# Towards Understanding Successful Novice Example Use in Blocks-Based Programming

Michelle Ichinco, Kyle J. Harms, Caitlin Kelleher
Dept. of Computer Science and Engineering
Washington University in St. Louis
St. Louis, MO, USA
{michelle.ichinco, harmsk, ckelleher}@wustl.edu

**Abstract** *Blocks-based environments are frequently used in settings where there is little or no access to teachers. Effective support for examples in blocks-based environments may help novices learn in the absence of human experts. However, existing research suggests that novices can struggle to use examples effectively. We conducted a study exploring the impacts of example-task similarity and annotation style on children's abilities to use examples in a blocks-based environment. To gain understanding of where and why children struggle, we examined the degree to which (1) children were able to map a task to its corresponding example, and (2) their programming behavior predicted task success. The results suggest that annotations improve task performance to an extent and that mappings and programming behavior can begin to explain the remaining problems novices have using examples.*

## 1. Introduction

Blocks-based programming environments, such as Scratch [1], App Inventor [2], and Looking Glass [3] have been growing in popularity. In this paper, we start to explore how novices use examples in a blocks-based programming environment. Understanding example use in blocks-based environments is important because effective example use can: (1) help to provide learning support for novices who lack access to formal classes and qualified teachers, and (2) be a useful skill as programmers transition to more independent projects.

Blocks-based environments are often used by children with very little programming experience. Due to a lack of qualified computer science teachers [4], children commonly use blocks-based environments for short classroom projects, extracurricular activities, or on their own. As a result, many blocks-based programmers rely on the programming environments and related online materials to provide learning opportunities. Current systems support learning through tutorials [5], games [6, 7], and intelligent tutoring [8]. However, in order to work toward a programming goal, novices likely need more context-specific information like they might find in example code that is similar to their goal.

Studies have found that programmers of all experience levels often use example code found online to accomplish particular tasks [9–13]. The ability to use examples effectively is an important skill for programmers to have, as it enables the programmers to continue to gain new skills as technologies change. Unfortunately, inexperienced programmers often struggle when attempting to reuse others' code [14]. Due to their difficulties understanding example code behavior, some novices report referencing example code as a model rather than adapting it and integrating it into their programs [12, 13]. Yet, research on programming and examples has primarily focused on selecting examples [15] or adapting them [16], rather than the process of example use.

In this work, we ran a study exploring how novice programmers use examples. Our first goal was to compare annotation styles and example-task similarities to understand how these commonly used styles of examples affect performance. We measured performance through task success and analogical mappings. The idea of analogical mappings comes from problem solving, where an analogical mapping refers to a relationship between corresponding parts of two problems. Often, that relationship can help a learner to solve a problem by figuring out how the solution of one problem relates to another problem they are trying to solve [17, 18]. Likely, using an example to solve a programming problem requires a mapping between the two snippets of code. If analogical mappings between example code and tasks indicate a likelihood to succeed, systems could use that information to determine when and how to provide support. In order to not affect the problem solving process, we collected analogical mappings after users completed tasks, which may not be able to indicate whether analogical mappings can predict success during a task. Thus, we hypothesized that novices' mappings between tasks and examples would correlate with task performance, which can begin to indicate how analogical mappings in programming relate to task success.

The example styles only had a small impact on both task performance and analogical mappings, so we performed a post-hoc analysis that looked at performance in terms of four "stages" of programming task completion. In blocks-based programming, programmers often need to (1) manipulate the GUI, and then (2) locate, (3) insert, and (4) correctly apply blocks in order to solve programming tasks. We used these

DOI reference number: 10.18293/VLSS2017-009

four stages to investigate two ways of understanding when novices are having trouble using examples: correctness of analogical mappings and programming behaviors. We hypothesized that participants' programming behaviors (i.e. manipulating the interface, editing the code, or executing the program) will have some predictive power, which we explore using decision trees.

To investigate these hypotheses, we ran and analyzed a study looking at novice programmers using examples in a blocks-based programming environment. Researchers can use the results of this study to inform the design of ways to help novice programmers use examples in blocks-based programming environments and educational systems for computer science using examples. The goal of this study is to better understand novices' example use in programming by answering 3 questions:

1. How does the interaction of annotations and example similarity affect novice programmers' performance on tasks using examples?
2. To what degree does the ability to map an example and target problem correlate with task success?
3. To what degree do programming behaviors predict task success?

## 2. Related Work

We first discuss related work surrounding examples in programming and in education. We then discuss how this work relates to other ways for novice programmers to learn independently in blocks-based programming environments.

### 2.1. Support for Programming with Examples

There is a broad range of work on programming with examples. However, most of it focuses on support for selecting appropriate examples or adapting an example for a new context, but does not look at the problems novices have in trying to use examples.

Systems focused on assisting in example selection either help programmers to find better examples or provide them directly. Specialized search tools for programming enable users to quickly perform keyword searches over API documentation [19], open-source projects [20–22], and code snippets presented on web pages [15, 21, 23]. Since keyword searches can return a large number of irrelevant results, some systems support specifying more constrained searches [24–28] or provide suggestions based on the programmer's current context [29]. Some programming environments provide access to examples by including a small set of pre-created examples [30–37], integrating example search directly into the environment [19, 20], enabling access to programs created by other users [1, 38, 39], or using direct manipulation to generate example code [29, 30, 33]. All of these systems aim to either make it easier to select an analogous example for a certain problem or attempt to suggest a useful example. Only one system that we know of, the Idea Garden, actually frames example use as analogical reasoning [40]. In the Idea Garden, analogy is introduced as a strategy for overcoming barriers, but the analogy was used mainly for selecting an example to use.

Programming systems currently support using examples, as well as integrating example code. Systems have added annotations as a way of supporting example use. Generally, these annotations either provide general descriptions of the code as a whole [15, 40] or provide specific information about certain parts of the code [16, 30, 41]. Other systems support programmers in integrating example code into programs, essentially removing the need for programmers to create mappings in order to use an example to solve their problem. For instance, Codelets provides a widget to allow easy modification of example code [42], while WebCrystal allows users to select which combination of features to integrate from an example [16].

Work on programming with examples is mainly in textual programming languages and focuses on selecting an example or supporting example integration, rather than understanding the issues novice programmers in blocks-based environments have while using examples.

### 2.2. Learning From Examples

The idea of supporting example use in order to improve independent learning opportunities is supported by two areas of research: (1) educational systems that introduce examples, and (2) theories of learning from examples. This study used examples more like programmers would find on the web because we wanted to simulate the experience of a novice programmer learning programming outside of a classroom, but the research on examples in education has inspired design choices in this study.

A number of educational systems provide support for example use [30, 43–45]. There are also systems [46, 47] that integrate examples into tutorial systems for programmers based on Caroll's theory on minimalist documentation [48, 49]. Similarly, other systems also provide sets of annotated examples to support learners, called 'case libraries' [50]. Case libraries provide sets of examples that relate to a problem learners are trying to solve [51]. This idea is supported by case-based reasoning theory, which focuses on having students learn through experiences and reflection [52]. The work on case libraries for case-based reasoning is limited and does not address programming examples. Furthermore, all of the examples in these systems are designed to fit within educational systems, rather than considering how novices can use examples to learn independently. They also do not address different types of example styles and how they affect novice programmer use. One study [53] looked at novice programmer example use, but only for one type of annotation style and mainly focused on the participants' descriptions of their difficulties, rather than on data that could possibly predict success or failure. In the discussion, we explore the relationship between the results in the two studies.

Research on learning from examples, such as worked examples and cognitive load theory, are important to consider

in thinking about novice programmer example use. Worked examples are a popular and well studied learning method found to be more effective than problem solving in some cases [54]. Worked examples are grounded in cognitive load theory, which is the mental effort for a novice to learn something new. Cognitive load theory can be reduced by improving instructional material, such as worked examples, to focus a learner's attention on the steps needed to solve a problem [55]. Worked examples have been used to teach a variety of topics, including mathematics [54] as well as programming [44, 56]. One recent study uses programming worked examples to study the effect of labels on learning [57]. Worked example research has also investigated differing degrees of example similarity, finding that both similar and dissimilar examples can be useful for learning [58, 59]. The research on worked examples supports the idea that the use of examples in learning can be highly effective, but worked examples have been primarily studied in classroom contexts. More work needs to be done to determine how the ideas from cognitive load theory and worked examples can apply to the types of examples programmers would find on the web, when programmers are more focused on completing a task than working through educational material.

### 2.3. Independent Learning for Novice Programmers

There are a variety of blocks-based programming environments that are used outside of classrooms and often provide some support for learning without a structured class, such as games, tutorials, and reuse.

Many blocks-based environments provide tutorials on their websites, such as Scratch [60] and App Inventor [61], which both have web pages providing video tutorials to get users started. However, video tutorials can be hard to use because it is often difficult to step forward or backward [62]. Furthermore, one study found that tutorials were not as effective as puzzles for novices for learning [63]. Some blocks-based environments are games-based, such as Blockly Games [64] and Code.org [6]. Games have been shown to be effective learning mechanisms for novice programmers [65], but often do not allow users to create their own project. Furthermore, learning through games does not apply to learning more advanced programming, where example use is critical.

Many blocks-based environments now provide the ability to share and reuse others' code, similar to the way experienced programmers sometimes use example code. Scratch [60] and Looking Glass [3] provide explicit "remixing" features. App Inventor [2], Kodu [66] and Touch Develop [67] also allow users to use other programmers' projects. However, research has found that novice programmers often have trouble selecting the code they need, which likely makes adapting others' code difficult for novices [14]. Research has been able to cluster behaviorally similar visual code for Scratch [68], which could help novices to find appropriate code more easily in blocks-based environments, but it is still unclear how to help them use example code more effectively.

Overall, prior work on examples in programming has focused on more experienced programmers in text languages. The research on examples in education supports the idea that examples can be an effective way for novices to learn new concepts. Novice programming environments provide a variety of learning supports, but none addresses understanding novices' behavior when using examples as models for learning.

## 3. Study

The goal of this study was to better understand novices' example use by exploring (1) how example similarity and annotations impact example use, (2) whether successful analogical mappings correlate with task success, and (3) whether programming behavior can be used to predict success.

For the study, we asked participants to complete twelve experimental programming and mapping tasks, equally divided between similar and different examples. Participants were randomly assigned to use examples with one of our three annotation styles or no annotations throughout all tasks. Participants completed the tasks in Looking Glass, a blocks-based programming environment for creating 3D animations, designed for middle school aged children (see Figure 4). In this section, we discuss the reasons for our study design and the details of how we ran the study.

### 3.1. Study Design Rationale

In this section, we explain how we chose our experimental materials (example similarity and annotations) and why we decided to look at how analogical mappings and behaviors relate to task success or failure.

#### 3.1.1. Example Task Design

We decided to vary the similarity between examples and task code to simulate the natural variety that would occur in found examples online. However, analogical reasoning research suggests that learners are more successful at completing problem solving tasks when they have an example that is similar to the target problem [17]. This work demonstrates the utility of two kinds of similarity: surface similarity refers to the correspondence between superficial features of the problem and example; structural similarity refers to the correspondence between the operations necessary to solve the problem and example [17]. To explore the impact of similarity, we created two kinds of tasks: similar and dissimilar example tasks.

The similar examples and programs have both structural and surface similarity because they share a similar code structure. For instance, in the similar task and example in Figure 1, the example and the solution use the exact same structure, a *Do together* block with two nested statements. For the dissimilar example tasks, the structures in the examples differ greatly from the solutions. As shown in Figure 1, the example has a *Do together* block with three statements, however the solution requires two *Do together* blocks each with two statements. Additionally, the dissimilar task and example differ in the types and number of objects used in the *Do together* blocks. We hypothesized that participants would be more successful at completing tasks with similar examples than tasks with dissimilar examples.

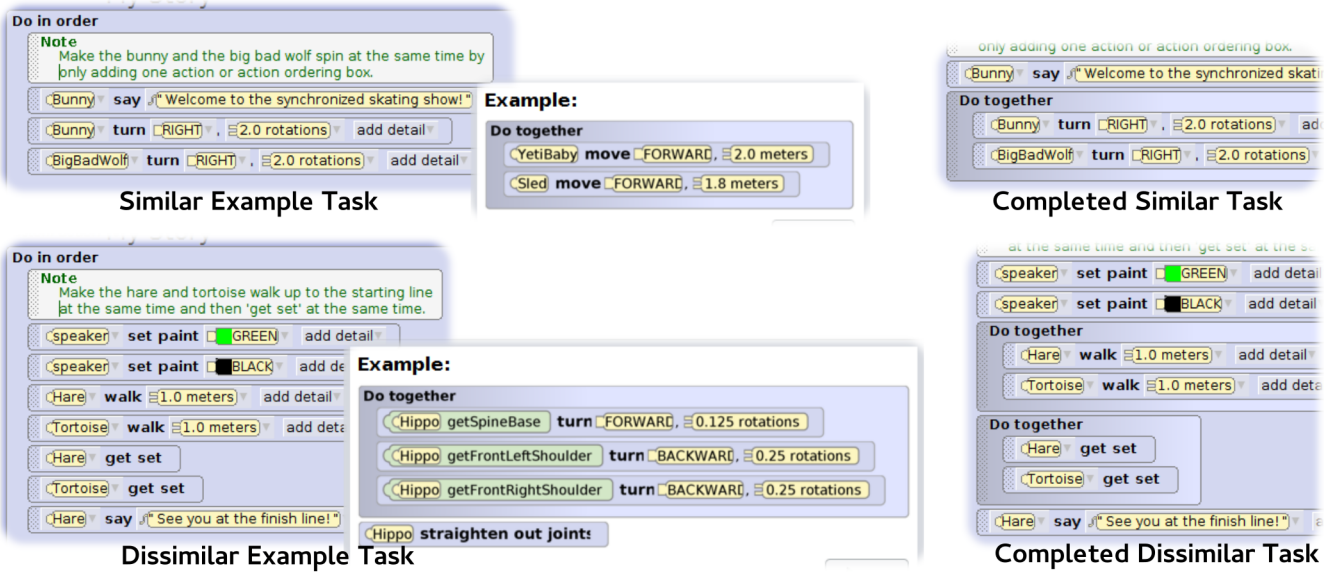


Figure 1: The similar and dissimilar example tasks for the simple parallel execution (*Do together*) task.

### 3.1.2. Example Annotations

We decided to compare example annotations because they are a common feature of found examples online, as well as in examples provided by support systems. Figure 2 shows the three annotation styles we selected and an example with no annotation (the control condition).

Below we describe the four annotation styles used in this study:

**Brief Summary:** provides a high-level description of code behavior, but it does not link explanations to individual lines of code. We selected this type of annotation because it is used in other systems and example work [15, 40]. We believe brief summaries could be useful in solving programming tasks because they can help a user to understand the overall behavior of the code, which could help with forming mappings and solving tasks.

**Line-Specific Notes:** provides descriptions of each salient line, shown near the associated code. We also selected this style as a comparison because it is commonly used for programming examples [16, 30, 41, 57]. Furthermore, line-specific notes could help a user who is confused about how a certain construct works to view information specifically about that part of the code.

**Visual Emphasis:** provides a red highlighted outline around the critical element or elements of the example. We designed this annotation based on formative testing, in which we found that simply circling the important part of an example in red helped novices to solve programming tasks. This could be because programmers can test the correctness of their code, so this annotation could provide them with enough information to identify the code elements involved in the solution so that they could then use that to try out possible solutions. A benefit of this annotation style is that it is quick to create and is not based on writing style, reading comprehension, or language.

**No annotation (Control):** the code example is shown without any textual information or highlighting.

### 3.1.3. Example-Task Analogical Mapping

Based on the idea that analogical problem solving is similar to completing a programming task using an example, we hypothesized that having novice programmers complete an analogical mapping task might provide insight into whether they correctly completed the task.

Cognitive psychologists define analogical problem solving as using one provided problem and solution (the base) to solve another problem (the target) [17, 18]. We believe that novice programming with examples is most closely related to analogical reasoning in mathematics [69]. To illustrate the model of analogical problem solving using mathematics, imagine a student solving problem 3x+2 = 11 using an example 2x-4 = 6, as shown in Figure 3. The student must first map the parts of the task and example that are related. In this case, the 3x and 2x are related, the + 2 and - 4 are related, and the =11 and =5 are related. The example solution might begin with moving 4 to the opposite side of the equation. Using the mapping, the learner would similarly subtract the 2 from both sides of the target equation. The mappings allow learners to adapt the sequence of steps necessary to solve the problem to the context of the target problem. In this case, the base and target problems have high surface similarity, meaning that the words are similar, making it easy to map the two [70].

Prior work is divided on whether the ability to generate correct mappings can predict task success. Gentner [18] describes the structure-mapping theory, which argues that there is a set of relations in the base problem that is also true for the target. The analogy is a mapping between the set of relations for the base and target problems. Gentner's work suggests that the primary difficulty associated with solving a problem using an analogy comes from mapping the example and the target. If participants

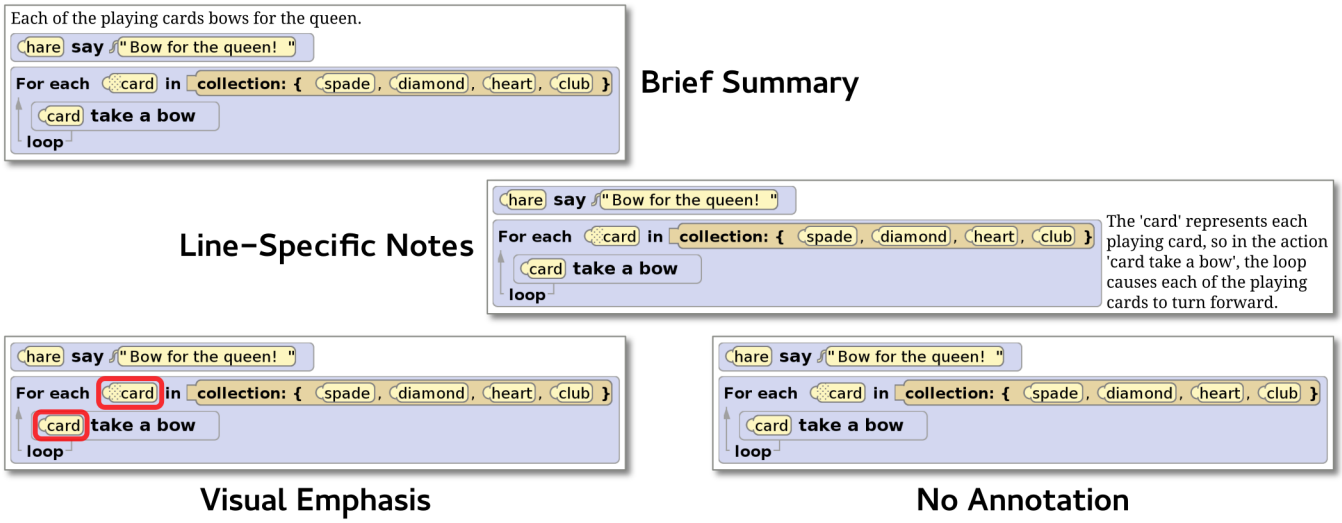


Figure 2: An example shown with three different styles of annotation and with no annotation.

can correctly map the example and target, they are highly likely to correctly solve the problem. In contrast, Novick and Holyoak's research [71] in the context of mathematical problem solving suggests that while mapping the example and target problems is necessary, it may not be sufficient to enable a learner to solve a problem using an analogy. In particular, when learners need to adapt the example to fit their target problem, some learners may succeed at mapping but struggle to construct a full solution [71].

Programming using an example shares some similarities with analogical reasoning in mathematics: novice programmers attempt to use a completed solution, in the form of an example to develop a related solution. However, there are also two important differences between analogies in mathematics and programming. First, novice programmers can incorporate testing into their problem solving process. Second, programming examples are not analogies in the traditional sense. Generally, an analogy provides both an analogous problem and the sequence of steps necessary to arrive at the solution. However, example code is essentially the completed solution. Because of these differences, it is important to evaluate how analogical mappings in programming relate to task success.

In order to understand whether correct mappings correlate

**Problem**

Solve for $x$? $3x + 2 = 11$

**Example**

Solve for $x$? $2x - 4 = 6$

$2x = 10$

$x = 5$

Figure 3: An example of analogical problem solving.

with success for novice programmers, we needed to collect mapping and task performance. Mapping can be operationalized as the ability to define relationships between elements in the example and target [17]. If a mapping is achieved, it occurs at some point during the task, possibly before the task is completed, making it difficult to collect mapping information without interrupting the task. We chose to collect the data after the task to prevent the data collection from changing the problem solving strategy.

#### 3.1.4. Example-Based Problem Solving Process

While previous work suggests that inexperienced programmers often struggle to make effective use of example code, relatively little is known about how novices attempt to solve problems using example code and what kinds of behaviors predict success or failure. Characterizing predictive behaviors may help to identify opportunities for future systems to better support example use. For instance, knowing whether difficulty finding specific code blocks is linked to success or failure can indicate how big of a hurdle the programming environment is in solving a task. This can be accomplished using log data from the programming tasks and using decision trees to understand which features predict success and failure. Decision trees are often used for prediction across many domains, and have been successfully used in human-computer interaction, such as in predicting interruptability [72]. We selected this analysis method after the study was complete to answer questions about programming behavior generated by the example-task mapping analysis.

### 3.2. Study Methods

We gave participants 90 minutes to complete a computing history survey, a training task, and 12 programming tasks with examples. The computing history survey asked participants about their experiences using computers and with programming in the past to confirm that they were eligible for the

study. Each participant was assigned to one of the four annotation conditions. For this evaluation, we used the novice programming environment, Looking Glass [3]. Looking Glass has similar complexity to other blocks-based programming environments in many respects, like having blocks organized in palettes.

#### 3.2.1. Participants

We recruited 99 participants between the ages of 10 and 15 for our study through the Academy of Science of St. Louis mailing list. The Academy of Science of St. Louis is an organization that provides opportunities for members to participate in science and technology programs city-wide. Community members also forwarded our email to a newspaper and a homeschool message board on their own. Since this work aims to address the problems of novice programmers who do not have access to formal computer science education in schools, we asked that participants have "minimal" programming experience, which we defined as 3 or fewer hours. This limit on the number of hours participants had coding also ensures that the participants did not have prior experience using the programming concepts in the tasks.

We analyzed the data for 80 participants (33 female, 47 male, age: M = 11.8, SD = 1.3). Each participant received a $10 gift card for Amazon.com in recognition of his or her participation. We excluded 19 participants: 13 had more than minimal programming experience (i.e. had programmed for more than three hours); 4 did not complete the study within the allotted time; and we had 2 study administration mistakes.

#### 3.2.2. Training Task

We asked participants to complete a training task designed to help familiarize them with the study format and the basic mechanics of the Looking Glass programming environment. The training task had two parts: (1) participants completed a simple program using an example, (2) participants mapped the example and training program on paper.

To complete the training program, participants assembled a simple three-line program in the correct order using an onscreen example. While completing the training program, participants could reference a mechanics help sheet that provided an overview of creating a simple program within the programming environment. This task was designed to introduce participants to the format of the programming tasks and basic interface mechanics. Through pilot tests, we found that this introduction helped to reduce the number of interface and task-related problems during the experimental tasks. Participants were free to reference the mechanics help sheet throughout all study tasks. There was no time limit for this task, but if participants seemed stuck, a researcher helped them to complete the task. Participants were also allowed to ask questions during this task.

After completing the training program, participants also completed a paper-based analogical mapping task. This mapping task asked participants to draw lines connecting elements in the example code with the introductory program (see Figure 5). As with the program training task, the mapping training task familiarized participants with the task instructions.

#### 3.2.3. Programming Tasks with Examples

We next asked participants to complete twelve programming tasks using examples. The tasks covered six programming concepts, listed here from easy to difficult: simple parallel execution, using a *for loop*, using an iterator within a *for each loop*, using an advanced API method unique to Looking Glass, setting a conditional for a *while loop*, and using a function's return value as an argument to a method call. For each programming concept, we developed similar and dissimilar example tasks. See Figure 1 for the similar and dissimilar example programming tasks for the simple parallel execution concept.

We chose programming concepts that greatly varied in difficulty to help provide insight into how concept difficulty affects novices' abilities to complete the tasks. We designed the tasks with the understanding that many would be challenging, especially for novice programmers with minimal programming experience, leading to an expected low overall task performance. This was purposeful because we wanted to explore both the times when novices succeeded using examples as well as when they had difficulties. As prior research suggests that novices are often unsuccessful in using examples, we included a significant proportion of tasks that our pilot tests indicated would have a low success rate in order to capture the challenges novices face when using examples.

As in the training task, each programming task consisted of: (1) modifying a program using an on-screen example (see Figure 4), and (2) completing a paper-based analogical mapping task (see Figure 5).

To complete each programming task, participants needed to modify an existing program to achieve a stated goal while meeting task constraints intended to ensure use of the targeted programming concept. For example, in looping tasks, we required that participants only add a certain number of code blocks in order to force them to use a loop in order to correctly complete the task. Figure 4 shows one programming task used in the study. Participants had at most five minutes to complete each programming task. If participants stated that they finished the task early, a researcher asked them if they were sure that they fulfilled all of the criteria of the task, but did not tell them if it was correct or incorrect. We did this to encourage participants to check their own work and try to make sure they fulfilled the criteria on their own, as pilot studies showed that participants sometimes did not always read directions carefully. We created the tasks and selected time limits based on formative and pilot testing.

To complete the mapping task, participants drew lines connecting code elements in the example to related elements in the program as shown in Figure 5. There was no time limit on the mapping task, but participants generally completed mapping tasks very quickly. Participants completed paper mappings after each programming task to prevent the mapping task from

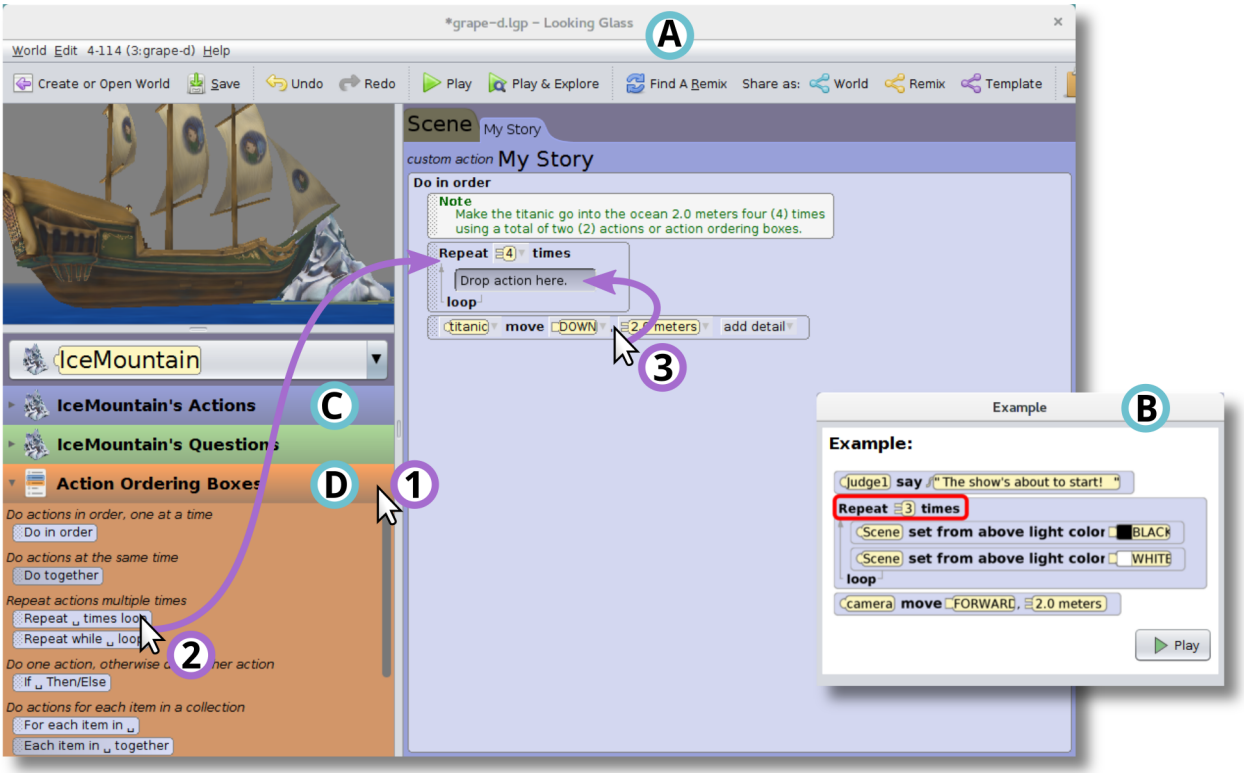


Figure 4: The dissimilar repeat task shown in the (A) Looking Glass programming environment with a (B) dissimilar example. Initially the (C) actions tab is selected. There are three steps to complete the task, shown with pointers in the figure. Step (1): the user clicks on the (D) constructs tab to transition from the *Exploring & Searching Stage* to the *Ready-to-Program Stage*. Step (2): the user drags the repeat construct into the code editor to transition from the *Ready-to-Program* to *Assembling Stage*. Step (3): the user drags the move statement into the repeat construct to correctly complete the task.

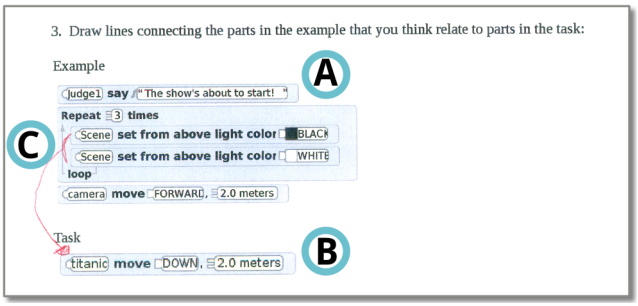


Figure 5: The dissimilar repeat mapping task completed by a study participant. We asked participants to connect the (A) example shown in the task with (B) the initial state of the program by (C) drawing lines connecting the two.

influencing their problem solving process. We acknowledge, however, that collecting mappings at the end of the task could overestimate how many participants had correct mappings during the tasks. Furthermore, it is possible that not having a time limit on mapping tasks may have allowed participants to figure out the mapping at the end of the task.

To account for learning effects, we used a Latin squares design to assign task orderings across participants. We varied the order of the six programming concepts using a balanced 6 x 6 Latin square. For each programming concept, a participant first completed a similar or dissimilar example task for that concept, immediately followed by the other example similarity type. For the six concepts, each participant completed three with a similar example task followed by a dissimilar example task and three with a dissimilar example task followed by a similar example task. Each participant saw one of the four example annotation types for all 12 tasks.

## 4. Data and Analysis

We analyzed three types of data: program performance, example-target mappings, and programming behavior.

### 4.1. Program Performance

We saved participants' task programs when they finished the task, which was when they either stated that they were finished or timed out after five minutes. We graded these final state programs for correctness using criteria based on previous work scoring similar programming tasks [17]. We assigned points for (1) correct usage of the target programming construct, (2) correct placement of the provided code statements relative to the target programming concept, and (3) a lack of extraneous changes. We created rubrics in which participants earned one point for each correct attribute in these three categories. One researcher then scored each participant's programs using the rubric.

| User Interface Behaviors |
|---|
| *Number of UI actions:* User interface actions such as exploring a drop down menu, changing a tab stage, etc.<br>*Time performing UI actions:* Continuous blocks of user interface actions spaced closer than one second apart.<br>*Number of example actions:* The number of times participants interacted with the example (i.e. by mousing over or playing it). It is important to note that this is, at best, an imperfect indicator of example engagement. While some users moused over or moved the example while studying it, we also observed participants who were clearly studying the example without use of the mouse.<br>*Time interacting with example:* Continuous blocks of interacting with the example using the mouse, spaced closer than one second apart.<br>*Idle time:* The amount of time the participant spent not doing anything. We computed this by looking for periods where the time between events was greater than fifteen seconds.<br>*Number of scene edits:* The number of times the participant changes the scene layout (e.g. moving the camera or changing the position of a 3D model). None of the tasks required that participants edit the scene, so this is always off-task behavior. |
| Code Editing Behaviors |
| *Number of irrelevant edits :* An irrelevant code edit does not make any progress towards a correct solution and does not touch any code elements used in achieving a correct solution.<br>*Number of semi-relevant edits:* A semi-relevant edit modifies an element of the code that is involved in a correct solution, but does so in a way that does not make progress towards a correct solution. For example, in the case where a participant needs to replace a numeric parameter with a call to a function, a semi-relevant edit might change that numeric parameter to a different numeric value.<br>*Number of relevant edits:* The number of relevant edits and the time at which the participant first made a relevant edit. We define a relevant edit as one that makes progress towards a correct solution.<br>*Number of edits:* This included all code edits, regardless of type.<br>*Number of tinkering edits:* We noticed that when participants gave up on solving tasks, they appeared to frequently transition to changing unrelated parameter values or experimenting with keyed parameters (e.g. animation styles, duration, etc). This often takes the form of "what does this do?" style experimentation. We refer to these types of changes as tinkering edits.<br>*Number of executions:* The number of times the participant executes their program. |

Table 1: Programming behavior features

## 4.2. Example-Target Mappings

To analyze the mapping tasks, two authors independently transcribed which components the lines were drawn between for 14% of participants, reaching high agreement (Cohen's $\kappa > .61$) for the mappings ($\kappa = .794$, $p < .001$). The authors then worked independently to transcribe the remaining participants' mappings. This was necessary because in a small number of cases, it was slightly ambiguous which blocks the mappings were connecting. We developed a set of correct mappings consistent with the program solution and recorded whether the transcribed mappings contained one or more of these correct mappings, similar to the analysis from Spellman et al. [73].

## 4.3. Programming Behavior

In addition to mappings and program performance, we also recorded log files of the actions that participants took while working on each task. This includes data such as interface interactions like opening a palette of blocks, time spent with their mouse interacting with example, and program modifications, as shown in Table 1. We use this log information to compute program solution progress, which we will discuss throughout our results section to provide more granularity than just success or failure. Our log parser analyzed participants' code editing behavior when working towards the solution of a task. Because each task has a known solution and a known interface state to reach that solution, we were able to ascertain whether their actions were relevant to the solution or were in no way related to the task. We also investigate whether this log data can predict success or failure on tasks using decision trees, which we will go into greater detail in the results section.

### 4.3.1. Program Solution Progress

The majority of our tasks required that participants work through a series of stages in order to arrive at a solution, which we will discuss throughout the results section. These stages are four key parts of a task in a blocks-based environment that we hypothesized could provide us with more in-depth information about where novices struggled:

**Exploring & Searching Stage:** The user has made no progress towards a solution. Typically, the user is exploring and searching the interface in an effort to advance their task. This stage may include both code changes and user interface actions that do not advance the user towards a solution. This is an essential part of many new programmers' experiences in blocks-based environments, where rather than being able to type a command, they must explore the environment in order to find the blocks they need to program.

**Ready-to-Program Stage:** The user interface is in a state where the program elements necessary to solve the problem are accessible. For example, if the user needs to add a loop to solve a problem, the palette containing the loop code block is visible in the interface, as shown in Figure 4-(1). For a blocks-based environment, this demonstrates progress toward solving a task because it likely indicates that the programmer has correctly determined which block they need and have also found it in the interface.

**Assembling Stage:** Any needed program elements have been added to the program, but the code is incorrect. Continuing the loop example, the loop is in the user's program but is incorrectly placed or does not contain all of the required statements, as shown in Figure 4-(2). At this point, in a blocks-based environment, all the programmer must do is re-arrange the blocks to create the correct solution.

**Completed Stage:** The program is completed and correct, as shown in Figure 4 after the action in (3).

For each task, we record the time at which a user first reaches each stage. We note that it is possible that a user can regress to a previous stage. This commonly happens with the *Ready-to-Program* stage because the user may change the state of the interface away from being ready to solve the problem. Our stage-based model was a post-hoc analysis designed to more deeply understand the behaviors affecting task success. For our stage-based analysis we excluded two programming concepts: using an iterator within a *for each loop* and calling an API method (four of the twelve tasks in total). These four tasks initially begin in the *Ready-to-Program Stage*, instead of the first stage, *Exploring & Searching* because the participants did not need to change the state of the interface to find the component needed to complete the task. We believe that analyzing the remaining 8 tasks for this part makes sense because in order to best understand exactly where the problems are taking place, we want to look at tasks where all four stages are required for successful task completion. All results unrelated to our stage-based analysis are based on the data from all twelve tasks.

## 5. Results

We seek to answer three questions: (1) how does the interaction of annotations and example similarity affect novice programmers' performance on tasks using examples, (2) to what degree does the ability to map an example and target problem correlate with task success, and (3) to what degree do programming environment and coding behaviors predict task success?

### 5.1. How do novice programmers perform on tasks using examples and how do annotations and example similarity affect performance?

To answer this, we discuss the overall task success, the effects of annotations on task success, and the effects of example-task similarity on task success.

#### 5.1.1. Overall Task Performance

Overall, participants completed 30.3% of tasks correctly. This result generally aligns with our expectation and the idea that novices often struggle completing tasks using examples. We will discuss this result further later in the paper. When completing most tasks, most participants did not move past the *Exploring & Searching Stage* (51.6%). Conversely, most participants failed to reach the correct solution (only 30.3% succeeded). Only 9.8% of tasks ended at the *Ready-To-Program Stage* and 8.2% of tasks ended at the *Assembling Stage* (as seen in Table 2). We also investigated whether gender may play a role in task performance by including it as a covariate in our analysis based on past research on gender in end-user programming [74]. However, we found no significant gender differences in any of the statistical tests we ran.

| Stage | % Tasks | % Correct Mappings | % Tasks | % Correct Mappings |
|---|---|---|---|---|
| | Ended at Stage | | Reached Stage | |
| Exploring & Searching | 51.6% | 45.6% | 100% | 55.2% |
| Ready-to-Program | 9.8% | 50.0% | 48.4% | 65.4% |
| Assembling | 8.2% | 56.8% | 38.6% | 69.3% |
| Completed | 30.3% | 72.7% | 30.3% | 72.7% |

Table 2: This table shows the percentages of tasks and mappings that ended at each stage and reached each stage. The percentage that ended at each stage shows how many tasks were at a stage when the tasks were over. The percentage that reached each stage demonstrates the amount of tasks that that got to each of the stages.

#### 5.1.2. Effects of Annotations

Using MANCOVA and Roy's largest root, there is a significant effect of annotations versus no annotation on program performance, $\Theta$ = .35, F(12, 66) = 1.93, $p < .05$. The three annotation conditions outperformed the no annotation condition. Separate univariate ANCOVAs revealed that there are no significant differences between the three annotation styles.

This effect was also present for the similar example tasks; participants who used similar examples with any annotation style significantly out-performed participants without any annotations, $\Theta$ = .21, F(6, 72) = 2.52, $p < .05$. However, we found no significant effect for dissimilar examples and annotation styles. We would have expected annotations to assist novice programmers in solving dissimilar example problems, since the annotations help novice programmers to map examples and problems. For dissimilar example tasks, participants may have needed more time to understand the mapping between the task and the example, which prevented them from having time to actually complete the task, though they may have realized how to complete it by the end of the task time.

We also wanted to know whether the annotations had any effect on which stage participants made it to for their tasks. We computed the percent of the tasks that finished in each stage, for each participant. Because the *Completed Stage* is functionally equivalent to program correctness results presented above, we report whether the annotations had any effect in the earlier stages. Using a MANCOVA and Roy's largest root, there is a significant effect of annotation versus no annotation on the stage participants reached, $\Theta$ = .10, F(3, 81) = 2.74, $p < .05$. Compared to the no annotation condition, participants with annotations were slightly more likely to finish their tasks at a higher stage than the no annotation condition. Separate univariate ANCOVAs revealed that there are no significant differences between the three annotation styles.

#### 5.1.3. Effects of Example-Task Similarity

As predicted, participants correctly completed more tasks using similar examples (Mdn = 2.2 of 6 points) than dissimilar examples (Mdn = 1.83 of 6 points), $p < .01$, $r$ = -.34. The low median task scores align with the low overall success rate. Using a MANCOVA and Roy's largest root, there is a significant effect of example-task similarity on the stage,

$\Theta$ = .06, F(2, 168) = 4.78, $p < .01$. We will discuss how annotations and example-task similarity affect mappings in the next section.

## 5.2. To what degree does the ability to map an example and target problem correlate with task success?

First we discuss the overall results for mapping and task success as well as example similarity, which both support the idea that mapping and task success are related. However, it turns out that mappings and performance actually have a relatively low correlation. Finally, we describe behaviors that may be influencing the low correlation.

### 5.2.1. Overall Mapping and Task Success

In the majority of tasks, participants either made no measurable progress towards a solution (51.6% ended the task in the *Exploring & Searching Stage*) or correctly completed them (30.3% ended the task in the *Completed Stage*). Table 2 shows the percentage of the total tasks that reached each stage and the proportions of tasks for the stages that had correct mappings. While there are some instances in which participants have identified the necessary code elements and failed to arrive at a fully correct solution (ending the task in the *Assembling Stage*), the ability to modify and test a program appears to enable those who successfully added the needed code element to complete the task. We found an increasing proportion of correct mappings for tasks ending in the later stages. For tasks ending in the *Exploring & Searching Stage*, 45.6% of tasks had correct mappings. By the *Completed Stage*, 72.6% of tasks had correct mappings. This trend is consistent with correct mappings contributing to task success.

### 5.2.2. Annotation and Mapping Results

Using MANCOVA and Roy's largest root, there is a significant effect of annotations versus no annotation on correct mapping of the example to the program, $\Theta$ = .38, F(12, 66) = 2.10, $p < .05$. Separate univariate ANCOVAs revealed that there are no significant differences between the three annotation styles. Compared to the no annotation condition, all participants whose examples had annotations constructed correct example-program mappings more often. There is also a significant effect of annotations versus no annotation on correct mappings when looking at both the similar example tasks, $\Theta$ = .23, F(6, 72) = 2.73, $p < .05$, and the dissimilar example tasks, $\Theta$ = .25, F(6,72) = 3.01, $p < .05$.

This suggests that even for similar examples, annotations are important in helping novices understand mappings. While this is what we expected for dissimilar tasks, as the annotations can help to fill in information missing when there is low surface similarity, this is unexpected for the similar example tasks. We would expect the surface similarity in the similar example tasks to make them doable without annotations. We believe one reason for this may be the difficulty level of some tasks. When a novice programmer is working on a task that is beyond their current level of understanding, surface similarity may not be enough to assist in understanding the correlation between an example and a problem, but an annotation can improve this.

### 5.2.3. Example Similarity and Mapping

A Wilcoxon Signed-Ranks test revealed that participants were more successful at similar example mapping (Mdn = 5 of 6 correct mappings) than dissimilar example mapping (Mdn = 3 of 6 correct mappings), $p < .001$, $r$ = -.65.

### 5.2.4. Mapping and Task Success Connection to Analogical Reasoning

While the upward trend of mappings for each stage and the similar and dissimilar example results start to support a relationship between mapping and task success, we found only a weak correlation, rb = .21, $p < .001$. This low correlation provides some support for Novick and Holyoak's findings that mappings are necessary but not sufficient for problem solving [71]. However, it is worth noting that nearly 28% of fully correct tasks did not have correct mappings, which suggests that some participants may be solving tasks using a strategy outside of analogical reasoning.

If Gentner's structure-mapping theory holds for programming, we should have seen a strong correlation between mapping success and program success. Structure-mapping theory states that the primary difficulty problem solving using analogies comes through mapping the problems. If learners can successfully map the problems, they should be able to correctly solve the target problem. In that scenario, we would only observe problems with executing that plan such as difficulty finding a needed program element. Instead, we saw a weak correlation. We explore possible reasons behind the weak correlation using the stages of task completion:

1. Correct mappings and incorrect tasks:
   - Participants may have developed their mappings too late in the task to use them. This could happen as a result of attempting to solve the problem without using the example initially, either through a desire to complete the task independently or because of difficulties understanding the example (discussed in Sec. 5.2.5 **Correct Mappings and Incomplete Solution Plans**). This is suggested by a large number of irrelevant edits in the early stages.
   - Participants may have generated full plans based on the mappings between the example and target problems but struggled to execute those plans within the programming environment (discussed in Sec. 5.2.6 **Correct Mappings and Difficulties Executing Solution Plans**). Many user interface actions in the early stages of a task could support that users were searching for how to execute their solution plans.
2. Incorrect mapping and correct task:
   - Though we based our mapping task on those used in psychology studies in the past, there are differences in blocks programming environments that

may have made the mapping task unclear (discussed in Sec. 5.2.7 **Mapping Task Design**).

We now describe each of these three cases and the data that supports these as possible reasons for the low correlation between mapping and performance.

#### 5.2.5. Correct Mappings and Incomplete Solution Plans

For 25.4% of tasks with correct mappings, participants' behavior suggests that they began with an incomplete solution plan and tested multiple variations to arrive at a solution. Adding the missing code element for a problem marks the boundary between forming a plan and beginning to carry out that plan. In our stage-based model, participants implemented their solution plans during the *Assembling Stage*.

If participants shaped solutions through working with the programming environment, we would expect to see more testing behavior through higher numbers of edits and program executions. To explore this, we divided the tasks in the *Assembling Stage* with correct mappings into low editing (0 or 1 edits beyond those necessary to solve the task's problem) and high editing (two or more extra edits) groups. A Wilcoxon Signed-Ranks test revealed that there is a significant difference in the number of program executions between the high (Mdn = 2) and low (Mdn = 1) editing groups, $p < .001$, $r$ = -.38. Behavior in the low editing group (74.6% of tasks) is consistent with Gentner's separation between planning a problem solution and executing it [70]. Behavior in the high editing group suggests that for 25.4% of tasks, participants began to execute an incomplete solution plan, supporting Novick and Holyoak's findings that mappings are not always sufficient to enable problem solving [71]. These types of difficulties spurred our third question about the ability of programming behavior using examples to predict success or failure.

#### 5.2.6. Correct Mappings and Difficulties Executing Solution Plans

Difficulties executing a solution plan may explain some task failures, but we failed to find evidence suggesting that execution difficulties are a large source of task failures.

If a participant is able to generate a plan but struggles to execute their plan, we would expect that task to end in the *Exploring & Searching Stage* (no progress) or *Ready-to-Program Stage* (interface correct) with correct mappings and a higher rate of user interface actions due to search behavior. A Wilcoxon Signed-Ranks test revealed that there is a marginally significant difference between the number of user interface actions among tasks ending in the *Exploring & Searching Stage* and the *Ready-to-Program Stage* with correct mappings (Mdn = 110) and without correct mappings (Mdn = 84), $p$ = .05, $r$ = -.14. This suggests that there are likely some tasks in which participants had a plan but struggled to execute it. However, it seems unlikely that difficulties executing a solution plan account for a large proportion of failed tasks.

#### 5.2.7. Mapping Task Design

In 27.3% of the tasks that reached the *Completed Stage* (9.1% of all tasks), participants produced incorrect mappings but correctly completed the program. Based on reviewing a random selection of 20% of these mappings, we observed that:

- Some participants struggled to represent mappings where a code element present in the example was related to something that needed to be added to their program. This meant that one correct mapping was to map a block in the example to an empty space in the task code. It was not clearly specified how to do this, so this was a weakness of the mapping task design. Since many tasks require that participants add programming constructs, we may need to explore alternative methods for capturing these mappings.
- In some cases, participants mapped sub-elements of a statement rather than full code statements. For example, a participant might map the method callers, names, and parameters for two statements. We did not rate sub-element mappings as correct, even when a mapping between their parent statements was correct. In these cases, it seemed more likely that participants were drawing lines between everything that was in the same location in the code, rather than understanding that the methods as a whole were the important related components. However, it is possible that participants who created sub-element mappings may have correctly understood the code.

### 5.3. Where in the process of solving a programming problem using an example do novices struggle and which behaviors predict success and failure?

In order to understand what challenges participants were having, we wanted to determine what kinds of programming behavior predict success and failure at each of the stages (*Exploring & Searching*, *Ready-to-Program*, *Assembling*, and *Completed*). To do this, we (1) identified predictive features from among the programming behavior features, as shown in Table 3, and then (2) used the subset of predictive features to train a decision tree that predicts successful completion of each of the stages in our stage model.

In this process, we used two classifiers: random forests and decision trees. A random forest is "a classifier consisting of a collection of tree-structured classifiers" where the input to each of the classifiers is an independent identically distributed random vector and each classifier "votes" for the most popular [75]. In our analysis, we used R's randomForest package, which implements Breiman and Cutler's algorithm for random forests [76]. A decision tree is a classifier that partitions the space based on the values of the internal nodes. Each leaf of the tree has the most likely target value based on the paths from the root that reach that leaf.

To identify predictive features for each stage, we trained a random forest of 500 trees using a combination of performance based features (see Table 1) and demographic features. The demographic features included age, condition, and gender.

| Stage | Predictive Features(%MSE explained) |
|---|---|
| Exploring & Searching | Idle Time (40.26)<br>Num. of Runs (28.94)<br>Num. of Code Edits (28.73)<br>Num. of UI Actions (19.47)<br>Num. of Tinker Edits (16.18)<br>UI Time (15.08)<br>Num. of Irrelevant Code Edits (13.78)<br>Num. of Semi-relevant Code Edits (11.23)<br>Age (10.16) |
| Ready-to-Program | UI Correct Time (27.95)<br>Num. of Runs (20.24)<br>Num. of UI Actions (17.21)<br>UI Time (14.04)<br>Idle Time (13.41)<br>Num. of Code Edits (12.80)<br>Example Time (10.74)<br>Num. of Irrelevant Edits (10.47) |
| Assembling | Num. of UI Actions (16.63)<br>UI Time (13.47)<br>Num. of Irrelevant Edits (12.63)<br>Num. of Relevant Edits (10.60)<br>Num. of Tinker Edits (10.11) |

Table 3: Predictive features for each stage

Note that all of the performance features are stage-specific. In predicting which tasks would achieve the *Ready-to-Program Stage*, we used only performance features for the previous stage, the *Exploring & Searching Stage*. Then, for each of the forests, we examined the variable importance statistics and identified the subset of features that improved the mean-squared error by more than 10% for use in constructing the decision tree.

Next, we trained an individual decision tree for each stage using our selected subset of features (see Table 3). We use the resulting decision trees to pull out behavioral differences between successful and unsuccessful participants at each stage. In training both the random forests and decision trees, we included only tasks that achieved the previous stage. Notice that most of the features used for this analysis do not explicitly focus on the example, but instead attempt to measure the behaviors that indicate difficulties using the example. There are several reasons for this: (1) we did did not use eye-tracking, so the main ways to measure example use were mouse movements and example executions, and (2) participants very rarely executed the example code.

First, we will discuss how we assessed our decision tree models and then we will explore the decision trees for each of the stages and discuss the features that predict success and failure for those stages.

#### 5.3.1. Decision Tree Model Quality

We assessed the quality of our three decision tree models in two ways:

First, we constructed a Baseline Model that always predicts the most common classification (success or failure) for that stage. Using a binomial test, we evaluate whether the decision tree performs significantly better than this baseline (see Table 4). We acknowledge that comparing to the Baseline Model is a relatively weak test of significance.

To provide additional insight, we also constructed a Null Mixed Logistic Regression model with two random factors: task ID and participant ID. This model leverages the fact that knowing the difficulty of the task and the general performance of a participant is often sufficient to make good performance predictions. As expected, the Null Mixed Logistic Regression models perform fairly well. It is important to note that they leverage task and participant information that we intentionally excluded from our decision tree model. Yet, the decision tree models achieve similar accuracy using purely behavioral data (see Table 4).

In training both the random forests and decision trees, we included only the subset of tasks that successfully achieved the previous stage.

#### 5.3.2. Predicting Ready-to-Program Stage Success

Figure 6 shows the decision tree that predicts whether a given task will achieve the *Ready-to-Program Stage* (correct interface state) given the programming behavior during the *Exploring & Searching Stage*. Any amount of code editing during the *Exploring & Searching Stage* is a strong predictor that task will not achieve the *Ready-to-Program Stage*. Specifically, tasks without the *Exploring & Searching Stage* code editing successfully reach the *Ready-to-Program Stage* 91% of the time; those with code editing successfully reach the *Ready-to-Program Stage* only 20% of the time. Among the tasks without code editing, those with task idle times of more than two minutes are dramatically less successful, reaching the *Ready-to-Program Stage* only 25% of the time. This behavior may indicate that participants did not know what to do and were reluctant to explore. Among the tasks with code edits during the *Exploring & Searching Stage*, running the program once or not at all increased the chances of successfully reaching the *Ready-to-Program Stage* to 60%. These participants made and tested a small number of changes before moving on to searching for the necessary code elements in the interface.

#### 5.3.3. Predicting Assembling Stage Success

A task successfully reaches the *Assembling Stage* when the participant adds a code element needed for task completion. Figure 7 shows the decision tree that predicts success at reaching the *Assembling Stage* based on the programming behavior in the *Ready-to-Program Stage*. The strongest predictor of successfully achieving the *Assembling Stage* is the number of user interface actions that occur in the previous stage. If a participant makes a large number of user interface actions (i.e. 26 or more), this may suggest that they reached the *Ready-to-Program Stage* (the correct interface state) by chance; 77% of these tasks end in the *Ready-to-Program Stage*. Additionally, tasks with a large number of user interface actions are less likely to have correct mappings. For tasks with 26 or more user interface actions, 52% have correct mappings. For those with fewer than 26 user interface actions, 69% have correct mappings. Finally, we note that if participants reached the

| Stage | # of Tasks | Baseline Model Accuracy | Null Mixed Logistic Regression Accuracy | Decision Tree Model Accuracy |
|---|---|---|---|---|
| Ready-to-program | 531 | 51.79% | 88.89% | 87.76%** |
| Assembling | 256 | 80.08% | 89.06% | 92.97%** |
| Completed | 205 | 78.05% | 82.93% | 84.39%* |

Table 4: Decision tree model quality. **p< .0001, *p< .05

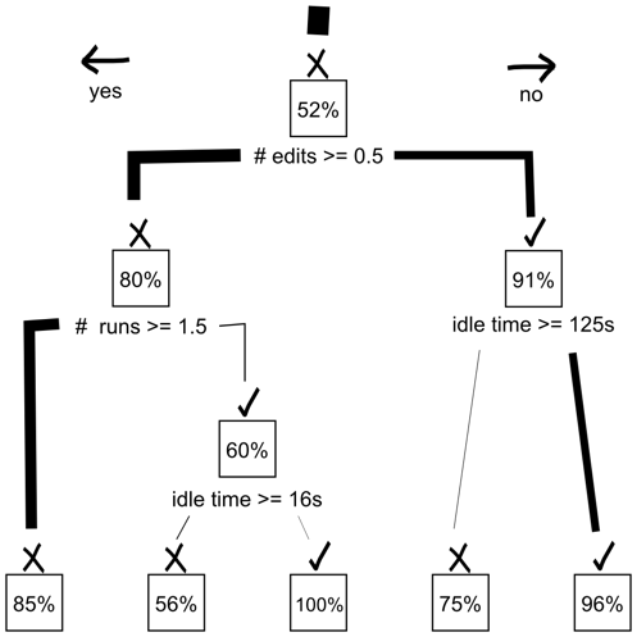


Figure 6: The decision tree predicting success at achieving the *Ready-to-Program Stage* based on *Exploring & Searching Stage* performance.
Yes/no: value of the inequalities.
Line thickness: percentage of tasks following each path.
Percentages in boxes: accuracy of each node.
'X': predict failure to reach stage.
Check marks: predict success at reaching stage.

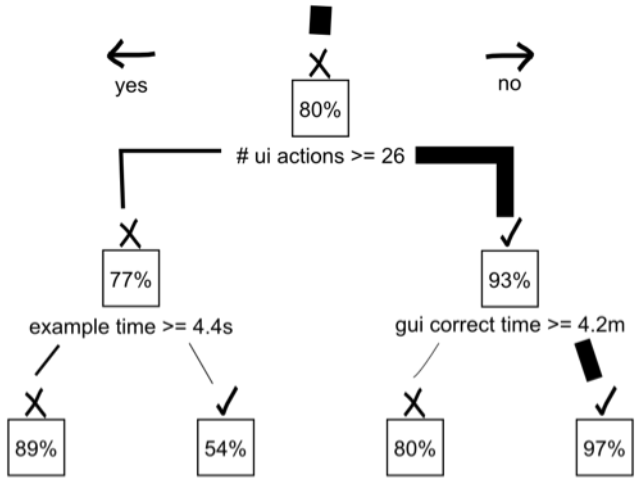


Figure 7: The decision tree predicting success at achieving the *Assembling Stage* based on the *Ready-to-Program Stage* performance.
Yes/no: value of the inequalities.
Line thickness: percentage of tasks following each path.
Percentages in boxes: accuracy of each node.
'X': predict failure to reach stage.
Check marks: predict success at reaching stage.

*Ready-to-Program Stage* late in the task, that often predicted failure.

#### 5.3.4. Predicting Completed Stage Success

Figure 8 shows the decision tree that predicts success at reaching the *Completed Stage*, based on the programming behavior in the *Assembling Stage*. Effectively, tasks break into three categories based on the number of user interface actions. Overall, tasks with fewer than 35 user interface actions were most successful: 88% achieve a correct task solution. Tasks with a mid-range (ranging from 35 to 85) number of user interface actions were least successful, only 22% arrived at a correct solution. Interestingly, tasks with the highest number of user interface actions were more successful than those in the mid-range: 54% achieved a correct solution. We note that user interface actions are strongly correlated with code editing, $p < .001$, $r(205)$=.64.

Once participants reach the *Assembling Stage*, they have all of the code elements necessary to correctly solve the task. So, moving from the *Assembling Stage* to the *Completed Stage* is a matter of placing the code elements in the right positions. The group with the lowest number of user interface actions shows a more selective approach to making code changes. In the middle, participants made a larger number of edits, but likely with less deliberation about each individual change. Since even with relatively short programs, there are a large number of potential edits that can be made, this strategy tended towards failure. Finally, the group that made the largest number of code edits shows an increase in overall success rates. This may be a result of a fast guess and test approach.

## 6. Discussion

We first go into further discussion on each of our primary topics: (1) example annotations, (2) analogical reasoning, and (3) programming behavior analysis as compared to a qualitative study on example use. Then, we discuss the importance of how this work fits into the larger picture of blocks-based programming environments.

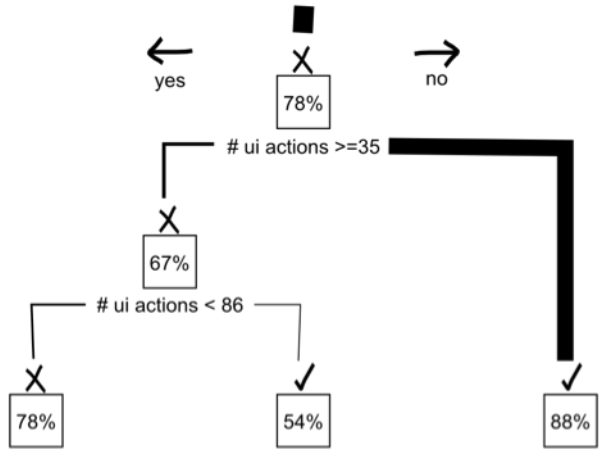


Figure 8: The decision tree predicting success at achieving the *Completed Stage* based on the *Assembling Stage* performance.
Yes/no: value of the inequalities.
Line thickness: percentage of tasks following each path.
Percentages in boxes: accuracy of each node.
'X': predict failure to reach stage.
Check marks: predict success at reaching stage.

### 6.1. High Failure Rates Regardless of Annotation

While we hypothesized that the different annotation conditions would provide different affordances for similar and dissimilar example mappings and performance, we did not find significant performance differences between the individual annotation styles for mapping or for task performance. One reason for this might be that each of the annotations provided extra information to lead participants to better mappings and solutions [17]. Yet, novices still had significant problems completing tasks in all conditions. Looking at the results, the two main issues causing low overall success were (1) inability to move beyond the *Exploring & Searching Stage*, and (2) incomplete solution plans with correct mappings. In order to help novice programmers in using examples, future work should address these two problems.

Our study results suggest that supporting novices in taking the first steps in using an example is crucial, similar to a finding that getting started is generally difficult for end-user programmers working on specified tasks [77]. In 51% of tasks, participants made no discernible progress towards a solution. Of those who made any progress, 62% arrived at the correct solution. One reason for this is that novice programmers may be overwhelmed at first, trying to figure out what to focus on and how to start. Only 45% of participants at this stage made correct mappings, so it may be critical to nudge novices back toward looking at and using an example if they have made a certain number of edits with no progress. Another issue might be that a programmer does not understand the example if their mapping is wrong, which could indicate the need for multiple examples of varying surface similarity, allowing novices to search for another example if the first one is confusing.

We showed that one reason behind failure is that once novice programmers reach the *Assembling Stage*, they often do not have a complete or correct plan for how to reach the solution. This means that although they understand the relationships between the task and the example, they do not know how to formulate a plan. The first step in helping novices in this situation is to be able to identify that they are having this specific issue. Based on our results, we believe systems should automatically keep track of whether a novice has added the correct component, whether they have reached a correct solution, and whether they have made multiple edits. Essentially, a system could leverage the stage model discussed in our results to keep track of progress and provide strategic advice. Future work could use this model as a starting point for educational strategies in programming, similar to the problem solving strategies introduced by Loksa et al. [78].

### 6.2. Programming Examples and Analogical Reasoning

In nearly 75% of tasks, participants' behavior is consistent with the prediction that mapping success is sufficient to enable task success. However, in the remaining 25% of tasks, participants made a series of code edits and program executions that suggests they did not have a full solution in mind when they began to make changes to the target program. If we can think of example code use as analogical problem solving, this opens up this topic to being able to apply other findings from analogical problem solving research to support for novice programming with examples. Two applications of analogical reasoning that could apply to novice programming are (1) using visual analogies as hints and (2) work on analogical reasoning across ages.

Research has found that visual analogies can be used as cues to hint at an analogy that had been presented earlier [79]. This could apply to novice programming because once a novice programmer has learned a concept once, they may still not realize that they should use it in another situation. In this case, a visual analogy might be useful because it would cue memory of previously provided information, without being repetitive. One study on programmers found that they naturally use examples as reminders [9], so this might be useful for novices as they are learning as well.

Work on analogical problem solving and the development of reasoning in children may also apply to example use in novice programming, since blocks-based environments exist for all ages. One study found that middle school students were stronger at solving analogy problems, but pre-schoolers were also capable of using an analogy to solve a new problem [80]. Further, middle school students often had similar performance as adults. This implies that we may be able to use similar support for middle school children and adults. Examples are also likely to be applicable for very young novice programmers, but the analogies must be very carefully selected [80].

### 6.3. Programming Behavior and Relation to Qualitative Study on Example Use

This study inspired a qualitative study on the barriers of novice programmer example use that looked at what novice programmers talked about during similar tasks to this study [53]. In the qualitative study, participants worked in pairs and only saw examples with the Visual Emphasis annotation. Findings included hurdles to success like being distracted by other exciting aspects of the programming environment, not understanding the example, not knowing where to find components in the interface, and trouble implementing solutions due to misunderstandings about the code.

Three hurdles from the qualitative study seem to align well with predictions in the decision trees: the context distraction hurdle, the example comprehension hurdle, and the programming environment hurdle.

- In the context distraction hurdle, participants talked about exploring the programming interface and trying out ideas they had to solve the task based on what they saw in the programming environment (rather than the example). This aligns with the decision tree in which having edits in the *Exploring & Searching Stage* often leads to failure, shown in Figure 6.
- The example comprehension hurdle captured participants talking about not understanding how an example worked or how it was relevant to a task. Spending long amounts of time with the example (as in the *Assembling Stage* decision tree in Figure 7) may indicate that a participant was trying to understand an example but struggling.
- In the programming environment hurdle, participants discussed having trouble finding a block they needed to solve a task, or having trouble editing their code in some manner. This aligns with the fact that a large number of UI actions in the *Ready-to-program Stage* likely means that a participant will not reach the *Assembling Stage*, as shown in the first prediction branch in Figure 7.

Novice programmers' descriptions of their difficulties on similar tasks aligning with the behavioral predictions in this analysis supports the value of using decision trees and behavioral data to explore example use and predict success.

### 6.4. Examples and Blocks-Based Programming

While examples are readily available for programmers in text languages, there are many fewer resources for programmers in blocks-based programming environments to find examples. Part of this may be due to the fact that it is not as easy to quickly copy and paste code to a forum like it is for text languages. However, the popularity of using example code in programming implies that it is important for the research community to address example use in blocks-based programming languages. This research topic is further compounded by the fact that most blocks-based programming language users are novices, so using examples is not as straightforward as it would be for experts. This study contributes a better understanding of novice blocks programmers using examples with varying annotations and similarities. This study also begins to explore two ways to predict success or failure of novice programmers using examples in blocks-based programming environments.

For the analogical mapping task, participants drew lines between components that were related in the example and the task. While there were several challenges in the implementation of this task, the challenges for a text-based language would be different. Instead of having discrete components, or blocks, that can be connected with lines, the participants would have to somehow decide which part of the code they wanted to connect and mark that in an understandable way. For novices, this might be more challenging because blocks of code are not necessarily as obvious in a text language. Furthermore, it would be much more difficult to group correct and incorrect mappings because participants would be able to draw lines between many more combinations of characters, whereas in a blocks-based programming environment, there are more constraints.

Our task stages and decision tree predictive modeling were also highly blocks-specific. For example, some of the stages were based on having the UI in the correct configuration to access the blocks needed to succeed in the task. This works across almost all blocks-based programming environments, where blocks are organized in palettes that programmers must correctly select to find the component that they need. In a text language, the intermediate steps would be different and likely more difficult to observe. Furthermore, code edits are easier to define in a blocks-based programming environment because it is easy to track when a code block has been added to a program, modified in a discrete way, or removed. In a text programming language, it would be more challenging to define when a code change has started and ended. While heuristics could be created for this purpose, the strategy used in this paper suits a blocks-based programming language more easily and effectively.

### 6.5. Limitations

In this study, we focused exclusively on the behavior of young novices as they attempt to use examples. Although young novice programmers may share some challenges with older novices, there are likely unique features about their approach to using examples. While we suspect that analogical reasoning has similar behavior in older children and adults, we do not know which aspects of the programming behavior we observed would apply to adult novices.

Additionally, participants were unfamiliar with both the target program and the example. Some prior work suggests that adult novices often attempt to integrate several snippets of found code to solve a problem [13]. This results in a situation similar to that of our study: novice programmers have code to modify that they do not fully understand and an example they want to apply to it. We think this is an important type of example use, but we acknowledge that it does not capture

all example use. The combination of a familiar, understood target program and an unfamiliar example is important and not addressed through this study.

## 7. Conclusions and Future Work

Overall, our results support other findings that completing programming tasks using examples is challenging for novice programmers. While similar examples and annotations help novice programmers perform better, it is clear that these are not enough support for novice programmers using examples. However, it is interesting that the three annotation conditions were not significantly different, indicating that the simple visual emphasis may be enough help even for novices to draw attention to the part of the example related to their task. This could be highly valuable if a system needed to automatically annotate examples, as visual emphasis is the only of the three where automation would be a viable option.

In terms of predicting success, analogical mapping seems like a promising method, but needs some improvement. Although we did not want to interrupt task flow to assess analogical mappings, future educational systems may benefit from integrating analogical mapping tasks into learning materials, which would allow them to measure analogical mappings during tasks. Furthermore, if future work could reduce the number of errors in mapping due to the way the mapping task was operationalized, there might be a higher correlation between mappings and success. Specifically, adding more constraints about which components can be mapped and how to map blocks in an example to missing blocks in a task are two important future directions for this evaluation method. We believe better constraints and a stricter time limit could vastly improve the consistency and accuracy of this method.

The stage-based analysis and decision tree models provide information about which programming behaviors impact success at each stage of the problem solving process. We found that in most tasks that do not succeed, participants did not progress past the *Exploring & Searching Stage*. This has important implications for the beginning of a task, when participants seem to need the most help. The large number of UI actions leading to failure in later stages gives some support to the idea of adding programming environment assistance to examples if programmers do not have the option to directly insert the example code into their program. These stages are highly applicable to other blocks-based programming environments where programmers normally must follow the same process to find the components they need in the interface, add them to their program, and then modify the program to complete the task. One promising future direction for this analysis of programming behaviors is in designing educational systems, such as intelligent tutoring systems, where a system could assess programmer behavior in real-time and use that to provide feedback to the learner.

## References


[1] M. Resnick, J. Maloney, A. Monroy-Hernndez, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[2] "MIT App Inventor | Explore MIT App Inventor." [Online]. http://appinventor.mit.edu/explore/

[3] "Looking Glass Community." [Online]. https://lookingglass.wustl.edu/

[4] M. Guzdial, "Why the U.S. is not ready for mandatory CS education," *Communications of the ACM*, vol. 57, no. 8, pp. 8–9, Aug. 2014.

[5] K. J. Harms, D. Cosgrove, S. Gray, and C. Kelleher, "Automatically generating tutorials to enable middle school children to learn programming independently," in *Proceedings of the 12th International Conference on Interaction Design and Children*. ACM, 2013, pp. 11–19.

[6] "Anybody can learn | Code.org." [Online]. http://code.org/

[7] M. J. Lee and A. J. Ko, "Personifying programming tool feedback improves novice programmers' learning," in *Proceedings of the Seventh International Workshop on Computing Education Research*. ACM, 2011, pp. 109–116.

[8] C. J. Butz, S. Hua, and R. B. Maguire, "A web-based Bayesian intelligent tutoring system for computer programming," *Web Intelligence and Agent Systems: An International Journal*, vol. 4, no. 1, pp. 77–97, 2006.

[9] 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*. ACM, 2009, pp. 1589–1598.

[10] J. Brandt, P. J. Guo, J. Lewenstein, S. R. Klemmer, and M. Dontcheva, "Writing code to prototype, ideate, and discover," *IEEE Software*, vol. 26, no. 5, pp. 18–24, 2009.

[11] B. Dorn and M. Guzdial, "Graphic designers who program as informal computer science learners," in *Proceedings 2nd International Workshop on Computing Education Research*, 2006, pp. 127–134.

[12] M. B. Rosson, J. Ballin, and J. Rode, "Who, what, and how: A survey of informal and professional web developers," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2005, pp. 199–206.

[13] M. B. Rosson, J. Ballin, and H. Nash, "Everyday Programming: Challenges and Opportunities for Informal Web Development," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2004, pp. 123–130.

[14] P. Gross and C. Kelleher, "Non-programmers identifying functionality in unfamiliar code: strategies and barriers," *Journal of Visual Languages & Computing*, vol. 21, no. 5, pp. 263–276, 2010.

[15] 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*. ACM, 2010, pp. 513–522.

[16] 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*. ACM, 2012, pp. 3205–3214.

[17] M. L. Gick and K. J. Holyoak, "Analogical problem solving," *Cognitive Psychology*, vol. 12, no. 3, pp. 306–355, 1980.

[18] D. Gentner, "Structure-mapping: A theoretical framework for analogy," *Cognitive Science*, vol. 7, no. 2, pp. 155–170, 1983.

[19] "Hoogle." [Online]. https://www.haskell.org/hoogle/

[20] "Java Examples - JExamples.com." [Online]. http://www.jexamples.com/

[21] "Google Code." [Online]. https://code.google.com/

[22] "Open Hub Code Search." [Online]. http://code.openhub.net/

[23] J. Stylos and B. A. Myers, "Mica: A web-search tool for finding API components and examples," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2006, pp. 195–202.

[24] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *Proceedings of the 27th International Conference on Software Engineering*. ACM, 2005, pp. 117–125.

[25] O. Hummel, W. Janjic, and C. Atkinson, "Code conjurer: Pulling reusable software out of thin air," *IEEE Software*, vol. 25, no. 5, pp. 45–52, 2008.

[26] D. Mandelin, L. Xu, R. Bodk, and D. Kimelman, "Jungloid mining: Helping to navigate the API jungle," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 48–61.

[27] N. Sahavechaphan and K. Claypool, "Xsnippet: Mining for sample code," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 413–430.

[28] S. Thummalapenta and T. Xie, "Parseweb: a programmer assistant for reusing open source code on the web," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2007, pp. 204–213.

[29] Y. Ye and G. Fischer, "Supporting reuse by delivering task-relevant and personalized information," in *Proceedings of the 24th International Conference on Software Engineering*. ACM, 2002, pp. 513–523.

[30] P. Brusilovsky, "WebEx: Learning from Examples in a Programming Course." in *WebNet*, vol. 1, 2001, pp. 124–129.

[31] R. Burow and G. Weber, "Example explanation in learning environments," in *Intelligent Tutoring Systems*. Springer, 1996, pp. 457–465.

[32] J. M. Faries and B. J. Reiser, "Access and use of previous solutions in a problem solving situation," Cognitive Science Laboratory, Princeton University, Tech. Rep. CSL Report 29, Jun. 1988.

[33] M. Guzdial and C. Kehoe, "Apprenticeship-based learning environments: A principled approach to providing software-realized scaffolding through hypermedia," *Journal of Interactive Learning Research*, vol. 9, pp. 289–336, 1998.

[34] L. Hohmann, M. Guzdial, and E. Soloway, "SODA: A computer-aided design environment for the doing and learning of software design," in *Computer Assisted Learning*. Springer, 1992, pp. 307–319.

[35] M. C. Linn, "How can hypermedia tools help teach programming?" *Learning and Instruction*, vol. 2, no. 2, pp. 119–139, 1992.

[36] M. C. Linn and M. J. Clancy, "Can experts' explanations help students develop program design skills?" *International Journal of Man-Machine Studies*, vol. 36, no. 4, pp. 511–551, 1992.

[37] H. Ramadhan, "An intelligent discovery programming system," in *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing: Technological Challenges of the 1990's*. ACM, 1992, pp. 149–159.

[38] A. S. Bruckman, "MOOSE Crossing: Construction, community, and learning in a networked virtual world for kids," Ph.D. dissertation, Massachusetts Institute of Technology, 1997.

[39] K. J. Harms, J. H. Kerr, M. Ichinco, M. Santolucito, A. Chuck, T. Koscik, M. Chou, and C. L. Kelleher, "Designing a community to support long-term interest in programming for middle school children," in *Proceedings of the 11th International Conference on Interaction Design and Children*, ser. IDC '12. New York, NY, USA: ACM, 2012, pp. 304–307.

[40] 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 *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2012, pp. 59–66.

[41] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do: suggesting solutions to error messages," in *Proceedings 28th International Conference on Human Factors in Computing Systems*, 2010, pp. 1019–1028.

[42] 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*. ACM, 2012, pp. 2697–2706.

[43] L. R. Neal, "A system for example-based programming," in *ACM SIGCHI Bulletin*, vol. 20. ACM, 1989, pp. 63–68.

[44] 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*. ACM, 1993, pp. 67–73.

[45] G. Weber and A. Mollenberg, "ELM-PE: A Knowledge-based Programming Environment for Learning LISP." in *Proceedings of ED-MEDIA 94–World Conference on Educational Multimedia and Hypermedia*, 1994, pp. 557–562.

[46] K. Østerbye, "Minimalist documentation of frameworks," in *ECOOP Workshops*, 1999, pp. 172–173.

[47] T. Vestdam, "Generating consistent program tutorials," in *Proceedings of NWPER2002-Nordic Workshop on on Programming and Software Development Tools and Techniques*, 2002.

[48] J. M. Carroll, P. L. Smith-Kerker, J. R. Ford, and S. A. Mazur-Rimetz, "The minimal manual," *Human-Computer Interaction*, vol. 3, no. 2, pp. 123–153, 1987.

[49] J. M. Carroll, *The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill*. MIT Press, 1990.

[50] J. L. Kolodner, M. T. Cox, and P. A. Gonzlez-Calero, "Case-based reasoning-inspired approaches to education," *The Knowledge Engineering Review*, vol. 20, no. 03, pp. 299–303, 2005.

[51] E. Domeshek and J. Kolodner, "Using the points of large cases," *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing*, vol. 7, no. 02, p. 87, May 1993.

[52] J. L. Kolodner and M. Guzdial, "Theory and practice of case-based learning aids," *Theoretical Foundations of Learning Environments*, pp. 215–242, 2000.

[53] M. Ichinco and C. Kelleher, "Exploring novice programmer example use," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2015, pp. 63–71.

[54] J. Sweller and G. A. Cooper, "The use of worked examples as a substitute for problem solving in learning algebra," *Cognition and Instruction*, vol. 2, no. 1, pp. 59–89, 1985.

[55] A. Rourke and J. Sweller, "The worked-example effect using ill-defined problems: Learning to recognise designers' styles," *Learning and Instruction*, vol. 19, no. 2, pp. 185–199, Apr. 2009.

[56] J. J. Van Merrinboer and M. B. De Croock, "Strategies for computer-based programming instruction: Program completion vs. program generation," *Journal of Educational Computing Research*, vol. 8, no. 3, pp. 365–394, 1992.

[57] B. B. Morrison, L. E. Margulieux, B. Ericson, and M. Guzdial, "Subgoals Help Students Solve Parsons Problems," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 2016, pp. 42–47.

[58] F. G. Paas and J. J. Van Merrinboer, "Variability of worked examples and transfer of geometrical problem-solving skills: A cognitive-load approach." *Journal of Educational Psychology*, vol. 86, no. 1, p. 122, 1994.

[59] J. Sweller, "Element interactivity and intrinsic, extraneous, and germane cognitive load," *Educational Psychology Review*, vol. 22, no. 2, pp. 123–138, 2010.

[60] "Scratch - Videos." [Online]. https://scratch.mit.edu/help/videos/

[61] "Tutorials for App Inventor | Explore MIT App Inventor." [Online]. http://appinventor.mit.edu/explore/ai2/tutorials.html

[62] S. Pongnumkul, M. Dontcheva, W. Li, J. Wang, L. Bourdev, S. Avidan, and M. F. Cohen, "Pause-and-play: automatically linking screencast video tutorials with applications," in *Proceedings of the 24th annual ACM Symposium on User Interface Software and Technology*. ACM, 2011, pp. 135–144.

[63] K. J. Harms, N. Rowlett, and C. Kelleher, "Enabling independent learning of programming concepts through programming completion puzzles," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2015, pp. 271–279.

[64] "Blockly Games." [Online]. https://blockly-games.appspot.com/

[65] M. J. Lee and A. J. Ko, "Comparing the effectiveness of online learning approaches on CS1 learning outcomes," in *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*. ACM, 2015, pp. 237–246.

[66] "Kodu | Home." [Online]. http://www.kodugamelab.com/

[67] "TouchDevelop - create apps everywhere, on all your devices!" [Online]. https://www.touchdevelop.com/

[68] S. Surisetty, C. Law, and C. Scaffidi, "Behavior-based clustering of visual code," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2015, pp. 261–269.

[69] L. E. Richland, K. J. Holyoak, and J. W. Stigler, "Analogy use in eighth-grade mathematics classrooms," *Cognition and Instruction*, vol. 22, no. 1, pp. 37–60, 2004.

[70] D. Gentner and C. Toupin, "Systematicity and surface similarity in the development of analogy," *Cognitive Science*, vol. 10, no. 3, pp. 277–300, 1986.

[71] L. R. Novick and K. J. Holyoak, "Mathematical problem solving by analogy." *Journal of Experimental Psychology: Learning, Memory, and Cognition*, vol. 17, no. 3, p. 398, 1991.

[72] S. Hudson, J. Fogarty, C. Atkeson, D. Avrahami, J. Forlizzi, S. Kiesler, J. Lee, and J. Yang, "Predicting human interruptibility with sensors: a Wizard of Oz feasibility study," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2003, pp. 257–264.

[73] B. A. Spellman and K. J. Holyoak, "If Saddam is Hitler then who is George Bush? Analogical mapping between systems of social roles." *Journal of Personality and Social Psychology*, vol. 62, no. 6, p. 913, 1992.

[74] L. Beckwith, C. Kissinger, M. Burnett, S. Wiedenbeck, J. Lawrance, A. Blackwell, and C. Cook, "Tinkering and gender in end-user programmers' debugging," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2006, pp. 231–240.

[75] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[76] L. Breiman and A. Cutler, "Random forests – classification description," 2007. [Online]. https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

[77] J. Cao, S. D. Fleming, and M. M. Burnett, "An exploration of design opportunities for "gardening" end-user programmers' ideas." in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2011, pp. 35–42.

[78] D. Loksa, A. J. Ko, W. Jernigan, A. Oleson, C. J. Mendez, and M. M. Burnett, "Programming, problem solving, and self-awareness: Effects of explicit guidance," *Proceedings International Conference on Human factors In Computing Systems*, 2016.

[79] M. Beveridge and E. Parkins, "Visual representation in analogical problem solving," *Memory & Cognition*, vol. 15, no. 3, pp. 230–237, 1987.

[80] K. J. Holyoak, E. N. Junn, and D. O. Billman, "Development of analogical problem-solving skill," *Child Development*, pp. 2042–2055, 1984.