PlanIT! A New Integrated Tool to Help Novices Design for Open-ended Projects

Alexandra Milliken
Wengran Wang
Veronica Cateté
Sarah Martin
North Carolina State University
Raleigh, North Carolina, USA
aamillik@ncsu.edu

Neeloy Gomes
Yihuan Dong
Rachel Harred
Amy Isvik
North Carolina State University
Raleigh, North Carolina, USA

Tiffany Barnes
Thomas Price
Chris Martens
North Carolina State University
Raleigh, North Carolina, USA
tmbarnes@ncsu.edu

ABSTRACT

Project-based learning can encourage and motivate students to learn through exploring their own interests, but introduces special challenges for novice programmers. Recent research has shown that novice students perceive themselves to be "bad at programming", especially when they do not know how to start writing a program, or need to create a plan before getting started. In this paper, we present PlanIT, a guided planning tool integrated with the Snap*!* programming environment designed to help novices plan and program their open-ended projects. Within PlanIT, students can add a description for their project, use a to do list to help break down the steps of implementation, plan important elements of their program including actors, variables, and events, and view related example projects. We report findings from a pilot study of high school students using PlanIT, showing that students who used the tool learned to make more specific and actionable plans. Results from student interviews show they appreciate the guidance that PlanIT provides, as well as the affordances it offers to more quickly create program elements.

CCS CONCEPTS

• **Social and professional topics** → **Model curricula**; **K-12 education**; **CS1**; • **Human-centered computing** → *Human computer interaction (HCI)*; • **Applied computing** → *Interactive learning environments*;

KEYWORDS

block-based languages, novice programmers, planning programming projects

ACM Reference Format:
Alexandra Milliken, Wengran Wang, Veronica Cateté, Sarah Martin, Neeloy Gomes, Yihuan Dong, Rachel Harred, Amy Isvik, Tiffany Barnes, Thomas Price, and Chris Martens. 2021. PlanIT! A New Integrated Tool to Help Novices Design for Open-ended Projects. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21), March 13–20, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3408877.3432552

SIGCSE '21, March 13–20, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8062-1/21/03…$15.00
https://doi.org/10.1145/3408877.3432552

1 INTRODUCTION

Introductory programming courses hope to teach students both "problem solving – devising an algorithm – and programming – converting the algorithm into [a program]" [15]. Increasingly, such courses focus on creative, open-ended coding projects, such as designing games [23], apps [4, 17], and simulations [11]. Such project-based learning can help broaden participation by improving learning and relevance to professional careers [20, 27] and improving motivation by connecting to student's personal interests and making an impact on the world [19, 28]. Open-ended programming projects can also support non-computing majors, who hope to use programming to create novel, meaningful artifacts, such as computational models, data analyses, and scientific simulations [10].

While creative and open-ended projects can be meaningful and motivating, prior work shows that students struggle with the design process needed to achieve them [26]. Furthermore, the resulting projects may be poorly-organized [9], with students struggling to explain how their programs demonstrate abstractions and algorithms. A few design tools exist to support students in this difficult process, but only those for UML diagrams have been successfully integrated with programming environments [15]. We present PlanIT, an integrated tool for planning open-ended projects. The goals of the system are to: (1) Provide planning scaffolds, (2) Teach effective planning processes, and (3) Make planning feel useful and worthwhile to students.

We conducted a pilot study using the PlanIT tool during a high school computer science internship program with 26 high school students. Students planned and programmed 3 open-ended game projects in Snap*!*, both with our tool and with a traditional planning worksheet. Half of the students used PlanIT for the second program and half for the third. Through analyses of interviews and the planning worksheets between the two groups, we found that students appreciated PlanIT's scaffolding, and incorporated elements from its interface into their later worksheet plans. The contributions of this paper are: (1) the design of a new planning tool that can be integrated with various programming environments, and its alignment to design criteria for design tools for teaching introductory programming, and (2) a study suggesting that the PlanIT digital

planning tool can support students to more effectively plan for open-ended programming projects.

2 RELATED WORK

Self-efficacy is a top early predictors of success in a CS1 course [22]. Many CS students experience low self-efficacy, causing them to drop out from CS courses [5]. In a survey study with 214 CS1 students to understand why they had lower self-efficacy, Gorson and O'Rourke found that students often formed negative self-assessments during programming, despite successes in completing programming projects. In particular, students were more likely to form negative self-assessments that led them to conclude they were bad at programming when they became frustrated by "not knowing how to start"; 15.42% of the students also incorrectly believed that needing to plan before programming shows low expertise [5].

However, the very problem-solving skills needed for planning are the same skills that have been shown to predict novice programmers' performance on programming problems [14]. In a survey of 559 novice students and 34 teachers on difficulties in learning and teaching programming, Lahtinen et al. found that *"design[ing] a program to solve a certain task"* is one of the most difficult aspects. Another research study on 12 undergraduate students creating pseudocode to solve programming problems Kwon found that students showed weakness in the strategic understanding needed to create useful plans that could be translated into programs [12].

Programming curricula have introduced planning as a part of the pedagogy into students' design and programming processes [7, 8, 26]. Jin's Cognitive Apprenticeship Learning [7] and related programming tutor frameworks with specific scaffolds for planning Jin et al., teach students to plan problem-solving strategies to construct closed-ended, short programs, yield higher learning gains over traditional approaches. Thomas et al. led a project called "Supporting Computational Algorithmic Thinking" (SCAT) that guided African-American middle school girls to design and code open-ended games for social change [26], and found that students have the most difficulties "articulating algorithms to describe user actions and related gameplay functionality and behavior", and also struggled in "expressing initial game idea". These findings show that while a guided planning experience is helpful to students in short programming tasks, there is still a need to facilitate planning for students creating open-ended programs.

In a 2018 systematic literature review, Luxton-Reilly et al. identified three main types of *design tools* to help students with problem solving for program design: pseudocode, flowcharts, UML diagrams. The pseudocode and flowcharting tools promoted learning, but none were integrated with programming environments. Authors of 2 main UML diagramming tools for introductory programming, *CIMEL ITS*[18] and *Green*[2], emphasized the need to integrate both plans and code into an integrated development environment to improve problem solving ability, to map plans to programs and vice versa, and to reduce the tedium of creating plans.

Alphonce and Martin created a set of 7 design criteria for the *Green* UML design tool: (1) *tailored plans*: aligning plans to constraints or affordances of the environment [1], (2) *code generation*: creating code from a plan, (3) *reverse engineering*: creating a plan from code, (4) *extensibility*: customizable functionality, (5) *run-time interactions*, (6) *refactoring support*, and (7) *set of relationships*. In the next section, we illustrate how we have enacted these criteria into the design of PlanIT.

[1]Tailored plans are called *restricted drawing* for UML diagrams in the framework

3 THE PLANIT TOOL

We designed PlanIT through an iterative design process with five cycles of low-fidelity prototyping and testing with non-majors in a college introductory CS1 course in Fall 2019. Our first prototype considered the elements needed to make a simple game in Snap*!*. We sought to create a flexible low-fidelity model for design plans that would allow us to gather user feedback and make quick refinements. In PlanIT v0.1, we used Google docs with instructional text for the project overview (description), actors (objects), important scene (an illustration of a prototypical moment in the game), game state (the minimal information needed to save and reload a game), a wishlist (things to create), and events.

We provided students with plans for implementing 2 assignments, then required them to use PlanIT v0.1 on Project 1. Based on plan grades and interviews with students on Project 1, we found that students misunderstood game states, and felt that planning was an extraneous and tedious task. Therefore, in PlanIT v0.2 we removed game states and required students to define just two events: one they knew how to program, and one they did not yet know. We also added a self-managed to-do list to track their progress. Our observations of students using PlanIT v0.2 showed that students (1) simultaneously edited their plans in no particular order, (2) realized that completing the plans helped them communicate and refine their designs, and (3) completed more interesting programs. This suggests that challenging students to plan complex things increased student perceptions of the value of planning.

3.1 Design Elements

In the next phase of the project we developed a standalone PlanIT v1.0 application using the Vue framework for integration into multiple environments. Layered over the Snap*!* interface using an iframe, PlanIT aligns well with Alphonce and Martin's design framework of seven criteria for design tools to support novice programmers in learning to design programs and translate those designs into programs. Figure 2 provides screenshots to illustrate how PlanIT implements Alphonce and Martin's framework. PlanIT has *tailored plans (TP)*, including creating actors and events, and *code generation (CG)* for actors and variables. It is also *extensible (E)* with free response components, e.g. the Important Scene description. PlanIT allows students to specify *sets of relationships (SR)* through events that relate actors and variables. Snap*!*, the current programming environment connected to PlanIT, provides *run-time interaction*. *Reverse engineering* and *refactoring support* in PlanIT are achieved by propagating changes to actors and variables.

3.2 Student Interface and Experience

We now present PlanIT's components: Description, Actors, Variables, Important Scene, Events, and To Do List. The **Description** and **Important Scene** components allow students to include text and a static image in their project plans, describing a critical decision point in their project. **My Actors** allows students to elaborate plans

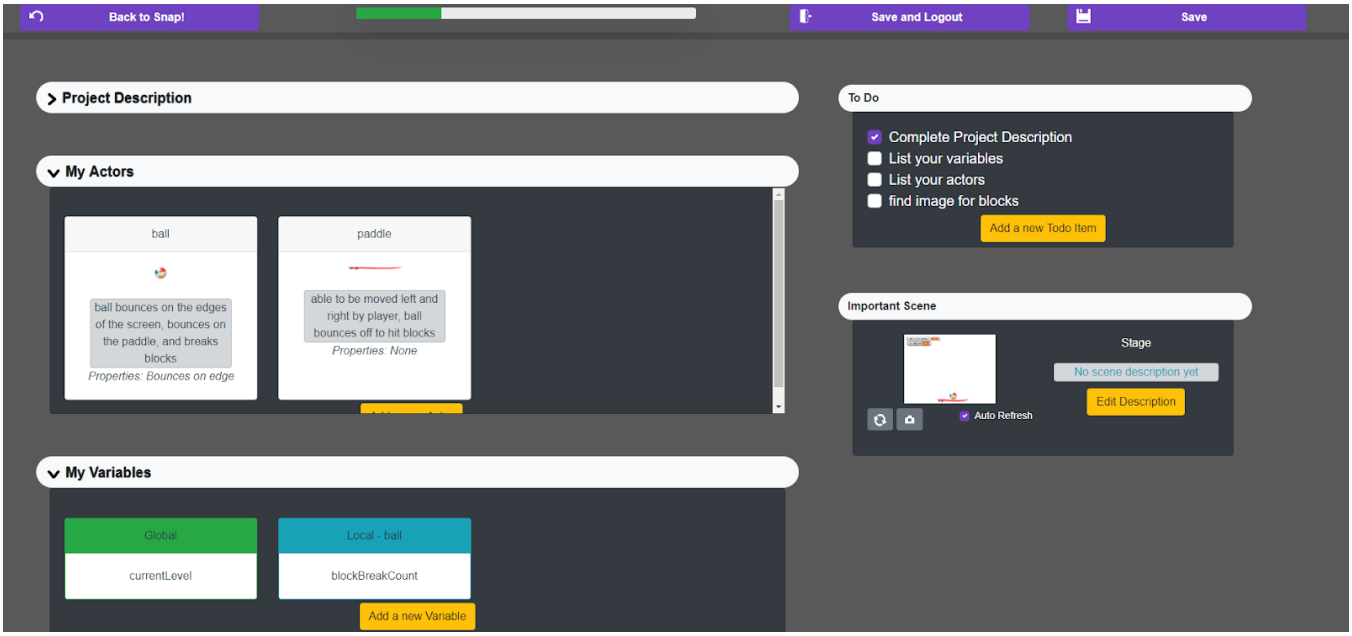

Figure 1: The PlanIT tool with Actors, Variables, To Do, and Important Scene components

including actor names, images, descriptions, and properties (e.g., "wrap around screen"). **My Variables** allows students to create, modify, or delete variables that they will need in their programming project. When a student adds a new variable, they specify its name and scope. PlanIT supports *code generation* and *refactoring support*, for both Actors and Variables by automatically adding or updating them in Snap*!*. Figure 1 shows the PlanIT interface with ball and paddle actors, a global variable called currentLevel, and a local variable for the ball actors, called brickBreakCount. The **Events component** has three options: (1) create a new event from an existing list of actions and triggers, (2) create an event from the example gallery with example behaviors with animated gifs, and (3) create a custom open-ended event. Figure 1 shows an event being created, using the trigger "When game starts" and action "change backdrop". Planning an event does not create code for students in Snap*!*, only a plan for creating an event. **To Do List** helps students manage their planning process and implementation by adding, modifying, completing and deleting tasks using a visual progress bar. The To Do List starts with three default tasks ("complete project description," "list your variables," and "list your actors") to encourage students to interact with the respective components.

4 METHODS

We performed a pilot study to evaluate the effectiveness of PlanIT in meeting our goals of 1) providing scaffolding, 2) teaching planning, and 3) making planning worthwhile. The study design involved students planning and programming three games, with all students planning and programming the first game using a traditional **planning worksheet**, containing a project description text area and instructions to use the remaining space to write down their plans, which may include text and images. For the second game, pairs of students were assigned randomly into 2 groups with the Early group using PlanIT and the other using the worksheet. For the third game, the Late group used PlanIT and the Early group used the worksheet. Students were interviewed after each game planning and programming process. Both groups planned and programmed

Figure 2: PlanIT provides (TP) tailored plans, (CG) code generation, (E) extensibility, and (SR) set of relationships

2 projects using a traditional planning worksheet and one with PlanIT, with the Early group using PlanIT between the two worksheets, and the Late group using PlanIT afterwards.

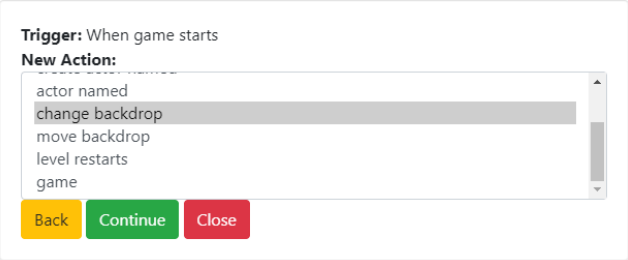

Figure 3: A contextual PlanIT menu for event creation

Participants: We recruited 26 high school students from an unpaid summer internship in CS at a large, public research university. Participants self-reported their gender (male/ female/ nonbinary/ prefer not to say) with 7 males and 19 females, and could select all that applied for race/ethnicity, with 2 White, 2 Black/African American, 19 Asian, 1 Other, and 2 Multiracial. The students self-reported their programming skill as "a little" (5 students), "some" (15 students), and "very strong" (6 students). We paired similar performing students together based on their performance within Crescendo, a self-paced coding practice tool [29].

4.1 Procedure

In the morning on Day One, all the students completed the same set of activities synchronously within the same Zoom meeting. In the morning, students completed a pre-survey, became familiar with Snap*!*, completed Crescendo activities and became familiar with planning and pair programming. The pre-survey collected their (1) experiences and opinions with coding, (2) completed computer science courses, and (3) experiences with planning past programming projects. In the afternoon, students planned and coded a simple arcade game, Asteroids, in their assigned pairs. The researchers reviewed instructions for the Asteroids activity and outlined the day's schedule with all the students in the main Zoom room. The researchers then opened the Zoom breakout rooms for pairs to plan their Asteroids game. They were given 20 minutes to complete the planning worksheet, where they could create a description for their program and openly plan any other relevant information. After planning, pairs shared their plans with the group and then returned to their breakout rooms to pair program for an hour.

On Days 2 and 3, we evenly split the student pairs between two groups. We define Early Pairs as those who used PlanIT on Day 2 and Late Pairs as those who used PlanIT later, on Day 3. On Day 2, students planned and implemented a breakout-style game. Breakout-style games are a subclass of the "bat-and-ball" genre. In a breakout game, the "bat" is a paddle, and the goal is to continuously bounce the ball off of the paddle, causing it to hit rows of tiles above it, eventually breaking all tiles. Students were shown a breakout-style game, but they were explicitly encouraged to create an original program of their own design. To help students create events for their Breakout game, the Early group had access to code example behaviors in addition to PlanIT [2]. The Late group used the planning worksheet, and had no access to the extra support provided by code examples during programming.

After an introduction to the programming activity and their respective planning environments, student pairs went to their individual breakout rooms and planned for 20 minutes. Students shared their ideas with the other students within the same condition and went back into their breakout rooms to program in pairs for 2 hours. They were allowed to ask for help when they got stuck on something during implementation. On Day Three, we had the same schedule as Day Two, but the Late group now had PlanIT and Example support. On this day, students programmed a game in the style of Space Invaders, a game where the user shoots at a group of enemies who are moving intermittently and shooting back at the user. On Day Two and Three, at the end of the day, we conducted interviews with each pair of students, interviewing pairs together, since time was limited and we were interested in the collective experience of the pair.

4.2 Data and Analysis Methods

We analyze student interactions within PlanIT, interviews, survey data, and planning artifacts to evaluate PlanIT's effectiveness for meeting our goals, as stated in section 1. We analyzed **PlanIT log data interactions**, e.g. adding an actor from the Actor menu, or unchecking a to-do item, to determine which were the most common. Two coders analyzed the interviews and planning worksheets by open coding for **thematic analysis** [1, 16]. The two coders completed their open coding blind to students' condition, using the following steps: (1) Become familiar with the data, (2) Generate initial codes of 25% of the data (individually), (3) Discuss initial codes and combine them into an initial codebook, (4) Two authors individually code all interview data with the codebook, (5) Meet, discuss, refine codes, (6) Two authors review all data using refined codes, and (7) Organize each tag into themes (together).

5 RESULTS AND DISCUSSION

5.1 Comparison of Planning Worksheets

To evaluate how well PlanIT met goal 2 to teach planning, we investigated its impact on students' planning processes outside the tool. We investigated how the 7 pairs in the Early group completed their *planning worksheet* on Day 1 (before using PlanIT) and on Day 3 (after using the tool). Two authors used open coding (described in Section 4.2 to identify 25 worksheet planning elements (e.g. used pseudocode, used conditionals, planned game mechanics), and categorized the level of planning detail as None, Low, Medium, or High. We then identified which planning elements changed for each pair between Day 1 and Day 3. We found that all 7 pairs in the Early group made meaningful changes to the way they completed the planning worksheets, respectively adding [3, 3, 2, 6, 2, 4, 2] elements each. Specifically, we found *all 7* pairs added PlanIT-inspired components (events, actors, variables, to do lists), 5 added references to specific programming concepts (input/output, conditionals, loops), 4 added other elements (e.g. game mechanics or specific Snap blocks) and 5 pairs increased the level of detail in their

[2]We provided the examples as *additional* support to help students program effectively, but we are not evaluating the effect of the examples in this paper.

descriptions. Pairs also removed a smaller number of elements, [3, 1, 1, 0, 0, 2, 0] respectively. Specifically 2 pairs removed programming concepts (conditionals, loops) and 2 pairs removed other elements (pseudocode, specific Snap blocks, pair programming roles).

These changes reflect some planning elements that are supported by PlanIT, which emphasized the role of actors, detailed descriptions, and the importance of planning variables, but did not support pseudocode, and did not emphasize the use of specific blocks. However, some elements of PlanIT (e.g. planning events) were not reflected in the majority of the students' future planning worksheets. This may be due to the amount of scaffolding built into event creation which heavily guided students during this process, i.e. making it more difficult to do or write down outside the tool than within it, or because students used the Events component less than they used the Actor, Variable, and To Do List components.

It is possible that some of this change in planning behavior was due to the experience of having completed projects between Day 1 and 3 (not just the use of PlanIT). Therefore, we also investigated differences between the Late Pairs' planning worksheets on Days 2 and 3. These 5 Late Pairs had also completed a project (on Day 1), but had not used PlanIT. We found that there were few changes in their planning worksheets. Two pairs used no different planning elements. Of the remaining 3 pairs, one added a unique title, one added references to specific blocks, and one (who had no plan on Day 1) added pseudocode, and programming elements. This indicates that the changes we saw in the Early groups' planning behavior were likely due to exposure to PlanIT. Our results suggest that by explicitly scaffolding the planning process (goal 1), a tool can also shape the way that students plan projects in the future, meeting goal 2. However, we also note that some of the students' natural planning behaviors (e.g. using pseudocode) should also be supported in PlanIT.

5.2 Interview Themes

Interviews took place on days 2 and 3 after they completed their programming projects. We used the thematic analysis described in Section 4.2 to determine the following themes, which provide evidence that PlanIT has met all three of its goals of scaffolding, teaching, and making planning worthwhile:

- Students appreciated integrated planning
- Students liked planning structure and support
- Students found PlanIT useful for a variety of planning tasks
- Students learned from PlanIT
- Students wanted more from PlanIT

In the following sections, participants are differentiated with numbers (e.g., P1, P2), and their gender and race are noted at the first instance using the following code: Asian (A), Black (B), Caucasian (C), Multi-racial (R), Male (M), and Female (F).

Students appreciated integrated planning An important feature of PlanIT is how it directly integrates a planning tool into the Snap*!* coding environment, allowing students to directly create code elements (e.g. Sprites, variables) as they plan. Students appreciated this tight integration, illustrating how PlanIT met goal 3 of making planning worthwhile: "the most helpful part of the Planner was... being able to make your sprites and your variables, so when we came back to it it was just already all set up" (P8-FC). Students noted this saved them time: "I feel like the [worksheet] took longer than [PlanIT] since... you can create and set variables and connect them to the things" (P14-FR). One student noted that PlanIT "doesn't feel like wasting time 'cause normally planning can feel like that" (P4-FA). Students also noted areas where PlanIT did *not* seem well- integrated with Snap*!*, such as when creating *events*. One student noted that events "take too much time, so we didn't put it in [our plan]" (P7-FA). Students felt like they "didn't really quite get the point of doing the events" (P3-FA), and log data confirms that events were one of the least used PlanIT features. We hypothesize that lack of direct integration with Snap*!* was a primary cause. Our results support prior work that students actively weigh the utility of programming support [21], and want to spend their time effectively working towards creating their project. Planning tools can therefore make planning feel worthwhile by integrating with programming environments and translating students' plans into progress towards implementation. This finding is consistent with phenomena described by Suchman, who suggests that the planning process of a design is not an isolated experience, but is rather an interconnected component of the context of the plan [24].

Students appreciated structure and support when planning This theme provides evidence that PlanIT meets goal 1 of scaffolding planning into steps, guiding students from determining a generic theme and description of the design, to determining actors, and then to defining events. Students appreciated this guided experience, because it " helped organize things in a way I didn't think about before" (P4-FA). Students also appreciated that PlanIT offers them cues to think of particular program elements, making their plan" more specific and thorough" (P6-FA). The scaffolded experience of PlanIT not only helped students *plan strategically*, but also prepared them to *program* in specific steps. For example, students explained that "with the to do list, and knowing what variables and what events and what features we were going to code beforehand, listing them all out, it made it a lot easier to mentally start coding and separate it into parts" (P12-FA). In comparison with the worksheet planner, students explained that PlanIT prevented them from " [diving] right in" (P12) and feeling "overwhelmed" (P12). Given the context of prior research showing that beginning CS students struggle to articulate structured programming strategies without explicit support [12], we believe that the scaffolded experience and structured components offered by PlanIT may help students bridge the gap between their own natural design language and the Snap*!* programming language.

PlanIT supported students in a variety of ways This theme provides evidence of goal 3 of making planning worthwhile. Students noted the system had organizational benefits: it "helped us with the logistics" (P5), and "helped us keep organized" (P4). PlanIT's workflow helped students start on a large, complex project, as it "prevented us from getting overwhelmed since we knew where to start planning" (P12). PlanIT also helped students to plan ahead: it "allowed us to think ahead more about what we were going to code" (P6). Students noted that PlanIT helped make their ideas more concrete, to "take your concepts and ideas, and actually turn it into something you can use" (P5-FA). These benefits – organization, decomposition, forethought and reifying ideas – are all proposed benefits of planning in the literature [6]. Some students noted that these benefits were *not found* with the planning worksheet. PlanIT

"was a lot more helpful than just using [the worksheet]," which "wasn't able to actually visualize the code we were gonna use to carry out the concept we had in mind" (P5). A student from the post survey said that "when we used the digital planner, it was easier to visualize...certain things before actually starting and I feel that we were able to accomplish more of what we wanted to do" (P8). This suggests the extra scaffolding provided by PlanIT was necessary to achieve these benefits, also addressing design goal 1.

Students learned from PlanIT This theme provides evidence for goal 2 of teaching planning proceses. Students felt that PlanIT taught them an alternative way to plan out a programming project. One student thought PlanIT "[gave] good insight about what is the **best way/process to organize code** for [a] specific function" (P9-FR). To further support this, a student from the post survey concurred and said "the planning tool was really helpful because **before I would've always just made a flowchart** (like the standard programming flowchart with rectangles/circles/parallelogram) and **never plan out the variables and actors, or just jump straight into the project and not plan at all. It showed me a way to effectively plan a program which also helped us use the Worksheet Planner**" (P5). On the post survey, one student explained "on the first day of the study I did not specifically plan out variables and events on the [worksheet], but after using [PlanIT] on the second day, I was inspired to specifically plan out the variables on the [worksheet] on the third day" (P10-FA). This agrees with our other results from Section 5.1, showing that students incorporated elements of PlanIT into their subsequent planning worksheets. This result suggests these students transferred the skill they learned to future tasks, addressing design goal 2. We found that the scaffolded experience with PlanIT not only serves as a tool for students to plan and program strategically within the task. It could also serve as a "worked example" [25] for students to learn the step-by-step planning strategies, and apply it for future programming tasks.

Students want more from PlanIT From the post survey and interviews, there were some suggestions for improving PlanIT to provide additional features and concepts that students often incorporated into their planning process when using the planning worksheet. One of these suggestions included having a place within PlanIT where students can jot down additional, unstructured notes. One student suggested "maybe like a section for additional notes that would go in the project description...so I wouldn't forget" (P9) smaller details or "miscellaneous ideas" they wanted to remember. Another suggestion was to provide space for students to pseudocode events or algorithms within their project. For example, one student said they wanted to "plan some pseudocode beforehand" to allow "some way to organize out [their] thoughts [on] how to implement [their] ideas in the planning into code" (P11-FA,12). Another suggestion for improving PlanIT was to have a more advanced and formattable To Do List component which would allow for subtasks or sections for organization. One student said they would like to "add smaller bullets under large ones, therefore [they] can organize your tasks from large general ideas to the smaller details" (P11). One suggestion requested by multiple students was the ability for students in a pair to simultaneously edit PlanIT (much like in a Google Document).

5.3 Most Useful Elements

During the interviews, many participants found it useful to plan for actors and variables, with five out of six Early group pairs mentioning its usefulness during post interviews on Day 2, and all five Late Pairs mentioning its usefulness on Day 3. One pair stated that the most helpful part of PlanIT was "being able to make your sprites and your variables" that, when switching to Snap, were "already all set up to go" (P17-MA,18-MC). These findings are supported by our analyzed trace data showing that adding actors and variables are some of the most commonly performed actions in PlanIT. Two pairs from the Early group mentioned the usefulness of the to do list during interviews on Day 2, with one stating that the "to-do list was very helpful" (P15-MA, 16-FC), and that they ended up creating a to-do list in the Planning Worksheet on Day 3 while "thinking of the snap planner used previously" (P11, 12). When it came to events, however, there was no overwhelmingly positive feedback from the participants. One participant pair found the events to be helpful (P15, 16). One participant felt "only new programmers would need to plan events because the events contain simple logistics" that more experienced programmers already understand. This feedback is supported by the trace data, which shows participants adding events only 8 times over the course of the study. Interestingly, participants added twice as many events over the course of the study while using the example gallery, adding a total of 15 example-based events. This is likely because adding events based on examples takes only a few clicks, while adding them manually requires users to make several selections in a multi-step process.

Checking off to-do list items was also a common action, likely due to the pregenerated to-do items. People want to check off to-do items to stay oriented toward their planned goals and to avoid dissatisfaction by leaving items unchecked [3], and this is no different for the participants of this study. Overall, the common actions relevant to the study of the students' planning process in an open-ended project suggest that students will make more use of PlanIT features directly affecting the Snap interface.

6 CONCLUSION

The contribution of this work includes a new integrated planning tool and a study demonstrating how it can help novice programmers plan challenging open-ended, creative projects in a meaningful and worthwhile way. Our results show that PlanIT has met goal 1 to provide students with scaffolding to make plans, especially via tailored plans for Snap*!* elements. Toward goal 2 of teaching planning, PlanIT influenced students to incorporate more elements from the tool (e.g. actors, variables) into future plans outside the tool. However, we also found that students were less likely to do some kinds of planning after using PlanIT de-emphasized other elements (e.g. pseudocode). This means that tools have the potential to shape how students plan, and also poses the potential for unintended consequences. Finally, our results show that when PlanIT meets the design criteria of code generation and reverse engineering that provide tight integration between plans and code, students find planning to be useful and worthwhile. Our investigation also revealed potential areas for improving PlanIT to meet targeted design criteria, with requests for more extensibility and the ability to more easily express a meaningful set of relationships.

7 ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under grant number 1917885. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Mohammed Ibrahim Alhojailan. 2012. Thematic analysis: A critical review of its process and evaluation. *West East Journal of Social Sciences* 1, 1 (2012), 39–47.

[2] Carl Alphonce and Blake Martin. 2005. Green: a customizable UML class diagram plug-in for Eclipse. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 108–109.

[3] Adam Burke, Cristi Shanahan, and Eka Herlambang. 2014. An exploratory study comparing goal-oriented mental imagery with daily to-do lists: Supporting college student success. *Current Psychology* 33, 1 (2014), 20–34.

[4] Dan Garcia, Brian Harvey, and Tiffany Barnes. 2015. The beauty and joy of computing. *ACM Inroads* 6, 4 (2015), 71–79.

[5] Jamie Gorson and Eleanor O'Rourke. 2020. Why do CS1 Students Think They're Bad at Programming? Investigating Self-efficacy and Self-assessments at Three Universities. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*. 170–181.

[6] A Gwande. 2010. The checklist manifesto. *New York: Picadur* (2010).

[7] Wei Jin and Albert Corbett. 2011. Effectiveness of cognitive apprenticeship learning (CAL) and cognitive tutors (CT) for problem solving using fundamental programming concepts. In *Proceedings of the 42nd ACM technical symposium on Computer science education*. 305–310.

[8] Wei Jin, Albert Corbett, Will Lloyd, Lewis Baumstark, and Christine Rolka. 2014. Evaluation of guided-planning and assisted-coding with task relevant dynamic hinting. In *International Conference on Intelligent Tutoring Systems*. Springer, 318–328.

[9] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code quality issues in student programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. 110–115.

[10] Amy J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, et al. 2011. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)* 43, 3 (2011), 1–44.

[11] Michael Kölling. 2010. The greenfoot programming environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 1–21.

[12] Kyungbin Kwon. 2017. Novice programmer's misconception of programming reflected on problem-solving plans. *International Journal of Computer Science Education in Schools* 1, 4 (2017), 14–24.

[13] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A study of the difficulties of novice programmers. *Acm sigcse bulletin* 37, 3 (2005), 14–18.

[14] Alex Lishinski, Aman Yadav, Richard Enbody, and Jon Good. 2016. The influence of problem solving abilities on students' performance on different assessment tasks in CS1. In *Proceedings of the 47th ACM technical symposium on computing science education*. 329–334.

[15] Andrew Luxton-Reilly, Ibrahim Albluwi, Brett A Becker, Michail Giannakos, Amruth N Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory programming: a systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. 55–106.

[16] Moira Maguire and Brid Delahunt. 2017. Doing a thematic analysis: A practical, step-by-step guide for learning and teaching scholars. *All Ireland Journal of Higher Education* 9, 3 (2017).

[17] Ralph Morelli, C Uche, P Lake, and L Baldwin. 2015. Analyzing Year One of a CS Principles PD Project. In *Proceedings of the ACM Technical Symposium on Computer Science Education*. 368–373. http://dl.acm.org/citation.cfm?id=2677265

[18] Sally H Moritz, Fang Wei, Shahida M Parvez, and Glenn D Blank. 2005. From objects-first to design-first with multimedia and intelligent tutoring. *ACM SIGCSE Bulletin* 37, 3 (2005), 99–103.

[19] Kylie A. Peppler and Yasmin B. Kafai. 2007. What Videogame Making Can Teach Us about Literacy and Learning: Alternative Pathways into Participatory Culture.. In *Proceedings of the Digital Games Research Association Conference*. http://eric.ed.gov/?id=ED521155

[20] Beatriz Pérez and Ángel L Rubio. 2020. A project-based learning approach for enhancing learning skills and motivation in software engineering. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 309–315.

[21] Thomas W Price, Joseph Jay Williams, Jaemarie Solyst, and Samiha Marwan. 2020. Engaging Students with Instructor Solutions in Online Programming Homework. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–7.

[22] Keith Quille and Susan Bergin. 2018. Programming: predicting student success early in CS1. a re-validation and replication study. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. 15–20.

[23] Alexander Repenning, Ryan Grover, Kris Gutierrez, Nadia Repenning, David C. Webb, Kyu Han Koh, Hilarie Nickerson, Susan B. Miller, Catharine Brand, Ian Her Many Horses, Ashok Basawapatna, and Fred Gluck. 2015. Scalable Game Design. *ACM Transactions on Computing Education* 15, 2 (apr 2015), 1–31. https://doi.org/10.1145/2700517

[24] Lucy Suchman. 1987. Plans and Situation Actions: The Problem of Human Machine Communication. (1987).

[25] John Sweller, Jeroen JG Van Merrienboer, and Fred GWC Paas. 1998. Cognitive architecture and instructional design. *Educational psychology review* 10, 3 (1998), 251–296.

[26] Jakita O Thomas, Yolanda Rankin, Rachelle Minor, and Li Sun. 2017. Exploring the difficulties African-American middle school girls face enacting computational algorithmic thinking over three years while designing games for social change. *Computer Supported Cooperative Work (CSCW)* 26, 4-6 (2017), 389–421.

[27] Elizabeth T Turner. 2012. Meeting learners' needs through project-based learning. *International Journal of Adult Vocational Education and Technology (IJAVET)* 3, 4 (2012), 24–34.

[28] Ian Utting, Stephen Cooper, and Michael Kölling. 2010. Alice, Greenfoot, and Scratch – A Discussion. *ACM Transactions on Computing Education* 10, 4 (2010). http://dl.acm.org/citation.cfm?id=1868364

[29] Wengran Wang, Rui Zhi, Alexandra Milliken, Nicholas Lytle, and Thomas W. Price. 2020. Crescendo: Engaging Students to Self-Paced Programming Practices. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 859–865. https://doi.org/10.1145/3328778.3366919