an Undergraduate Research Grant from Northwestern University's Office of the Provost.


## References


1. Theresa Beaubouef and John Mason. 2005. Why the high attrition rate for computer science students: some thoughts and observations. SIGCSE Bull. 37, 2 (June 2005), 103-106. http://dx.doi.org/10.1145/1083431.1083474
2. Jens Bennedsen and Michael E. Caspersen. 2007. Failure rates in introductory programming. SIGCSE Bull. 39, 2 (June 2007), 32-36. http://dx.doi.org/10.1145/1272848.1272879
3. Pierre Dillenbourg and Daniel Schneider. 1995. Mediating the Mechanisms Which Make Collaborative Learning Sometimes Effective. International Journal of Educational Telecommunications. 1 (2), pp. 131-146. Charlottesville, VA: Association for the Advancement of Computing in Education (AACE).
4. Allan Fisher and Jane Margolis. 2002. Unlocking the clubhouse: the Carnegie Mellon experience. SIGCSE Bull. 34, 2 (June 2002), 79-83. http://dx.doi.org/10.1145/543812.543836
5. Dastyni Loksa and Andrew J. Ko. 2016. The Role of Self-Regulation in Programming Problem Solving Process and Success. In Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16). https://doi.org/10.1145/2960310.2960334
6. Charlie McDowell, Linda Werner, Heather Bullock, and Julian Fernald. 2002. The effects of pair-programming on performance in an introductory programming course. SIGCSE Bull. 34, 1 (February 2002), 38-42. http://dx.doi.org/10.1145/563517.563353
7. Charlie McDowell, Linda Werner, Heather Bullock, and Julian Fernald. 2003. The impact of pair programming on student performance, perception and persistence. In Proceedings of the 25th International Conference on Software Engineering (ICSE '03).
8. Jaime Sánchez and Ruby Olivares. 2011. Problem solving and collaboration using mobile serious games. In Computers & Education, Volume 57, Issue 3, 2011, Pages 1943-1952, ISSN 0360-1315. http://www.sciencedirect.com/science/article/pii/S0360131511000935