Interaction of Instructional Material Order and Subgoal Labels on Learning in Programming

A Thesis
Presented to
The Academic Faculty

by

Laura M. Schaeffer

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Psychology

Georgia Institute of Technology
December 2015

COPYRIGHT © 2002 BY LAURA M. SCHAEFFER

Interaction of Instructional Material Order and Subgoal Labels on Learning in Programming

Approved by:

Dr. Richard Catrambone, Advisor
School of Psychology
Georgia Institute of Technology

Dr. Frank Durso
School of Psychology
Georgia Institute of Technology

Dr. Wendy Rogers
School of Psychology
Georgia Institute of Technology

Date Approved: December 2, 2015

TABLE OF CONTENTS

Page

LIST OF TABLES …… v

LIST OF FIGURES …… vi

SUMMARY …… vii

CHAPTER

1 INTRODUCTION …… 1

Improving Transfer Through Subgoal Labeling …… 2

Expository Instructions …… 2

Worked Examples …… 3

Present Study …… 7

Hypotheses …… 8

2 METHOD …… 9

Participants …… 9

Design …… 9

Materials …… 10

Procedure …… 14

3 RESULTS …… 15

Demographics …… 15

Cognitive Load Assessment …… 16

Performance Assessments …… 20

4 DISCUSSION …… 35

Further Work …… 37

APPENDIX A: SUBGOAL LABELED EXPOSITORY SCRIPT …………………38

APPENDIX B: NON-SUBGOAL LABELED EXPOSITORY SCRIPT …………...41

APPENDIX C: SUBGOAL LABELED WORKED EXAMPLE SCRIPT …………43

APPENDIX D: NON-SUBGOAL LABELED WORKED EXAMPLE SCRIPT…….46

APPENDIX E: SUBGOAL LABELED PRACTICE PROBLEM GUIDE …………48

APPENDIX F: NON-SUBGOAL LABELED PRACTICE PROBLEM GUIDE…….52

APPENDIX G: ASSESSMENT ONE…………………………………………………55

APPENDIX H: ASSESSMENT TWO, PART ONE……………………………….56

APPENDIX I: ASSESSMENT TWO, PART TWO………………………………57

APPENDIX J: ASSESSMENT THREE …………………………………………….58

APPENDIX K: ASSESSMENT FOUR ………………………………………………60

REFERENCES……………………………………………………………………….61

LIST OF TABLES

Page

Table 1: Predictors of Performance ……………………………………………………….16

Table 2: Descriptive Statistics for Cognitive Load Assessment ……………………….18

Table 3: Descriptive Statistics for Problem Solving Task ……………………………...22

Table 4: Descriptive Statistics for Explanation Task …………………………………...28

Table 5: Descriptive Statistics for General Procedure Task ……………………………30

Table 6: Descriptive Statistics for Time on Instructional and Assessment Period …….32

LIST OF FIGURES

Page

Figure 1: Puzzle Piece Style Code Used to Program Features in App Inventor…………7

Figure 2: Subgoal "Set Properties" Presented as a Callout in App Inventor Video.........11

Figure 3: Mean Scores and Standard Deviations for Intrinsic Cognitive Load…………17

Figure 4: Means Scores and Standard Deviations for Extraneous Cognitive Load…….19

Figure 5: Mean Scores and Standard Deviations for Germane Cognitive Load………..20

Figure 6: Mean Scores and Standard Deviations for the Problem Solving Task……….21

Figure 7: Distribution of Scores for Problem Solving Task: Performance……………..23

Figure 8: Mean Scores and Standard Deviations for Problem Solving Task: Written …24

Figure 9: Mean Scores and Standard Deviations for Problem Solving Task: Subgoals..26

Figure 10: Mean Scores and Standard Deviations for Grouping Structurally Similar Steps in the Explanation Task………………………………………………28

Figure 11: Mean Scores and Standard Deviations for Functional Labels in the Explanation Task…………………………………………………………..29

Figure 12: A Significant Interaction Between Subgoal Labeling and Instructional Order was Present in the General Procedure Task …………………………31

Figure 13: Mean and Standard Deviations of Time Spent on Instructional Period …….32

Figure 14: Mean and Standard Deviations of Time on Assessments…………………...33

SUMMARY

Expository instructions, worked examples, and subgoal labels have all been shown to positively impact student learning and performance in computer science education. This study examined whether learning and problem solving performance differed based on the sequence of the instructional materials (expository and worked examples) and the presence of subgoal labels within the instructional materials. Participants were 138 undergraduate college students, age 17-25, who watched two instructional videos on creating an application in the App Inventor programming language before completing several learning assessments. A significant interaction showed that when learners were presented with the worked example followed by the expository instructions containing subgoal labels, the learner was better at outlining the procedure for creating an application. These manipulations did not affect cognitive load, novel problem solving performance, explanations of solutions, or the amount of time spent on instructions and completing the assessments. These results suggest that the order instructional materials are presented have has little impact on problem solving, although some benefit can be gained from presenting the worked example before the expository instructions when subgoal labels are included. This suggests the order the instructions are presented to learners does not impact learning. Previous studies demonstrating an effect of subgoal labels used text instructions as opposed to the video instructions used in the present study. Future research should investigate how these manipulations differ for text instructions and video instructions.

CHAPTER 1

INTRODUCTION

In order for America to be competitive in our modern global economy, comprehension of science, technology, engineering, and mathematics (STEM) is crucial. Over the last 50 years, half of the U.S. gross domestic product's growth can be attributed to advancements in STEM fields (Beatty, 2011). Today's students need to develop their skills to a level much higher than what was deemed necessary for yesterday's society (National Science Foundation [NSF], 2007). Currently, STEM education in the United States is falling short of desired standards (NSF, 2007). Specifically, employers often struggle to find applicants with the desired computer and problem solving skills needed to fill positions (Committee on Highly Successful Schools or Programs in K-12 STEM Education, 2011). The NSF (2007) recommended improving STEM education to address the deficit of high quality STEM specialists, as well as to increase understanding of STEM subjects in the general population.

Unlike other STEM subjects that have been studied for many years, such as mathematics or physics, computing education research is a comparatively young field. We know relatively little about effectively teaching programming skills. For example, we do not know how mental models differ between learners who successfully grasp difficult computing concepts versus the mental models of those who do not, let alone how to help students develop their mental models (Cooper, Grover, Guzdial, & Simon, 2015). Over the last decade, the NSF has sponsored several efforts to encourage research in computing education. One of those efforts was a panel that identified the novice's ability to transfer what they have learned as a key area for further research (Cooper et al., 2015). Transfer is an indicator of successfully grasping difficult computing concepts, and promoting transfer is a goal in programming education.

Improving Transfer Through Subgoal Labeling

Learners have difficulty solving novel problems, or problems that require steps that are different from worked example problems they have already encountered (Catrambone, 1995; Reed, Dempster, & Ettinger, 1985; Ross, 1987, 1989). This difficulty stems from learners tending to fixate on superficial aspects of examples as opposed to the goal structure of the problem. When learners understand the goal structure of the example problems, they become more successful at solving novel problems (e.g., Catrambone, 1995).

Subgoals are part of the task structure and organize solution steps into a meaningful hierarchy; subgoals are specific to problems within a particular domain (Catrambone, 1994; Catrambone, 1998). Subgoal labels assist learners in noticing and learning the subgoals and organizing their problem solving knowledge. This organization is demonstrated when learners who received instructions with subgoal labels tended to explain their problem solutions using the subgoals (Catrambone, 1995a; Margulieux, 2013). Subgoal labels within instructions have improved transfer in many domains, including computer programming, and have been shown to be most effective when provided in both expository instructions and worked examples (Margulieux, 2013).

Expository Instructions

Expository instructions usually consist of both declarative information, such as terminology, and procedural information (Trafton & Reiser, 1993). Procedural instructions describe and explain how to carry out a task (Eiriksdottir & Catrambone, 2011).Procedural instructions are often written at a more general level than worked examples, so they can be applied to a variety of situations. The learner is equipped with the high level concepts needed to solve novel problems within the domain (Catrambone, 1990). This allows students who master general procedural instructions to be able to solve novel problems better than students who receive more specific instructions

(Catrambone, 1990). However, because general procedural instructions do not have the same level of detail as more specific instructions, such as a worked example, more detailed information must be inferred. Regardless of the exact difficulty of the inference, it is still more difficult to infer details than to have the details stated.

Worked Examples

Worked examples demonstrate how a specific instance of a task is performed (Eiriksdottir & Catrambone, 2011). Worked examples provide a concrete application of the problem solution's abstract concepts, rules, and general directions (Charney & Reder, 1987; Pirolli & Recker, 1994; Wiedenbeck, 1989). Worked examples are generally structured as a problem statement followed by the steps needed to arrive at the solution. This allows the learner to become familiar with the task and increase their understanding of how to carry out the task (Eiriksdottir & Catrambone, 2011). Because worked examples provide detailed information, learners are able to more easily apply the same procedure to a similar problem than if they had been given more abstract information (Catrambone, 1990). Learners who use worked examples have also been shown to perform similar tasks more quickly than learners who used only procedural instructions (Catrambone, 1990).

One drawback of typical worked examples is that they do not inherently provide the learner with any general methods or reasoning behind decisions (Eiriksdottir & Catrambone, 2011). When given a worked example, the learner must infer information such as the nature of the task, the purpose of each step, rules governing the steps, subgoals, and organization (LeFevre & Dixon, 1986; Pirolli & Recker, 1994). In limited cases learners have been shown to infer general methods when several worked examples are presented, but usually guidance is needed for such connections to be made (Rumelhart & Norman, 1981).

Presenting the learner with both procedural instructions and worked examples has been shown to produce the benefits associated with each type of instructional material while reducing the drawbacks. Catrambone (1995) showed that presenting procedural text with a worked example aided both initial performance and transfer.

There is reason to believe the order in which the instructions are presented might affect the learner's ability to process them. Several lines of research suggest that students perform and learn better when given a worked example followed by procedural texts (Alfieri, Nokes-Malach, & Schunn, 2013; Anderson, 1990; Dale, 1946). Dale (1946) argued that when learning math, students should first be introduced to concrete objects (e.g., five fingers as opposed to an abstract five), and then work up semi-concrete ideas. If the material does not relate to a student's experience with the items in the equation, the formula will not mean anything (Dale, 1946). Dale (1946) concluded that the role of the teacher is to take the child from concrete experiences to significant and important generalizations. Other studies also suggest that it is better to give people principles for the concept or procedure that they are trying to learn after they view the cases (Alfieri, Nokes-Malach, & Schunn, 2013).

Another theory, from the inductive teaching research literature, suggests that worked examples provide the "why" behind the principles and procedure (Prince & Felder, 2006). The specifics from worked examples cause the learner to generate a need for more information, such as the rules, procedures, and principles. This curiosity then motivates the learner to incorporate and apply the instructions.

It has been noted that new information is best learned when the learner has a knowledge base to support the information, and they are unlikely to learn if the new information has few apparent connections to what they already know. Advance organizers have been used to provide such a foundation (Ausubel, 1968; Novak, 1977). Advance organizers can be used as an effective way to bridge the gap between the novice's knowledge and the basis on which the instructions function (Ausubel, 1968).

Ausubel (1968) proposed three ways in which advance organizers facilitate learning. First, advance organizers activate the learner's prior knowledge making the new information more familiar and meaningful. Additionally, "advance organizers at an appropriate level of inclusiveness, by making subsumption under specifically relevant propositions possible (and drawing on other advantages of subsumptive learning), provide optimal anchorage," (Ausubel, 1968, p. 137). Ausubel goes on to state that this facilitates both initial learning as well as making the information resistant to obliterative subsumption. Obliterative subsumption occurs when specific details of information are lost because they were not firmly anchored in the cognitive structure. For example, no longer being able to differentiate between a timber wolf and a gray wolf, although at one point able to do so shows the distinction between the two types of wolves has been obliterated.

Finally, advance organizers decrease dependence on sheer memorization in favor of a meaningful understanding of the information. When information is not meaningful, the learner is able to only memorize procedures without fully understanding the ideas. Memorization of a procedure without understanding the procedure reduces the chances of the learner solving novel problems within the domain.

A worked example might serve as an advance organizer because it gives the learner a base on which to apply the latter expository information. A worked example introduces the learner to the type of situation to which the expository information is applicable, mobilizing the learner's prior knowledge. Therefore, instructional materials might be more effective if the worked example is presented before the expository information.

However, according to Ausubel (1968), instructions aid mental organization better when progressing from abstract ideas to specific details. Information is organized with the abstract ideas and concepts at the top levels of our cognitive structure and specific information at the lower levels. Therefore, Ausubel (1968) argued that the

sequence used to teach new information should mimic the cognitive structure, presenting abstract ideas first and then proceeding to the specific information. This argument for organization would support giving students expository instructions, describing the general ideas of the task, before the worked example, which demonstrates a specific application of the abstract principles.

Additionally, presenting specific details first, such as those found in the worked example, might cause the learner to focus on applying the expository instructions to problems that are very similar to the worked example. Consequently, the learner might have a more difficult time generalizing the instructions to other situations. Because of this, presenting the worked example first might hinder the learner's ability to use the abstract principles when solving novel problems. However, subgoal labels might help learners compensate for this effect because they explicitly provide the higher level functions found within the worked example and the expository instructions.

In summary, subgoal labels have been shown to improve problem solving and increase transfer to novel problems because they provide a framework for solving problems, help create mental representations, and reduce demands on working memory. Advance organizers bridge the gap between what learners already know, and what they need to know for the task at hand. Additionally, advance organizers anchor the concepts to previously formed cognitive structures. Worked examples might function as an advance organizer because they can aid the learner in understanding what they know and what they do not know. The concrete nature of the worked example might be able to anchor the concepts to cognitive structures. Research has also shown that a learner must understand concrete concepts before they can understand abstract concepts. Therefore, presenting the worked example before the expository instructions might result in better problem solving than when the expository instructions are presented first, and including subgoal labels within the instructions could further improve problems solving. This relationship has not yet been investigated.

Present Study

The present study investigated the effect of instructional material order and subgoal labels in learning computer programming. Participants were taught how to use the programming language Android App Inventor to create a Fortune Teller application (app). The App Inventor programming environment uses a drag-and-drop interface to create apps for Android devices (see Figure 1).

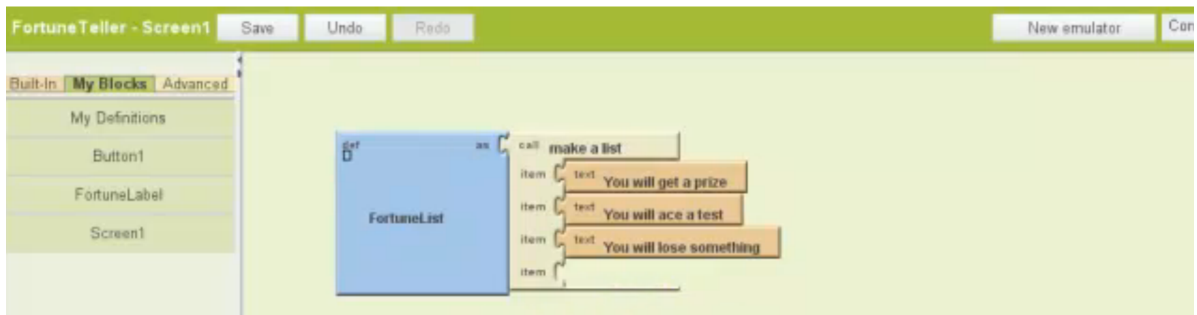


Figure 1. Puzzle piece style code is used to program features in App Inventor.

Drag-and-drop programming is ideal for novices because instead of writing code, the learners drag components from a menu and fashion them together like puzzle pieces. Creating code in this way has been shown to be easier for novices to comprehend than other types of programming environments (Hundhausen, Farley, & Brown, 2009). Presumably, the video format reduces extraneous cognitive load because the video presents information in both the auditory and visual channels, as opposed to overloading the visual channel when all the information is presented as text. Additionally, the video demonstrates the auditory narration, which reduces the need for the learner to scan the interface when trying to align the text instructions to the App Inventor interface. This has

the additional advantage of allowing the manipulations to affect performance in a controlled experimental session.

Videos were used to convey the App Inventor instructions because videos have been shown to be a natural and efficient way for learners to gain knowledge of direct-manipulation interfaces (Palmiter & Elkerton, 1993; Palmiter, Elkerton, & Baggett, 1991). Participants also used a practice problem guide to practice creating the Fortune Teller app before being tested. Trafton and Reiser (1993) showed that learners who study and practice newly learned material are better able to apply the material than learners who are not given the opportunity to practice.

Hypotheses

1. A main effect of instructional order: Instructional materials that present the worked example before the expository instructions would lead to better performance than instructional materials that present the expository instructions before the worked example.
2. A main effect of subgoal labeling: Instructional materials with subgoal labels would lead to better performance than instructional materials without subgoal labels.
3. An interaction between instructional order and subgoal labeling: Instructional materials with subgoal labels presented with the worked example first followed by the expository instructions will perform better than all other groups.

These hypotheses apply to all dependent variables because the dependent variables all measure different aspects the mental representations and organization of the learned material. The exceptions to this are the hypotheses about the different cognitive load measures, which are discussed more thoroughly in the results section.

CHAPTER 2

METHOD

Participants

Participants were 138 undergraduate students from the Georgia Institute of Technology recruited through Sona and compensated with course credit. The sample consisted of 71 females, 66 males, and one participant who did not indicate gender. The mean age was 19.3 years with a standard deviation of 1.93. Participants were screened based on their experience with computer science and familiarity with App Inventor. These qualifications were necessary because the instructional materials were designed for novices. Novices were defined as having taken no more than one computer science course and not having previous experience with App Inventor. This definition was based on Margulieux, Guzdial, and Catrambone's (2012) study that used similar materials and found there was not a statistically significant difference in performance between participants who had not taken any computer science courses compared to participants who had taken one course.

Design

The experiment was a two-by-two, between subjects, factorial design. The first independent variable was the order subjects received the instructional materials: expository followed by worked example or worked example followed by expository. The second independent variable was presence of subgoal labels: present or absent. The dependent variables consisted of performance on a cognitive load task, time (measured in seconds) spent on the instructional materials, performance on three assessment tasks (to

determine organization of domain knowledge and problem solving performance), and time spent on each task.

Materials

The materials comprised the following: demographic questionnaire, instructional materials (expository instructional video, worked example instructional video, and practice problem guide), and assessment materials (cognitive load assessment, problem solving task, explanation task, and generalization task). The demographic questionnaire gathered information on predictors of performance in computer science, including age, gender, computer science experience, and comfort with computers (Rountree, Rountree, Robins, & Hannah, 2004).

The expository instructional videos (see Appendices A and B) contained general procedural instructions and declarative information, such as definitions, necessary for creating an app in App Inventor. The worked example video (see Appendices C and D) demonstrated how to create a specific app, the Fortune Teller app. App Inventor tutorials provided by the ICE Distance Education Portal (Ericson, 2012) were used as the foundation for the instructional materials in the study. Subgoals were created by Margulieux (2013) using the Task Analysis by Problem Solving (TAPS) method developed by Catrambone et al., (2012).

When applicable, the videos used callouts to present the subgoal labels (see Figure 2).

Figure 2. The subgoal "Set Properties" presented as a callout in App Inventor video demonstration.

These were text boxes containing the subgoal labels appearing on screen while the narration continues explaining the steps needed to achieve the subgoal. For example, one aspect of creating an app is setting the outputs, or behaviors, that the app will have. As demonstrated in Appendix A, the subgoal labeled expository instructional video includes a subgoal callout containing the text "Set outputs from My Blocks" that represents the purpose of the following voice over:

> "Similarly, to programming the feature, you'll also need to define what output you want. These outputs will almost always come from My Blocks. From the previous example, if you want to display the text on a label, then you'll need to add the block 'set label.text' to the 'when button.click' block."

As shown in Appendix B, the non-subgoal labeled expository instructional video is identical to the above excerpt, but does not include the subgoal label callout “Set outputs from My Blocks.”

The final instructional material was the practice problem guide, which was a scaffolded worked example (see Appendices E and F). Scaffolding is used as a transitional step between worked examples and independent problem solving (Paas & van Gog, 2009). The stages of scaffolding can vary (Pea, 2004), but in the present study, the practice problem guide provided learners with the steps necessary for creating the Fortune Teller app without giving them guidance on how to carry out the steps (e.g., where in the menus to find blocks). Examples demonstrating how to create an app are inherently lengthy and complex. Therefore, the scaffolded example used the same Fortune Teller app in the worked example video. This allowed the participant to learn information quickly while allowing session times to be kept short to reduce cognitive fatigue.

Multiple assessments were used to investigate the effects of the instructional manipulations. The first assessment was a cognitive load survey (see Appendix G). The cognitive load survey was adapted from the materials created by Leppink et al. (2013). The survey was originally used to measure cognitive load of materials presented in a statistics class, but Leppink et al. (2013) stated the survey is not restricted for use within a particular domain and minor changes in terminology could make the survey applicable to other fields. Therefore, in questions 2 and 9, the word “formulas” was changed to “procedures.” In question 8, the word “statistics” was changed to “programming” in order to make the survey terminology applicable to the present study. The 10-item questionnaire measures intrinsic load, which is demand on working memory due to the complexity of the task (questions 1, 2, and 3), extraneous load, which is unnecessary demand on working memory due to the design of the materials (questions 4, 5, and 6)

and germane load, which is the demand on working memory necessary to learning the material (questions 7, 8, 9, and 10).

The second assessment consisted of four problem solving tasks (see Appendix H) where participants were instructed to add or modify features of their Fortune Teller app. This assessment measured participants' problem solving performance on novel tasks using App Inventor. First, the participants solved the problems using the App Inventor interface. Then, they were asked to write the necessary steps used to accomplish the tasks (see Appendix I). Writing the steps allowed the participant to demonstrate declarative knowledge that they might not have been able to apply using the interface. For example, when attempting to solve the problems in the interface, a participant who knew the second step of a prompt but did not know the first step (upon which the second step was dependent) would not have been able to demonstrate that they knew the second step. However, during the writing portion, the participant would have then been able to write down step two even if they did not know step one. Therefore, the writing portion of the assessment measured declarative knowledge that the participant might not have necessarily been able to use procedurally.

The third assessment was the explanation task (see Appendix J). Correct solutions to the four problem solving tasks were given to the participants. Participants were asked to group steps of the problem solving task solution. They were then asked to label their groups by describing what goal was met for each grouping. This assessment measured how well participants could group steps based on structural similarity, and how well they could explain the solutions.

The fourth and final assessment was the generalization task (see Appendix K). The generalization task asked participants to describe the general procedure that they would use to create an app within a given set of constraints. A correct response to this task included the fundamental steps needed to make the app while excluding unnecessary

details. This assessment was used to measure how well the participants could use abstract principles to outline the task procedure they learned earlier in the session.

Procedure

Each session lasted between 60 and 90 minutes. Participants were randomly assigned to one of four conditions. All participants then completed the demographic questionnaire. Participants then began the instructional period where they watched both instructional videos (the expository video and the worked example video) before using the practice problem guide to practice creating an app.

After the instructional period, the participants began the assessment period. During the assessment period, the participants were not able to use the materials from the instructional period. However, they were able to use the App Inventor website and refer to the app they created during the instructional period as an aid to problem solving (Margulieux, 2013). Participants then completed the cognitive load assessment followed by the problem solving assessments. Next, they completed the explanation task. Finally, they completed the generalization task. Participants were then debriefed and thanked for their time.

CHAPTER 3

RESULTS

Demographics

Answers to the demographic questionnaire were analyzed for predictors of performance on the assessments. Expected ease of learning programming (on a scale of 1-7) was correlated with spending more time on the written problem solving task, r (136) = .20, p =.02, which means that participants who thought learning programming would be easier also tended to spend more time on the written problem solving task. SAT verbal scores correlated with performance on the written problem solving task, r (87) = .23, p =.03, but this was true only for participants who received the worked example before the expository instructions (see Table 1). This finding suggests that the presenting the worked example before the expository instructions might particularly benefit students with high verbal skills.

Table 1. *Predictors of Performance*

	Worked Example First		*Expository First*	
	Subgoals	*No labels*	*Subgoals*	*No labels*
Expected Ease				
M	3.63	3.85	3.60	3.65
SD	1.26	1.31	1.48	1.50
r	.14	.21	.23	.20
p	.43	.25	.20	.28
SAT Verbal				
M	686.50	665.60	664.40	677.37
SD	68.62	66.71	62.72	85.37
r	.48	.41	.23	-.20
p	.04	.04	.28	.41

Note: Expected ease of learning computer programming on a 7-pt. scale (1-Very Difficult and 7-Very Easy).

Cognitive Load Assessment

The cognitive load questionnaire assessed the amount of mental effort that participants experienced during the instructions. Intrinsic cognitive load is the complexity of the information that is innate to the material (Paas, Renkl, & Sweller, 2003). This study did not manipulate intrinsic complexity, and therefore a significant difference among conditions was not expected. There were no significant differences among the groups for intrinsic load (see Figure 3).

Figure 3. Mean scores and standard deviations for Intrinsic Cognitive Load Assessment. No significant differences were found among the groups.

The main effect of instructional material order was not significant, which means the order in which the materials were presented did not effect intrinsic load, F (1, 128) = 1.99, p =.16. The main effect of subgoal labels was also not significant, which means the presence or absence of subgoal labels within the instructions did not effect intrinsic load, F (1, 128) = 0.77, p = .38. The interaction between instructional order and subgoal labels was not significant, F (1, 128) = 0.12, p = .73 (see Table 2).

Table 2. *Descriptive Statistics for Cognitive Load Assessment*

	Worked Example First		*Expository First*	
	Subgoals	*No labels*	*Subgoals*	*No labels*
Cognitive Load	*M (SD)*	*M (SD)*	*M (SD)*	*M (SD)*
Intrinsic	1.81 (1.36)	2.13 (1.68)	1.53 (1.53)	1.67 (1.51)
Extraneous	2.90 (1.91)	2.98 (2.07)	2.70 (2.18)	3.03 (2.21)
Germane	5.32 (2.38)	4.42 (2.19)	5.68 (2.39)	5.25 (1.92)

Note: Scores out of a maximum 10 points.

Extraneous load refers to features of instructions that hinder learning (Paas et al., 2003). In the present study, instructions were the same with the exception of additional functional explanations (i.e. subgoal labels) that were expected to benefit learning. The extraneous information in the instructions was the same among the conditions, and therefore differences in extraneous cognitive load were not expected. Therefore, hindrance to schema acquisition was not expected, so differences in cognitive load were not expected. Results supported this hypothesis, and no differences were found among groups for extraneous load (see Figure 4): main effect of instructional material order, F (1, 128) = 0.04, p = .84; main effect of subgoal labels, F (1, 128) = 0.32, p = .57; interaction, F (1, 128) = 0.12, p = .73.

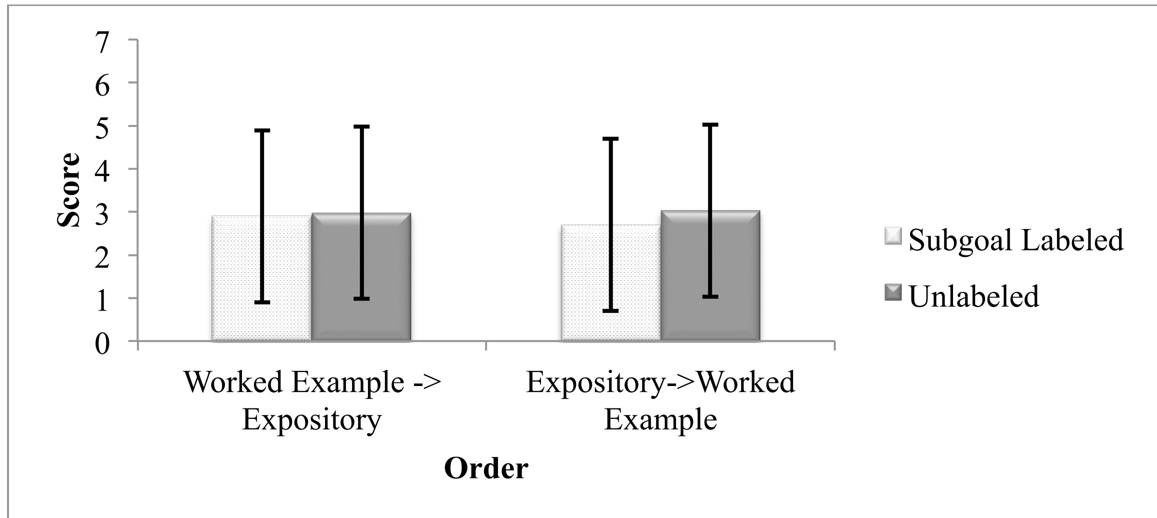


Figure 4. Mean scores and standard deviations for Extraneous Cognitive Load Assessment. No significant differences were found among the groups.

Germane load refers to working memory resources that are being used to enhance learning (Paas et al., 2003). Subgoal labels and presenting the worked example before the expository instructions were both hypothesized to focus working memory resources on important details of the worked example, which could be reflected in the germane cognitive load score. However, this hypothesis was not supported (see Figure 5).

Figure 5. Mean scores and standard deviations for Germane Cognitive Load Assessment. No significant differences were found among the groups.

No significant differences were found among the groups for germane load: main effect of instructional material order, $F(1, 125) = 0.01$, $p = .93$; main effect of subgoal labels, $F(1, 125) = 0.21$, $p = .65$; interaction, $F(1, 125) = 0.16$, $p = .69$. In summary, the order of instructional materials and subgoal labeling did not impact cognitive load.

Performance Assessments

The following assessments were scored following the method developed by Margulieux et al. (2012), which has been shown to have high statistical power (due to partial scoring methods discussed later) and high interrater reliability. Two raters scored each of the assessments; interrater reliability was measured with an intraclass correlation coefficient of absolute agreement (ICC(A)).

Problem solving tasks

Performance in App Inventor

For this task, participants were asked to modify or add different features of an app and were scored by awarding one point for each correct action in App Inventor taken towards the problem solutions for up to a maximum score of 22. ICC(A) for this assessment was .89. There were no significant differences found among the groups when modifying and adding features in App Inventor (see Figure 6): main effect of instructional material order, F (1, 124) = 0.04, p = .84; main effect of subgoal labels, F (1, 124) = 0.32, p = .57; interaction, F (1, 124) = 0.08, p = .77 (see Table 3).

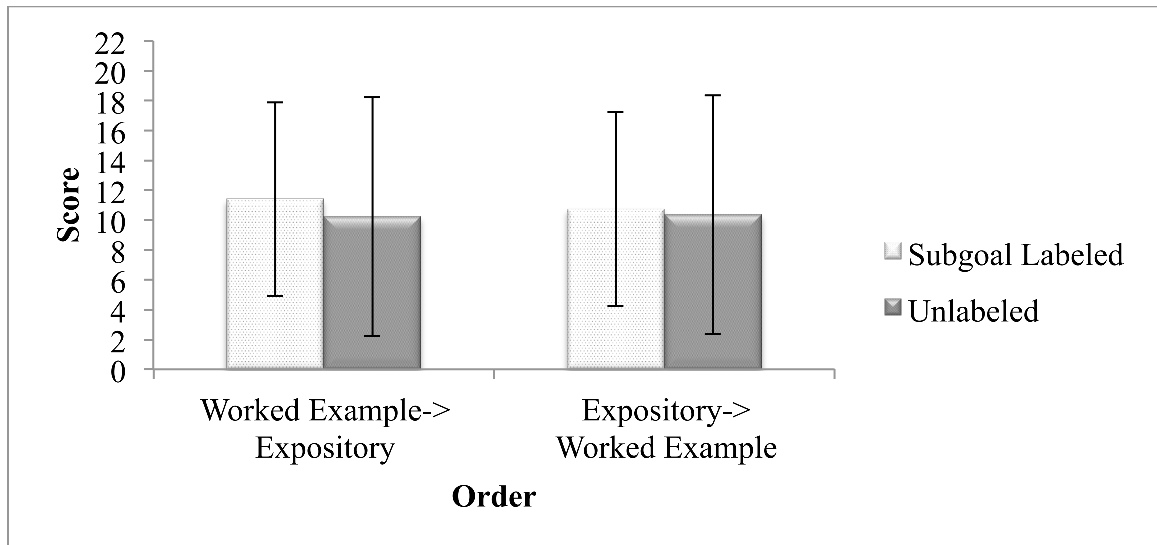


Figure 6. Mean scores and standard deviations for the Problem Solving Task: Performance in AppInventor. No significant differences were found among the groups.

Table 3. *Descriptive Statistics for Problem Solving Task*

	Worked Example First		*Expository First*	
	Subgoals	*No labels*	*Subgoals*	*No labels*
	M (SD)	*M (SD)*	*M (SD)*	*M (SD)*
App Inventor Performance	11.41 (7.43)	10.25 (6.44)	10.75 (7.99)	10.37 (8.44)
Written Performance	10.91 (7.13)	10.15 (6.82)	8.67 (6.68)	10.42 (6.67)
Attempted Subgoals	6.41 (3.61)	6.06 (3.05)	5.75 (3.51)	6.22 (3.32)

Further inspection of the data revealed that the data were not normally distributed (see Figure 7). The residuals were examined and were not normally distributed, violating the normality assumption of the ANOVA. Therefore, a Kruskal-Wallis H test was used to determine if there were differences in the performance score among the four instructional groups. The mean rank of performance scores was not statistically significantly different between groups, $\chi^2(3) = .789$, $p = .852$.

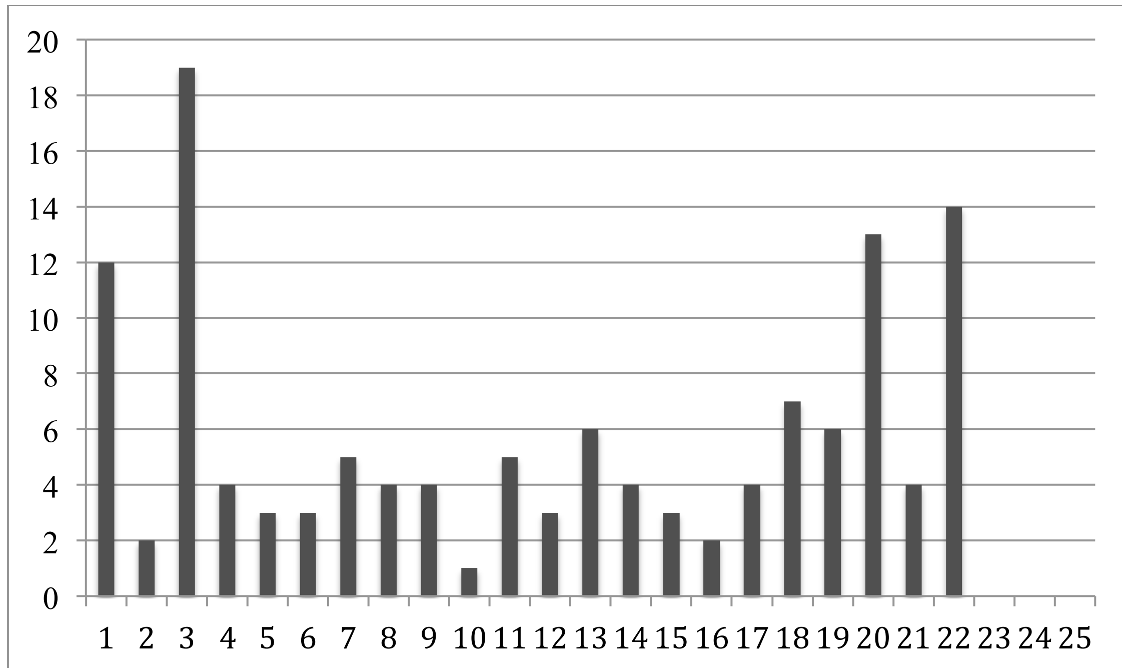


Figure 7. Distribution of scores for Problem Solving Task: Performance in AppInventor.

This was unexpected because prior research suggests that subgoal labels benefit problem solving by helping learners to represent their problem solving knowledge in a way that allows more flexible transfer (e.g. Catrambone, 1998; Margulieux, 2013). For the main effect of subgoal labels, the present study showed $\eta^2_p = 0.00$, and the observed power was 0.09 compared to est. $\omega^2 = .32$ found in Margulieux's (2013) study. The present study saw a very small effect size that would have needed a large sample to reveal any significant differences.

Written Performance.

Participants were awarded one point for each correct step written towards achieving the problem solution for up to a maximum score of 22, and the ICC(A) for this

assessment was .91. There was no main effect of instructional material order, F (1, 128) = 0.69, p = .41, meaning that the participants did not display additional declarative knowledge due to the order the instructional materials were presented. There was also no main effect of subgoal labels, F (1, 128) = 0.18, p = .68, or interaction, F (1, 128) = 1.12, p = .29 (see Figure 8).

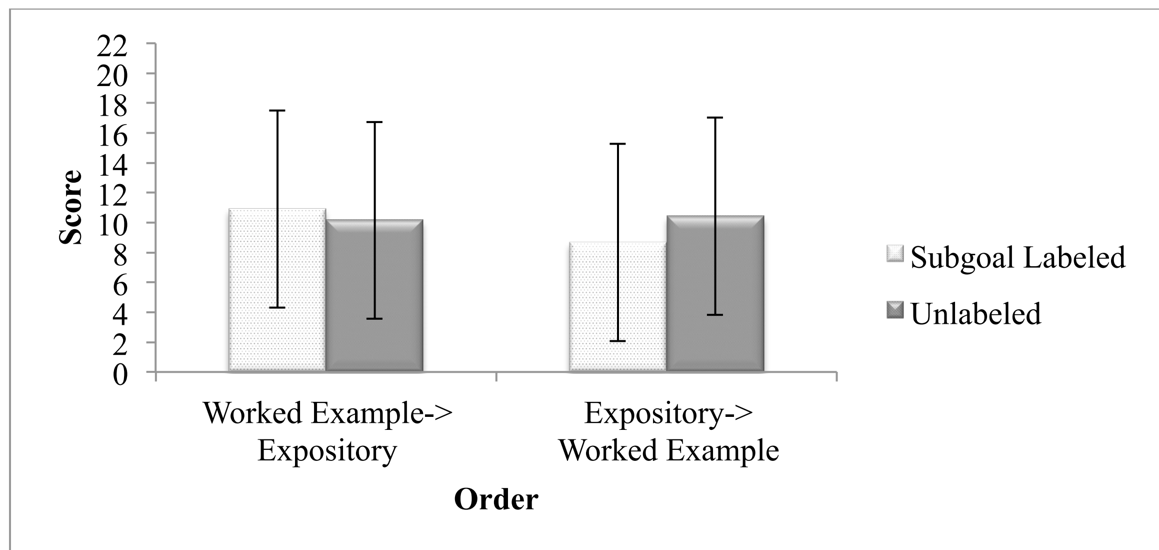


Figure 8. Mean scores and standard deviations for Problem Solving Task: Written Steps. No significant differences were found among the groups.

Additional inspection of the data revealed that the data were not normally distributed. The residuals were examined and discovered to not have a normal distribution, violating the normality assumption of the ANOVA. Therefore, a Kruskal-Wallis H test was used to determine if there were differences in written performance score among the four instructional groups. The mean rank of the written performance scores was not statistically significantly different between groups, $\chi^2(3) = 1.64$, p = .65.

These results did not support the hypothesis that instructional order and subgoal labels would affect the declarative knowledge concerning how to modify and add features to an app in App Inventor.

Attempted Subgoals Written Performance

The written assessment was also scored by how many functional solution components the participants attempted, regardless of whether the steps were correct. The correct solutions were divided into the subgoals needed to achieve them. An attempted subgoal demonstrated that the participant knew the components needed to work towards the solution, even if the participant did not know the specific details for correct execution. Participants earned one point for each subgoal that was attempted. Margulieux (2013) operationally defined an attempted subgoal as "listing at least one step required to complete the subgoal, 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), or describing the subgoal," (p. 19). Participants could earn up to 10 points. The ICC(A) for this assessment was .93. There was no main effect of instructional material order, meaning that presenting the worked example before the expository instructions did not effect the number of subgoals the participant attempted, F (1, 124) = 0.18, p = .68. Additionally, there was no main effect of subgoal labels, meaning that the presence of subgoal labels did not increase the number of subgoals the participant attempted, F (1, 124) = 0.01, p = .92. The interaction was not significant, F (1, 124) = 0.46, p = .50 (see Figure 9).

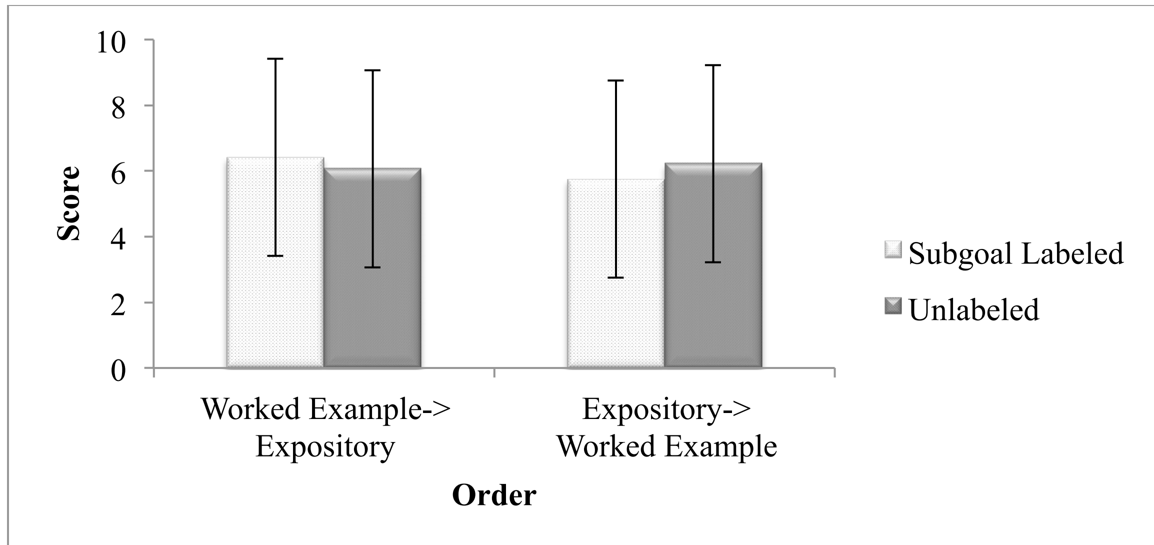


Figure 9. Mean scores and standard deviations for Problem Solving Task: Attempted Subgoals Written Performance. No significant differences were found among the groups.

However, further examination revealed the data and the residuals did not have normal distributions violating the normality assumption of the ANOVA. A Kruskal-Wallis H test was used to determine if there were differences in the attempted subgoal score among the four instructional groups and found the mean rank of attempted subgoal scores was not statistically significantly different between groups, $\chi^2(3) = .821$, $p = .84$. These results did not support the hypothesis that presenting the worked example before the expository instructions and including subgoal labels within the materials would assist the learner in organizing their problem solution.

In conclusion, the participants were asked to solve novel problems in App Inventor, operationalized by asking them to modify or add features of an app in App Inventor. For additional measurement sensitivity, they were asked to write the steps necessary to complete the aforementioned problem solving tasks. Performance in App

Inventor, the written attempted subgoals, and the written correct steps did not reveal a significant effect of instructional material order, subgoal labels, or interaction.

Explanation Task.

In order to measure how well participants could organize and explain problem solutions, participants were given the solutions and instructed to meaningfully group and label the solution steps. Participants were awarded one point for each group that contained only structurally similar steps, for up to a maximum of 10 points. ICC(A) for this assessment was .98. As illustrated in Figure 10, there were no significant differences on grouping structurally similar solution steps based on instructional material order, F (1, 116) = 0.02, p = .89, subgoal labels , F (1, 116) = 0.11, p = .74, or interaction , F (1, 116) = 0.06, p = .81 (see Table 4).

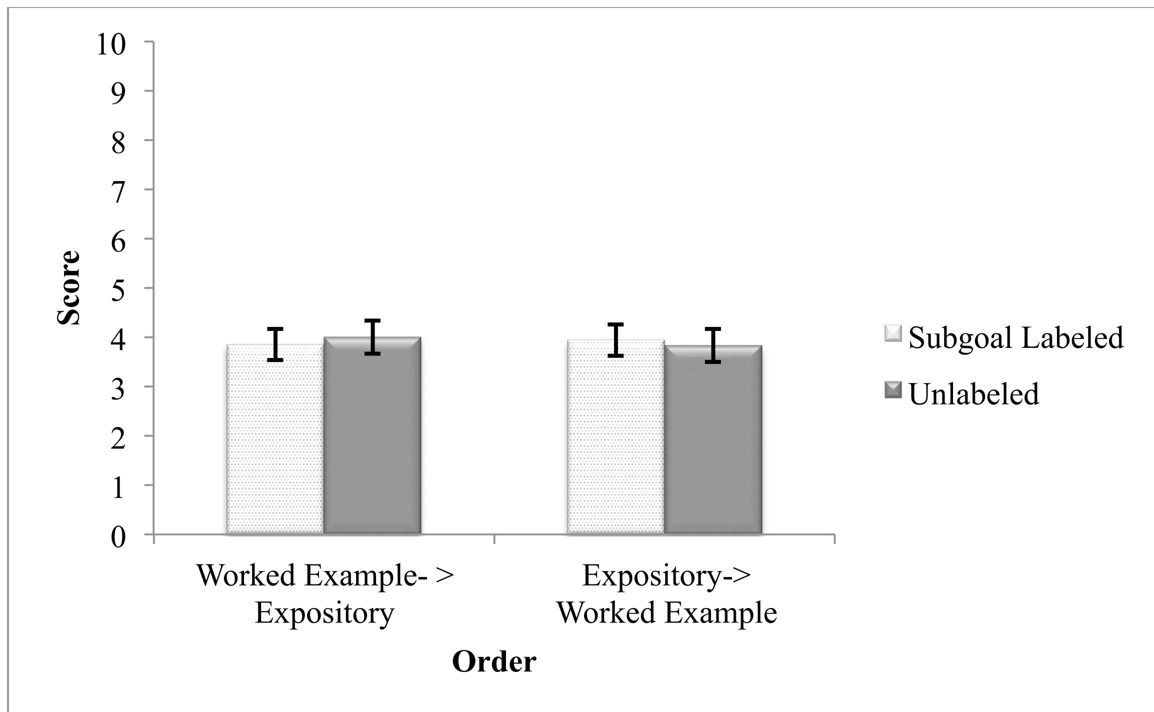


Figure 10. Mean scores and standard deviations for grouping structurally similar solution steps in the Explanation Task. No significant differences were found among the groups.

Table 4. *Descriptive Statistics for Explanation Task*

	Worked Example First		*Expository First*	
	Subgoals	*No labels*	*Subgoals*	*No labels*
	M (SD)	*M (SD)*	*M (SD)*	*M (SD)*
Grouping	3.90 (1.88)	3.87 (1.78)	4.03 (2.01)	3.83 (1.91)
Explanations	1.46 (1.72)	1.24 (1.80)	1.26 (1.87)	1.45 (1.83)

Note: Scored out of a possible nine points.

Labels were scored for whether they described the function of the group of steps. For each label, participants earned one point if the explanation identified the purpose of the grouped steps. These results are illustrated in Figure 11. There were no significant differences based on instructional material order, F (1, 136) = 0.00, p = .98, subgoal labels , F (1, 136) = 0.00, p = .97, or the interaction , F (1, 136) = 0.47, p = .50. The hypothesis that the order the materials were presented, labeling of subgoals, and the interaction would effect performance on organizing and explaining problems solutions was not supported.

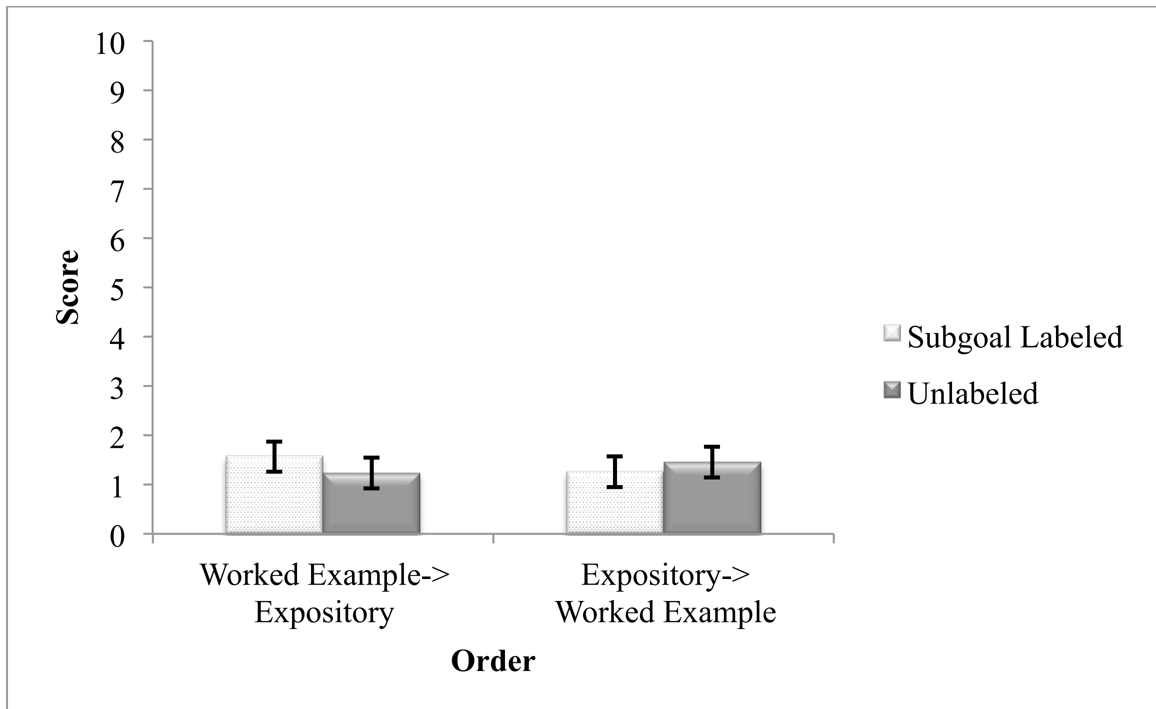


Figure 11. Mean scores and standard deviations for providing functional labels for grouped solution steps in the Explanation Task. No significant differences were found among the groups.

General Procedure Task

The general procedure asked participants to describe the general process they would use to create an app. One point was awarded for each structurally necessary feature the participant described, for up to a maximum score of 6. ICC(A) for this assessment was .99. There was no main effect of instructional material order, F (1, 132) = 0.58, p = .45 There was also no main effect of subgoal labels, F (1, 132) = 1.31, p = .26 (see Table 5).

Table 5. *Descriptive Statistics for General Procedure Task*

Worked Example First		*Expository First*	
Subgoals	*No labels*	*Subgoals*	*No labels*
M (SD)	*M (SD)*	*M (SD)*	*M (SD)*
2.85 (1.46)	2.03 (1.22)	2.12 (1.47)	2.40 (1.29)

Note: Score out of six possible points.

However, there was a significant interaction between the instructional material order and subgoal labeling as depicted in Figure 12, F (1, 132) = 5.49, p = .02. Simple main effects analysis showed that participants who received subgoal labels were able to provide more steps of the general process for creating an app than those who did not receive subgoal labels when presented with the worked example before the expository instructions, p = .02, but there were no differences between the subgoal labeled group and the group without subgoal labels when the expository instructions were presented before the worked example, p= .40.

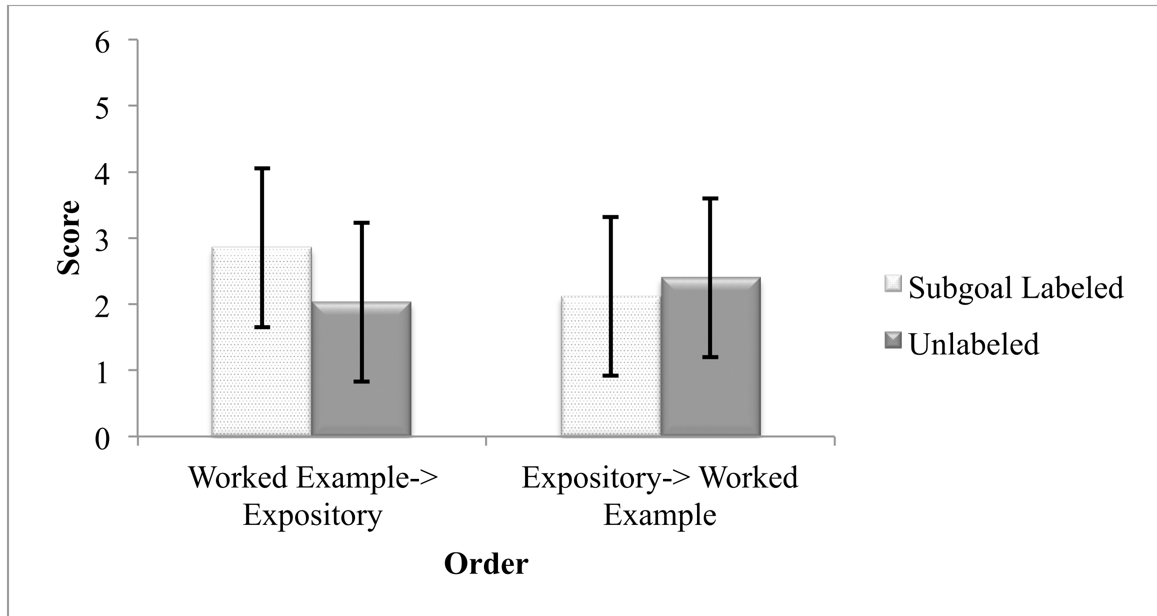


Figure 12. A significant interaction between subgoal labeling and instructional order was present in the General Procedure Task where participants were asked to outline the procedure for creating an app.

Time Spent on Materials

As illustrated in Figure 13, there were no significant differences among groups for time spent on the instructional materials. The order the instructional materials were presented did not effect the time spent on the instructional materials, F (1, 124) = 0.10, p = .76. The presence of subgoal labels did not effect the time spent on the instructional materials, F (1, 124) = 0.44, p = .51. The interaction was also not significant, F (1, 124) = 0.64, p = .43 (see Table 6).

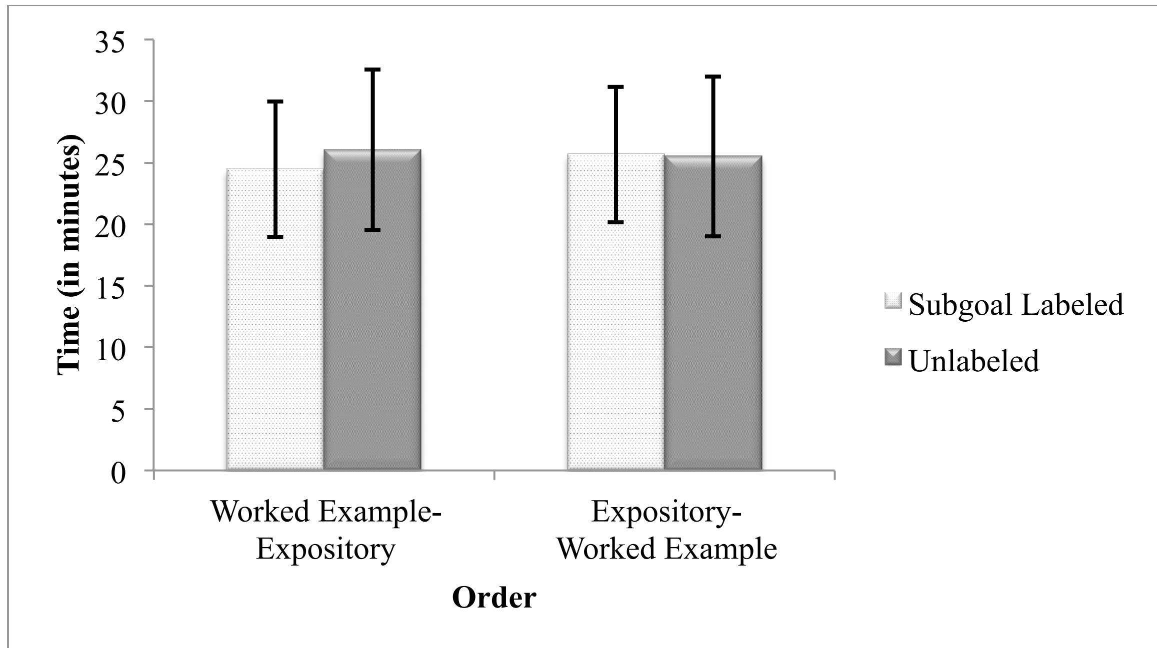


Figure 13. Mean and standard deviations of time (in minutes) spent on the instructional period. No significant differences were found among the groups.

Table 6. *Descriptive Statistics for Time Spent on Instructional and Assessment Periods*

	Worked Example First		*Expository First*	
	Subgoals	*No labels*	*Subgoals*	*No labels*
	M (SD)	*M (SD)*	*M (SD)*	*M (SD)*
Instructional Period	24.37 (4.71)	26.04 (5.11)	25.66 (6.54)	25.52 (7.37)
Assessment Period	36.32 (13.28)	38.27 (13.64)	35.17 (11.47)	35.69 (12.30)

Note: Time measured in minutes.

Margulieux (2013) found that providing subgoal labels decreased the time spent on instructions, and believed this occurred because the subgoal labels help the learner

chunk steps together, making them easier to remember. Additionally, Margulieux (2013) argued that the subgoal labels could have helped the learner find their place when referring back to previous instructions. The difference between these studies for time spent on instructional materials could be due to Margulieux (2013) using mostly text instructions, while the present study utilized more videos; glancing over a text document is likely less time consuming than scrolling through a video, even if subgoal labels are being used to find a specific section.

There were no significant differences among groups for time spent on the assessments. The order the instructional materials were presented did not effect the time spent completing the assessments, *F* (1, 124) = 0.69, *p* = .41 (see Figure 14).

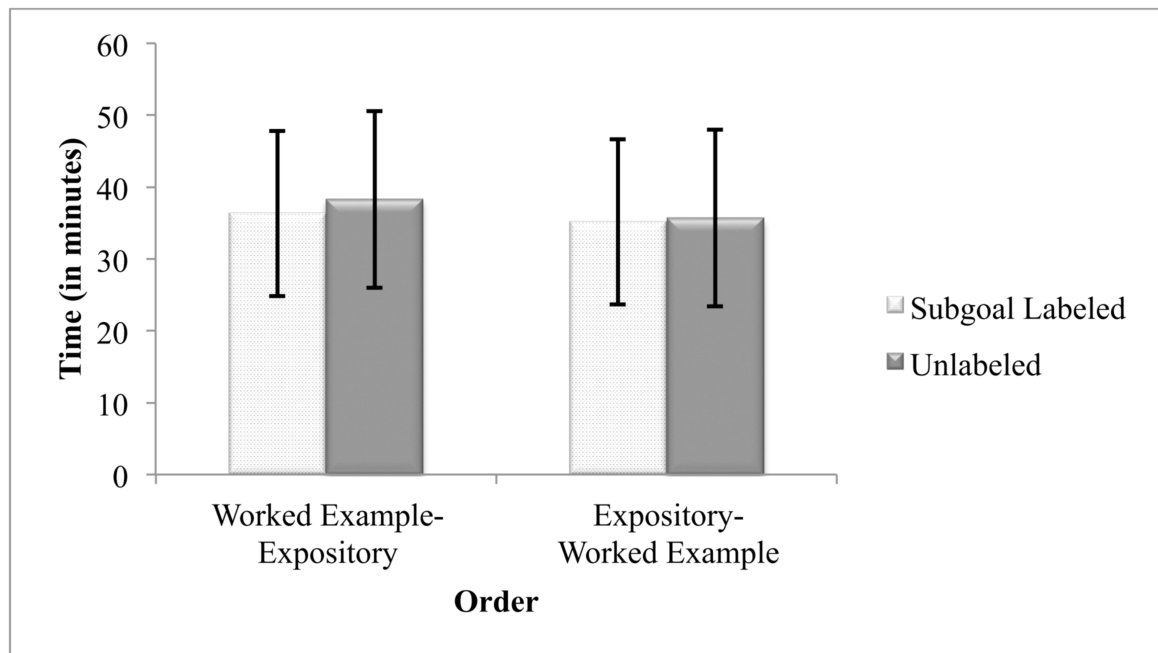


Figure 14. Mean and standard deviations of time (in minutes) spent on the assessments. No significant differences were found among the groups.

Additionally, the presence of subgoal labels in the instructions did not reduce the time spent completing the assessments, F (1, 124) = 0.30, p = .58. The interaction was also not significant, F (1, 124) = 0.10, p = .75. Margulieux (2013) found that those who received subgoal labels performed better on the problem solving tasks and in less time than those who did not receive subgoal labels. The present study did not replicate these findings.

To summarize, the results from the generalization task supported the third hypothesis which stated instructional materials with subgoal labels presented with the worked example first followed by the expository instructions will perform better than all other groups. All other assessments did not support any of the three hypotheses.

CHAPTER 4

DISCUSSION

The present study showed limited evidence that the instructional material order and subgoal labels affect a learner's performance in computer programming. This study suggests that similar learning occurs regardless of whether the worked example is presented before or after the expository instructions. The exception to this is that when asked to provide a general outline for creating an app, participants whose instructions contained subgoal labels and received the worked example before the expository instructions performed better than the other groups.

The reasoning behind presenting the worked example before the expository instructions was partly based on the literature about advance organizers. The benefit of an advance organizer lies on relating the new information to the existing cognitive structures. However, it is possible that the given instructions were not aligned with the participants' cognitive structures. The distribution of scores for the problem solving task in Figure 7 show that although some students did well, many performed poorly. It is plausible that the instructions might have been at an appropriate level for the high performers, but not for the low performers. For the participants who did not do well, the worked example might not have been able to bridge the gap between what the learners already knew and what they were about to learn. Instead, the instructions might have just been new information that was not easily anchored to existing cognitive structures. Additionally, the inductive teaching literature shows that learners are unlikely to learn new information when there are few apparent connections to what the learner already

knows. If the instructions were not at the proper level for the learner, then it follows that presenting the worked example first would have no added benefit.

Contrary to previous research such as Margulieux (2013), subgoal labels did not affect problem solving performance. There are several possible reasons that results in this study differed from results of previous research on subgoal labels. The main difference in research materials between this study and Margulieux (2013) is the media used for the expository instructions. Margulieux (2013) used a text document to convey this information, whereas the present study narrated the text document during a video. This might have reduced the cognitive load as well as ambiguity of these instructions because the learner did not need to mentally transpose the text information to the App Inventor interface. Additionally, auditory information is more transient than text on a piece of paper; each piece of auditory information lasts for only a short period of time compared to text information that is continually present. Instructions presented through videos tend to be processed at a more superficial level than text instructions (Palmiter & Elkerton, 1993). Therefore, the subgoal labels in the videos might not have been processed to the same extent as when they were presented in a text document. As discussed previously, subgoal labels are thought to provide a framework for problem solving and aid in the creation of mental representations. However, if the information was not presented for a long enough duration, or processed the necessary extent, the learner would not be able to form these connections. Future research should investigate the effectiveness of subgoal labels in videos compared to subgoal labels in text instructions.

Finally, the samples might have been substantially different in some way. For example, on the problem solving assessments in the present study, those who received

subgoal labels scored lower than those who received subgoal labels in Margulieux's (2013) study. On the other hand, those who did not receive subgoal labels in the present study scored higher than those who did receive subgoal labels in Margulieux's (2013) study. Additionally, on the explanation task, all groups scored lower than the groups in Margulieux's (2013) study, and the subgoal labeled groups scored lower than Margulieux's (2013) unlabeled groups. The majority of the students in the present study were participating at the end of the semester. This could have caused the participants to be more stressed than participants in other studies due to upcoming exams and project deadlines. This might have caused the participants to be distracted and less attentive which might have negatively affected their performance on these tasks. It is tenable that the videos benefitted low performers by increasing their familiarity with the App Inventor layout and reducing the cognitive load of applying text instructions to the App Inventor interface. However, if the participants were not properly focused and attentive, they might not have been learning the information to the level necessary to perform well on the last few assessments.

Further Work

Further research should broaden the sample to include groups other than undergraduates. Finally, this study focused on performance on the same day the task was learned. Testing after a delay would reveal how well the instructions were incorporated and applied long term. Much instruction aims to teach knowledge and skills that will be used not just on tasks on the day of instruction, but on future tasks. Investigating knowledge that is retained days and weeks after instruction is more reflective of the real-world application of this type of instruction.

APPENDIX A

SUBGOAL LABELED EXPOSITORY SCRIPT

In this session you will create an app that shows a picture of a fortune teller in a button. When you click the button, your fortune will be displayed. The fortune will be picked randomly from a list of possible fortunes.

To create the app, you'll use two different components of App Inventor.

In the App Inventor Designer

This is the first screen that comes up when you start a new project, and this is where you will set up the components of your app.

Create components

Components are the pieces that provide your app functionality, such as a *button* that users can press or a *label* to display information. You'll create components in the App Inventor Designer by selecting which type of component you want to create and dragging it to the screen. The components are on the left of the screen and are organized under different "palettes" which each have a theme (e.g., media or animation).

Set properties

You'll be able to change the properties of each component in the App Inventor Designer as well. For example, you can change how big a *button* is or change the font of a *label* to bold in the "Properties" section on the right of the screen. The properties that can be set depend on the component that is being manipulated.

In the App Inventor Blocks Editor

The Blocks Editor is opened by click on the "Open the Blocks Editor" button in the Designer, and this is where you will program the components of your app.

Handle events from My Blocks

Blocks are the user and computer actions that you'll piece together to program your app. My Blocks is the section of blocks that contains the blocks for the components of your app; that is, if you create a *button*, then the blocks for the *button* will be in My Blocks. To program a feature of your app, you'll first need to define which input, from the user or computer, will start the program. These inputs will almost always come from My Blocks. For example, if you want to create a feature, so text is displayed when a button is clicked, you'll need to start with the block "when button.click," so the program knows after what action to display the text.

Set outputs from My Blocks

Similarly, to programming the feature, you'll also need to define what output you want. These outputs will almost always come from My Blocks. From the previous example, if you want to display the text on a label, then you'll need to add the block "set label.text" to the "when button.click" block.

Set conditions from Built-in

The Built-in blocks are blocks that are not dependent on which components your app has. Built-in blocks allow you to add features, such as variables, to your app with which the user will not directly interact. You can use these blocks to create conditions for your program. From the previous example, if you wanted the program to randomly select the text to be displayed from a list of text items, then you'd need to create a list and add the "call select list item" block to the "set label.text" block.

Define variables from Built-in

Variables are a value that can be changed. By defining a variable, you are giving that value a name that can be used in a program. From the previous example, the text that is displayed from the list of text items is a variable. Because the text that is displayed can change, you'll need to attach the variable block to the "call select list item" block.

The following video will demonstrate how to use Android App Inventor and show you how to make the Fortune Teller app.

At this time, please watch the video by clicking on the Media Player icon at the bottom of your screen. Make sure that you wear headphones while watching the video. This video will demonstrate how to create this app. When you’re done watching the video, use the following steps to create your own Fortune Teller app.

APPENDIX B

NON-SUBGOAL LABELED EXPOSITORY SCRIPT

In this session you will create an app that shows a picture of a fortune teller in a button. When you click the button, your fortune will be displayed. The fortune will be picked randomly from a list of possible fortunes.

To create the app, you'll use two different components of App Inventor.

In the App Inventor Designer

This is the first screen that comes up when you start a new project, and this is where you will set up the components of your app.

Components are the pieces that provide your app functionality, such as a *button* that users can press or a *label* to display information. You'll create components in the App Inventor Designer by selecting which type of component you want to create and dragging it to the screen. The components are on the left of the screen and are organized under different "palettes" which each have a theme (e.g., media or animation).

You'll be able to change the properties of each component in the App Inventor Designer as well. For example, you can change how big a *button* is or change the font of a *label* to bold in the "Properties" section on the right of the screen. The properties that can be set depend on the component that is being manipulated.

In the App Inventor Blocks Editor

The Blocks Editor is opened by click on the "Open the Blocks Editor" button in the Designer, and this is where you will program the components of your app.

Blocks are the user and computer actions that you'll piece together to program your app. My Blocks is the section of blocks that contains the blocks for the components of your app; that is, if you create a *button*, then the blocks for the *button* will be in My Blocks. To program a feature of your app, you'll first need to define which input, from the user or computer, will start the program. These inputs will almost always come from

My Blocks. For example, if you want to create a feature, so text is displayed when a button is clicked, you'll need to start with the block "when button.click," so the program knows after what action to display the text.

Similarly, to programming the feature, you'll also need to define what output you want. These outputs will almost always come from My Blocks. From the previous example, if you want to display the text on a label, then you'll need to add the block "set label.text" to the "when button.click" block.

The Built-in blocks are blocks that are not dependent on which components your app has. Built-in blocks allow you to add features, such as variables, to your app with which the user will not directly interact. You can use these blocks to create conditions for your program. From the previous example, if you wanted the program to randomly select the text to be displayed from a list of text items, then you'd need to create a list and add the "call select list item" block to the "set label.text" block.

Variables are a value that can be changed. By defining a variable, you are giving that value a name that can be used in a program. From the previous example, the text that is displayed from the list of text items is a variable. Because the text that is displayed can change, you'll need to attach the variable block to the "call select list item" block.
The following video will demonstrate how to use Android App Inventor and show you how to make the Fortune Teller app.

At this time, please watch the video by clicking on the Media Player icon at the bottom of your screen. Make sure that you wear headphones while watching the video. This video will demonstrate how to create this app.

When you're done watching the video, use the following steps to create your own Fortune Teller app.

APPENDIX C

SUBGOAL LABELED WORKED EXAMPLE SCRIPT

Let's create an app that will tell our fortune. We will create a new project and will call it "Fortune."

Create Component

What we are going to want is a button that has a picture of a fortune teller in it so we will drag a button over from the basic palette.

Set Properties

Under the properties we'll change the text for the button, clear that out, and set an image for it. So we will add an image. Under fortune teller I've already saved an image for the gypsy. I need to get rid of the text for the button, the default text, so clear that out. I'm using internet explorer which doesn't always set the width correctly so I'm going to go ahead and click "fill parent" on that one and set height to 300.

Create Component

The other thing I'm going to want is a label, a little bit of text to tell the user to hit the button to get their fortune. So I'm going to add a label from the "basic" palette as well, underneath.

Set Properties

I'll change the text for that to say "Press the button to see your fortune." I'm going to change the name of that label to be "fortuneLabel." That is all we need, a button and label, and that's where we'll put the actual fortune is the text of that label.

Then we are ready to program it over on the blocks editor. So I already see that I have "Button1" and "fortuneLabel." What I want to do is have a whole bunch of different possible fortunes and then use a random number to select which fortune to show you when you click the button. So to do that I need to make a list that has several different fortunes in it.

Define Variables from Built-in

To make a list that is a type of variable so I will go into "Built-in" and "Definitions" and I will create a variable, defvariable. I will change the name by clicking on variable. I will call it "fortuneList." I'm going to make that as a list and in "Built-In," "Lists" there is "make a list." That's what I will set fortuneList to and I can add items to it and the kind of items I am going to add are text items. I will go into "text." I can change what is displayed on the text. So I will start making some fortunes. I can keep adding items here by just getting another text block, and every time I add one there is another empty space so I can make as many as I want. So I am making several nicer fortunes and maybe I will make a not-so-happy fortune. So that's making my list and setting it to a variable fortune list to a list of different possible fortunes.

Handle Events in My Blocks

Now I am going to My Blocks when "button1" is clicked I want to set the text of my labels, my "fortuneLabel," there is a "set text."

Set Output from My Blocks

I want to set it to one of those fortunes from the list, just randomly.

Set Conditions from Built-in

The way that I can do that is in "Built-In" in "Math" there is away to get a random number, so I can get a random integer from some value to some value. I want to use that random integer to get an item from the list. Under "Lists" there is a "select list item" from the list given an index. That is what I am going to use. So I will set the text to "select list item" from the list. Which list? Well, if I go back to "My Blocks" "My Definitions" I will say the "fortuneList." Which index to use? Well, I am going to use the "Built-in" "Math," get a "random integer," call it "random integer" from one to- instead of 100 here I am going to use the length of my list. I am going to do it programmatically instead of with a hard code number because that way if I add things to my list it will still work. So there is a way to get the length of the list. And I have to tell it which list again. Under "My

Blocks" "My Definitions" is my "fortuneList." When the button is clicked, we are going to set the label, "fortuneLabel.Text" to select a list item from the fortune list using the index that is a random integer between one and the length of the list. You can try it out.

APPENDIX D

NON-SUBGOAL LABELED WORKED EXAMPLE SCRIPT

Let's create an app that will tell our fortune. We will create a new project and will call it "Fortune." What we are going to want is a button that has a picture of a fortune teller in it so we will drag a button over from the basic palette.

Under the properties we'll change the text for the button, clear that out, and set an image for it. So we will add an image. Under fortune teller I've already saved an image for the gypsy. I need to get rid of the text for the button, the default text, so clear that out. I'm using internet explorer which doesn't always set the width correctly so I'm going to go ahead and click "fill parent" on that one and set height to 300.

The other thing I'm going to want is a label, a little bit of text to tell the user to hit the button to get their fortune. So I'm going to add a label from the "basic" palette as well, underneath.

I'll change the text for that to say "Press the button to see your fortune." I'm going to change the name of that label to be "fortuneLabel." That is all we need, a button and label, and that's where we'll put the actual fortune is the text of that label.

Then we are ready to program it over on the blocks editor. So I already see that I have "Button1" and "fortuneLabel." What I want to do is have a whole bunch of different possible fortunes and then use a random number to select which fortune to show you when you click the button. So to do that I need to make a list that has several different fortunes in it.

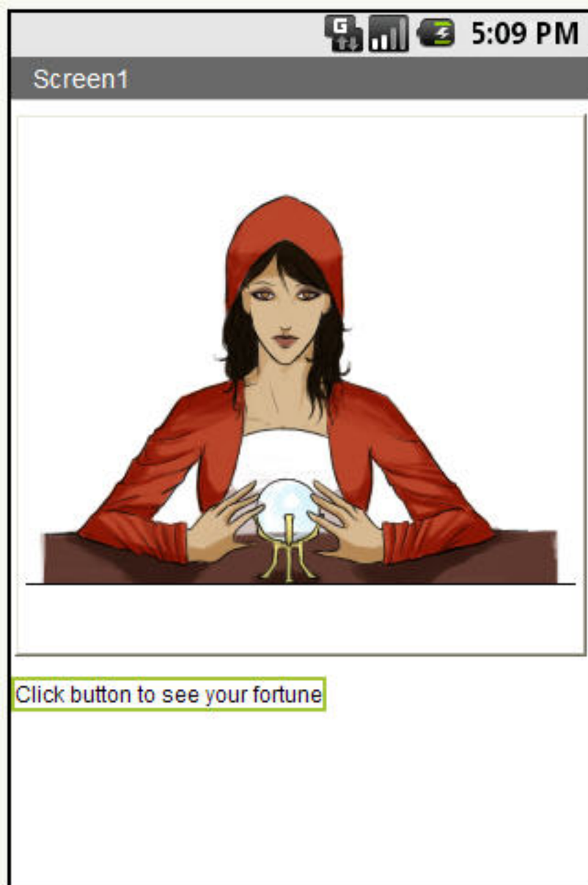
To make a list that is a type of variable so I will go into "Built-in" and "Definitions" and I will create a variable, defvariable. I will change the name by clicking on variable. I will call it "fortuneList." I'm going to make that as a list and in "Built-In," "Lists" there is "make a list." That's what I will set fortuneList to and I can add items to it and the kind of items I am going to add are text items. I will go into "text." I can

change what is displayed on the text. So I will start making some fortunes. I can keep adding items here by just getting another text block, and every time I add one there is another empty space so I can make as many as I want. So I am making several nicer fortunes and maybe I will make a not-so-happy fortune. So that's making my list and setting it to a variable fortune list to a list of different possible fortunes.

Now I am going to My Blocks when "button1" is clicked I want to set the text of my labels, my "fortuneLabel," there is a "set text." I want to set it to one of those fortunes from the list, just randomly. The way that I can do that is in "Built-In" in "Math" there is away to get a random number, so I can get a random integer from some value to some value. I want to use that random integer to get an item from the list. Under "Lists" there is a "select list item" from the list given an index. That is what I am going to use. So I will set the text to "select list item" from the list. Which list? Well, if I go back to "My Blocks" "My Definitions" I will say the "fortuneList." Which index to use? Well, I am going to use the "Built-in" "Math," get a "random integer," call it "random integer" from one to- instead of 100 here I am going to use the length of my list. I am going to do it programmatically instead of with a hard code number because that way if I add things to my list it will still work. So there is a way to get the length of the list. And I have to tell it which list again. Under "My Blocks" "My Definitions" is my "fortuneList." When the button is clicked, we are going to set the label, "fortuneLabel.Text" to select a list item from the fortune list using the index that is a random integer between one and the length of the list. You can try it out.

APPENDIX E

SUBGOAL LABELED PRACTICE PROBLEM GUIDE



1. Go to the Android App Inventor website by clicking the Firefox icon that is on the bottom of your screen.
2. Create a new project by clicking on *New* and naming the project "fortune" and your participant number (e.g., “fortune1”). Ask the moderator if you do not know your participant number.

In the Designer

Create Component

3. From the basic palette drag out a *Button*.

Buttons are components that users touch to perform some action in your app. Buttons detect when users tap them. Many aspects of a button's appearance can be changed. You can use the Enabled property to choose whether a button can be tapped.

Set Properties

4. Set the image source to "gypsy.jpg". This file will be located in the “Media” folder on the desktop.
5. Clear the default text.
6. Set the width to fill the parent's width and the height to 300 pixels.

Create Component

7. From the basic palette drag out a *Label*.

Labels are components used to show text. A label displays text which is specified by the Text property. Other properties, all of which can be set in the Designer or Blocks Editor, control the appearance and placement of the text.

8. Place the *Label* underneath the gypsy image.

Set Properties

9. Set the text to "Click button to see your fortune".
10. Rename it to "fortuneLabel".

In the Blocks Editor

11. Open the blocks editor.

Define Variables from Built-in

12. Click on "Built-In" and "Definition" and pull out a *def variable.*

A ***variable*** creates a value that can be changed while an app is running, and gives that value a name. Variables are global in scope, which means you can refer to them from any code in the app, including from within procedures.

When you create a variable, App Inventor will automatically create two associated blocks, and place them in the My Definitions drawer in My Blocks:

- The global block gets the value of the variable.
- The set global block changes the value of the variable.

13. Click on the "variable" and replace it with "fortuneList". This creates a variable called "fortuneList".
14. Click on "Lists" and drag out a *call make a list*

Make a list creates a list from the given blocks. If you don't supply any arguments, this creates an empty list, which you can add elements to later.

15. Click on "Text" and drag out a *text text* block and drop it next to "item".

Text contains a text string.

16. Click on the rightmost "text" and replace it with your first fortune.
17. Repeat steps 15 and 16 to create 3 additional fortunes.

Handle Events from My Blocks

18. Click on "My Blocks" and "Button1".
19. Drag out a *when Button1.Click*.

Set Output from My Blocks

20. Click on "fortuneLabel"
21. Drag out a *set fortuneLabel.Text to* and drop it in the *when Button1.Click*

Set Conditions from Built-in

22. Click on "Built-In" and "Lists"
23. Drag out a *call select list item*

Select list item selects the item at the given index in the given list.

24. Click on "My Blocks" and "My Definitions"
25. Drag out a *global fortuneList* and put that next to the area marked "list".
26. Click on "Built-In" and "Math"
27. Drag out a *call random integer* and drop it in the area marked "index".

Random integer returns a random integer value between the given values, inclusive. The order of the arguments doesn't matter.

28. Remove the "100" number block next to the "to" area by throwing it in the trash.
29. Click on "Lists"
30. Drag out a *call length of list* and drop it in the "to" area.

Length of list returns the number of items in the list.

31. Click on "My Blocks" and "My Definitions"
32. Drag out a *global fortuneList* and drop it after the area marked "list" in *call length of list*.

Now you have a Fortune Teller app!

APPENDIX F

NON-SUBGOAL LABELED PRACTICE PROBLEM GUIDE

1. Go to the Android App Inventor website by clicking the Firefox icon that is on the bottom of your screen.
2. Create a new project by clicking on *New* and naming the project "fortune" and your participant number (e.g., “fortune1”). Ask the moderator if you do not know your participant number.
3. From the basic palette drag out a *Button*.

Buttons are components that users touch to perform some action in your app. Buttons detect when users tap them. Many aspects of a button's appearance can be changed. You can use the Enabled property to choose whether a button can be tapped.

4. Set the image source to "gypsy.jpg". This file will be located in the “Media” folder on the desktop.
5. Clear the default text.
6. Set the width to fill the parent's width and the height to 300 pixels.
7. From the basic palette drag out a *Label*.

Labels are components used to show text. A label displays text which is specified by the Text property. Other properties, all of which can be set in the Designer or Blocks Editor, control the appearance and placement of the text.

8. Place the *Label* underneath the gypsy image.
9. Set the text to "Click button to see your fortune".
10. Rename it to "fortuneLabel".
11. Open the blocks editor.
12. Click on "Built-In" and "Definition" and pull out a *def variable.*

A ***variable*** creates a value that can be changed while an app is running, and gives that value a name. Variables are global in scope, which means you can refer to them from any code in the app, including from within procedures.

When you create a variable, App Inventor will automatically create two associated blocks, and place them in the My Definitions drawer in My Blocks:

- The global block gets the value of the variable.
- The set global block changes the value of the variable.

13. Click on the "variable" and replace it with "fortuneList". This creates a variable called "fortuneList".
14. Click on "Lists" and drag out a *call make a list*

Make a list creates a list from the given blocks. If you don't supply any arguments, this creates an empty list, which you can add elements to later.

15. Click on "Text" and drag out a *text text* block and drop it next to "item".

Text contains a text string.

16. Click on the rightmost "text" and replace it with your first fortune.

17. Repeat steps 15 and 16 to create 3 additional fortunes.

18. Click on "My Blocks" and "Button1".

19. Drag out a *when Button1.Click*.

20. Click on "fortuneLabel"

21. Drag out a *set fortuneLabel.Text to* and drop it in the *when Button1.Click*

22. Click on "Built-In" and "Lists"

23. Drag out a *call select list item*

Select list item selects the item at the given index in the given list.

24. Click on "My Blocks" and "My Definitions"

25. Drag out a *global fortuneList* and put that next to the area marked "list".

26. Click on "Built-In" and "Math"

27. Drag out a *call random integer* and drop it in the area marked "index".

Random integer returns a random integer value between the given values, inclusive. The order of the arguments doesn't matter.

28. Remove the "100" number block next to the "to" area by throwing it in the trash.

29. Click on "Lists"

30. Drag out a *call length of list* and drop it in the "to" area.

Length of list returns the number of items in the list.

31. Click on "My Blocks" and "My Definitions"

32. Drag out a *global fortuneList* and drop it after the area marked "list" in *call length of list*.

Now you have a Fortune Teller app!

APPENDIX G

ASSESSMENT ONE

A ten-item questionnaire for the measurement of IL (Items 1, 2, and 3), EL (Items 4, 5, and 6), and GL (Items 7, 8, 9, and 10) developed from Leppink et al. (2013).

All of the following questions refer to the activites that just finished. Please respond to each of the questions on the following scale (0 meaning *not at all the case* and 10 meaning *completely the case*). 0 1 2 3 4 5 6 7 8 9 10

[1] The topic/topics covered in the activity was/were very complex.

[2] The activity covered procedures that I perceived as very complex.

[3] The activity covered concepts and definitions that I perceived as very complex.

[4] The instructions and/or explanations during the activity were very unclear.

[5] The instructions and/or explanations were, in terms of learning, very ineffective.

[6] The instructions and/or explanations were full of unclear language.

[7] The activity really enhanced my understanding of the topic(s) covered.

[8] The activity really enhanced my knowledge and understanding of programming.

[9] The activity really enhanced my understanding of the procedures covered.

[10] The activity really enhanced my understanding of concepts and definitions.

APPENDIX H

ASSESSMENT TWO, PART ONE

Complete the following tasks in App Inventor.

1. Italicize the fortune presented in your fortune teller app.

2. You can create a ball that moves around your screen at a set heading (in degrees, 0 degrees is towards the right, 90 degrees is towards the top), set interval (in milliseconds), and set speed (in pixels). Make a ball that moves at a rate of 5 pixels every 250 milliseconds towards the right of the screen (hint: animation components must be on a canvas).

3. Create a list of colors and make the ball change to a random color whenever it collided with something.

4. Make the ball change direction (called heading in App Inventor) to 90 degrees more than its current direction whenever it is touched.

APPENDIX I

ASSESSMENT TWO, PART TWO

Write the steps you would take to italicize the fortune presented.

You can create a ball that moves around your screen at a set heading (in degrees, 0 degrees is towards the right, 90 degrees is towards the top), set interval (in milliseconds), and set speed (in pixels). Write the steps you would take to make a ball that moves at a rate of 5 pixels every 250 milliseconds towards the right of the screen (hint: animation components must be on a canvas).

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.

Write the steps you would take to make the ball change direction (called heading in App Inventor) to 90 degrees more than its current direction whenever it is touched.

APPENDIX J

ASSESSMENT THREE

The sheet you received has the steps to the solutions of the problems that you were just working on. The steps are correct and in the correct order. Please group the steps of these solutions that you think go together (either by circling them or drawing a bracket around them). "Go together" is open to your interpretation, but think of it as if you were trying to put headers into the solution to group steps in some meaningful way. If you do not think any of the steps go together, you do not have to group any steps. If you group steps together, please provide a label or description of why you think those steps go together.

Write the steps you would take to italicize the fortune presented.

Select Label

Click "FontItalic"

You can create a ball that moves around your screen at a set heading (in degrees, 0 degrees is towards the right, 90 degrees is towards the top), set interval (in milliseconds), and set speed (in pixels). Write the steps you would take to make a ball that moves at a rate of 5 pixels every 250 milliseconds towards the right of the screen (hint: animation components must be on a canvas).

Drag out Ball

Set Heading to 0

Set Interval to 250

Set Speed to 5

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.

Drag out “def variable”

Add “call make a list” and put it in “as”

Add colors to list

Drag out “when Ball1.CollidedWith”

Add “set Ball1.PaintColor” and put it in “do”

Add “call select list item” and put it in “do”

Add “global color” and put it in “list”

Add “call random integer” and put it in “index”

Delete “100” from “to”

Add “call length of list” and put it in “to”

Add “global color” and put it in “list”

Write the steps you would take to make the ball change direction (called heading in App Inventor) to 90 degrees more than its current direction whenever it is touched.

Drag out “when Ball1.Touched”

Add “set Ball1.Heading” and put in “do”

From math, add “+” block

Add “Ball1.Heading” and “90” to the “+” block

APPENDIX K

ASSESSMENT FOUR

Describe the general procedure you would take to create an app that has an image and a sound, so that the sound played when the image was touched. You do NOT need to list the specific steps, just the general procedure.

A good first step would be, “Make a component for the image.”

A bad first step would be, “Drag an image sprite from the palette to the canvas,” because it’s too specific.

REFERENCES

Alfieri, L., Nokes-Malach, T. J., & Schunn, C. D. (2013). Learning through case comparisons: a meta-analytic review. *Educational Psychologist*, *48*(2), 87-113.

Anderson, J. R. (1990). *The adaptive character of thought*. Hillsdale, NJ: Lawrence Erlbaum.

Ausubel, D. P. (1968). *Educational psychology: A cognitive view.* New York: Holt, Rinehart and Winston.

Beatty, A. (Rapporteur), Committee on Highly Successful Schools or Programs for K-12 STEM Education, National Research Council. (2011). *Successful STEM education: A workshop summary.* Retrieved from http://www.nap. edu/catalog. php?record_id=13230

Catrambone, R. (1990). Specific versus general procedures in instructions. *Human-Computer Interaction*, *5*, 49–93.

Catrambone, R. (1994). Improving examples to improve transfer to novel problems. *Memory & Cognition, 22*(5), 606-615.

Catrambone, R. (1995). Aiding subgoal learning: Effects on transfer. *Journal of Educational Psychology, 87*(1), 5-17. doi: 10.1037/0022-0663.87.1.5

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.

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.

Charney, D. H., & Reder, L. M. (1987). Initial skill learning: An analysis of how elaborations facilitate the three components. In P. Morris (Ed.), *Modelling cognition* (pp. 135–165). Chichester, UK: Wiley.

Committee on Highly Successful Schools or Programs in K-12 STEM Education, National Research Council. (2011). *Successful K-12 STEM education: Identifying effective approaches in science, technology, engineering, and mathematics.* Retrieved from http://www.nap.edu/catalog.php?record_id=13158

Cooper, S., Grover, S., Guzdial, M., & Simon, B. (2014). A Future for Computing Education Research. *Communications Of The ACM*, *57*(11), 34-36. doi:10.1145/2668899

Dale, E. (1946). *Audiovisual Methods in Teaching.* Dryden Press, New York, NY.

Eiriksdottir, E., & Catrambone, R. (2011). Procedural instructions, principles, and examples: How to structure instructions for procedural tasks to enhance performance, learning, and transfer. *Human Factors, 53*(6), 749-770. doi: 10.1177/0018720811419154

Ericson, B. (2012, February 12). ICE Distance Education Portal. Retrieved from http://ice.cc.gatech.edu/dl/?q=node/641

Hundhausen, C. D., Farley, S. F., & Brown, J. L. (2009). Can directed manipulation lower the barriers to computer programming and promote transfer of training?: An experimental study. *ACM Transactions in CHI, 16*(3). doi: 10.1145/1592440.1592442

Karreman, J., Ummelen, N., & Steehouder, M. (2005, July). Procedural and declarative information in user instructions: What we do and don't know about these information types. Paper presented at the *IEEE International Professional Communication Conference*, Limerick, Ireland.

LeFevre, J.. & Dixon, P. (1986). Do written instructions need examples? *Cognition and Instruction, 3*, l-30. doi: 10.1207/s1532690xci0301_1

Leppink, J., Paas, F., Van der Vleuten, C. M., Van Gog, T., & Van Merriënboer, J. G. (2013). Development of an instrument for measuring different types of cognitive load. *Behavior Research Methods*, *45*(4), 1058-1072. doi:10.3758/s13428-013-0334-1

Margulieux, L. E. (2013) *Subgoal Labeled Instructional Text and Worked Examples in STEM Education.* Unpublished master's thesis, Georgia Institute of Technology, Atlanta, Georgia.

Margulieux, L. E., Guzdial, M., & Catrambone, R. (2012). Subgoal-labeled instructional material improves performance and transfer in learning to develop mobile applications. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, 71-78. doi: 10.1145/2361276.2361291

National Science Foundation. (2007). *A national action plan for addressing the critical needs of the U.S. science, technology, engineering, and mathematics education system*. Retrieved from http://www.nsf.gov/pubs/2007/nsb07114/nsb07114.pdf

Novak, J.N. (1977). *A Theory of Education*. New York: Cornell University Press.

Paas, F., Renkl, A., & Sweller, J. (2003). Cognitive load theory and instructional design: Recent developments. *Educational Psychologist, 38(1), 1-4.*

Paas, F., & van Gog, T. (2009). Principles for designing effective and efficient training of complex cognitive skills. In F. T. Durso (Ed.) *Reviews of Human Factors and Ergonomics (Vol. 5),* Santa Monica, CA: HFES, pp. 166-194.

Palmiter, S., & Elkerton, J. (1993). Animated demonstrations for learning procedural computer-based tasks. *Human-Computer Interaction, 8*(3)*,* 193-216. doi:10.1207/s15327051hci0803_1

Palmiter, S., Elkerton, J., & Baggett, P. (1991). 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

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

Pirolli, P., & Recker, M. M. (1994). Learning strategies and transfer in the domain of programming. *Cognition and Instruction*, *12*, 235–275.

Reed, S. K., Dempster, A., & Ettinger, M. (1985). Usefulness of analogous solutions for solving algebra word problems. *Journal Of Experimental Psychology: Learning, Memory, And Cognition*, *11*(1), 106-125. doi:10.1037/0278-7393.11.1.106

Ross, B. (1987). This is like that: The use of earlier problems and the separation of similarity effects. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 13,* 629-639.

Ross, B. (1989). Distinguishing types of superficial similarities: Different effects on the access and use of earlier problems. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 15,* 456-468.

Rountree, N., Rountree, J., Robins, A., & Hannah, R. (2004). Interacting factors that predict success and failure in a CSI course. *SIGCSE Bulletin, 33*(4), pp. 101-104.

Rumelhart, D. E., & Norman, D. A. (1981). Analogical processes in learning. In J. R. Anderson (Ed.), Cognitive skills and their acquisition (pp. 335-360). Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Trafton, J. G., & Reiser, B. J. (1993). *Studying examples and solving problems: Contributions to skill acquisition*. Technical report, Naval HCI Research Lab, Washington, DC, USA.

van Merriënboer, J. G., Clark, R. E., & de Croock, M. M. (2002). Blueprints for Complex Learning: The 4C/ID-Model. *Educational Technology Research And Development*, (2), 39. doi:10.2307/30221150

Wiedenbeck, S. (1989). Learning iteration and recursion from examples. *International Journal of Man-Machine Studies*, *30*(1), 1–22.