Exploring Novice Programmer Example Use

Michelle Ichinco and Caitlin Kelleher
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri, United States
{michelle.ichinco, ckelleher}@wustl.edu

***Abstract*— Both experienced and novice programmers use examples while programming, whether from tutorials, forums, or source code. Novice programmers, however, often find it challenging to use unfamiliar example code. Little is known about the challenges of using examples, making it difficult to design support for novice programmer example use. We ran an exploratory study of novices using examples to complete programming tasks. To analyze programming behaviors, we define the 'realization point' as the time when the participants discover the crucial concept in an example. Our results show that participants spend more time after the realization point using the concept from the example than they do identifying which part of the example to use. We describe hurdles and strategies, types of tasks and their behaviors, and finally, implications for supporting example use.**

Keywords—novice programming; code examples; code reuse

I. INTRODUCTION

Programmers of all skill levels leverage example code while performing programming tasks [1]. Though experienced programmers can easily understand and integrate example code into their programming problem, inexperienced programmers often have a much harder time understanding how to use the example code in their own programs [2]. Two significant populations susceptible to this problem are school-aged novice programmers using systems like Scratch [3] and Looking Glass [4], and end-user programmers [5].

While prior work found that novices struggle to make effective use of programming examples, the specific causes are not clear. Yet, when used effectively, examples can be a powerful learning resource. For example, research in mathematics education found that an example based approach enabled middle school students to learn three years worth of algebra in two years [6]. A deeper understanding of the ways in which novices attempt to solve problems using examples and the ways in which they struggle could inform the design of new example supports within novice programming environments [3], [4] and potentially lead to greater educational impact. We aimed to answer two questions: 1) what hurdles do novice programmers encounter and 2) what strategies do they use while attempting to use examples?

To answer these questions, we ran an exploratory study of novice programmers using example code to solve programming tasks. Using transcripts of participant conversations and logs of program interactions, we describe the strategies novice programmers used and what challenges they encountered. We then relate the strategies and hurdles to a new concept we define in this paper, the 'realization point'. Finally, we suggest how systems can support novice programmers in overcoming hurdles.

II. RELATED WORKS

We situate this work within two main bodies of research: A) understanding the behaviors of programmers, especially those using examples or reusing code, and B) existing systems designed to support example code use for programmers. These bodies of work span two use cases of programming with examples: using example code as a resource for learning and reusing example code in a new program. However, this work is relevant in both cases, as identifying barriers to understanding example code has the potential to assist in either scenario.

A. Understanding programmer behavior

This work is based around two areas of work on programmer behavior: programmers using examples or reusing code and general programming behaviors. Studies on programmer behavior while reusing code do not focus on in-depth analysis of novice programmer example use, while studies of non-expert programming behavior inspire our study.

1) Examples and code reuse

Several studies seek to understand how programmers use example code naturally, though some focus on experienced programmers as opposed to novices. One such study explores experienced programmers reusing example code and, similar to this work, describes the programmers' behaviors during tasks using example code [7]. Rosson and Carroll find that expert programmers 'debugged into existence' and only used examples as an initial source of information. It is important to note that this study took place before example code was widely available online. Another study looked at how programmers search the internet throughout programming tasks and find that programmers used online code examples for learning and reminding themselves of what they already know [1]. They also discovered that programmers started to use code they found before fully understanding it and made mistakes while adapting copied code. However, we do not know how these behaviors and problems apply to novices.

Few studies focus on non-expert programmers and those that do only briefly discuss code examples and reuse as a part of larger works. In analyzing the practices of informal web

development, Rosson, Ballin and Nash found that programmers often use example code as a model when looking for general ideas of ways to design websites [2]. However, when they try to use the code, the programmers cannot effectively integrate it into their projects. To our knowledge, no work focuses specifically on the behaviors, strategies, and confusions of novice programmers using examples.

2) End user and novice programming behavior

This work was inspired by the study design and analyses of previous research on the challenges and strategies of non-expert programmers. At a high level, researchers have studied the general behaviors exhibited during programming, such as debugging [8] and barriers in learning programming [9]. Both of these studies collected transcriptions to find out what confused participants, similar to our study. More specific to reusing code, research has also investigated the behaviors of non-expert programmers during mashup programming [10] and when attempting to locate functionality in unfamiliar code [11]. However, the body of research on novice and end user programming behavior lacks a focused study of example use.

B. Supporting programmers' example use

Based on the existing work on examples in education and programming, educational systems and programming environments often include support for using examples.

1) Educational systems

Educational systems for programming often provide examples similar to this study, where code is available along with the ability to run the code. Researchers have worked on example selection [12], as well as presenting examples as learning material for learning programming [13]–[15]. Another tool, the 'Explainer', provides support for learning from programming examples based on previous learning theories, by allowing programmers to view multiple forms of the example as well as programming plans [16]. Redmiles found that with Explainer, participants were more consistent and direct in how they completed tasks. Yet, these studies focus primarily on the design of the systems, as opposed to understanding how novice programmers use examples and what issues they have.

2) Programming systems

End user programmer systems focus on enabling correct selection of examples and supporting the repurposing of example code, but tell us little about what programmers are confused about as they try to use the examples. Some tools, like Blueprint [17] and Fishtail [18], integrate example search into programming environments, improving programmers' abilities to search for examples without having to switch contexts. Other tools integrate with web browsers. Mica [19] and Codetrails [20] augment searches to improve the results for programmers looking for examples. In addition to improved example search, Snipmatch also supports integration into code, similar to Codelets and Webcrystal [21]–[23]. On the other hand, Looking Glass provides a way for novice programmers to select which part of a larger program they want to reuse [24]. These studies often compare programmers' success with and without the tools, but do not address the behaviors of programmers using examples. In this work, we seek to describe how novices utilize examples when programming.

III. Study Setup

We ran an exploratory study to understand the hurdles encountered and strategies used by novice programmers working with examples.

A. Participants

We recruited 21 children aged 10-15 from the St. Louis Academy of Science mailing list. We screened participants to ensure that they had three or fewer hours of programming experience. Three children had more than three hours of programming experience, so they participated in another study instead. Our 18 participants had an average age of 11.4 (σ = 1.4); 10 participants were female.

For each session, we randomly assigned participants to pairs, such that in the end, we had 9 pairs of participants. We had participants work in pairs because formative work showed that children were not actively 'thinking out loud' on their own, even when instructed to do so. We found that this strategy was effective in getting most pairs to have continuous conversations about the tasks, but we acknowledge that having the participants work in pairs changed the dynamics of the situation and likely improved performance.

B. Materials

We augmented a novice programming environment with examples and authored completion programs and examples.

1) Looking Glass

We conducted this study using Looking Glass [4], a drag and drop novice programming environment where the output of a program is a 3D animation. In this study, we augmented Looking Glass with example code in an un-closable dialog box, as shown in Fig. 1-C. The example always had a red outline around the important concept for emphasis. We chose to provide this emphasis because early testing indicated that this red highlighting could assist novice programmers in identifying the important part of example code. We did this instead of comments to describe the example, like in [17], [25], because our goal was to find out what problems novices have using examples. Having an explanation of the example might help novices to use the example, while we believe the emphasis merely gave direction without explanation. Furthermore, most examples that programmers find online are not annotated, though this type of emphasis could be added automatically.

2) Completion programs

We created six completion programs based on six concepts of varying difficulty, selected based on formative testing. Each program completion task focused on a unique programming concept: simple parallel execution, a for loop, an unfamiliar API method, using a function as a parameter, a while loop condition, and a for each loop iterator. The instructions for each task ask participants to add to or modify the given program to create a specific animation. The completion program was always a very simple program, only including basic programming statements similar to those participants had seen in the training task. The solution for each task required adding the complex concept in the example code, such as simple parallel execution, as shown in Fig. 1-C.

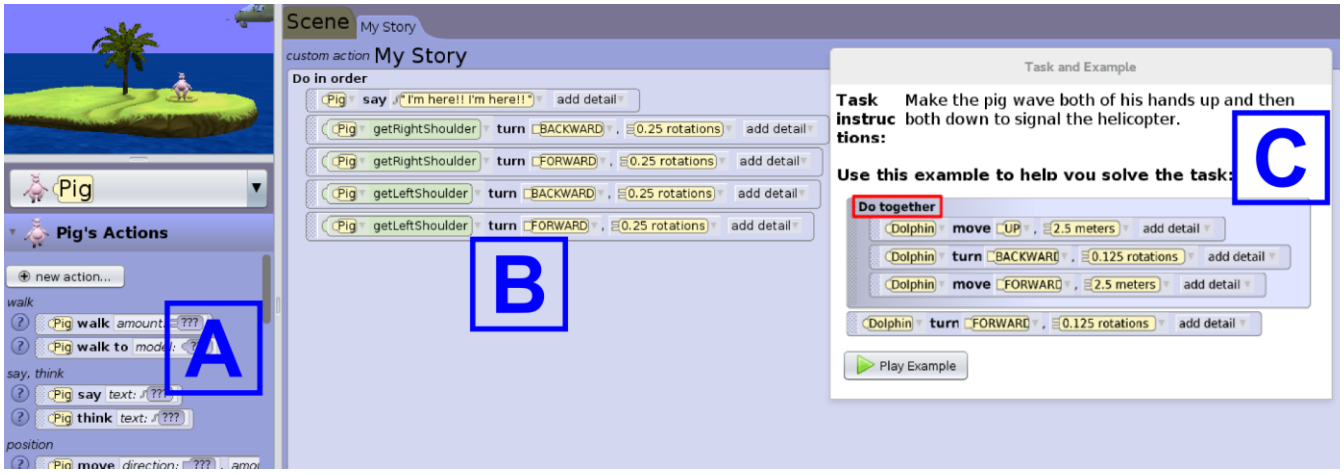

Fig. 1 A) Participants can drag and drop the programming blocks from this menu into their program. B) The program shown here is a completion program given to a participant at the beginning of the task. We asked participants to modify the programs. C) This dialog was added to the programming environment only for this study. It includes the task instructions and the example code.

3) Examples

We created a code example for each program completion task to simulate a well-selected example found online. Each example contained the concept necessary to complete the associated task. However, in order to prevent the tasks from being obvious, we used formative testing to ensure that the example did not directly map to the solution. For example, in Fig. 1-B, participants needed to add two 'Do together' blocks and rearrange the statements, while the example only shows one block in Fig. 1-C.

C. Study Design

This study took a total of 90 minutes. Participants completed a demographic and computing history survey, a training task, and six program completion tasks, as shown in Fig. 2. If participants finished early, they were allowed to work on optional program completions (which we did not analyze), or create their own program. In this work, we focus only on the six programming completion tasks. We allowed participants to ask questions at any point during the study.

1) Training Task

Pairs first completed a training task that was designed to familiarize them with the programming environment. They received an instruction sheet with directions and images that showed where to find essential elements in the interface.

2) Program Completion Tasks

Pairs then completed six program completion tasks, which they saw in one of six orders that were balanced across participants. In these tasks, participants worked on completing a program (Fig. 1-B), given instructions and an example (Fig. 1-C). For each program completion task, participants had a total of eight minutes to work on the task, split into two four-minute halves. We selected the number of tasks and task times based on formative and pilot studies.

Fig. 2 Each pair completed one training task and six programming tasks

After the first 4 minutes of the task, there was a mid-task interview, during which the researcher asked the participants questions about what they had tried so far. The purpose of this interview was to encourage participants to discuss their thought process and to encourage participants to use the example, if they had not yet used it. Encouraging participants to use the example halfway aligns with our goal of understanding example use. At the end of the mid-task interview, pairs had another 4 minutes to complete the task. We encouraged participants to keep trying if they told us they completed the task but it was not correct, which likely increased success rates. Once the task was complete, the researcher performed a final interview for that task.

D. Data

We collected demographic and computing history survey data, logs from the programming environment during the sessions, audio logs, and task programs.

IV. Analysis Methods

We analyzed completion program correctness and audio recordings from this study.

A. Program Correctness

We scored each task as either correct or incorrect based on the instruction critera given to participants. In four cases, tasks did not fit one of the criteria, but they used the correct concept fully and correctly, so we also marked those as correct. For example, one criteria was to not add extra code blocks into a loop task to ensure that they used the loop instead of repeated code blocks. However, if the resulting code used the loop correctly but they had extra code statements added elsewhere, we still counted solutions as correct.

B. Audio Recordings

To analyze the audio recordings, we transcribed them, created two sets of labels to categorize the focus area and processes, and determined the 'realization point' for each task.

1) Transcription

We transcribed a total of 7.6 hours of audio from the program completion tasks in order to analyze what participants were saying as they completed the tasks. We then broke the transcriptions up into segments in which participants were focused on a single topic, such as a question and an answer.

2) Labels

We created two sets of labels to categorize how participants spent their time during programming tasks with examples. We wanted to know 1) which part of the interface or task the participants were focusing on, and 2) what they were doing or talking about within that context. To capture these ideas, we created two sets of labels, one for the focus area of the statement (such as the instructions or the example code, shown in the top of Table I), and the other for the process the participants were completing at that point (such as describing something or talking about an idea, shown in the bottom of Table I). We then labeled the transcriptions. To obtain inter-rater agreement, two authors took a set of 20% of the task transcripts and iterated four times on 1/5 of that set to help clarify the labels. The authors then achieved >80% agreement on the whole 20% set labeling the transcripts independently. One author labeled the remaining transcriptions.

3) Realization Point

Through our analysis of the audio transcriptions, we discovered that all but two tasks had a definitive point when the participants first noticed which part of the example to use. We believe this is a valuable feature of example use, and call the point when a participant first talks about the critical element of the example the 'realization point'. We believe that identifying realization points and looking at behavior before and after the realization points is a new way of analyzing the behavior of programmers working with examples. The realization point separates the task into two parts: 1) the time before participants know what concept to use, and 2) the time the participants spend trying to figure out how to apply that information to the task code.

We can objectively determine the point in the transcription when one of the participants first mentions the necessary concept in the example. One possible limitation of the realization point is that participants may have thought about the concept before they said it out loud. The natural flow of conversation between participants in most pairs, however, makes it likely that that participants talked about their realization right away.

TABLE I. TRANSCRIPT LABELS

Focus area labels
Instructions: Talking about or reference the task instructions. Programming Environment: Talking about a part of the programming environment without mention of the task or example code. Example Code: Reading or talking about the example code, specifically referring to objects or parameters used in the example. (Example or Task) Execution: Focusing on executing either the task program or the example code, as differentiated by task logs. Task Code: Reading or talking about the task code, specifically referring to objects or parameters used in the task. Unknown/Other Off-topic: Not talking about the task
Process labels
Description: Reading, paraphrasing or explaining part of the focus area. Description-Realization: Describing something when they make a realization or describing their realization. This is often signaled by an "Oh!"- like statement. Description-Don't Understand: Describing something and making an explicit statement about not understanding how something works. Idea: Talking about an idea for something to complete the task. It may be abstract, concrete, or not even explicitly stated. Ideas can also be negative, such as telling their partner not to do a certain thing.(*This does not include actions like "play the example.") Idea-Realization: Talking about an idea about what to do next in which they seem to suddenly understand what needs to happen. This is often signaled by an "Oh!"-like statement. Idea-Don't know how: Talking about an idea about something to do next to solve the task, but they do not know how to carry out the idea. Evaluation-Working: Declaring that their program is correct. Evaluation-Possibly working: Declaring that that their program might be working. Evaluation- Not working: Declaring that their program does not work. Unknown/Other

V. CORRECTNESS AND TIMING

In this section, we report the overall correctness, task times, and times before and after the realization point to give a high level idea of what the task data looks like.

A. Program Correctness and Task Time

Out of 54 total tasks, participants correctly completed 37 tasks (69%) and failed to complete 17 tasks (31%). On average, it took participants 5.27 minutes (σ = 2.24 min.) to complete a task, including those who spent the whole 8 minutes and did not finish the task. Task times ranged from 1.63 to 8 minutes, as participants had up to up to 8 minutes in total. In one of the 54 tasks, the timer did not stop the participants at the 8 minute mark, but from the logs we can determine what they accomplished within the 8 minutes and only analyzed that period of the task.

B. Realization Point

Across all tasks, pairs spent an average of 1.9 minutes (σ = 1.5 min.) before the realization point, ranging from 0.2 minutes to 7 minutes. For only 2 of the 54 tasks, participants never reached a realization point, so we exclude these task times from the averages for both before and after realization times. After the realization point, pairs spent 3.4 minutes on average (σ = 1.9 min.), with times ranging between 0.3 and 7.5 minutes. Notice that participants spent longer after the realization point than before (3.4 vs. 1.9 min.), which suggests that using the concepts from the example was more challenging than identifying them.

VI. HURDLES AND STRATEGIES

Because behavior before and after the realization point differs, we first describe two hurdles that occur before the realization point, followed by those after the realization point.

Then, we describe three strategies. We call these 'hurdles' because many pairs overcome the challenges on their own.

A. Context distraction hurdle

Often, participants spent time at the beginning of a task exploring the task code and programming environment or generating ideas from those contexts. For instance, in a few tasks, participants wanted to move a UFO to the ground. Even though the instructions told them they could not use numerical values to accomplish this, a few pairs wanted to explore the different numerical values to see how they worked. In another task, a participant wanted to explore a parameter called 'as seen by' after the pair talked about not having any ideas about how to complete the task. In this task, the participants actually needed to insert a function, but first they want to explore what 'as seen by' does: "Wait, can you, wait click 'as seen by,' just out of curiosity, a little more. Just try one of those things: begin gently, begin gently and… do you know?" We can also see this hurdle through the transcription labels, where tasks have multiple *task code- idea* and *programming environment- idea* labels before participants looked at the example.

B. Example comprehension hurdle

In some cases, participants' confusion about the example prevented them from using it or being able to generate ideas based on it. After having the researcher suggest that they use the example during the mid-task interview, one pair had the following conversation: "Play example. I don't get how that's supposed to help us. Yeah, I have no idea." In this case, the participants did not understand how the example was related to their task, so they did not even consider using it to prompt ideas. In other cases, participants did not understand what was happening in the example, such as one participant who describes an example where a ghost moves toward a treasure chest until the two objects overlap. In this quote, the participant was reading part of the example code: "Ghost move toward treasure chest. Huh. That's weird. Hmmm." However, he does not read the next part of the code, which is the critical component. In these cases, the transcripts often have *example code* or *example execution* labels early in the task with a much later realization point.

C. Programming environment hurdle

After participants discovered which programming concept to use, they sometimes could not find it in the programming environment. For example, a pair of participants has this conversation about using the 'repeat loop': "then you do repeat two times. How? But it says that you can repeat. Where is the times thing? I don't see that. Stop. Oh here, jump. We got that. I was just trying to find the…" At that point, the participants have been talking about the repeat loop for two minutes and it is time for the mid-task interview, so they tell the researcher about their problem finding the repeat: "so I was kind of confused because we can't find the. […] We can't find how to do the repeat."

In other cases, participants found what they wanted to use, but could not figure out how to select or move it to accomplish their goal. One such participant had a clear idea of what they wanted to do, but did not know how to accomplish it: "Well we take the collection and put it where the girl was so that it moves them all up at once. Okay, so how are we supposed to do this?" These types of issues are commonly labeled *programming environment: don't know how* and *programming environment: description- don't understand* in the transcriptions.

D. Code misconception hurdle

Sometimes participants had misconceptions about how their programs worked. In these cases, participants thought they knew what to do to complete the task, but that idea was actually incorrect. One participant incorrectly thought that changing the ordering of their code would make two things happen at the same time: "Maybe you put the right shoulder, maybe you switch those around. So put this one right there and that one right there. Why would we do that? Cause then it would go in sync." However, their real problem was that they needed multiple parallel execution blocks. Sometimes, these misconceptions led participants to generate new ideas that helped them to succeed, but misconceptions added to task time, as they required participants to debug the problem. In other cases, code misconception hurdles were followed by code comprehension hurdles, in which participants expected their code to do one thing, but it did another.

E. Code comprehension hurdle

Participants sometimes talked about not understanding how their code worked: "Why is he not on the ground," "Let's see how this works out. Why didn't the rabbit move," "What the heck happened with this jump," and "What did we do? I thought he'd jump again." In these questions, participants had an expectation of what would occur when they executed their code, but that expectation was not met. Responding to these questions lead to other hurdles, like context distraction (A), but also spurred strategies like idea generation (F) and code-example comparison (G). Common labels for these types of problems are *task execution: description-don't understand.*

F. Idea generation strategy

After the realization point, if participants did not have a plan for how to actually use the programming concept to solve the task, many still generated ideas based on the task code. We classify behaviors as part of this strategy if they are not based on the example code nor on a previous failed attempt. One participant asked their partner a slew of questions about what to do next "Do we have to put that up there or what? Do we move them in there or something? For it to work? Do we move this?" These questions refer to multiple different possible next actions, none of which the participant seems to base on any specific rationale. Another participant stated "Huh. I have no idea what you're supposed to do, but I'll try something." While this process can be haphazard, the willingness to keep trying often resulted in success. The *task code: idea* and *execution* labels often accompany this strategy.

G. Code-example comparison strategy

Revisiting the example after the realization point while trying out ideas helped participants to complete the task. For example, a pair of participants were working on a task where they need to get a girl to walk a certain distance and then have a rabbit run away. Solving the task depended on them figuring out to use the expression 'not overlapping', but the *not* operator

had to be added separately. They first get the 'overlapping part' and then return to the example and eventually figure out that they are missing the 'not': "Okay. Now, when I play it, she walks up, but the rabbit doesn't run. Overlapping. Overlapping with… Play. It doesn't do it. That's weird. Not is true. But here it's just is true … That looks like the example. Yeah, but it's got this whole red line around it, but it's got this not thing." After participants have worked with the task code for a little while, they are better able to identify meaningful differences between the example and task programs.

H. Example emphasis strategy

Some participants stated that the red outline helped them find the important part of the example, even though we did not provide any explanation of the outline (see Fig.1). When asked how they decided to use a certain concept, one participant stated, "we just saw the outline." Another participant asked the researcher "where is the repeat? We saw it outlined." We provided visual emphasis because we wanted participants to have a cue to help them move through the task, but we did not want to provide hints as to how the example actually worked.

VII. Task Behavior Groups

Overall, this data contains a variety of task behavior profiles. Fig. 3 shows a graph of the 54 tasks where the x-axis is the time before the realization point and the y-axis is the time after the realization point. We noticed that there are tasks that spent much more time than the average before and after the realization point, as well as tasks that were overall completed much more quickly than most. In this section, we wanted to explore what happened in these extreme cases. To do this, we selected 5 tasks (approximately 10% of the data) that performed best and worst before and after the realization point:

- **Long conclusion**: the 10 tasks where pairs spent the longest time after the realization point (5 correct, 5 incorrect)
- **Slow start**: the 10 tasks where pairs spent the longest time before the realization point (5 correct, 5 incorrect)
- **Quick**: the 5 tasks correctly completed the quickest
- **No realization**: the 2 tasks where participants never reached a realization point

For each of the groups, we describe their behaviors, hurdles, and strategies based on the transcriptions. Fig. 4 shows a set of relevant transcription labels for this discussion and the average count of each label.

A. Long conclusion group

Since participants, on average, spent more of their task time after the realization point, we wanted to know what caused long conclusions, shown in the top grouping of Fig. 3.

1) Correct long conclusion

Tasks in this group were slowed down by the number of ideas participants had, as well as participants' incorrect expectations of the code. Likely, participants successfully completed these tasks because they continued to generate ideas, and because they revisited the example. While participants in other groups spent time talking about not understanding why the task code executed a certain way, participants in this group revisited the example to try to figure out how their code and the example differed. Fig. 4 also shows that this group had the most programming environment ideas, but not many statements where the participants talked about not understanding or not knowing how to find a code block. This likely means that they just needed to try a few ideas before finding what they needed. Behaviors after the realization point included two main hurdles: *code misconception* and *programming environment*, but participants used the *code-example comparison* strategy and the *idea*

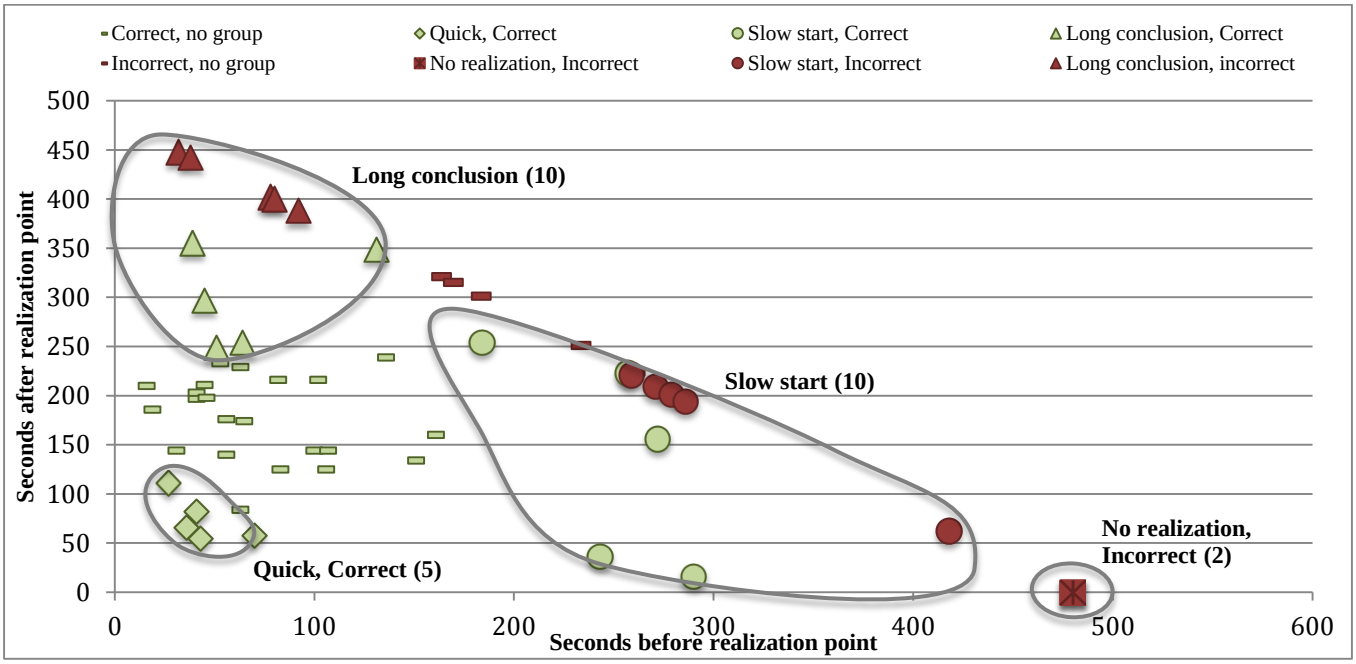

Fig. 3 Time before realization point vs. time after realization point, with correctness and behavior group annotated with color and shape

generation strategy.

2) Incorrect long conclusion

The tasks in the incorrect long conclusion group seem to have been the most slowed down by the programming environment (see Fig. 4). This means that after the realization point, participants spent time trying to find code blocks or struggling with system mechanics. However, participants still used the *idea generation* and *code-example comparison* strategies, during which they thought of ideas from the task code and executed the code to see if the ideas worked. Unfortunately, participants in this group were the most confused about how their code worked, which likely meant that they generated many incorrect ideas. Overall, these tasks had similar hurdles and strategies to tasks completed correctly: *programming environment* and *misconceptions* hurdles and the *idea generation* strategy. These tasks, though, also suffered from the *code comprehension* hurdle.

B. Slow start group

In this section, we discuss both the correct and incorrect tasks during which participants spent the most time before the realization point (the middle group in Fig. 3).

1) Correct slow start

Participants spent a long time before the realization point on these tasks primarily due to the *distraction* hurdle and because they did not always fully understand the task instructions. In these tasks, participants did not appear to look at the example before they created a plan based on the task code or programming environment. Accordingly, the first time participants have an *example code* focus label is not until near the realization point. The study context may have also contributed to the extended time before realization for some correct slow start tasks. In order to control what programs participants worked on for the study, we had to provide participants with tasks and instructions, which not all participants may have been motivated by or understood immediately. Transcriptions for these tasks show that correct slow start tasks had on average one *instruction-description don't understand* label in their transcripts, which was the highest of all of the groups (see Fig. 4).

2) Incorrect slow start

Interestingly, as shown in Fig. 3, the incorrect slow start tasks have similar times before the realization point to the correct tasks. This means that both groups of tasks had similar amounts of time after the realization point to complete the task, so lack of time did not contribute to the incorrect end state. On incorrect slow start tasks, participants had the *context distraction* hurdle, but these tasks seem to have a different pattern than those completed correctly. Task transcriptions in this group contain the *example code* or *example execution* labels near the beginning of the task, but participants do not return to it again until the researcher reminds them during the mid-task interview, possibly caused by example comprehension hurdles.

C. No realization group

Participants working on tasks in the 'no realization group', shown on the bottom right of Fig. 3, do not reach a realization likely because they do not discuss the example code even though they execute the example (see Fig. 4). This likely means that they do not know how it would be useful. Consequently, the *example comprehension* hurdle will be especially important to resolve, as it can prevent participants from even realizing what concept to use. Unexpectedly, however, participants working on these tasks do not use the idea generation strategy, shown by the small number of *task code: idea* labels. Most likely, participants during these tasks were overwhelmed, which is supported by the fact that both of these tasks were first in the series of six for the two pairs of participants.

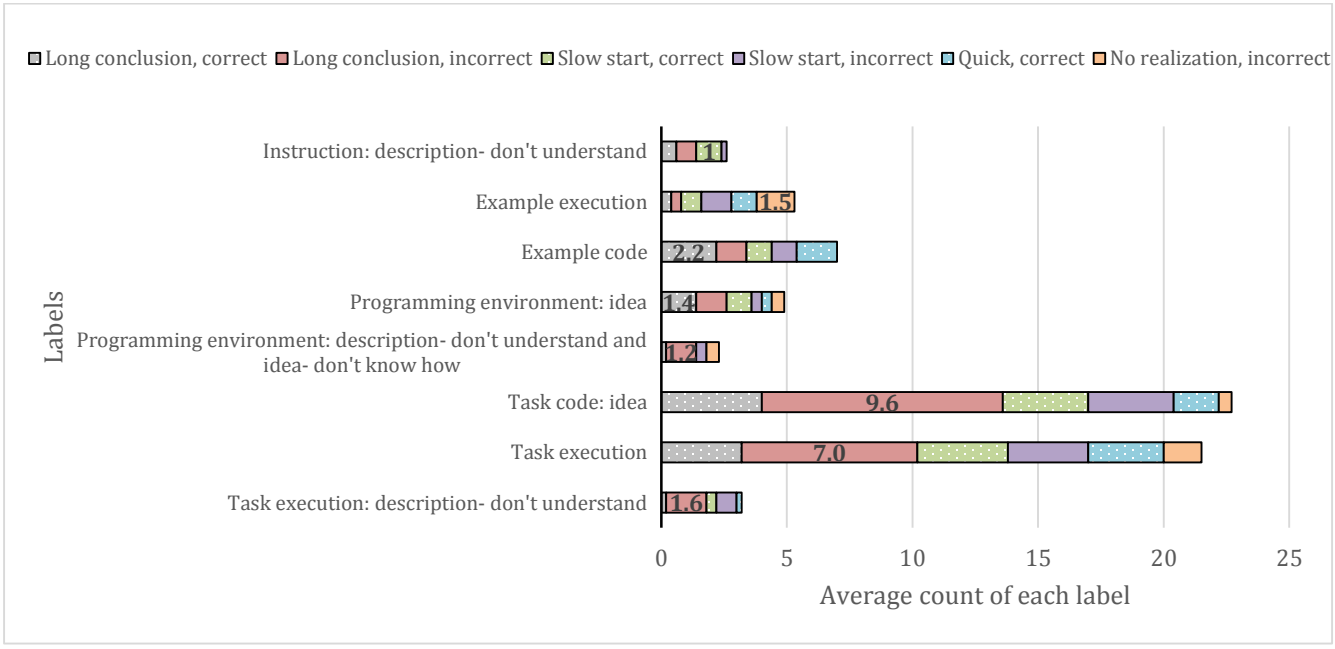

Fig. 4 Important labels and the average count for each of the behavior groups.The largest value is shown for each label.

D. Quick group

Participants who completed a task extremely quickly primarily described the instructions and task and generated ideas from the example early, rather than getting distracted. After the realization point, transcripts from these tasks have zero or one *programming environment* labels, which means that the participants did not have many conversations or questions about where to find code blocks. However, some participants in the quick group did use the idea generation strategy: "at first we tried putting them all in the do together box and then we tried putting two out and then one out and then put another box and put them in it." Since these were often simple tasks, participants could guess about different configurations and still complete the tasks quickly.

Participants in this group also used the outline and code-example comparison strategies. They discover the correct concept to use almost immediately, mainly by finding it in the example. However, participants may have also noticed these concepts in previous tasks. In the quick group, on average, the tasks had 1.6 *example code* labels and 1 *example execution* label. Two of the five tasks in this group contained the code-example comparison strategy when the participants did not necessarily grasp the concept well enough to complete the task directly. In the other three tasks, participants did not need more information to correctly complete the tasks, or quickly generated several ideas, which happened to work.

VIII. Discussion

In this study, we explore how novice programmers use examples to complete programming tasks. Specifically, we look at the case where a novice programmer is highly unfamiliar with their own code, as well as the example. The combination of many new concepts can create an overwhelming experience. Yet, this situation likely encompasses the experiences of many end user and novice programmers when they begin and look to examples as a way to try to accomplish their goals.

A key result of this work is that the time spent before or after the realization point can indicate the types of problems participants likely experienced. In *slow start* tasks, participants focus on the task and programming environment before addressing the example. In the *long conclusion* group, participants notice the key to the example early, but still struggle to solve the task. We believe these groupings can suggest ways to design support for novice programmers using examples.

A. Implications of slow start behavior

When participants had slow starts, it was often because of the example comprehension and context distraction hurdles.

Participants sometimes took a long time to reach the realization point because they were executing the example code more than reading the example code. The majority of the support provided for understanding examples accompanies the example code, but this might indicate that we should consider ways to augment the example execution. For example, this type of support could be more along the lines of a debugger than a textual annotation. Furthermore, some participants did not understand how the example related to their own code, which prevented them from trying harder to understand it.

Since participants were new to both the programming environment and task, spending time becoming familiar with those aspects of the task can be valuable. Thus, we do not always want to force novices past the context distraction hurdle. However, especially in educational contexts, we may want to nudge novice programmers to return to the example once they feel comfortable with the code and environment.

B. Implications of long conclusion behavior

Interestingly, many participants had quite a bit of trouble completing tasks even after the realization point. Our analysis of participant behavior starts to explain why participants still struggled after the realization point: programming environment, code misconception and comprehension hurdles.

The programming environment hurdle is specific to visual programming environments, where programmers may not be able to find a code block. However, this issue it is not necessarily specific to the first 90 minutes of programming. Even if a novice programmer has become familiar with the programming environment, they still might not know where to find a code block that they have not used before. One way to improve examples to help novices would be to augment examples with assistance to find code blocks in the interface.

For the code misconception and code comprehension hurdles, we may be able to help novices by encouraging more revisiting of the example and by helping them to make a plan from the example. While some participants revisited the example while they were working on using the programming concept to complete the task, this was rare, yet helpful. Instead, many participants either used the idea generation strategy or 'debugged into existence', based on their misconceptions and code comprehension hurdles [26]. The participants who tried a few ideas and then returned to the example to see how their code was different seemed to be more effective in generating ideas that succeeded. However, the long conclusion pattern likely occurs because at the realization point, participants are not familiar enough with the task to generate a complete plan to solve the task. This means that just augmenting an example with a lot more information would probably cause novices to be even more overwhelmed when they first look at it. Instead, we would want to encourage participants to return to the example and provide support that they can request when they need it.

While there is more to be learned about how novices use examples, we believe that the results of this study can inform the design of new example support for novices.

Acknowledgments

Many thanks to Julian Ozen and Evan Balzuweit for their help! This material is based upon work supported by the National Science Foundation under Grant No. 1054587.

References

[1] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2009, pp. 1589–1598.

[2] M. B. Rosson, J. Ballin, and H. Nash, “Everyday programming: Challenges and opportunities for informal web development,” in *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, 2004, pp. 123–130.
[3] “Scratch | Home | imagine, program, share.” [Online]. Available: http://scratch.mit.edu/. [Accessed: 19-Sep-2012].
[4] “Looking Glass Community.” [Online]. Available: https://lookingglass.wustl.edu/. [Accessed: 24-Feb-2013].
[5] C. Scaffidi, M. Shaw, and B. Myers, “Estimating the numbers of end users and end user programmers,” in *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*, 2005, pp. 207–214.
[6] X. Zhu and H. A. Simon, “Learning mathematics from examples and by doing,” *Cogn. Instr.*, vol. 4, no. 3, pp. 137–166, 1987.
[7] M. B. Rosson and J. M. Carroll, “The reuse of uses in Smalltalk programming,” *ACM Trans. Comput.-Hum. Interact. TOCHI*, vol. 3, no. 3, pp. 219–253, 1996.
[8] C. Kissinger, M. Burnett, S. Stumpf, N. Subrahmaniyan, L. Beckwith, S. Yang, and M. B. Rosson, “Supporting end-user debugging: what do users want to know?,” in *Proceedings of the working conference on Advanced visual interfaces*, 2006, pp. 135–142.
[9] A. J. Ko, B. A. Myers, and H. H. Aung, “Six learning barriers in end-user programming systems,” in *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, 2004, pp. 199–206.
[10] J. Cao, Y. Riche, S. Wiedenbeck, M. Burnett, and V. Grigoreanu, “End-user mashup programming: through the design lens,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2010, pp. 1009–1018.
[11] P. Gross and C. Kelleher, “Non-programmers identifying functionality in unfamiliar code: strategies and barriers,” *J. Vis. Lang. Comput.*, vol. 21, no. 5, pp. 263–276, 2010.
[12] P. Brusilovsky and G. Weber, “Collaborative example selection in an intelligent example-based programming environment,” in *Proceedings of the 1996 international conference on Learning sciences*, 1996, pp. 357–362.
[13] L. R. Neal, “A system for example-based programming,” in *ACM SIGCHI Bulletin*, 1989, vol. 20, pp. 63–68.
[14] P. Brusilovsky, “WebEx: Learning from Examples in a Programming Course.,” in *WebNet*, 2001, vol. 1, pp. 124–129.
[15] G. Weber and A. Mollenberg, “ELM-PE: A Knowledge-based Programming Environment for Learning LISP.,” 1994.
[16] D. F. Redmiles, “Reducing the variability of programmers’ performance through explained examples,” in *Proceedings of the INTERACT’93 and CHI’93 Conference on Human Factors in Computing Systems*, 1993, pp. 67–73.
[17] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, “Example-centric programming: integrating web search into the development environment,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2010, pp. 513–522.
[18] N. Sawadsky and G. C. Murphy, “Fishtail: from task context to source code examples,” in *Proceedings of the 1st Workshop on Developing Tools as Plug-ins*, 2011, pp. 48–51.
[19] J. Stylos and B. A. Myers, “Mica: A web-search tool for finding api components and examples,” in *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*, 2006, pp. 195–202.
[20] M. Goldman and R. C. Miller, “Codetrail: Connecting source code and web resources,” *J. Vis. Lang. Comput.*, vol. 20, no. 4, pp. 223–235, 2009.
[21] D. Wightman, Z. Ye, J. Brandt, and R. Vertegaal, “Snipmatch: using source code context to enhance snippet retrieval and parameterization,” in *Proceedings of the 25th annual acm symposium on user interface software and technology*, 2012, pp. 219–228.
[22] S. Oney and J. Brandt, “Codelets: linking interactive documentation and example code in the editor,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2012, pp. 2697–2706.
[23] K. S.-P. Chang and B. A. Myers, “WebCrystal: understanding and reusing examples in web authoring,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2012, pp. 3205–3214.
[24] P. A. Gross, M. S. Herstand, J. W. Hodges, and C. L. Kelleher, “A code reuse interface for non-programmer middle school students,” in *Proceedings of the 15th international conference on Intelligent user interfaces*, New York, NY, USA, 2010, pp. 219–228.
[25] J. Cao, I. Kwan, R. White, S. D. Fleming, M. Burnett, and C. Scaffidi, “From barriers to learning in the idea garden: An empirical study,” in *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*, 2012, pp. 59–66.
[26] M. B. Rosson and J. M. Carroll, “Active Programming Strategies in Reuse,” in *Proceedings of the 7th European Conference on Object-Oriented Programming*, 1993, pp. 4–20.