Subgoal-Labeled Instructional Material Improves Performance and Transfer in Learning to Develop Mobile Applications

Lauren Margulieux
Georgia Institute of Technology
School of Psychology
Atlanta, GA, 30332-0170, USA
1-404-894-7556
l.marg@gatech.edu

Mark Guzdial
Georgia Institute of Technology
School of Interactive Computing
Atlanta, GA, 30332-0760, USA
1-404-894-5618
guzdial@cc.gatech.edu

Richard Catrambone
Georgia Institute of Technology
School of Psychology
Atlanta, GA, 30332-0170, USA
1-404-894-2682
rc7@prism.gatech.edu

ABSTRACT
Mental models are mental representations of how an action changes a problem state. Creating a mental model early in the learning process is a strong predictor of success in computer science classes. One major problem in computer science education, however, is that novices have difficulty creating mental models perhaps because of the cognitive overload caused by traditional teaching methods. The present study employs subgoal-labeled instructional materials to promote the creation of mental models when teaching novices to program in Android App Inventor. Utilizing this and other well-established educational tools, such as scaffolding, to reduce cognitive load in computer science education could promote the creation of mental models by novices and could reduce barriers to learning programming.

Categories and Subject Descriptors
K.3.2 [Computers and Education]: Computer and Information Science Education – *computer science education, information systems education.*

General Terms
Measurement, Performance, Experimentation, Human Factors, Languages, Theory.

Keywords
Subgoal learning; mental models; cognitive load, instructional text, educational videos

1. INTRODUCTION
As a domain that focuses on problem solving, CS is similar to other procedural domains, such as physics and mathematics, so many of the educational tools from these other domains can be applied to CS. The computational tools used to solve CS problems, however, are more complex than many of the computational tools used in other domains. For example, to solve physics problems, students must use computational tools with which they are already familiar, such as a calculator. To solve CS problems, on the other hand, CS students must learn how to use the computational tools (programming languages) needed to solve the problem and implement the solution. Though CS instructors can emulate the graduated learning that is used in other procedural domains with novice CS learners, the added cognitive load from using a novel programming language increases the demand on working memory and creates a barrier to learning CS and presents a major problem in CS education (Anderson, Farrell, & Sauers, 1984).

Many computer science education researchers have approached this barrier by trying to predict success in programming classes by using background information about the students (e.g., Rountree, Rountree, Robins, & Hannah, 2004; Wilson & Shrock, 2001). Factors that have been analyzed as predictors of success include age, gender, past experience with programming (both formal and informal), mathematical background, self-efficacy (both general and CS-specific), previous academic performance, and previous non-programming experience with computers (e.g., Rountree et al., 2004). These studies have identified a few weak predictors for success in programming. Most of their research, however, has been inconclusive (e.g., no effect of gender), contradictory (e.g., Dehnadi et al., 2009, found no correlation between success and previous programming experience, whereas Wilson & Shrock, 2001, did), or too specific to be practical (e.g., Rountree et al, 2004, found that students whose background was not in science, year in school was second or third, and expected grade was not an A, failed more frequently that students who did not meet those criteria). A meta-analysis by Dehnadi et al. (2009) found one factor that is a strong predictor of success in a programming course: a mental model of programming knowledge. Students with a mental model of programming that they applied consistently to solving computing problems had an 85% pass rate compared to students without a consistent model, who had a 36% pass rate (Dehnadi et al., 2009).

1.1 Mental Models
A mental model is a representation of how an action will affect a system. For example, a car mechanic can mentally simulate how a particular engine adjustment will affect how a car runs. Similarly, a competent programmer will have a mental model that will allow him or her to mentally "run" part of a program in order to predict how a change to a line of code will affect the program. Mental models are important for reasoning in procedural domains because they enable learners to integrate simple skills to achieve a complex skill. Mental models help this process by allowing reasoners to hierarchically classify information and focus on high level problem solving without getting distracted by low level

details (van Merriënboer, Clark, & de Croock, 2002). For example, mental models allow people to conceptualize how to achieve a higher function math skill, such as solving for a variable, without allocating attention to lower function math skills, such as addition and multiplication (because those details are not initially relevant for the higher level goal). The hierarchical structure of mental models in procedural domains also allows people to construct simplified and parsimonious explanations for complex phenomenon because they do not need to incorporate lower level mechanisms into higher level explanation (Norman, 1983).

The problem, however, is that novice programmers have trouble creating mental models for programming knowledge. For example, one problem that novices have is that they try to carry over syntax rules from other languages, such as English and algebra, which complicates learning programming syntax (Davis, Linn, Mann, & Clancy, 1993). Furthermore, novices' mental models emphasize different knowledge than CS experts' mental models (Brooks, 1990). Experts in CS focus on deeper structural aspects of problems because they have mental models to classify problems, whereas novices are misled by incidental features of the problem because they have mental models for syntax (Atkinson, Derry, Renkl, & Wortham, 2000). Therefore, novice solution structures tend to be bottom-up and details-first which often makes their solutions needlessly convoluted (Anderson et al., 1984; Guzdial, 1995).

1.1.1 Creating Mental Models

CS education researchers recognize the need to help learners develop mental models (Du Boulay, 1986). CS instruction, however, tends to emphasize the product of the design and development process, but does not emphasize the process itself, leading to students creating mental models of syntax rather than structure (Brooks, 1990). Moreover, teachers assign grades to the running code but not the process that produced it (Linn & Clancy, 1992). Davis et al. (1993) and Linn and Clancy (1992) argue that to fix this problem, novices need more instruction in how to construct mental models for programming in general rather than more instruction in a specific programming language.

To help students create mental models, Kirschner, Sweller, and Clark (2006) advocate guided instruction over unguided or minimally guided instruction because guided instruction provides students with the instructor's mental model framework into which students can integrate new information. Providing a mental model framework reduces the cognitive load required to learn new information which leads to better long-term learning. In contrast, the problem-based approach, a type of minimally guided instruction, asks students to solve problems as a way of learning. When students begin solving problems too early in the learning process, however, the overload on their working memory inhibits the creation of a mental model and the process of learning (Kirschner et al., 2006). The problem-based approach, however, is analogous to the conventional methods used to introduce programming languages to novices.

Conventional methods of teaching computer science ask novices to solve problems using unfamiliar knowledge while applying novel code construction rules. These methods lead to poor performance on program-writing problems due to cognitive overload and lead to common complaints such as "I don't know where to start," or "You've taught me so many details, I don't know which ones to use," (Clancy & Linn, 1990). Possible solutions to this problem are to reduce the intrinsic cognitive load (cognitive load associated with the material being taught) or to reduce the extraneous cognitive load (cognitive load associated with **how** the material is being taught; Sweller, 2010).

1.2 Reducing Cognitive Load

The only way to decrease intrinsic cognitive load for novices is to reduce the amount of information being used to solve the problem (Sweller, 2010). To reduce the amount of information, components of programming can be isolated so that students are not trying to learn multiple aspects at once. Students can first be taught and tested on how to conceptually solve a problem without concerning themselves with syntax. Drag-and-drop programming languages, such as Android App Inventor and Scratch, replace writing code with dragging components from a menu; this approach reduces the cognitive load associated with syntax because users can easily understand it, which allows users to focus on conceptually solving a problem (Brennan, 2009). Drag-and-drop programming languages are also meant to be easy for users of all ages, backgrounds, and interests and to allow users to "tinker" with components, joining code commands together, similar to "Lego bricks" (p. 63; Resnick et al., 2009). This approach might also help novices create mental models focused on the structure of solutions rather than on syntax.

1.2.1 Worked Examples

To reduce extraneous cognitive load, the present study uses a few techniques; the first of which is worked examples. Anderson et al. (1984) argues that in CS, problem solving by novices is guided by making structural analogies to worked examples. Providing worked examples helps students learn more than instructional text does because worked examples provide information about the application of domain principles (Catrambone, 1996). Additionally, worked examples provide a solution for a learner to study before the student is able to solve problems independently (Atkinson et al., 2000). Worked examples are most effective when labeled subgoals are incorporated because this presentation emphasizes the conceptual structure of the problem solution being taught (Catrambone, 1996).

1.2.2 Subgoal Labels

The main technique that the present study used to reduce extraneous cognitive load is subgoal-labels. Subgoals are "task structures to be learned for solving problems in a domain," (Catrambone, 1994). Subgoal-oriented worked examples have caused problem solving performance improvements in a number of procedural fields, such as statistics (e.g., Catrambone, 1998). Subgoal labels group steps of a worked example into a meaningful unit and help students identify the structural information from incidental information. Learning subgoals can also reduce cognitive load when problem solving because the student has fewer possible problem-solving steps on which to focus (i.e., subgoals [consisting of multiple steps] versus individual steps) similar to functional programming (Clancy & Linn, 1990). Furthermore, subgoal-labeled worked examples might provide students with mental model frameworks. Students who were given labels for subgoals used those labels when explaining how they solved a problem, suggesting that is how they mentally organized information (Catrambone, 1996).

Apprising learners of the underlying structure of the worked examples promotes self-explanation strategies such as "identifying the main features or points of an examples and their underlying purpose" and "connecting concepts in the text and examples" (Bielaczyc, Pirolli, & Brown, 1995; Renkl & Atkinson,

2002), and self-explanation has additional benefits. "Self-explanation directs cognitive resources to deal with relevant [information] and reduces the effect of extraneous cognitive load," (Sweller, 2010, p. 136). Perhaps because self-explanation can reduce cognitive workload, a greater number of self-explanations are related to more successful learning. Learners should be encouraged to explain examples to themselves either through direct training in self-explanation strategies or through the example's structure (e.g., using subgoal labels as described above; Atkinson et al., 2000). Direct training's impact on performance progressively weakens after training and is less potent in the absence of a teacher. In contrast, the structure of an example has a constant impact on performance over time and regardless of the presence of a teacher, possibly because the student is extrinsically guided to self-explain (Atkinson et al., 2000; Renkl & Atkinson, 2002).

1.2.3 Beyond Worked Examples

Worked examples help reduce extraneous load, but learners' benefit from them is capped. According to the ACT-R model, worked examples are important only in the first two of four stages of learning (Anderson, Fincham, & Douglass, 1997). The theory of knowledge compilation also supports that learners need more than worked examples to learn to solve problems in a domain. It postulates that in order to fully develop problem-solving skills, learners need to solve problems because solving problems allows students to create rules from their knowledge. Trafton and Reiser (1993) similarly found that learners presented with interleaved examples and problems took less time on novel problems than learners presented with blocks of examples and problems. Their experiment also demonstrated that giving practice problems above the participant's learning level inhibits learning because of increased cognitive load. Therefore, practice problems must only cover material that the student has learned.

To reduce the demand on working memory when transitioning from worked examples to practice problems, one technique that can be used is scaffolding. Scaffolding can be used as an intermediate step between giving a learner worked examples and asking them to solve problems on their own; it gives the student a problem to solve and some of the components of the solution to guide his or her solution (Pea, 2004). This extra step between guided instruction and unguided instruction allows learners to develop problem-solving mental models for the domain (Kirschner et al., 2006).

1.3 Present Study

The present study employed subgoal labels, worked examples, and scaffolding to reduce intrinsic and extraneous cognitive load on novices learning to program. These techniques were expected to improve their performance on assessment tests and enable their development of mental models of programming knowledge. To assess programming knowledge, students were asked to solve problems using a drag-and-drop programming language, so students did not need in-depth knowledge of the language to solve the problems. Because drag-and-drop programming involves selecting pieces of code instead of writing pieces of code, this approach might allow students to focus on making mental models of conceptual structures rather than language syntax.

The present studies manipulated the instructional material that learners received; that is, a participant either received conventional instructional material from the projects section of ICE Distance Education Portal (http://ice.cc.gatech.edu/dl/?q=node/641) created by Barbara Ericson, or he or she received instructional material adapted to include subgoal labels. The materials were identical except for the added subgoal labels (see Figure 1). During their two sessions, participants watched a video demonstration of an application (app) being created (worked example), created the app using a text guide (scaffolding), and modified components of or added components to their apps without guidance (practice problems).

To assess their knowledge, participants were asked to write the steps that they would take to program a feature (i.e., either a component or a block) for their app. Their answers were scored based on whether or not they attempted the subgoals required to program the feature and whether or not they completed the subgoal correctly.

Subgoal-Labeled Materials

Handle Events from My Blocks

1. Click on "My Blocks" to see the blocks for components you created.
2. Click on "clap" and drag out a *when clap.Touched* block

Set Output from My Blocks

3. Click on "clapSound" and drag out *call clapSound.Play* and connect it after *when clap.Touched*

Conventional Materials

1. Click on "My Blocks" to see the blocks for components you created.
2. Click on "clap" and drag out a *when clap.Touched* block
3. Click on "clapSound" and drag out *call clapSound.Play* and connect it after *when clap.Touched*

Figure 1. Sample Materials from Two Groups

1.3.1 Development of Instructional Materials

The present study used both video demonstrations of the task (i.e., a video of someone doing the task) and text instructions for how to complete the task. Palmiter and Elkerton (1993) found that video demonstrations can quickly and naturally show users how to learn a direct-manipulation interface, but they concluded that simply watching demonstrations might lead to superficial processing of a task. While participants who viewed the video demonstrations performed tasks on the immediate test more quickly and accurately than participants who read text-only instructions, video-demonstration participants' performance on the delayed test, which was one week later, was much worse than text-only participants, whose speed and accuracy remained about the same (Palmiter & Elkerton, 1993; Palmiter, Elkerton, & Baggett, 1991). Given that video demonstrations are a useful aid in learning a complex task that uses an interface, that participants enjoy video demonstrations more than text instruction, and that text instruction leads to better transfer and retention, the present study used both methods of instruction (Palmiter & Elkerton, 1993).

For the subgoal condition, subgoal labels were incorporated into both the video demonstration and the text instruction. Given that participants in Palmiter's and Elkerton's (1993) video demonstration group reported that they felt like they were "'memorizing sequences of clicks'…without understanding the task" (p. 210), the subgoal callouts (see Figure 2 for an example) might help engage the subgoal participants during the video. The subgoal labels included in the materials were developed using the

TAPS procedure (Catrambone, Gane, Adams, Bujak, Kline, and Eiriksdottir, 2012) in consultation with subject-matter experts, Mark Guzdial and Barbara Ericson (see Figure 3 for list of subgoal labels).

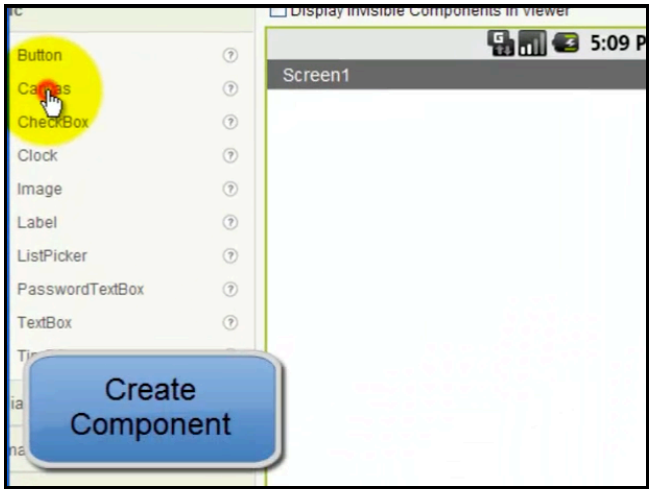

Figure 2. Sample of Subgoal Callout in Video Demonstration

Subgoals

- Create components
- Set properties
- Handle events from My Blocks
- Set outputs from My Blocks
- Define variable from Built-In
- Set conditions from Built-In
- Emulate app

Figure 3. Subgoals Used In Instructional Material

1.3.2 Development of Assessment Tasks

Assessment tasks were developed based on the material that participants were exposed to in the sessions (e.g., assessment one was based on the material taught in session one). During the assessment, participants were asked to write down the steps that they would take to create a feature (e.g., component or block) in App Inventor. Half of the assessment tasks were classified as "near transfer" tasks meaning that they followed an identical structure to tasks completed in the instructional period but substituted blocks, components, or properties of the **same** type. For example, the second task in assessment one asked participants to program the clap sound to play when the phone was tilted down. To complete this task, participants could follow the same steps that they used in the instructional period to program the drum sound to play when the phone was tilted to the right, but they had to replace the drum sound with the clap sound and the x-axis acceleration sensor with the y-axis acceleration sensor.

The other half of the assessment tasks were classified as "far transfer" tasks meaning that they followed the same general scheme as tasks completed in the instructional period but substituted blocks, components, or properties of a **different** type. For example, the third task in assessment one asked participants to program an ImageSprite to move 5 pixels to the right if touched. To complete this task, participants had to integrate steps from several tasks from the instructional period (e.g., using a "Math" block), but the subgoals that needed to be achieved to complete the assessment task were the same as the subgoals that needed to be achieved to complete tasks in the instructional period. There were no statistically significant performance differences between near and far transfer tasks either within participants or between groups.

Hints were given on assessment tasks that asked participants to use features that they had not used before. The hints directed participants to the correct feature but did not tell them how to use that feature (see Figure 4). Instructional material was not available to the participants during the assessment, but participants had access to the App Inventor interface and the apps that they had created during that session. Participants were allowed access to the apps that they had made, so the apps could serve as memory cues and reduce cognitive load.

1.5 Write the steps you would take to make the screen change colors depending on the orientation of the phone; specifically, the screen turns blue when the pitch is greater than 2 (hint: you'll need to make an orientation sensor and use blocks from "Screen 1" in My Blocks).

2.1 Write the steps you would take to add a tambourine to your Music Maker app (create the component only).

3.3 Write the steps you would take to create a list of colors and make the ball to change to a random color whenever it collided with something.

Figure 4. Sample of Assessment Tasks

2. EXPERIMENT ONE

2.1 Method

2.1.1 Participants

Participants were 40 students recruited from Georgia Institute of Technology. To participate in the experiment, students must have been at least 18 years of age, and they must not have completed more than one computer programming or computer science class. Experience with Android App Inventor disqualified students. Information about participant's age, gender, academic field of study, SAT scores, high school and college GPA, year in school, number of complete credits, computer science experience, primary language, number of math courses completed, subjective comfort with computers, and expected difficulty of learning a programming language were collected to be analyzed as possible predictors of performance (Rountree et al., 2004). None of these demographics correlated with performance except that expected difficulty of learning a programming language correlated positively with amount of time spent on assessment tasks, $r = .38$, $p = .02$.

2.1.2 Procedure

The study consisted of two one-hour sessions which were one week apart and was conducted using a computer-based learning environment (i.e., all instructional material was presented to the participants through a computer). During the instructional period for the two sessions, students learned how to create two apps in Android App Inventor using various components such as animations, sounds, and accelerometer input. Android App Inventor was chosen because it is a drag-and-drop program language. By watching videos of an app being created, creating their own apps with guidance, and modifying and adding to their

apps, participants learned how to create components in the App Inventor Designer then program the components in the App Inventor Blocks Editor.

In the first session, participants completed a demographic questionnaire, and then they had 40 minutes to study the first app's instructional material. Next, participants had 15 minutes to complete the first assessment task. In the second session, participants had 10 minutes to complete the second assessment task, which measured their retention. Then participants had 25 minutes to study the second app's instructional material followed by 25 minutes to complete the third assessment.

2.2 Results and Discussion

Each solution for the assessment tasks was deconstructed into the components (i.e., subgoals) that were necessary to successfully complete the solution; participants were given a point for each subgoal that they attempted and each subgoal that they completed correctly. Attempting a subgoal was operationally defined as listing at least one of the steps required to complete the subgoal or listing a step that would achieve a similar function (e.g., for a "set properties" subgoal, listing a step to change a property regardless if it was the correct property). There were 46 subgoals across the assessment task solutions, so participants could get a maximum score of 46 for both the attempted and correct measurements. Interrater reliability was high with a one-way random model intraclass correlation coefficient of agreement (ICC(A)) of .97, Cronbach's alpha of .98, and $r = .96$, $p < .001$. Participants were also given a score for the number of questions that they attempted (operationally defined as writing something for an answer) to account for participants who did not complete the assessments in time. Additionally, the amount of time that participants took to complete each assessment was measured.

2.2.1 Attempted Subgoals

Participants in the subgoal group ($n = 20$) attempted more subgoals ($M = 34.70$, $SD = 6.12$) than the conventional group ($n = 20$, $M = 29.42$, $SD = 7.40$), F (1, 38) = 5.91, $MSE = 45.91$, $p = .02$, $\omega^2 = .14$, $f = .38$. Furthermore, though the number of attempted questions was not correlated with group or correct subgoals, the number of attempted questions was correlated with attempted subgoals, $r = .52$, $p = .001$. Linear regression was used to test if number of attempted questions and instructional group accounted for different parts of the variance of number of attempted subgoals. In the linear regression both group and attempted questions are significant predictors of attempted subgoals, $\beta = .32$, $p = .047$, and $\beta = .38$, $p = .02$, respectively.

These statistics mean that the participant group is a significant predictor of attempted subgoals with other predictors held constant, and group uniquely accounts for 14% of the variance for attempted subgoals. Furthermore, the effect size, which represents the magnitude of the difference between the two groups in units of standard deviations, equates to subgoal participants attempting on average 2.57, or 6%, more subgoals than conventional participants. These results could mean that participants in the subgoal group can better identify the subgoals necessary to complete the solution whether or not they complete the solution correctly. If this is true, then being better able to identify the subgoals necessary for a solution could be explained by having a mental model for the computer programming information that was learned and how to solve problems in the domain. Subgoal labels can help learners create better mental models because subgoals can provide a mental model framework that could be used to organize new information more efficiently. Based on these results alone, inferences about participants' mental models cannot be made, but Experiment Two addresses this issue.

2.2.2 Correct Subgoals

Participants in the subgoal group completed more subgoals correctly ($M = 28.10$, $SD = 7.22$) than the conventional group ($M = 20.63$, $SD = 6.72$), F (1, 38) = 11.16, $MSE = 48.71$, $p = .002$, $\omega^2 = .23$, $f = .53$. These statistics mean that 23% of the variance for correct subgoals was accounted for by group, and the effect size equates to subgoal participants answering on average 3.69, or 8%, more subgoals correctly than the conventional group. These results support the hypothesis that participants in the subgoal group would perform better on the assessment tasks than those in the conventional group. This difference could be due to the subgoal participants learning the subgoals better. As described earlier, learning subgoals can reduce extraneous cognitive load by highlighting the structure of examples, promoting self-explanation, creating mental models early in the learning process, and chunking problem-solving steps (Catrambone, 1998). If extraneous cognitive load was reduced, subgoal participants could have learned more effectively than conventional participants and performed better on the assessment tasks. Though cognitive load theory would predict these results, the present study does not directly measure cognitive load, so the theoretical mechanism underlying the results cannot be definitively determined.

2.2.3 Time on Task

The subgoal group finished the assessments faster (M = 40.64 min., SD = 7.48 min.) than the conventional group (M = 45.45 min., SD = 5.11 min.) as well, F (1, 38) = 5.48, MSE = 41.07, p = .03, $\omega^2 = .13$, f = .37. Additionally, the correlation between time and number of correct subgoals was nonsignificant (r = .06, p > .05), which suggests that participants did not rush through the assessments at the cost of accuracy. These statistics mean that 13% of the variance for time spent on tasks is attributable to instructional group, and the effect size translates into the subgoal group finishing the tasks on average 2 minutes and 18 seconds faster than the conventional group. This result could suggest that participants in the subgoal group learned the subgoals more effectively, which allowed them to transfer what they learned more easily than those in the conventional group.

2.2.4 Defining the Variable Problem

The third question of assessment three asked participants to create a list, which was similar to the list that they created during the instructional period in the second session (see Figure 5). An important part of completing this task is defining the variable that contains the list because, without defining the variable, the list cannot be used in the program (e.g., other parts of the program would not be able to reference the list); that the variable happens to be a list is an incidental feature of this app. For this reason, the subgoal label used for these steps of the instructional material was "define variable." Interestingly, though the groups performed similarly for creating the list, participants in the subgoal group were more likely to define the variable in this assessment task ($M = .55$, $SD = .51$) than the conventional group ($M = .05$, $SD = .23$), F (1, 38) = 15.12, $MSE = .16$, $p < .001$, $\omega^2 = .29$, $f = .61$.

These statistics mean that 29% of the variance for correct subgoals is attributable to group, and the effect size translates into 23% more of the subgoal group defined the variable than the conventional group. This result could mean that the subgoal label helped the subgoal participants to learn the subgoal and recognize

the underlying structure of the example, which helped them transfer what they learned in the instructional period to the assessment task. That is, the subgoal label could have helped the subgoal participants recognize that defining the variable was an important task in creating the app. However, this result could also mean that the subgoal label "define variable" simply doubled the subgoal participants' exposure to the idea of defining a variable, and that the extra exposure helped them to remember those steps. That is, the subgoal label could have helped subgoal participants remember the steps to define a variable during the assessment without helping them to understand the structure of the task. Further probing of participants' problem solving strategy would be required to better determine the cause of this result.

Subgoal-Labeled Materials

Define Variables from Built-in

1. Click on "Built-In" and "Definition" and pull out a *def variable.*
2. Click on the "variable" and replace it with "fortuneList". This creates a variable called "fortuneList".
3. Click on "Lists" and drag out a *call make a list*
4. Click on "Text" and drag out a *text text* block and drop it next to "item". Click on the rightmost "text" and replace it with your first fortune.

Conventional Materials

5. Click on "Built-In" and "Definition" and pull out a *def variable.*
6. Click on the "variable" and replace it with "fortuneList". This creates a variable called "fortuneList".
7. Click on "Lists" and drag out a *call make a list*
8. Click on "Text" and drag out a *text text* block and drop it next to "item". Click on the rightmost "text" and replace it with your first fortune.

Assessment Task

Write the steps you would take to create a list of colors and make the ball to change to a random color whenever it collided with something.

Figure 5. Instructional Materials by Group and Assessment Task for which the Solution Includes Defining a Variable

2.2.5 Retention

To test retention of knowledge, participants took an assessment at the start of the second session, which was one week after the first session. During this assessment, they had access to the App Inventor website but not access to a previously created app like they did in the other assessments.Therefore, they did not have a memory cue for creating features of an app other than the website itself. There were a total of 11 subgoals in the correct solutions for this assessment. On this retention assessment, subgoal participants completed more subgoals correctly (M = 5.95, SD = 2.61) than conventional participants (M = 4.05, SD = 1.75), F (1, 38) = 7.06, MSE = 4.97, p = .01, ω^2 = .16, f = .42. These statistics mean that 16% of the variance for correct subgoals is attributable to instructional group, and the effect size equates to the subgoal participants answering on average .92, or 8%, more subgoals correctly than the conventional participants. This result suggests that participants in the subgoal group retained knowledge about App Inventor better than those in the conventional group. Similar to previous results, this result suggests that subgoal participants learned the material better than conventional participants. This difference could be due to lower extraneous cognitive load while learning, which would allow more mental resources for germane cognitive load and long-term learning.

3. EXPERIMENT TWO

3.1 Method

3.1.1 Participants

Participants were 12 students recruited from Georgia Institute of Technology. Criteria for participation were the same as for Experiment One. The same demographic information about participants that was collected in Experiment One was collected in Experiment Two. None of these demographics correlated with performance.

3.1.2 Procedure

The procedure for Experiment Two was identical to Experiment One except that while participants completed the assessment tasks, they engaged in a talk-aloud protocol. The talk-aloud protocol asked participants to explain their goals or strategies for completing the assessment tasks and also to identify the features for which they searched while working on the tasks. Before starting assessment one, participants practiced the talk-aloud procedure by playing a game of tic-tac-toe with the experimenter. During the assessment tasks, the experimenter did not provide information about how to complete a task but did provide information or instruction about the protocol (e.g., the experimenter could encourage the participant to talk more). Assessments in Experiment Two were not timed due to the talk-aloud protocol. As a result, participants had as much time as they wanted to work on assessment tasks.

3.2 Results and Discussion

In addition to scoring participant responses for attempted and correct subgoals, participants were also scored on the number of subgoal labels they used when describing their strategies and goals when solving tasks and the number of blocks they dragged out while solving assessment tasks.

3.2.1 Attempted and Correct Subgoals

Due to the small number of participants in this experiment (N=12), there was not enough power in the null-hypothesis-significance-testing framework to achieve statistically significant results. The attempted and correct subgoals were still analyzed by effect size between groups. Similar to results from Experiment One, the effect size for attempted subgoals was .42, and the effect size for correct subgoals was .59. These effect sizes suggest that if the same number of people who participated in Experiment One had participated in this experiment, the same statistically significant difference in performance between groups that were observed in Experiment One would have been observed in this experiment. Besides replicating the results from the first experiment, these results mean that without time constraints during the assessments, subgoal participants again performed better than conventional participants. This conclusion suggests that subgoal participants learned the material better perhaps because the subgoal labels allowed them to learn the subgoals better. If they learned the subgoals better, they also might have created better mental models of the material, and they might have experienced less cognitive load while learning the material.

3.2.2 Subgoal Labels in Descriptions

Participants who had subgoal labels in their instructional materials used those labels when describing their strategies and goals while solving the assessment tasks (M = 5.75, SD = 4.27). That participants used these labels during the talk aloud protocol suggests that the information that they learned is mentally organized under these labels. Organizing information under subgoal labels might help participants create better mental models and perform better than participants in the conventional group.

3.2.3 Number of Blocks

Participants in the subgoal group were less likely to drag out blocks while working through assessment tasks (M = 18.83, SD = 13.09) than those in the conventional group (M = 49.85, SD = 5.66), F (1, 10) = 9.84, MSE = 148.14, p = .02, ω^2 = .62, f = .91. These statistics mean that 62% of the variance for number of blocks was accounted for by group, and the effect size equates to subgoal participants dragging out 8.5 fewer blocks on average than the conventional group. Thus, another benefit to having their knowledge apparently organized by subgoals is that learners in the subgoal group were more efficient in their problem solving (i.e., dragging out fewer blocks).

These results could mean subgoal participants did not need as much external representation of the problem state to solve the problem suggesting that they represent the problem state more internally than the conventional group. If subgoal participants had better mental models than the conventional participants, they might have been better able to internally represent the problem state compared to the conventional participants.

4. CONCLUSION

One well-established reason that novices struggle to learn programming is because of the cognitive overload that they experience (Anderson et al., 1984). Cognitive overload not only prevents information from being stored in long-term memory, it also hinders the development of mental models (Kirschner et al., 2006). A mental model of programming knowledge, however, is a strong predictor of success in CS classes (Dehnadi et al., 2009). Furthermore, the stunted development of mental models compounds the difficulty of learning additional information about programming (Sweller et al., 2010). The purpose of the present study was to employ techniques to reduce cognitive load and to promote the creation of mental models and test whether or not those techniques improved performance on assessments of programming knowledge.

The results of the present study could support that subgoal-labeled materials help novices learn subgoals, which reduces the extraneous cognitive load imposed on novices learning programming. Learning subgoals could have reduced extraneous cognitive load in a few ways. It could have reduced extraneous cognitive load by highlighting the essential features of worked examples, by chunking problem-solving steps, and by promoting self explanation (Catrambone, 1998; Sweller, 2010). This reduction in extraneous cognitive load might have allowed students to learn faster because more of their mental resources were available for germane cognitive load which is responsible for creating mental models and storing information in long-term memory (Kirschner et al., 2006). Furthermore, subgoals emphasize the structure of solutions, which aids the development of mental models (Atkinson et al., 2000).

The results could also support that the subgoal labels aided novices in developing mental models early in the learning process. In addition to reducing cognitive load, subgoal labels are a type of guided instruction that could give learners a framework for a rational mental model that they could have filled in with information that they learned (Kirschner et al., 2006). In turn, mental models reduce cognitive load required to process new information, which increases long-term learning, and long-term learning reduces the cognitive load required to process new information on the same topic (Kirschner et al., 2006). More research is still needed to understand the connection between subgoal-labeled materials and mental models in CS.

Many students view CS classes as difficult and frustrating, so they avoid taking them, even though the knowledge could be beneficial to them as the prevalence of technology increases (Clancy & Linn, 1990). A major goal for CS education researchers is to dispel this stigma associated with CS classes (Kolodner et al., 2008). A key idea in this paper is that *instructional design matters*. The two groups did not differ in the *content* of the instruction but in the *design* of that material (e.g., whether subgoals were made explicit). The two groups performed significantly differently, with the subgoal group performing better on several measures. We believe that CS learning was enhanced through the application of instructional design principles. Thus, instructional design principles can be useful in achieving our goals as CS education researchers and CS educators to help more students to gain knowledge about computing.

5. ACKNOWLEDGMENTS

Our thanks to the NSF for grant CNS-1138378 and to the GVU Center for the grant that made this research possible.

Our thanks to Joe Dagosta, Hannah Fletcher, and Catherine Hwang for help collecting and scoring data.

6. REFERENCES

[1] Anderson, J. Farrell, R. & Sauers, R. 1984. Learning to program in LISP. *Cognitive Science, 8*(2), 87-129. DOI= 10.1207/s15516709cog0802_1

[2] Atkinson, R. K., & Derry, S. 2000. Computer-based Examples Designed to Encourage Optimal Example Processing: A Study Examining the Impact of Sequentially Presented, Subgoal-oriented Worked Examples. *Proceedings of ICLS 2000 International Conference of the Learning Sciences*, 132-133.

[3] Atkinson, R. K., Derry, S. J., Renkl, A., & Wortham, D. 2000. Learning from Examples: Instructional Principles from the Worked Examples Research. *Review of the Educational Research, 70*(2), 181-214. DOI= 10.3102/00346543070002181

[4] Bielaczyc, K., Pirolli, P., & Brown, A. L. 1995. Training in self-explanation and self-regulation strategies: Investigating the effects of knowledge acquisition activities on problem solving. *Cognition and Instruction, 13*, 221-252.

[5] Brennan, K. 2009. Scratch-Ed: an online community for scratch educators. In *Proceedings of the 9th international conference on Computer supported collaborative learning - Volume 2* (CSCL'09), Angelique Dimitracopoulou, Claire O'Malley, Daniel Suthers, and Peter Reimann (Eds.), Vol. 2. International Society of the Learning Sciences 76-78.

[6] Brooks, R. 1990. Categories of programming knowledge and their application. *Int. J. Man-Mach. Stud.* 33, 3 (August 1990), 241-246. DOI=10.1016/S0020-7373(05)80118-X http://dx.doi.org/10.1016/S0020-7373(05)80118-X.

[7] Catrambone, R. 1994. Improving examples to improve transfer to novel problems. *Memory and Cognition, 22*, 605-615.

[8] Catrambone, R. 1996. Generalizing solution procedures learned from examples. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 22*, 1020-1031. DOI= 10.1037/0278-7393.22.4.1020

[9] Catrambone, R. 1998. The subgoal learning model: Creating better examples so that students can solve novel problems. *Journal of Experimental Psychology: General, 127,* 355-376. DOI= 10.1037/0096-3445.127.4.355

[10] Catrambone, R., Gane, B. D., Adams, A. E., Bujak, K. R., Kline, K. A., & Eiriksdottir, E. 2012. Task Analysis by Problem Solving (TAPS): A Method for Uncovering Expert Knowledge. Unpublished manuscript, School of Psychology, Georgia Institute of Technology, Atlanta, GA.

[11] Clancy, M.J., & Linn, M.C. 1990. Functional fun. In *Proceedings of the twenty-first SIGCSE technical symposium on Computer science education* (SIGCSE '90), James E. Miller and Daniel T. Joyce (Eds.). ACM, New York, NY, USA, 63-67. DOI=10.1145/323410.319085 http://doi.acm.org/10.1145/323410.319085

[12] Davis, E., Linn, M., Mann, L., & Clancy, M. 1993. Mind your Ps and Qs: Using parentheses and quotes in LISP. In C. R. Cook, J. C. Scholtz, and J. C. Spohrer (Eds.), Empirical Studies of Programmers: Fifth Workshop, (pp. 62-85). Norwood, NJ: Ablex. 1993.

[13] Dehnadi, S., Bornat, R., & Adams, R. 2009. Meta-analysis of the effect of consistency on success in early learning of programming. *21st Annual Workshop of the Psychology of Programming Interest Group* (p. 10pp)

[14] Du Boulay, B. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, *2*(1), 57-73.

[15] Guzdial, M. 1995. Centralized mindset: a student problem with object-oriented programming. In *Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education* (SIGCSE '95), Curt M. White, James E. Miller, and Judy Gersting (Eds.). ACM, New York, NY, USA, 182-185. DOI=10.1145/199688.199772 http://doi.acm.org/10.1145/199688.199772

[16] Kirschner, P., Sweller, J., & Clark, R. 2006. Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational Psychologist, 41*(2), 75-86. DOI= 10.1207/s15326985ep4102_1

[17] Linn, M., & Clancy, M. (1992). The case for case studies of programming problems. *Communications of the ACM, 35*(3), pp 121-132.

[18] Norman, D. A. (1983). Some observations on mental models. In D. Gentner & A. Stevens (Eds.), *Mental models* (pp. 7-14). Retrieved from http://books.google.com/

[19] Palmiter, S., Elkerton, J., & Baggett, P. 1993. Animated demonstrations versus written instructions for learning procedural tasks: A preliminary investigation. *International Journal of Man-Machine Studies, 34,* 687-701. DOI= 10.1016/0020-7373(91)90019-4

[20] Pea, R. 2004. The social and technological dimensions of scaffolding and related theoretical concepts for learning, education, and human activity. *Journal of the Learning Sciences*, *13*(3), 423-451. DOI= 10.1207/s15327809jls1303_6

[21] Renkl, A., & Atkinson, R. K. 2002. Learning from examples: Fostering self-explanations in computer-based learning environments. *Interactive Learning Environments, 10*(2), 105-199.

[22] Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., & Kafai, Y. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (November 2009), 60-67. DOI=10.1145/1592761.1592779 http://doi.acm.org/10.1145/1592761.1592779

[23] Rountree, N., Rountree, J., Robins, A., & Hannah, R. 2004. Interacting factors that predict success and failure in a CS1 course. In *Working group reports from ITiCSE on Innovation and technology in computer science education* (ITiCSE-WGR '04). ACM, New York, NY, USA, 101-104. DOI=10.1145/1044550.1041669 http://doi.acm.org/10.1145/1044550.1041669

[24] Sweller, J. 2010. Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational Psychology Review*, *22*(2), 123-138. DOI= 10.1007/s10648-010-9128-5

[25] Trafton, J. G., & Reiser, B. J. 1993. The contributions of studying examples and solving problems to skill acquisition. In *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society* (pp. 1017-1022). Boulder, CO.

[26] van Merriënboer, J., Clark, R., & de Croock, M. 2002. Blueprints for complex learning: The 4C/ID-model. *Educational Technology Research and Development*, *50*(2), 39-61. DOI= 10.1.1.113.8484

[27] Wilson, B., & Shrock, S. 2001. Contributing to success in an introductory computer science course: A study of twelve factors. *SIGCSE Bull.* 33, 1 (February 2001), 184-188. DOI=10.1145/366413.364581 http://doi.acm.org/10.1145/366413.364581