

BUILDING BRIDGES: DUAL-MODALITY INSTRUCTION AND INTRODUCTORY  
PROGRAMMING COURSEWORK

By

JEREMIAH J. BLANCHARD

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL  
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT  
OF THE REQUIREMENTS FOR THE DEGREE OF  
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2020

© 2020 Jeremiah J. Blanchard

To my dear departed Pops, who pushed me never to settle for less than the best of my capabilities, and who always engaged with a gentle touch and kindness. While the world is poorer for your absence, your words continue to echo inspiration in your progeny, and your actions have set an example of love we can only hope to someday achieve. We miss you!

## ACKNOWLEDGMENTS

I thank my advisors, Drs. Lisa Anthony and Christina Gardner-McCune, for the endless hours working with an unconventional student. The time they have spent helping me to hone my research and writing skills is a gift I hope I can pay forward to others in the future. Looking back, it is hard to convey how much I have learned and how much they have changed the course of my life. Thanks also go to my committee members – Drs. Kristy Boyer, Corrine Huggins-Manley, David Weintrop, and Joseph Wilson – for reading drafts, providing generous feedback, and helping and providing expertise throughout the proposal and defense process. I also thank David Bau, whose invitation to work on the Pencil Code project opened the door to my eventual dissertation research.

From the bottom of my heart, I thank my kind and ever-patient spouse, Kyoko, for supporting my long and winding path through graduate school for nearly two decades. Without her constant encouragement and incredible resolve, I could never have completed this journey. Her sacrifices – not only supporting me, but also caring for our children during my long writing nights and conference trips – were instrumental in the completion of this dissertation.

# TABLE OF CONTENTS

	<u>page</u>
ACKNOWLEDGMENTS .....	4
LIST OF TABLES .....	10
LIST OF FIGURES .....	12
ABSTRACT .....	14
CHAPTER	
1 INTRODUCTION .....	16
1.1 Research Motivation.....	17
1.2 Research Opportunities.....	19
1.3 Research Questions & Hypotheses .....	20
1.4 Overview of Work.....	22
2 BACKGROUND .....	24
2.1 Learning to Program.....	24
2.1.1 Language Challenges.....	24
2.1.1.1 Syntax .....	25
2.1.1.2 Semantics .....	26
2.1.1.3 Perceptions .....	27
2.1.2 Developing Expertise.....	27
2.1.2.1 Sensorimotor .....	28
2.1.2.2 Preoperational.....	29
2.1.2.3 Concrete-operational.....	29
2.1.2.4 Formal operational.....	30
2.1.2.5 Summary .....	30
2.2 Programming Assessment.....	31
2.2.1 Concept Inventories.....	31
2.2.2 Item Response Theory .....	33
2.3 Visual Languages & Environments.....	35
2.3.1 Development of Visual Programming Environments .....	36
2.3.1.1 Early visual programming systems.....	36
2.3.1.2 Blocks-based interface development .....	37
2.3.2 Contemporary Blocks-Based Environments.....	39
2.3.3 Efficacy of Blocks-Based Environments.....	40
2.3.3.1 Effectiveness as learning environment .....	41
2.3.3.2 Moving from blocks to text.....	42
2.3.3.3 Special considerations - environments are not equal.....	44
2.4 Multi-Modal Environments .....	45
2.4.1 Hybrid Modality Environments .....	45

2.4.2	Dual-Modality Programming Environments .....	48
2.4.2.1	Unidirectional translation (blocks to text).....	49
2.4.2.2	Bidirectional translation.....	51
3	STUDY OF PERCEPTIONS OF PROGRAMMING .....	54
3.1	Study: Construct Perceptions in Children (Summer Camp).....	55
3.1.1	Study Context.....	55
3.1.2	Procedure .....	55
3.1.3	Qualitative Measures .....	57
3.1.4	Coding Process.....	58
3.1.5	Programming Definition Themes.....	58
3.1.6	Findings – Perceptions of Programming .....	59
3.1.7	Findings – Perceptions of Constructs .....	60
3.1.8	Influence on Course of Research.....	61
3.2	Position Paper: Bridging Blocks and Text.....	62
4	STUDY OF DUAL-MODALITY PROGRAMMING ENVIRONMENTS.....	64
4.1	Development: Python Variant of Pencil Code.....	64
4.1.1	Description of Work .....	65
4.1.2	Development .....	65
4.1.2.1	Language interpreter runtime .....	66
4.1.2.2	Python routines.....	66
4.1.2.3	Palette (text-to-blocks mapping) .....	66
4.1.3	Results.....	66
4.2	Development: Custom Dual Modality Assessment (Python Text/Blocks) .....	68
4.2.1	Description of Work .....	68
4.2.2	Development .....	68
4.2.3	Impact on Course of Research .....	70
4.3	Study: Perceptions and Concept Assessment (Middle School).....	71
4.3.1	Study Context.....	71
4.3.2	Participants.....	72
4.3.3	Study Design .....	73
4.3.4	Data Collection.....	73
4.3.4.1	Surveys.....	74
4.3.4.2	Assessments .....	74
4.3.5	Data Analysis .....	75
4.3.6	Findings .....	76
4.3.7	Discussion .....	79
5	FINAL STUDY: LEARNING & DUAL-MODALITY INSTRUCTION.....	82
5.1	Research Questions & Hypotheses .....	84
5.1.1	Performance Comparison in Dual-Modality vs Text Instruction.....	85
5.1.2	Performance Comparison by Prior Experience.....	86
5.1.3	Classroom Experience of Dual-Modality Instruction.....	88

5.2 Amphibian: A Dual-Modality-Representation IDE Plugin for Java.....	89
5.2.1 Using the Amphibian Plugin.....	90
5.2.2 Architecture.....	93
5.2.2.1 The Droplet Editor.....	93
5.2.2.2 IntelliJ IDE Plugin Framework.....	94
5.2.2.3 Logging mechanism.....	95
5.3 Dual-Modality Curriculum.....	95
5.3.1 Instruction.....	96
5.3.2 Assignments.....	97
5.3.3 Ethical Considerations.....	98
5.3.3.1 Faculty review.....	98
5.3.3.2 Delay of pure-text instruction.....	98
5.3.3.3 Cognitive overload.....	99
5.4 Instrument Evaluation Study.....	99
5.4.1 Context & Data Collection.....	100
5.4.2 Question Analysis.....	100
5.5 Study: Dual-Modality Instruction, CS Learning, and Classroom Experience (CS1).....	102
5.5.1 Study Design.....	103
5.5.2 Participants.....	106
5.5.3 Data Collection.....	107
5.5.3.1 Examinations, assessments, and demographic surveys.....	107
5.5.3.2 Perception surveys and usage logs.....	108
5.5.3.3 Bias control.....	108
5.6 Analysis Methods: Dual-Modality Instruction and Learning.....	109
5.6.1 Examinations and Assessments.....	110
5.6.1.1 Hypotheses & expectations.....	110
5.6.1.2 SCS1 assessment questions.....	112
5.6.1.3 Course examination questions.....	112
5.6.1.4 Analysis tests.....	114
5.6.2 Surveys, logs, and notes.....	114
5.6.2.1 Qualitative data.....	115
5.6.2.2 Quantitative data.....	116
5.6.2.3 Surveys.....	116
5.6.2.4 Usage logs.....	116
5.6.2.5 Instructor notes.....	118
5.6.3 Summary.....	118
<b>6 LEARNING &amp; DUAL-MODALITY INSTRUCTION: FINDINGS &amp; DISCUSSION.....</b>	<b>119</b>
6.1 Performance Comparison in Dual-Modality vs Text Instruction.....	119
6.1.1 Course Exam Results.....	119
6.1.1.1 Code reading & definitional questions.....	120
6.1.1.2 Code writing questions.....	120
6.1.2 SCS1 Results.....	121
6.1.3 Performance Comparison Discussion.....	121
6.1.3.1 Course Exam performance comparison discussion.....	122
6.1.3.2 SCS1 performance comparison discussion.....	124

6.1.4	Performance Comparison Summary.....	125
6.2	Performance Comparison by Prior Experience.....	126
6.2.1	Course Exam Results.....	126
6.2.1.1	Code reading / definitional questions .....	127
6.2.1.2	Code writing questions .....	129
6.2.2	SCS1 Results.....	132
6.2.3	Prior Experience Discussion .....	132
6.2.3.1	Course exam discussion .....	133
6.2.3.2	SCS1 discussion .....	136
6.2.4	Performance Comparison by Prior Experience Summary.....	137
6.3	Classroom Experience of Dual-Modality Instruction .....	139
6.3.1	Student Perceptions of Dual-Modality Instruction.....	140
6.3.1.1	Participants with only text experience .....	143
6.3.1.2	Participants with only blocks or with both blocks and text experience.....	144
6.3.1.3	Participants with no prior programming experience .....	145
6.3.1.4	Perceptions of dual-modality instruction discussion .....	146
6.3.2	Use of Dual-Modality Materials.....	148
6.3.2.1	Dual-modality materials results.....	148
6.3.2.2	Dual-modality materials discussion.....	151
6.3.3	Instructor Experience.....	152
6.4	Findings & Discussion Summary.....	155
7	CONTRIBUTIONS.....	157
7.1	Foundational Studies (Perceptions of Programming & Dual-Modality Representations) .....	157
7.2	Technical: Python Pencil Code Variant & Amphibian Dual-Modality Java Plugin.....	158
7.3	Empirical: Learning and Dual-Modality Approaches to CS Instruction.....	158
7.4	Instructional: Perceptions in Dual-Modality Programming Environment .....	159
8	CONCLUSIONS.....	160
8.1	Problem.....	160
8.2	Proposed Solution .....	160
8.3	Early Work.....	161
8.3.1	Perceptions of Programming Investigations .....	161
8.3.2	Initial Evaluation of Perceptions & Learning .....	161
8.4	Final Study.....	162
8.4.1	Amphibian Dual-Modality Java Language IDE Plugin for IntelliJ IDEA .....	162
8.4.2	Dual-Modality Instruction & Curriculum.....	163
8.4.3	Instrument Evaluation.....	163
8.4.4	Study of Dual-Modality Instruction and CS Learning .....	163
8.4.5	Analysis of Learning and Dual-Modality Instruction.....	164
8.4.6	Examination of Student Perceptions and Instructor Experience.....	165
8.5	Contributions.....	166
8.6	Future Work.....	166
8.7	Summary.....	168



## APPENDIX

A	CONFERENCES, PUBLICATIONS, & DEVELOPMENT .....	170
	Published / Completed.....	170
	In Progress .....	170
B	TIMELINE FOR DOCTORAL WORK .....	171
C	MIDDLE SCHOOL STUDY: DEMOGRAPHIC QUESTIONNAIRE .....	172
D	MIDDLE SCHOOL STUDY: PERCEPTION QUESTIONNAIRES .....	173
	Personal Perceptions (Pre, Mid, & Post).....	173
	Mid-Survey Only, By Condition.....	173
	Text Condition .....	173
	Blocks Condition.....	173
	Hybrid Condition .....	173
	Post-Survey Only, All Conditions .....	174
E	CS1 STUDY: DEMOGRAPHIC QUESTIONNAIRE .....	175
F	CS1 STUDY: PERCEPTION QUESTIONNAIRES.....	177
	Personal Perceptions (Pre-Survey Only).....	177
	Blocks/Text Perceptions (Pre, Mid, Post) .....	177
	Hybrid Instruction Perceptions (Mid, Post).....	177
	Weekly Survey.....	178
G	CUSTOM ASSESSMENT .....	179
H	ITEM ANALYSIS: CUSTOM ASSESSMENT IN CS1 COURSE.....	208
I	ITEM ANALYSIS: SCS1 IN CS1 COURSE.....	209
J	CONDITION AND EXPERIENCE INTERACTIONS .....	210
K	PLUGIN EVENT COUNTS AND CATEGORY MAPPING .....	211
L	CS1 STUDY CODEBOOK AND RESULTS TABLE BY MODULE NUMBER .....	212
M	DISCUSSION WITH CURRICULUM COMMITTEE CHAIR.....	216
	LIST OF REFERENCES .....	218
	BIOGRAPHICAL SKETCH.....	229

## LIST OF TABLES

<u>Table</u>	<u>page</u>
2-1 Questions on SCS1 by Discrimination Factor & Difficulty (Parker & Guzdial, 2016)....	33
2-2 Summary of Visual Environment Affordances.....	45
2-3 Summary of Multimodal Environment Affordances.....	53
3-1 Interview Questions by Topic.....	56
3-2 Expectations of Perceptions based on Programming Experience.....	57
3-3 Programming Definition Themes.....	59
3-4 Percentage and Number of Participant Responses by Theme.....	60
3-5 Percentage of Participants Saying Constructs EASY for N > 4 (14.3%).....	61
3-6 Percentage of Participants Saying Constructs HARD for N > 4 (14.3%), & If / Loop.....	61
4-1 Number of Questions by Concept, Type, Difficulty .....	71
4-2 Questions Comparing Blocks & Text Programming.....	74
5-1 Course Topics & Mode for Instructional Intervention .....	97
5-2 Module Survey Questions (Weekly) .....	103
5-3 Demographic Groups by Condition.....	107
5-4 Measures by Research Question .....	109
5-5 Independent Variables .....	110
5-6 RQ1 – Dual-Modality Instruction and Question Performance - Hypothesis.....	111
5-7 RQ2 - Dual-Modality Instruction vs. Text Instruction by Experience - Hypothesis .....	111
5-8 List of Topics in Common by Exam .....	113
5-9 Dependent Variables.....	115
5-10 Time Window for Lecture Slide Usage by Module .....	117
5-11 Time Window for Plugin Usage by Module.....	118
6-1 Results Summary for Course Exams (Scores as Percent).....	121

6-2	Results Summary for SCS1 (Scores as Percent).....	121
6-3	Course Exam Interactions: Condition x Experience .....	128
6-4	Mean & Standard Deviation, Final Exam: Condition x Experience .....	128
6-5	Mean & Std. Deviation, Exam 1, Writing: Condition x Experience.....	130
6-6	Mean & Std. Deviation, Exam 2, Writing: Condition x Experience.....	131
6-7	SCS1 Interactions: Condition x Experience (See Appendix J for Means / Std. Dev.)....	132
6-8	“Dual Mode Instruction is Helpful”, Range by Experience .....	141
6-9	Common Codes and Examples .....	142
6-10	Responses: Why Dual-Modality Instruction is Helpful (n=63) (>5% of Students).....	142
6-11	Responses: Why Dual-Modality Instruction is Not Helpful (n=63) (>3% of Students) .	142
B-1	Doctoral Work Timeline (Chronological) .....	171
J-1	Mean & Standard Deviation: Condition x Experience .....	210
K-1	Table of Event Counts and Percentages by Module (Chronological).....	211
K-2	Mapping if Event Name to Event Category.....	211
L-1	Codebook: Why Dual-Modality Instruction is Helpful / Not Helpful .....	212
L-2	Table of Code Counts of Responses Indicating Instruction was Helpful, by Module ....	214
L-3	Table of Code Counts of Responses Indicating Instruction Not Helpful, by Module ....	215

## LIST OF FIGURES

<u>Figure</u>	<u>page</u>
1-1 “Hello world!” program in Scratch (blocks-based) and C (text-based) .....	18
2-1 Standard Item Curve. (Sudol, 2010).....	34
2-2 Guttman (a), Easy (b), Linear (c), & Descending (d) Curves (Sudol, 2010).....	35
2-3 Early version of AgentSheets (A. Repenning) [107].....	37
2-4 Blocks-Based Interfaces: LogoBlocks [54] .....	38
2-5 a) Alice environment [104] (left) and b) Scratch environment [77] (right).....	40
2-6 Hybrid modality environments: a) BlueJ [63] and b) Greenfoot [51].....	48
2-7 Pencil Code: Blocks-based mode, text-based mode, output window [9] .....	53
3-1 Eclipse IDE [141].....	63
4-1 Pencil Code architecture, with added Python-variant modules highlighted in gray .....	67
4-2 Pencil Code Python variant: Blocks-based mode, text-based mode, output window .....	67
4-3 Custom assessment: a) blocks / text variants (left) and b) isomorphic variants (right) ....	70
4-4 Timeline of time spent in text / dual / blocks modes by condition .....	70
4-5 Distribution of survey Likert responses.....	70
5-1 Amphibian Blocks Mode editor showing a) tabs for switching between modes, puzzle-piece connection, b) blocks representation of the current program, and c) block toolbox from which users can drag and drop constructs.....	92
5-2 Amphibian Blocks Mode editor showing a) Java object-oriented constructs and b) drop-down menus used for types and modifiers .....	93
5-3 Amphibian architecture with new elements highlighted in gray: a) Modifications to the Droplet Editor and b) Architecture of the IntelliJ Plugin.....	94
5-4 Example of switching from text to blocks mode: a) successful change to blocks mode and b) syntax error message.....	95
5-5 Instructional material – presentation in blocks, followed by conversion to text .....	97
5-6 Curriculum assignment documentation – sample code in blocks and text.....	98

5-7	Gantt chart showing date ranges for surveys, examinations, and SCS1 assessment.....	105
5-8	Definitional (left) and code reading (right) question samples from Exam 1 .....	112
5-9	Code writing question from Exam 1 (abbreviated).....	113
6-1	Boxplot of Final Exam scores by condition and prior programming experience .....	1288
6-2	Boxplot of Exam 1 writing scores by condition and prior programming experience .....	128
6-3	Boxplot of Exam 2 writing scores by condition and prior programming experience .....	128
6-4	Percentage of students indicating dual-modality instruction was helpful, by module....	141
6-5	Percentage of students accessing lecture slides and using plugin, by module.....	150
6-6	Percentage of events of each type by module.....	150

Abstract of Dissertation Presented to the Graduate School  
of the University of Florida in Partial Fulfillment of the  
Requirements for the Degree of Doctor of Philosophy

BUILDING BRIDGES: DUAL-MODALITY INSTRUCTION AND INTRODUCTORY  
PROGRAMMING COURSEWORK

By

Jeremiah J. Blanchard

August 2020

Chair: Lisa Anthony  
Cochair: Christina Gardner-McCune  
Major: Computer Engineering

Blocks-based programming environments have become commonplace in introductory computing courses in K-12 schools and some college level courses. In comparison, most college-level introductory computer science courses teach students text-based languages which are more commonly used in industry and research. However, the literature provides evidence that students may face difficulty moving to text-based programming environments even when moving from blocks-based environments, and some perceive blocks-based environments as inauthentic. Bi-directional dual-modality programming environments, which provide multiple representations of programming language constructs (such as blocks and text) and allow students to transition between them freely, offer a potential solution to issues of authenticity and syntax challenges for novices and those with prior experience in blocks by making clear the connection between blocks and text representations of programs. While previous research has investigated transition from blocks-based to textual environments, there is limited research on dual-modality programming environments.

The goal of my dissertation work is to identify how use of bi-directional dual-modality programming environments connects with learning in introductory programming instruction at

the college level. I have developed a bi-directional dual-modality Java language plugin and evaluated the use of said tool within an introductory computer science (CS1) course. In my work I analyzed understanding and retention of specific computing / programming concepts, how any connections vary according to prior programming experience, and in what ways dual-modality programming environments affect the classroom learning experience.

## CHAPTER 1 INTRODUCTION

When learning computer science, students must master several skills, including computational thinking, fundamentals of programming, and computer science theory [125]. To practically apply computer science skills, students must work in a language's semantic structure and syntax while learning about programming environments. They must also progress through stages of expertise over time. To draw on the Neo-Piagetian framework, students must develop expertise in programming by moving from the *sensorimotor stage* (in which they know what a program does, but not how it does it) to later *concrete* and *formal operational stages* (in which they can understand programs by breaking them into chunks and considering their abstract function) [71]. Finally, students must be able to translate their ideas into code that runs within the target environment.

Computer science instructors and educational researchers have recognized the positive role that appropriate scaffolding can play in programming instruction to help address these challenges; this has motivated the development of instructional programming environments to scaffold the learning of computational thinking [58]. For example, blocks-based environments, such as Scratch and Alice, were developed by researchers in computer science education to decouple the learning of syntax from programming, computational thinking, and computer science theory by allowing students to program without text, eliminating syntax barriers [54, 30, 77]. The elimination of syntax errors may contribute to a reduction in student cognitive load, allowing students to master computational thinking skills without needing to master syntax at the same time [72]. While blocks-based environments have shown promise in improving learning and perception by obviating syntax issues [89, 31], they do not currently address syntax challenges students must ultimately face when transitioning to production language



environments. Dual-modality programming environments, such as Pencil Code and its Droplet Editor, may be able to help bridge this understanding and help students delve into syntax by providing blocks and text representations of the same program [8, 7]. In my dissertation work, I investigate the use of blocks, text, and especially dual-modality programming environments for introductory CS learners at varied stages of their education, culminating in a study to evaluate how and if the use of dual-modality instruction in CS1 courses correlates with learning and how said instruction affects the classroom experience.

### **1.1 Research Motivation**

My motivation to conduct computer science education research springs from my experience teaching in K-12 classrooms and at the college level, especially concerning accessibility of computing education. Much of my early work focused on students' perceptions of programming and constructs (e.g., loops, variables, blocks). I explored how these perceptions related to student motivations, interest in, and learning of, computer science in blocks-based environments. As researchers began to note challenges that remained even when moving from blocks-based to text-based environments (such as difficulty learning syntax) [134], the focus of my work shifted to analyzing the perceptions of blocks and text.

My recent work aims to identify how to alleviate those remaining challenges (in student perception and learning) when transitioning from blocks to text, particularly in the CS1 course that I teach. Programming instruction has traditionally made use of text-based production languages [124], such as C++ [1], Java [111], and Python [38]. While this has the benefit of anchoring instruction in practical languages used in the industry, it presents difficulties for students. Even for languages with simple structure, syntax errors and semantics are a hurdle [136]. These language challenges are coupled with learning computational thinking and basic computer science theory, which compounds cognitive load in early CS instruction [72, 87].

Blocks-based languages are also now frequently used to teach computing in K-12 classrooms, so many students enter early programming courses with blocks-based experience (Figure 1-1) [10]. However, research accounts suggest that, even when starting in blocks, some students nevertheless struggle with the same syntax challenges of text-based languages as learners who begin learning in text [74]. In particular, students who move from blocks to text have noted the disparity in difficulty between some blocks-based environments and text-based languages [74]; my early work (outlined in later sections) provides evidence that students who move from blocks directly to text perceive text-based programming as more frustrating than those who work only in text from the start. Student frustration may also stem from an inability to connect blocks constructs and their text counterparts.

Dual-modality block-text systems, offering both text and blocks-based representations, were developed to provide a connection for students between blocks-based representations common in learning environments and text used in production languages [7]. These environments offer promise in that they may be able to help students overcome syntax challenges and reinforce semantics when moving between blocks-based and text-based representations. By linking textual and blocks-based modes of the same language, dual-modality blocks-text systems may facilitate chunking and abstraction in text [71] by visually nesting code blocks, such as function or condition constructs.

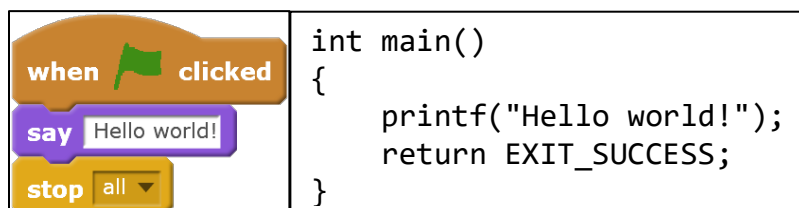


Figure 1-1. “Hello world!” program in Scratch (blocks-based) and C (text-based).

## 1.2 Research Opportunities

Dual-modality programming environments are relatively new developments and there is limited research into their effectiveness and connection to learning, with only a small number of studies conducted, and many of those on a small scale [136, 134, 9]. In particular, David Bau's work with Pencil Code – an educational, web-based programming platform – studied how students use bidirectional dual-modality programming environments [9]. Bau conducted a study with eight public middle school students across four after-school sessions who had no prior programming experience. Students were permitted to freely use blocks and text modes. While on the first day, all students used blocks most of the time, they progressed to using text more often each day. By the last day, they were working in text 95% of the time, of their own volition. It is possible that this transition occurred because students became more confident working in text, suggesting that they no longer perceived text as intimidating. While Bau's work did not directly address perception or learning, it did offer insight by providing evidence that suggested the students became more accustomed to text representations in dual-modality programming environments.

David Weintrop's is among the most comprehensive body of work on the subject of dual-modality programming environments [136, 134]; he studied students moving from dual-modality programming environments to text-only development. Weintrop's study was conducted over the course of a year with 90 students enrolled in a public high school's introductory programming class. Weintrop developed mechanisms – assessments, surveys, and tool log files – to measure student attitudes, perceptions, and conceptual understanding in blocks and text modes. His work also compared how student experiences differ by starting in pure-text, pure-blocks, and dual-modality programming environments before moving to pure-text environments. Weintrop's studies suggest that dual-modality programming environments provide some of the affordances

of both blocks and text – they helped foster confidence in students (like blocks) but are also perceived as authentic programming experiences (like text languages). The dual-modality programming environment used a different language (JavaScript) than the development in pure text (Java) [134] due to the curricular requirement to teach in Java and tool limitations (which were only available at the time in JavaScript and CoffeeScript variants).

These important studies open new questions and opportunities for research. It is an open question as to whether changing languages in addition to changing environment modes—as was done in Weintrop’s study—may constitute a significantly higher cognitive load [118] than merely transitioning from a dual-modality programming environment to a text-based one. It may also inhibit transfer [126], as the differences between the languages could constitute farther transfer [6] in comparison to working within the same language, but merely a different environment. As blocks-based languages are now frequently used to teach computing in K-12 classrooms [10], an evaluation of dual-modality programming environments—and in what ways they provide an effective bridge to text—is particularly useful at the college level, where most students first encounter textual languages.

### 1.3 Research Questions & Hypotheses

Previous work studying dual-modality and multimodal environments opened new questions to explore regarding their potential use and place in education which have explored as part of this dissertation. Specifically, my work addresses the following open questions in the literature:

**RQ1. How do students perform in code reading and writing after learning with dual-modality instruction, as compared to students learning with traditional (text-based) approaches to instruction in CS1 courses?** Learners progress through multiple stages of development as they grow via problem solving practice to eventually become experts [71]. The

benefits of dual-modality compared to pure-text environments are likely to differ depending on the student's current state of cognitive development. Some research has been done to evaluate student perceptions and patterns in classrooms using dual-modality programming environments [134]; in my own work with middle school students, those who worked in dual-modality programming environments held positive perceptions of text more often than those who moved straight from blocks to text [14]. However, no comparative analysis of learning outcomes as compared to traditional (text-based) environments has been done. Examining how dual-modality instruction connected to student cognitive development, and in what conditions, would help advance early computer science instruction.

**RQ2. How does prior programming experience affect students learning in dual-modality instruction as compared to students learning in traditional (text-based) approaches to instruction in CS1 courses?** Students in early computer science courses are a diverse population with different experiences. Some have prior text experience programming, some have worked in blocks, and others have none. In my early research, students with prior programming experience more frequently held nuanced perceptions of programming compared to those who had none [15]. Short of offering completely separate or tailored instruction for each student, introductory courses must find effective ways of serving all of these populations. Identifying how these environments can support learning and how that support might differ based on prior experience can help instructors and researchers improve student experiences. By helping instructors and researchers tailor tools and instruction to students with different varied programming backgrounds, we can create development-appropriate experiences for students to engage with programming and computer science content to build their skills and knowledge.

**RQ3. What are student perceptions of dual-modality programming environments and instructional approaches, and how do they change over time, in the context of a CS1 course?** There are reports in study interviews that suggest some students perceive text languages as hard and intimidating [54]. Blocks-based environments have shown promise in alleviating the syntax challenges when learning programming concepts [133], but some students who work in blocks continue to struggle with negative perceptions when they move to text [74]. My early work showed that some students who move from blocks directly to text find the experience frustrating [14]. If dual-modality programming environments alleviate these negative perceptions, they may contribute to improved motivation and confidence, which have been shown to improve retention within the discipline [80]. Identifying how the student and instructor experience change when using dual-modality instruction would provide guidance for instructors considering their use in the classroom.

#### **1.4 Overview of Work**

My dissertation work aims to identify how dual-modality (blocks-to-text) learning environments support computer science instruction, how learning outcomes change based on prior programming experience, and in what ways they affect the classroom learning experience. Research to investigate the efficacy of dual-modality programming environments would be a valuable contribution to computer science education because, if it can be shown that they are effective in helping students learn computer science, they can be used to bridge from blocks-based learning environments—which provide helpful scaffolding for novices—to production languages which are used in research and industry [78, 81]. My work seeks to provide the tools to facilitate use of dual-modality programming environments, an empirical study of how they support learning, and evidence of their effectiveness within a classroom setting. To measure support of learning in dual-modality programming environments:

- I have developed a general-purpose Java language IDE plugin providing blocks-to-text transition in real time (and vice versa) based on Pencil Code's Droplet editor;
- I have constructed custom dual-mode representation curricula materials with text and blocks representations, for use in UF's CS1 course; and
- I have conducted a study in which I collected survey responses and assessment data from a baseline group taught using traditional, text-based approaches to instruction in UF's CS1 course, and separately an intervention group using dual-modality tools and instruction in a different semester of the same CS1 course.

My contributions include a) an analysis of dual-modality tools and curricula and how they support the learning of computing concepts; b) a dual-modality blocks-to-text IntelliJ IDEA plugin for the Java language; and c) analysis of student perceptions of dual-modality instruction and the classroom experience from my perspective as an instructor implementing the use of dual-modality tools and instruction in an introductory CS college classroom.

The first part of this dissertation describes the relevant background (Chapter 2). This is followed by three descriptions of three related but distinct studies – perceptions of programming (Chapter 3), perceptions of dual-modality programming environments (Chapter 4), and learning in dual-modality programming environments (Chapter 5). The final section describes my findings (Chapter 6), work timeline (Chapter 7), identifies my contributions (Chapter 8), and ends with a conclusion (Chapter 9) establishing how my work fits into the body of literature within computing education.

## CHAPTER 2 BACKGROUND

Computer science education research is built upon a fusion of general educational theory with computer-science-specific practice and research. In this section I outline the background work which builds upon traditional educational theory to develop CS-specific approaches and tools.

### 2.1 Learning to Program

Learning to program requires concurrent development of multiple skills. In addition to general problem solving and computational thinking competency, students must also learn theory. At the same time, students must evolve through the stages of expertise development – from seeing programs as a black box to advanced chunking and abstraction [71]. These abilities are distinct and each carries its own **cognitive load**—that is, mental effort in working memory required when solving problems [118, 95]. Programming has also historically been taught using text-based languages, meaning those challenges specifically associated with learning syntax often surface in early programming instruction. Additionally, in order to improve the efficacy of programming instruction, it is helpful to have measurement instruments to identify if and to what degree learning has taken place [117]. This section identifies the general challenges of learning to program, as well as those specific to text-based instruction, and describes the development of concept inventory instruments that can be used to assess learning in programming.

#### 2.1.1 Language Challenges

Text languages, which are commonplace in industry and college education, present challenges to the novice programmer with respect to syntax and semantics. This section addresses these challenges from the perspective of the development stages that learners go through as they progress from novice to expert.



### 2.1.1.1 Syntax

Language syntax has been recognized as a barrier students face when learning computer science [72]. McIver and Conway developed the GRAIL language in an attempt to minimize syntax errors; the main goals were to **maximize readability** and **minimize unproductive errors** [86]. In a follow-up, McIver studied students using GRAIL and LOGO to compare error rates [85]. Participants in this study were provided with development environments that were identical, with only the languages differing; the participants were given a series of eight exercises to complete. When participants attempted to run their programs, a snapshot of the program text was captured and stored, and program errors were then analyzed and split into syntax errors (such as use of an incorrect keyword) and logic errors (such as an algorithm with incorrect steps). McIver found that students using GRAIL—designed to minimize syntax problems—had not only a lower number of syntax errors, but also a lower number of logic errors, suggesting that students who face fewer syntax challenges can reduce their logical errors as a result.

Later, Ko et al. identified syntax as a contributing factor in four of six learning barriers they examined within programming systems [61]. They found that identifying the right interface for a task—such as when a user knows what task to accomplish but cannot determine (select) the correct construct—created *selection barriers*. For example, a user in a simulation environment may know that a task requires moving a character to a particular location on the screen but may have difficulty identifying which syntax will accomplish that goal. As construct names / text are not necessarily intuitively tied to what they do, syntax challenges can act as *use barriers* – even if a user knows that an “if” statement provides conditional execution, they may not be able to properly construct the conditional expression. Difficulty in knowing which constructs can connect to and work with one another establish *coordination barriers*, such as trying to break out

of a looping expression using a “switch” statement (which cannot be done in many languages); and misassembly of constructs can result in difficulty understanding why a program does or does not do something (*understanding barriers*) – a user who misunderstands operator precedence may miscalculate a value by arranging the operators incorrectly.

### **2.1.1.2 Semantics**

In addition to unique syntax, each programming language construct has specific semantics – the meaning of the construct. Deciphering this meaning requires an understanding not only of the syntax of the language, but also the overall context of the construct within a block of code. Programmers at concrete and formal operational stages of reasoning development, when reading code, perceive text as a composition of constructs using an internal mental model of those constructs – they chunk out blocks of code and summarize their meaning [71]. Students in the preoperational stage, by contrast, see lines of code as individual elements, rather than as abstract chunks; this increases their cognitive load and limits their ability to reason abstractly [71]. Ko’s work also noted that semantics were a key aspect of two of the six learning barriers, specifically use barriers (such as using the wrong parameters) and coordination barriers (e.g., misunderstanding how constructs interact with one another) mentioned previously [61]. Semantics also play a critical role in debugging, as practitioners must read, trace, and develop abstract models of sections of code [2].

Taken together, syntax and semantic challenges represent a potentially significant hurdle for students to overcome. Further exploration of the relationship between syntax / constructs and perceptions of difficulty and intimidation could help researchers and teachers address those negative perceptions that may impact student interest and learning.

### 2.1.1.3 Perceptions

Text-based languages, especially production languages, have been noted as presenting perception-based challenges; the literature suggests that text-based languages can be intimidating, especially for women [12]. Begel and Klopfer, in developing StarLogo TNG, conducted focus groups to identify the strengths and weaknesses of the previous and new platforms; they found that women and girls consistently felt intimidated by (text-based) programming languages, who viewed them as male-oriented [12]. This may impact future motivations to study computer science [59]. Results also suggest that text-based languages suffer from association with “uninteresting” tasks [134]. Association with uninteresting / boring / “uncool” work tasks has also been implicated in limiting motivation among students in minority populations [37]. Visual languages attempt to address many of these issues by making the environments more inviting and approachable and by incorporating games, simulation, and multimedia [107, 54, 104].

### 2.1.2 Developing Expertise

Early attempts to understand development of programming skills followed the constructivist / Piagetian tradition [96]. In this framework, students learn by constructing their own knowledge via assimilation (bringing new information into existing frameworks) and accommodation (reframing mental representation to match new experiences). The Piagetian framework, however, is closely tied to mental development and age-based maturity [20]. In Piaget’s framework, the *sensorimotor* stage encompasses infancy (through age of 2) and is characterized by a lack of internalized thinking, while the *preoperational* stage is described as intuitive (rather than logical) and lasts until the age of 7. In the concrete operational stage (through age 11), children apply logic, but only to their concrete inputs; finally, in the formal operational stage (from age 11), children can reason fully in the abstract. Lister instead proposed

applying the Neo-Piagetian framework – which decouples age and maturity from development of skills – to account for cognitive development in the domain of programming [71]. Lister proposed four stages of reasoning development in programming within this framework: sensorimotor – knowing programs produce a result, but not why, preoperational – understanding lines of code, concrete operational – reasoning about familiar, real-world situations, and formal operational – reasoning about unfamiliar, hypothetical situations. This framework is supported by empirical evidence from Corney et al [32] and think aloud studies by Teague et al [120]. In this section I briefly summarize each stage as it relates to programming skill development.

It is established in psychology literature that humans have limited capacity in their short-term or **working memory**, as argued by Miller in 1956 [90]. In order to cope with these limitations, experts employ chunking as a mechanism to recall information and ideas [45]. Information is broken into chunks which are stored in long-term memory; these chunks can be recalled as a single concept in working memory, reducing the number of unique ideas that must be in the working memory at a particular moment in time, and thereby reducing cognitive load [45]. As such, the development of chunking and abstraction methods is tied intimately with the evolution of advanced stages in the Neo-Piagetian framework of development.

#### **2.1.2.1 Sensorimotor**

In the sensorimotor stage, students see programming as a “black box” – they know the code produces a result but do not see the executing program as a sequence of instructions on a machine. They lack conceptual understanding of constructs and programs, even at a definitional level. Lister (and later Corney et al) identified students in this stage as those who could not read code and trace its execution with at least 50% accuracy [71, 32]. Students in this stage, lacking an understanding of the constructs themselves, are unable to engage in abstraction.

### 2.1.2.2 Preoperational

Students in the preoperational stage understand the function of individual lines of code. They understand the deterministic nature of computer programs functionally and conceptually – that is, they understand the *definition* of constructs – but often cannot summarize sections of code to determine overall meaning or function. According to Lister’s framework [71], these students can accurately read code and trace its execution with at least 50% accuracy but struggle to relate the function of lines of codes with respect to one another, to explain what a section of code does, or to develop diagrams describing the function of code. They may be able to write simple programs but cannot meaningfully think abstractly about programs.

### 2.1.2.3 Concrete-operational

Concrete operational reasoning requires the ability to engage in abstraction and to understand the meaning of sections of code as they relate to concrete and familiar situations. Students in this stage of development can read, trace, and write code. They can also engage in abstract thinking about programs, explain blocks of code, and draw diagrams describing code, but are restricted to those situations with which they have experience – they typically cannot abstract away solutions and apply them to distantly related problems – i.e., apply them to a new task. Notably, the McCracken working group identified abstraction as a key challenge students continue to struggle with at the end of most introductory computer science (CS1) courses [84]. Being able to break code into sections, and then evaluate the function of that code as a whole – rather than merely tracing code execution – is a fundamental distinction between preoperational and concrete operational development stages. This chunking mechanism facilitates abstraction of code into ideas that do not require line-by-line tracing [71]. Lister also argued that students in this stage understand three key properties – *reversing*, *conservation*, and *transitive inference* [71]:

- **reversing** – computational operations can be “undone”; e.g. after shifting items in a list to the left, the operation can be reversed by shifting the same items in the list to the right.
- **conservation** – equivalence of code across transformations that maintain the specification (targeted task); e.g., equivalence of two programs that find a minimum value in a set of values.
- **transitive inference** – relationships in data (often math); e.g., if a program organizes data to guarantee that  $x > y$ , and separately that  $y > z$ , then it also organizes the data such that  $x > z$ .

#### 2.1.2.4 Formal operational

As the most developed stage, formal operational reasoning is, in the words of Corney et al, “what competent programmers do, and what we’d like our students to do” [32]. These students can read, trace, and write code; they understand the constructs conceptually; and they can reason abstractly about programs. Lister described these students, based on the work of the McCracken working group, as capable of engaging in abstraction and deconstruction in order to develop solutions and iterate on them [71, 84]. In addition to being able to reason about familiar situations, persons at this stage of development can reason abstractly about unfamiliar ones.

#### 2.1.2.5 Summary

The Neo-Piagetian framework suggests students progress through four stages of development as they move from novice to expert in a field such as computer science: **sensorimotor** (seeing a program as a “black box”), **preoperational** (understanding lines of code and being able to trace execution), **concrete operational** (able to apply abstractions of solutions in similar situations), and **formal operational** (able to apply abstractions of solutions to unfamiliar situations). Research has suggested that while most students progress beyond sensorimotor levels in CS1 courses [73], the majority are at preoperational or concrete operational stages (with most showing a limited degree of concrete operational thinking) [32].

Prior research provides clues as to intervention strategies that may be applicable when helping students progress in the development stages. Code tracing has been identified as a key skill differentiating sensorimotor and preoperational stages of development, while abstraction is noted as critical to concrete and formal operational stages. Evidence suggests that techniques such as **lightweight sketching** – stepping through instructions while using a written *memory table* to track variable values (rather than trying to keep them memorized) – helps students learn to read code and trace through programs [138]. In his consideration of abstraction in computing instruction, Kramer suggested abstractions could be effectively taught by building on the work of Huitt and Hummel [55] - namely, by having students explore hypothetical questions, encouraging them to explain their problem-solving process, and by approaching instruction from a conceptual (rather than fact based) perspective [68].

## **2.2 Programming Assessment**

Several approaches can be used to evaluate student learning in technical fields, including examination of artifacts created by students and formal assessments [52, 70]. Standardized assessment instruments, if developed in a way that makes strong arguments for their validity and reliability, can provide compelling evidence of the effectiveness of approaches to instruction [122]. It is also important to be able to identify the validity of an instrument within a specific context by analyzing questions on an assessment individually and collectively [123]. This section identifies key aspects of learning assessment and instruments used for this purpose.

### **2.2.1 Concept Inventories**

A concept inventory is one type of instrument that can be used to measure competency. Tew and Guzdial proposed a language independent assessment of CS1 concepts [123]. Tew and Guzdial proposed a multi-step process to define the test's content and verify its validity and reliability. Tew later developed the Foundational CS1 (FCS1) Assessment to evaluate basic

computer science competency in a language independent manner [122]. The FCS1 is made up of multiple-choice questions that are categorized as definitional, tracing, and code-completion questions. Content of the exam was defined by an examination of topics from textbooks used in CS1 courses and ACM/IEEE guidelines. The topics covered by the test include variables, operators, program control, arrays, and recursion. Tew conducted three separate studies to verify programming language independence [122].

Building on the results of the work by Tew and later the 2013 ITiCSE working group under Utting, Parker and Guzdial developed an isomorphic version of the FCS1 in order to expand on the instruments available to the research community [98]. Isomorphic variants of questions are developed by changing variables and answer choices but keeping the topical area and style consistent with the original [98]. This new instrument, the Second CS1 Assessment (SCS1), was developed to mitigate the risk of saturation of any one assessment (and any impact on its validity). In a study with 183 participants, Parker and Guzdial found a strong correlation between participant scores on the FCS1 and the SCS1, which were given to participants one week apart from one another, and argued on this basis that the SCS1 is valid.

It is notable that both the FCS1 and SCS1 questions are, on average, considered very difficult, and not all questions provide the same level of discrimination (Table 2-1). Most questions on the assessment (22 of 27) were answered correctly by less than 50% of the participants, and none of the questions were considered easy (85%-100% answering correctly) [98]. There were also limitations with respect to discrimination quality of questions, with 7 of 27 questions considered to be *poor* discriminators (discrimination factor of less than 0.1), 15 of 27 considered *fair* discriminators (factor of 0.1-0.3), and only 5 of 27 considered *good* discriminators (factor greater than 0.3). Luckily, these limitations can be addressed, depending



on the circumstance. While the SCS1’s difficulty poses challenges when measuring lower levels of performance, this difficulty also means that the assessment has a higher ceiling – i.e., there is more “room” for high performance to be measured. In addition, the majority of the questions (20 of 27) provide fair or better discrimination.

Table 2-1. Questions on SCS1 by Discrimination Factor & Difficulty (Parker & Guzdial, 2016)

Discrimination Factor	Hard (< 50%)	Medium (50% - 80%)	Easy (85%+)	Total
Poor (<0.1)	7	0	0	Data
Fair (0.1-0.3)	14	1	0	Data
Good (0.3+)	1	4	0	Data
Total	22	5	0	Data

### 2.2.2 Item Response Theory

Item response theory (IRT) is a common way that an argument for the reliability of an instrument can be made. It is rooted in the probability that a person of a certain ability level will score correctly on a particular item on an instrument (such as a question on a test) based on a response curve [5]. IRT is used around the world for large-scale assessments, including extensively in research and use by Educational Testing Service (ETS), which develops and administers the SAT, GRE, and AP examinations [25]. Two commonly used IRT models are one-parameter (1PL/Rasch) and two-parameter (2PL) logistical models. IRT can be applied to a set of data via *item analysis*. An item analysis can be performed on questions to identify the **difficulty parameter** of a question and, if a multi-parameter logistic model is used, a **discrimination parameter** (which measures how well an item discriminates between those of higher and lower ability – also called a **slope parameter**) [101].

Sudol and Studer presented one approach to item analysis of a set of response samples using the R language. Their work allows researchers to easily build graph visualizations of difficulty and discrimination on a per-item basis [115]. Sudol and Studer also described several

item curves; these graphs plot ability vs probability, where mean student ability is zero. First among the described curves is the Standard Item Characteristic Curve (Figure 2-1). They suggested that items (questions) that best discriminate among average participants mirror this graph. These questions have a steep and positive slope at zero in ability, indicating that performance increases with ability. Most performance difference is within one standard deviation from the mean in such questions. They also discussed a Guttman Item curve (Figure 2-2a), which is seen when measuring knowledge that is likely recall-based or when there is poor item fit, and an Easy Item curve (Figure 2-2b), where most participants – even those with low ability – perform well. Sudol and Studer also described problematic curves. These include Linear Items (Figure 2-2c), which may indicate mixing of multiple concepts into a question – a violation of the assumptions of the model – and possibly other problems, and Descending Curves (Figure 2-2d), which indicate an inverse relationship between performance and ability. Using Item Analysis, instruments such as concept inventories can be evaluated for reliability and validity with different populations.

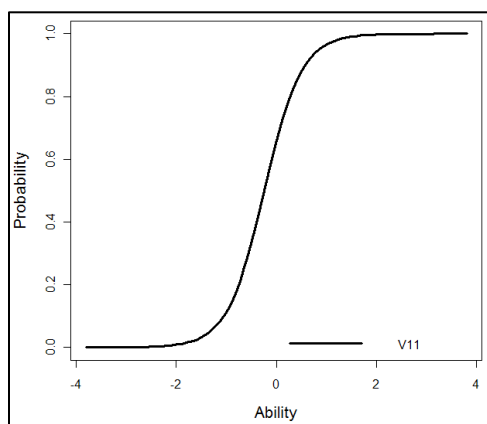


Figure 2-1. Standard Item Curve. (Sudol, 2010).

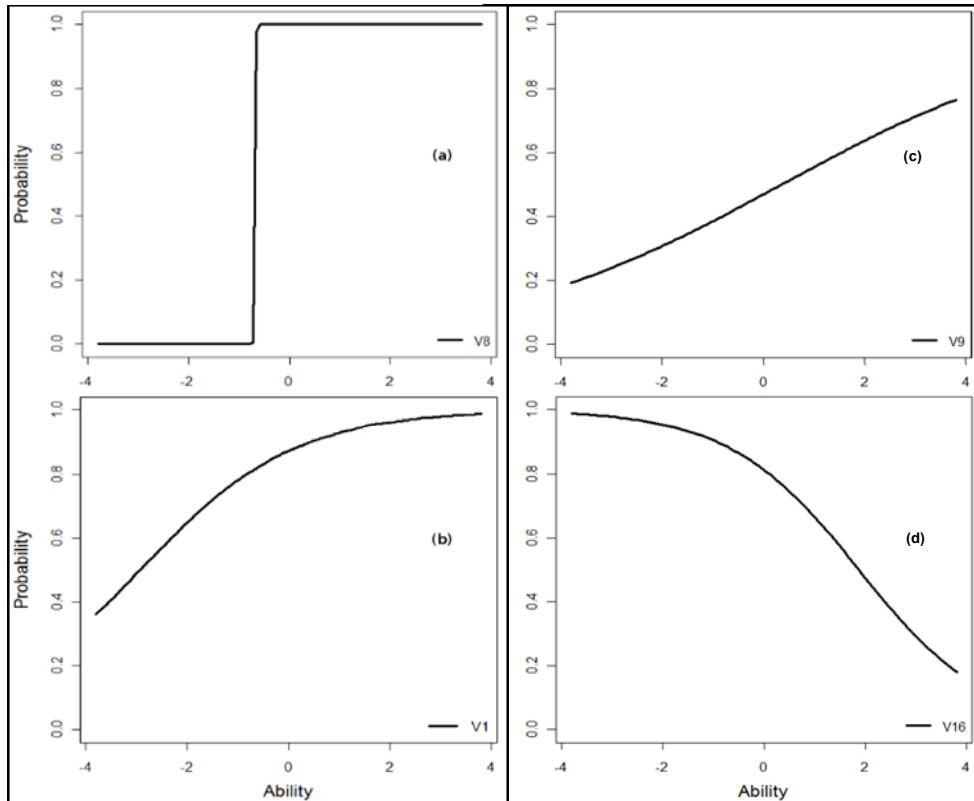


Figure 2-2. Guttman (a), Easy (b), Linear (c), & Descending (d) Curves (Sudol, 2010).

### 2.3 Visual Languages & Environments

In addition to providing a pedagogical framework and methods of measurement, instructional tools and environments can help facilitate learning in computer science and other fields. To address the challenges of text-based languages for novices and provide scaffolding, visual and (especially blocks-based) languages have been in development that have shown promise in helping more students learn computer science concepts [31, 34]. The earliest of these are graph systems, which evolved from flow charts and diagrams, and were intended to be accessible to non-programmers [56, 83]. Later systems integrated simulations and were rooted in the constructionist philosophy; they intentionally provided an area for play and engagement with learning as an explicit target [96]. Modern blocks-based languages incorporated the ideas of these earlier systems, but also added additional scaffolding to facilitate educational goals

[54]. These blocks-based environments usually use colorful palettes to create an approachable environment for novices of computer science and denote semantic roles (such as control or statement blocks). They also often use visual and/or audible cues (such as puzzle-piece connectors and clicks) to convey construct connections and semantics [54].

### **2.3.1 Development of Visual Programming Environments**

Visual environments and representations of algorithms were inspired by the desire to make programming more accessible and easier to understand [54, 56, 83]. Modern visual environments aim to decouple syntax from algorithmic thinking (through the use of visual constructs that snap together), reduce intimidating perception (through friendly color schemes and recognizable shapes), and introduce interesting functionality to boost motivation (via multimedia integration) [77, 54, 28]. This section evaluates the historical motivations of these visual languages—many related to the challenges of text-based instruction—and examines their design, application, and evaluation. It also considers their evolution over time, evaluations of their efficacy, and open questions in the literature regarding them.

#### **2.3.1.1 Early visual programming systems**

Work in visual programming environments evolved in part from visual flow charts and diagrams, such as Prograph and Fabrik [83, 56]. These early system designs were in part meant to create executable variants of data and control flow diagrams (a la flow charts). Graph-based symbolic systems laid the groundwork for object- and agent-based visual simulation frameworks as the object-oriented paradigm developed and influenced computer science research and languages [109, 11]. These simulation environments were designed around object (agent) manipulation and constructionist philosophy, which is based on learning through the building of mental models over time through exploration [107, 112, 97]. KidSim and AgentSheets (Figure 2-3) integrated spatial and temporal affordances through which agents could be created and

modified via grid-structured containers [108, 33]. These containers were the early precursors to blocks-based programming systems [43].

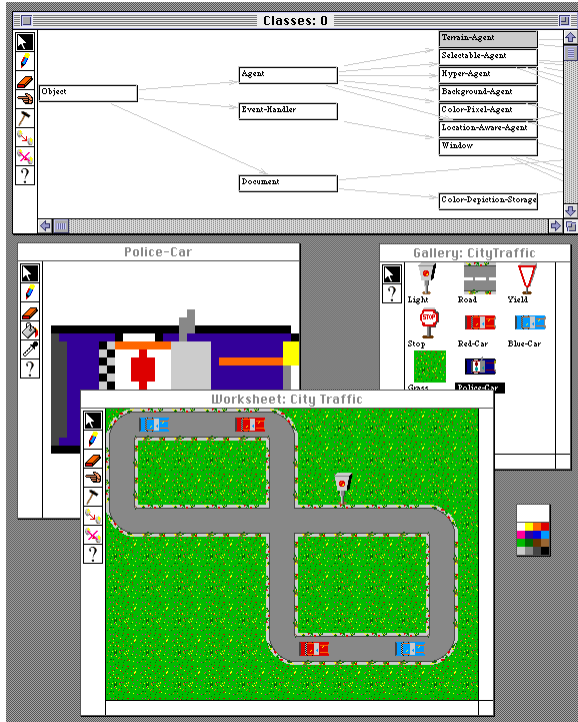


Figure 2-3. Early version of AgentSheets (A. Reppenning) [107].

### 2.3.1.2 Blocks-based interface development

At MIT, systems inspired by robotics were being developed with a focus on accessibility to wider audiences. These eventually gave birth to blocks-based programming environments, beginning with LogoBlocks. LogoBlocks (Figure 2-4) is a puzzle-style, blocks-based programming environment, built on the ideas of prior diagram-based and agent-based systems [54]. LogoBlocks was based on earlier work on LEGOsheets, which itself was based on AgentSheets [43]. LogoBlocks also was created to serve as a development environment for the LEGO Programmable Brick [11]. LogoBlocks programs are written to be perfectly translatable into Brick Logo, a variant of Logo used for the LEGO Programmable Brick. It was paired with a compiler that first converted LogoBlocks programs into Brick Logo.

The design work in LogoBlocks focused on visual affordances and cues to facilitate understanding. Early iterations of LogoBlocks used a grid-based model a la Agentsheets [107]. Later iterations, seeking to depart from the rigid nature of the grid-based approach, removed the grid itself. To maintain spatial relationships between elements, visually interlocking connectors were added. Connectors varied by block type. For example, action blocks in a sequence were listed vertically; each action block contained an ACTION\_TOP and ACTION\_BOTTOM connection point. The ACTION\_TOP connector of one block could connect to the ACTION\_BOTTOM of another block. Blocks with matching (complementary) connection points were also built to snap together (both visually and audibly) when in close proximity; this allowed for more free form use of the blocks. To make blocks easily identifiable, each block type had a unique color, shape, and label [54]. The latching mechanism was devised to simplify connection of constructs and relieve students of the need to precisely place blocks on the canvas.

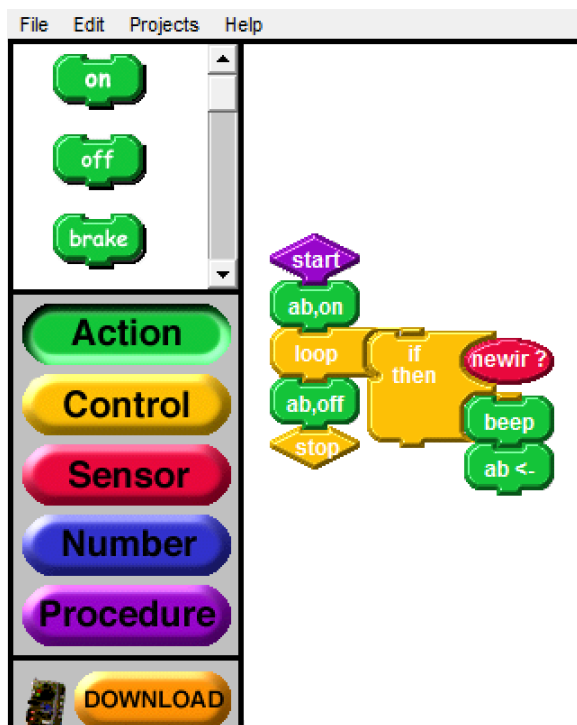


Figure 2-4. Blocks-Based Interfaces: LogoBlocks [54].

### 2.3.2 Contemporary Blocks-Based Environments

While work on LogoBlocks progressed, simulation-based, object-oriented development—and work to make it more accessible to a wider audience—continued with the Alice environment (Figure 2-5a). Alice was at first developed as a scripting environment for 3D computer graphics for storytelling [104]. Alice focused especially on usability concerns (principally, the *law of least astonishment*) for the non-programmer audience; its principal goal was to reduce the dependency on mathematical underpinnings when working in 3D graphics [27]. To this end, Alice abstracted hardware, graphical APIs, and linear algebra into a simplified, “plain-language” programming interface [26]. Later work on Alice focused on its potential as a platform for learning to program and for practicing algorithmic thinking by building and scripting 3D worlds, as it allowed for exploration with real-time visualizations [34]. While Alice started out as a text-scripted environment in Python, later versions converted to a blocks-based approach to sequence commands in an object-oriented style inspired by Java [29], and as of Alice 3, visually constructed code is perfectly translatable in Java to make transition to Java from Alice easier [23].

Scratch [77] (Figure 2-5b), started in MIT’s Media Lab, built on the puzzle-piece-style block constructs of LogoBlocks [54] and real-time visualizations of Alice [27]. While the environment of Scratch was 2D (compared to Alice’s 3D environment), it incorporated many of the usability features from both Alice and LogoBlocks to create a platform that was friendly and accessible to children [77]. Scratch’s environment featured the same interlocking and snap-together mechanisms developed and refined in the LogoBlocks project [54] and added the multimedia elements from Alice [104]. Scratch’s web-centric design encouraged students to share their creations with other users, helping users to develop a vibrant community based on

remixing the work of others [92]; this practice has been shown to increase interest in continuing to program [37] and helped disadvantaged students connect as a community in local computer clubs [76]. Scratch was also explicitly designed to include parallel execution affordances as a main feature; the event-driven structure of the system allows multiple instruction threads to execute concurrently, usually engaged through input or messaging between agents in the environment [110].

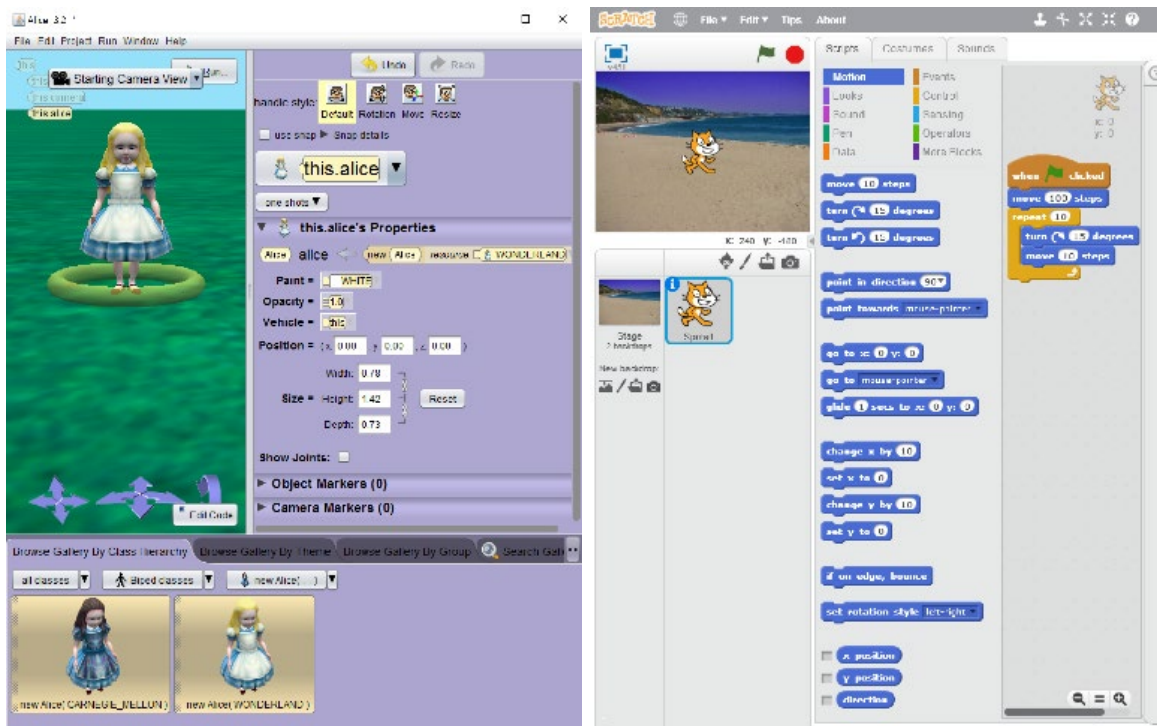


Figure 2-5. a) Alice environment [104] (left) and b) Scratch environment [77] (right).

### 2.3.3 Efficacy of Blocks-Based Environments

Effectiveness of these environments can be measured according to different criteria and by using a few different metrics. For example, the goal may be to see if the environment can be an effective instructional tool; to compare the performance of students using a new environment or language and one already in use; to prepare students for more advanced programming and computer science coursework; or it may be some combination of these. The scaffolding provided



by these environments aims to reduce or eliminate syntax errors (by eliminating syntax through the use of blocks), facilitate understanding of semantics and construct relationships (via colors and puzzle-piece-style snapping), and improve motivation and interest (by providing multimedia environments to work within).

### **2.3.3.1 Effectiveness as learning environment**

In general, these tools have been used successfully to help students to learn. Studies have provided evidence that the scaffolding provided by these environments eliminates problems with syntax errors, allowing students to employ computational thinking, while also nurturing an appreciation of trial-and-error approaches necessary for programming and debugging [75, 132, 94]. Some studies, particularly those with Alice, also show students developed a strong sense of objects and their contexts within programs [31, 131]. Cooper et al. ran an experiment with a small number of college students (N=21) identified as weak CS majors (those who had no prior programming experience and were not prepared for calculus) to test the effectiveness of Alice as an environment to help students learn computer science [31]. Eleven (11) students took an Alice-based preparation course, either before or alongside CS1; ten (10) students did not and served as the control group. The students who took the Alice-based course in addition to CS1 performed better in the CS1 course, suggesting a prior course in Alice can help students succeed in CS1 courses. However, as the authors noted, these results may be biased by self-selection of motivated students into the Alice course. Meerbaum-Salant et al. evaluated instruction using Scratch to teach 9<sup>th</sup> grade boys and girls (N=46) [89]; they found that students using Scratch improved their knowledge of computer science and programming through its use. They also found that students internalized core CS concepts (including initialization, loops, variables, and concurrency) and improvement in cognitive performance levels (including understanding, applying, and creating). However, the impact was not uniform – some concepts were more

readily internalized than others. For example, while 75% of participants correctly answered questions regarding *conditional* loops, only 52% answered correctly regarding *bounded* loops.

With respect to effectiveness compared to other environments, results diverge somewhat depending on the circumstances and are mixed. Wang et al. conducted a quasi-experiment at the high school level with two groups of students (N=166), one learning C++ and the other learning via Alice [131]; they found that students in the Alice coursework performed slightly better overall than those in the C++ group. However, in a study comparing students (N=154) receiving instruction via pseudocode and those using Alice, Garlick and Çelikel found that students using Alice performed more poorly than those learning via pseudocode [42]. It is possible that method of instruction or differences between the C++ and pseudocode approach played a role in the difference in results; more research may help elicit in which situations Alice and similar environments may be helpful. Weintrop studied the difference between blocks-based and text-based environments with two groups of students in high schools. One group started in a blocks-based environment, while the other started with text (JavaScript or CoffeeScript); at the midpoint, both groups changed to the Java language in text. He found that those students who worked in the blocks-based environment outperformed the text-based group on an assessment of algorithm construct concepts used in programming at the midpoint (before the switch to Java). However, their scores at the end of the course—after the completion of the Java portion—were comparable; in other words, the students were not worse off, performance-wise, for having used the blocks-based environments.

### **2.3.3.2 Moving from blocks to text**

While studies have shown that blocks-based environments can be effective learning tools, some data suggest that students struggle to switch from blocks to text later. In Weintrop's study [134], participants starting in blocks reported statistically significant increased levels of

confidence from the start to just before changing to Java (the midpoint of the study), but a statistically significant decrease in confidence in the second phase (after switching from blocks to text). The text-condition students, however, did not show statistically different changes in confidence over the course of the study. One possible explanation may be that, while blocks-based environments boost morale and motivation—and possibly early learning—over time this benefit decreases as students become more accustomed to computational thinking and thus get less benefit from the reduced cognitive load of the blocks-based environments. It is also possible that the motivational challenges of text-based languages impact performance in similar ways regardless of when they are introduced (whether before or after learning fundamentals), possibly related to perceptions of difficulty and authenticity, resulting in the same overall ability level in participants upon completion of the entire course. Data collected during my own study with middle school students (Section 4.3) suggests that students who start in blocks perceive text more negatively after switching from blocks to text, unlike their counterparts who have worked exclusively in text. The scaffolding provided by blocks-based languages is intended to address issues related to perception and the challenges of syntax; however, more research is needed to determine how this scaffolding impacts student learning and motivation in the long term. It is also notable that, while Weintrop's results showed improvement in student performance on ability tests while in the blocks-based environments, once students moved to text, the learning outcome advantage dissipated. While not the primary topic of this dissertation, additional study of how switching students from blocks to text environments compares with teaching them exclusively in text environments could help identify the impact these environments have on learning over longer periods. This dissertation will help provide a foundation for later research comparing purely text-based and blocks-to-text approaches to instruction.

### **2.3.3.3 Special considerations - environments are not equal**

It is important to note that the blocks-based environments discussed are targeted to different age groups. For example, while Scratch targets elementary through early high school students, Alice was originally created to serve students at the undergraduate level, though today it is used in K-12 classrooms [26, 75]. Design differences change the implementation of scaffolding and structure of the language representation. For example, Scratch makes use of brightly colored, snap-together puzzle piece blocks meant to appeal to younger children, while Alice utilizes simple rectangular blocks in more subdued colors [77]. Likewise, Alice's variants employ an explicitly object-oriented data model, while Scratch's interface is limited to a handful of base object types (most notably sprites and backdrops) [77, 132]. The differences in scaffolding implementation could also impact the effectiveness of the environments and the evaluation of them. For example, Scratch uses puzzle piece affordances that help novices associate constructs that can be used together with one another visually and audibly; Alice's drag-and-drop object creation mechanism reinforces the concepts of classes as blueprints and objects as entities. Further research of these differences, and how they impact students of different ages, could help instructors and researchers determine which features are more effective with different age groups.

Modern blocks-based environments, like Alice and Scratch, also provide unique affordances for some aspects of programming. For example, parallel, multithreaded programming is a topic that is becoming more crucial with the proliferation of multicore processors. Both Alice and Scratch provide visual, message- and event-based frameworks that simplify multithreaded execution in a form accessible even to children. If research showed that these approaches help students learn parallel programming more easily or quickly, teachers could more effectively prepare students in less time. Previous studies comparing text and blocks have

differed; however, those studies used different text-based languages. Research comparing different text-based languages to blocks-based environments, in the same situation, could help determine if language played a role. Scaffolding to smooth the transition from blocks to text may also reduce the friction of changing to text-based languages. Further research could help determine if this is the case. Knowing this could help the teaching community identify the best approach for introductory programming.

Table 2-2. Summary of Visual Environment Affordances

Environment	Type	Predecessors	Snap	Color	Shape	Icon
Prograph	Graph [83]	-	No	No	Yes	Yes
Fabrik	Graph [56]	-	No	No	Yes	Yes
AgentSheets	Grid [107]	-	No	Yes	No	Yes
KidSim	Grid [112]	-	No	No	No	Yes
LEGOSheets	Grid [43]	AgentSheets, Logo [43]	No	Yes	No	Yes
LogoBlocks	Blocks [11]	BrickLogo [11]	Yes	Yes	Yes	No
Alice 2.0, 3.0	Blocks [132, 35]	Alice 1.0	No	Yes	No	No
Storytelling Alice	Blocks [59]	Alice 2.0	No	Yes	No	No
Scratch 1.0, 2.0	Blocks [77]	LogoBlocks [77]	Yes	Yes	Yes	Part
Scratch Jr	Blocks [105]	Scratch 2.0 [105]	Yes	Yes	Yes	Yes

## 2.4 Multi-Modal Environments

Another approach to introducing students to programming is to use multimodal development environments—that is, environments that use different types of representations (such as blocks and text) to represent code and/or relationships. For purposes of this dissertation, I will refer to **hybrid modality** environments as those where different representations are used for different types of information and/or different relationships, and **dual-modality** programming environments as those which provide multiple representations of the same information and/or relationships.

### 2.4.1 Hybrid Modality Environments

The earliest multimodal environments used different representations distinctly to represent different types of information (i.e., hybrid modality). BlueJ (Figure 2-6a), an evolution

of the Blue Environment [62], mixes graphical representations of class and object relationships in the Java language with text representations of their definitions [63]. BlueJ is centered on an objects-first design; early on it distinguishes between classes, which are designed as templates, and objects, which must be explicitly, and visually, created from those classes [67]. The classes and their relationships—including inheritance—are displayed using diagrams, similar to earlier systems like Prograph and Fabrik [83, 56]. Objects, meanwhile, are displayed in a separate *object bench* to distinguish them from the classes. Users can edit the code for a class by “opening” the class, which displays the class’s text in an editor. Users can also run methods for testing purposes and inspect object values through a context menu.

Preliminary studies using BlueJ have suggested it holds promise for students in computer science courses. Hagan and Markham surveyed 120 college students who used BlueJ asking them to answer the question “How much does BlueJ help you learn Java programming?” on a scale of 1 to 7 (1 being “a great deal” and 7 being “very little”); 62% of respondents answered “1”, “2”, or “3” (more helpful), 16% answered “4” (neutral), and only 22% answered “5”, “6”, or “7” (less helpful) [50]. In comparing students who used BlueJ for an introductory course and those who did not, Borstler et al. found that students using BlueJ had lower dropout rates and higher pass/fail ratios [19]. Van Haaster and Hagan surveyed students in the two earliest computer science classes where BlueJ was optional; all students elected to use BlueJ, and their failure rates were reduced compared to the previous two years of classes (though they noted that the language was also different, previously having been C++) [48].

Greenfoot (Figure 2-6b) was later developed by many from the team that originally created BlueJ; it takes its inspiration from BlueJ and Karel the Robot, a robot simulator for learning programming [63, 51, 103]. Similar to BlueJ, Greenfoot is based on an objects-early

model. It is meant to address younger populations—including high school students—and to combine the simplicity of microworlds (in the vein of Karel) with the flexibility and object modeling of BlueJ. Greenfoot uses a grid-based world similar to the one used by AgentSheets and KidSim [107, 112]. It also carries over many features directly from BlueJ, including class/object distinctions, direct method invocation, and object inspection [63]. The combination of the world model and object/class visualization model allows for complex development [64].

Greenfoot was designed specifically to, among other things, motivate students when studying computing. Research has suggested working in Greenfoot may help student motivation, though it is not clear if it has helped students learn more effectively [129, 3]. Vilner et al. surveyed 325 students who worked in Greenfoot. Most of these students said they enjoyed using Greenfoot, and about half said it helped them understand inheritance; however, these was no statistical different in grades attributable to using Greenfoot [129]. Al-Bow et al. worked with students in a high school summer camp (9<sup>th</sup> and 10 grades). These students showed improvements in attitude, including enthusiasm and pride, and they were able to complete the tasks in the camp [3].

Recent versions of Greenfoot have implemented frame-based editing [66]. The frame-based editing model seeks to prevent syntax errors, as blocks do, while maintaining the expressive nature of text valued by experts [65]. In frame-based editors, scoping is defined by frames – which represent the boundaries of programming constructs – and which contain other constructs. As in blocks-based environments, the constructs are delineated visually; however, in frame-based editors, new frames are created using key combinations – such as pressing the “V” key to create a new variable – rather than dragging and dropping of constructs from a toolbar [66]. Generally, frame-based editors are intended to work with a keyboard workflow – and

therefore maintain similarity with text-based source editing – with the stated goal of being useful to learners for a longer period than blocks-based programming environments [65].

While some initial work has indicated that students have positive perceptions of frame-based editing environments [106], there is limited research in how the effectiveness of multimodal environments compares to other environments, especially after moving to pure-text environments [36]. Further work would be needed to determine how these environments compare to text- and blocks-based frameworks.

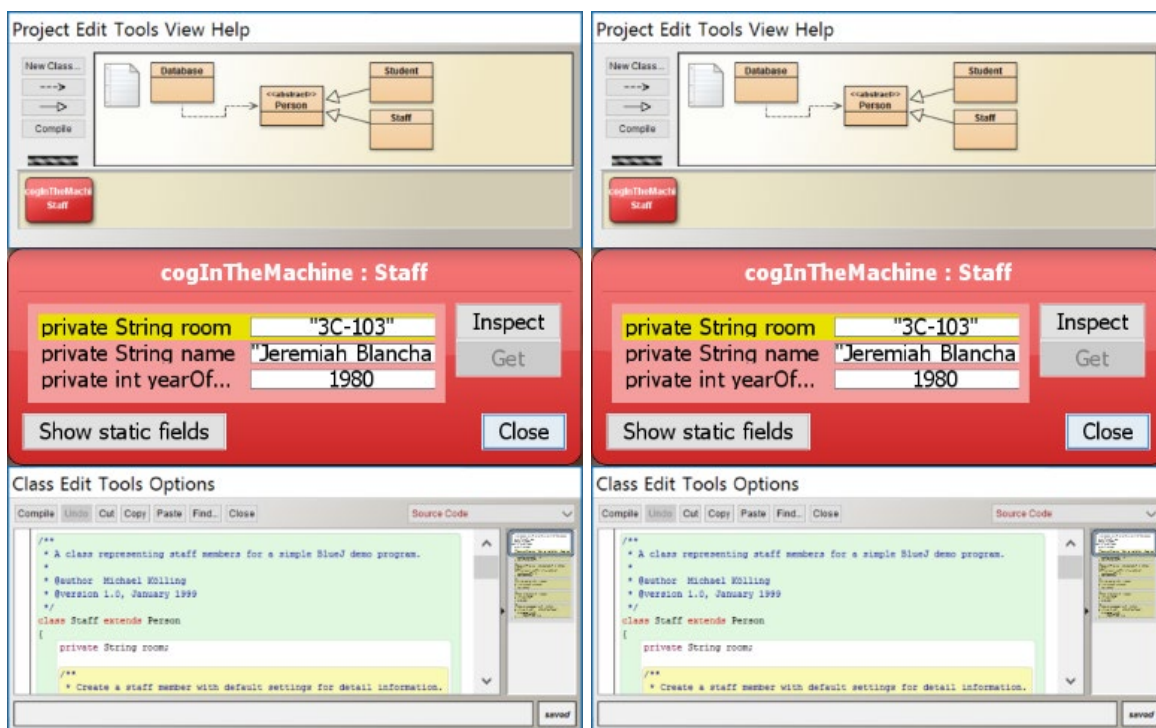


Figure 2-6. Hybrid modality environments: a) BlueJ [63] and b) Greenfoot [51].

## 2.4.2 Dual-Modality Programming Environments

Dual-modality programming environments provide a means to translate between representations – usually from blocks into text or vice versa – to help students understand the relationships between visual representations of constructs and their text-based counterparts [134]. Some of these allow only translation from blocks to text, while others also allow text to be



translated into blocks (i.e., they have bidirectional translation) [7, 35]. By providing a practical and usable environment that allows quick transition between blocks and text of the same program semantics, these environments may be able to overcome the actual and/or perceived difficulty of text-based languages. In addition to more clearly connecting constructs to their syntax, these environments may provide scaffolding for students to develop skills in chunking and abstraction [71] by “blocking” sections of text – i.e., creating blocks from the text (and thereby associating these text constructs with a visual area). By combining the affordances of blocks-based environments that show how constructs can be used together and how they connect, and by showing explicitly the equivalence to text variants of the same constructs, they hold promise in addressing coordination and understanding barriers [61]. If language difficulties and learning barriers associated with text representations could be alleviated, students may be able to develop competency in computing more effectively. In this section, I explore current frameworks offering translation between blocks and text and explore open questions in the literature regarding them.

#### **2.4.2.1 Unidirectional translation (blocks to text)**

Many environments offer one-way translation of blocks into text code. While Alice’s early versions were text-based, visual programming (and eventually blocks-based programming) was used in later versions. Alice 2 introduced a “code export” feature that allowed users to export a printable HTML file [79]. Later, Alice 3 displayed a grammatically correct Java translation of blocks in a separate window which users could turn on and off via the menu. In addition, Alice 3 projects can be converted into a complete Java project. Dann et al. studied the use of Alice 3’s translation features and how they impact transfer when moving to Java [83]; they found that student performance on final exams improved dramatically—from 60.8% in a prior semester using Alice 2 to 85.0% and 81.5% in semesters using Alice 3 and Java translation

features. This lends credence to the notion that connecting blocks and text representations explicitly – such as via a multi-modal environment – provides strong support for transfer of student knowledge from blocks to text. It should be noted that final exams were not identified as valid CS assessment instruments and that only one of the instructors taught both the old and new versions of the course; either or both of these concerns could influence the data gathered in the study. Some exam questions may not be measuring core computing concepts, but other related factors – such as language- or platform-specific applications. In the same vein, different teachers are likely to present materials in very different ways and using different approaches, potentially impacting student learning and retention.

Another unidirectional blocks-to-text tool is Google’s Blockly library. Blockly allows developers to build blocks-based programming editors. It provides a blocks-based development environment that can be translated into multiple programming languages. Users of Blockly-based applications can drag and drop blocks representing constructs to build algorithms, and these blocks representations can be translated into syntactically correct code in text-based languages [139]. Several languages are supported by default, including JavaScript and Python, and additional language implementations can be added. Blockly was designed and updated based on feedback and observations from user testing [40]. An explicit goal of Blockly’s blocks-to-text design is to provide an “exit strategy” and also support the authenticity of the blocks representation [40]. The design aims to facilitate movement to pure-text representations by providing direct conversion of blocks into text in production languages. Blockly is currently in use in several environments, including App Inventor and Code.org [139]. Wagner et al. studied the use of App Inventor specifically in a K-12 summer camp [130]. They found that showing

students the blocks representation of Java using Blockly, then the text representation of the same Java constructs, helped students understand how to build an application.

#### **2.4.2.2 Bidirectional translation**

The Pencil Code project introduced a dual-modality development editor—the Droplet Editor—that allows users to switch between blocks-based and text-based representations of the same program in real-time [7, 9]. A crucial difference is that, unlike one-way translation features (like those in Alice and Blockly), Droplet is bidirectional—it can convert from blocks to text, but also from text to blocks (Figure 2-7). This allows users to transition between blocks and text at their own pace, as they can return to the blocks mode (or switch to text mode) at any time. Bau et al. found that, in a small group of middle school students (N=8), use of the text-based editor increased over time with experience, suggesting that as students became accustomed to computer science and programming, they began using the text mode more often [9]. Later work by Weintrop & Holbert showed that students most often switch from text back to blocks when adding new or unfamiliar constructs [135]. Together, these findings suggest that that students made use of the scaffolding of the blocks-based environment to reduce cognitive load for new concepts in order to gain familiarity, but once those concepts were mastered, they moved to text.

In addition to blocks-based environments (noted above), Weintrop also tested bidirectional environments, and in particular the JavaScript variant of Pencil Code. [134]. He found that, like students who began in a blocks-based environment, students working in the bidirectional dual-modality programming environment (Pencil Code) scored more highly than those who began in the text environment [134]. Like those starting in the blocks-based environment, those starting in the dual-modality programming environment scored about the same at the end of the class after switching to Java [134]. However, unlike the blocks-based environment students, the dual-modality programming environment students had an increase in

interest in taking computer science courses in the future *after switching to text*, matching the trend in their performance, which also rose. The reason for this is an open question. It may be evidence that students in a dual-modality programming environment experienced less of a shock moving to the text environment, resulting in a lower negative impact on their perceptions and performance alike. It should be noted that students in the dual-modality programming environment had a decrease in interest in taking future courses in computer science until the switch to Java, unlike the students working in the blocks-based environment; further study could help researchers understand if this change in interest is related to the languages, the programming environments, or other factors.

There are contextual considerations that may limit the impact and applicability of this study's results more broadly within computer science. In Pencil Code, the Droplet Editor's use is limited to drawing applications and animations using a turtle interface; as the study's population consisted of high school students, the interface may not provide sufficiently interesting material to motivate students. Age may also be a factor; younger students may more readily take to the blocks-based and dual-modality programming environments and view them as more authentic than older students. Weintrop's work in particular was also done in an environment that involved changing languages at the midpoint of the course (as the latter half of the course was in Java, rather than CoffeeScript or JavaScript); this required students to learn not just new syntax, but also new control structures, potentially increasing the cognitive load on students and impacting student perceptions and/or learning. My work, as outlined in this dissertation, has focused on investigating how the use of dual-modality programming environments and instruction influences perceptions of text programming as well as how such instruction supports learning in the classroom, including work with middle-school and college-age students in a single language

– Python and Java, respectively. As such, my work benefits the research and education communities by providing evidence of student perceptions and learning in other age groups and without changing the instructional language.

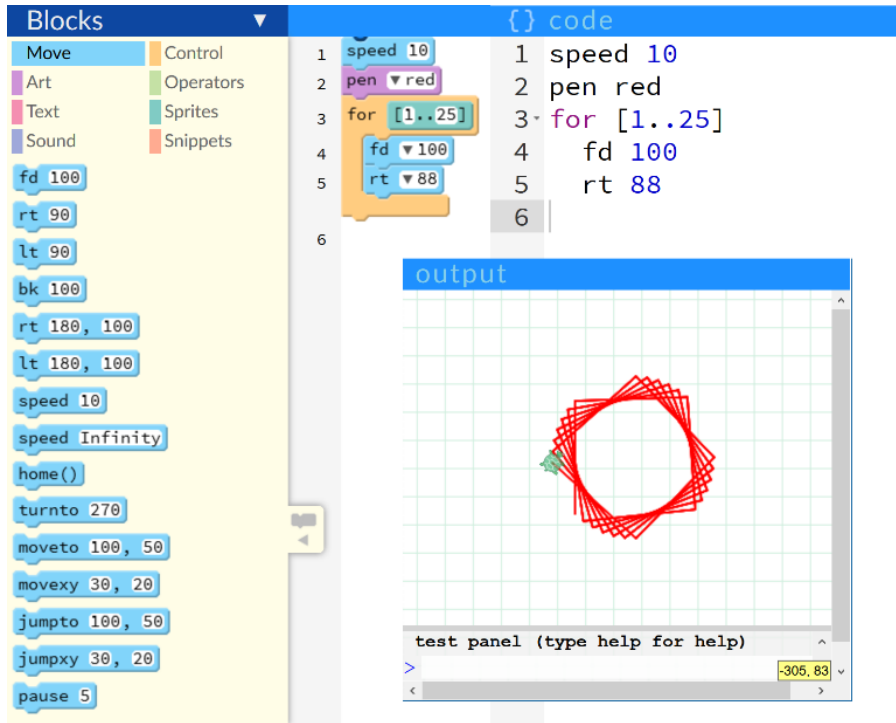


Figure 2-7. Pencil Code: Blocks-based mode, text-based mode, output window [9].

Table 2-3. Summary of Multimodal Environment Affordances

Environment	Type	Predecessors	To Text?	Bi-Dir.?
Alice 2.0	Blocks [132]	Alice 1.0	No	No
Storytelling Alice	Blocks [59]	Alice 2.0	No	No
Alice 3.0	Blocks [35]	Storytelling Alice, Alice 2.0 [35]	No	Yes
Blockly	Blocks [139]	-	No	No
Pencil Code (Droplet)	Dual [7]	Pencil Code – Text	No	Yes
BlueJ	Hybrid [67]	Blue [63]	Yes	Yes
Greenfoot	Hybrid [51]	BlueJ, Karel [51]	No	Yes

## CHAPTER 3 STUDY OF PERCEPTIONS OF PROGRAMMING

My work in the area of computer science education research has been largely shaped by my professional experiences within the teaching field, in which I have been working for over 15 years on a full time, part time, and volunteer basis. The challenges faced by underrepresented groups at varied age groups – from Kindergarten to the college level – first inspired me to examine how students’ perceptions of programming and the design of different tools impact learning and interest in computing. Eventually, my studies of perceptions led to questions about the efficacy of learning environments and how they can be improved. As such, my early work involved examinations of perceptions, environments, and learning measurements.

I began my work by examining perceptions of blocks-based environments (Section 3.1) and how students move from them to text languages (Section 3.2). I wanted to explore how children perceived programming: specifically, **how do children perceive the act of programming, what constructs do they perceive as hard or easy, and how does it differ based on prior experience?** To identify these perceptions and explore their connection to learning, I conducted a study during a summer camp with young children in 2015 and presented a poster at SIGCSE 2018 [15]. I also developed instruments to measure perceptions of blocks-based constructs with elementary school students for use in the study. In addition, based on my own teaching experience and available literature, I believed that students with experience in blocks-based languages nevertheless sometimes struggled when moving to text environments; I wondered, **how can we help ease students into text from blocks-based environments?** As such, I submitted a position paper at IDC’s “Every Child a Coder” workshop, which focused on bridging blocks and text representations (Summer 2015) [57].

### **3.1 Study: Construct Perceptions in Children (Summer Camp)**

I conducted an initial study of programming and language construct perceptions of elementary school students as part of a summer camp program in 2015. The purpose of this study was to identify what role prior programming experience played in perception of the act of programming and specific language constructs. This study's results provided guidance for my work focusing on blocks, text, and dual-modality programming environment analysis. I presented the results of this study as a poster at SIGCSE [15].

#### **3.1.1 Study Context**

The study was conducted as part of a children's summer camp program held at a medium-sized university in the southeastern United States in 2015. Two separate week-long summer camps on game development were conducted (one each in July and August of 2015), eight hours each day (9AM to 5PM). During the camps, participants learned programming through video game development and created games for about four hours per day. The summer camps used the Scratch environment and a modified variant of Google's CS-First curriculum for games [140]. They also visited campus studios, heard from industry guests, and played computational-thinking games. All activities were guided by three camp facilitators without backgrounds in computer science who were trained for one week in the CS-First curriculum by computer scientists. The camps were not designed around the purpose of this study; instead, I studied how participants' perceptions of programming changed after experiencing these camps, which are like many other computing camps offered around the world.

#### **3.1.2 Procedure**

I recruited children to participate in the study from the camp attendees; no compensation was provided to the children. Participants were recruited via an email that was sent to guardians of summer camp attendees before the camp began, with IRB-approved consent forms available

to guardians on the first day of camp. Voluntary assent was also obtained from the children at the beginning of the camp before the questions were asked. It was made clear to guardians and children that participation or non-participation in the study would not impact their experiences or opportunities during the camp. Of 51 summer camp attendees, I collected data on the responses of 28 / 55% of the children (16 in July 2015 and 12 in August 2015). Seven participants were female and 21 were male; 13 of 28 (46.4%) children indicated they had prior programming experience (generally, Hour of Code [102] or Scratch [110] activities).

I collected data over two weeks about the participants’ perceptions of programming via a series of semi-structured interview questions which I verbally administered on the first, second, and last days of the summer camp (Table 3-1). Most interviews were audio-recorded; some participants preferred their responses be written down. I asked participants if they had ever programmed before and what they thought programming was on the first day, as well as their initial impressions of programming. I asked about their opinions on Scratch programming constructs on the second day, after they had started using them. Follow-up questions on the last day were planned, but not asked in the July cohort due to logistical issues. In August, I also asked questions on the last day. Specifically, I repeated questions about their impressions and asked about their desire to program again in the future.

Table 3-1. Interview Questions by Topic

Topic	Questions
Prev. Experience	Have you ever programmed before?
Prog. Definition	Do you know what programming is? What is it, in your own words?
Prog. Impressions	Do you think programming is fun or interesting? Why do you think that? How do you think programming can be useful to people?
Constructs	What programming ideas were hard / easy / fun? Why do you think that?
Future Interest	Do you want to program again in the future? Why / why not?



### 3.1.3 Qualitative Measures

Based on existing literature and my own experience teaching children programming, I had some expectations regarding student perceptions about programming. Since children without experience lack understanding of what programming involves [46], I thought I would see evidence that they conceptualize programming according to results, e.g., artifacts they can see and interact with or have learned about from peers, parents, and teachers, but not the lower-level processes and functions involved in creating artifacts – e.g., providing instructions and communicating with devices (Table 3-2). This would also align with the Neo-Piagetian *sensorimotor* stage of development, in which learners know what an artifact is and does, but not how it works. In contrast, I thought children with experience would have gained insight and understanding through practice and application, allowing them to conceptualize these low-level functions – in line with what we might expect from learners at *preoperational* or later cognitive stages. If this expectation were confirmed, it would be possible to tailor instruction for children based on experience level as their knowledge matures. Past work has shown that some constructs are used more often than others by novices [24, 21], so I expected that perceived difficulty of constructs would follow similar patterns. Based on my own anecdotal experience teaching young children in the TurtleArt environment [18], students took naturally to loops, but sometimes struggled with if-else branching, so I expected that children would identify loop-based constructs as easier to work with than if-based ones, and vice-versa.

Table 3-2. Expectations of Perceptions based on Programming Experience

No Prior Programming Experience	Prior Programming Experience
Conceptualize programming according to results, e.g., artifacts	More mature understanding of process and functionality
Conceptualize programming according to what they have learned from peers, parents, and teachers	insight & understanding gained through personal practice & application to create artifacts

### 3.1.4 Coding Process

Using an inductive qualitative coding approach as described by Auerbach and Silverstein [4], my advisors and I created and assigned codes to the participants' interview responses. Responses to questions addressing specific constructs (blocks in Scratch) were qualitatively coded by participant identification of the construct. Sometimes participants referred to specific sets or subsets of constructs (e.g., “the green ones” or “the ifs”); these were assigned to set or subset codes (e.g., “SUBSET: IF”). To validate the reliability of the code book for characterizing the participants' interview responses, I computed interrater reliability on the second round of coding, in which all 3 researchers qualitatively coded all responses from 33% of the August participants. Since there were multiple raters, I used Fleiss' kappa [47]. Since each response could have multiple codes, each possible code was transformed into a yes/no variable to compare raters' consistency in assigning codes. Interrater reliability was computed for each question and possible response code, and average agreement between coders was  $\kappa = 0.8045$ , characterized as *substantial* by typical interpretations of kappa [128].

### 3.1.5 Programming Definition Themes

For questions about perceptions of programming in general, codes were grouped into themes using an inductive card-sorting approach (Table 3-3) [114]. I performed the initial card sort on codes for the relevant questions. These themes and the codes were reviewed and discussed by all researchers until consensus was reached. I identified themes that emerged related to my expectations, including responses that focus on results of programming: (1) Creation – a way to create an artifact (such as a program or media content); and (2) Helping – aiding people or society through robots and assistive technology. I also identified themes for responses that dealt with the process and function of programming: (3) Control – exerting control

over something / someone (such as a computer or robot); and (4) Communication – transfer of messages or using an encoding medium to transfer information (such as instructions or ideas).

Table 3-3. Programming Definition Themes

Theme	Description	Example Response
Creation	way to create an artifact	“Programming is making things for a computer”
Helping	aiding people / society	“It could help the elderly walk”
Control	exerting control over some entity	“making... any character... do something”
Communication	transfer of messages / information	“It is telling the computer what to do”

### 3.1.6 Findings – Perceptions of Programming

Before the study, I had expected that children without prior programming experience would perceive programming in terms of its results, and that children with prior experience would have a more mature understanding of process and functionality. Participants were asked at the beginning of the camp a) if they knew what programming is, and if so, to provide a definition; and b) how programming can be useful to people. Their responses to these two questions were then coded and combined (Table 3-4). Many with and without experience defined programming in part by referring to creation of artifacts (67.9%, n=19) and helping people (50%, n=14). However, few students without experience referred to communication (6.7%, n=1) or control (13.3%, n=2); this is in contrast to students with experience, who referred in larger numbers to communication (30.8%, n=4) and control (69.2%, n=9). This pattern matched my expectation that children without prior experience tend to perceive programming in terms of its results, especially of artifacts and helping people. It also provided evidence that participants with prior programming experience perceive programming at least in part in terms of function and process (though not to the exclusion of other ideas like creation). It also suggested that children’s perceptions broaden to include lower-level aspects of programming as they gain experience.

Table 3-4 summarizes the percentage of participants whose responses fit in each theme; as responses could have more than one theme expressed, numbers do not sum to 100%.

Table 3-4. Percentage and Number of Participant Responses by Theme

Experience?	Creation	Helping	Communication	Control
No	66.7% (10)	40.0% (6)	6.7% (1)	13.3% (2)
Yes	69.2% (9)	61.5% (8)	30.8% (4)	69.2% (9)
All	67.9% (19)	50.0% (14)	17.9% (5)	39.3% (11)

### 3.1.7 Findings – Perceptions of Constructs

Based on my prior experience teaching young children to program, I believed that some control structures would be more intuitive to inexperienced children than others: specifically, that loop-based structures would be easier and if-based branching would be harder. To explore whether participants perceived loop-based constructs as easy compared to other constructs, on the second day, I asked students which constructs they found easy and why. Again, I compared participants with and without prior programming experience, and in particular I examined how loop-based constructs compared with if-based constructs.

The 15 inexperienced participants most often identified loops (40%), simple events, (60%), and motion (53.3%) as easy – and loops were indicated far more often than if constructs (13.3%). It is notable that both of the students who identified if-based constructs as easy also identified loop-based constructs as well. These results suggest that these novices found loop-based constructs – particularly the variants found in Scratch – easier to learn and work with than other constructs. By comparison, among the 13 participants with prior programming experience, a majority identified both loop-based (53.8%, n=7) and if-based (53.8%, n=7) constructs as easy, along with events (61.5%, n=8), motion (53.8%, n=7), and visuals (“Looks” blocks in Scratch) (46.2%, n=6). This suggested that the difference in perception of these control structures subsides with experience. Table 3-5 shows the percentage of participants who identified

constructs as easy within the most commonly mentioned sets; again, participants could identify more than one construct in a response.

I also asked participants which constructs they found hard and why on the second day. Table 3-6 shows the percentage of participants who identified at least one construct as difficult within the most commonly mentioned sets. Among participants without prior experience, only 13.3% (n=2) identified loop-based constructs as hard. Other construct types were noted much more frequently – particularly coordinate-based motion (26.7%, n=4) and sensing (26.7%, n=4) blocks. For the 13 participants with prior experience, other construct types were more often identified – specifically, broadcast (23.1%, n=3) and events (30.8%, n=4).

Table 3-5. Percentage of Participants Saying Constructs EASY for N > 4 (14.3%)

Exp?	If	Loop	Color	Events	Motion	Visuals	Sound
No	13.3% (2)	40.0% (6)	13.3% (2)	60.0% (9)	53.3% (8)	20.0% (3)	26.7% (4)
Yes	53.8% (7)	53.8% (7)	30.8% (4)	61.5% (8)	53.8% (7)	46.2% (6)	30.8% (4)
All	32.1% (9)	46.4% (14)	21.4% (6)	60.7% (17)	53.6% (16)	32.1% (9)	28.6% (8)

Table 3-6. Percentage of Participants Saying Constructs HARD for N > 4 (14.3%), & If / Loop

Exp?	If	Loop	Coord.	Events	Broadcast	Sensing
No	13.3% (2)	6.7% (1)	26.7% (4)	6.7% (1)	13.3% (2)	26.7% (4)
Yes	7.7% (1)	15.4% (2)	7.7% (1)	30.8% (4)	23.1% (3)	7.7% (1)
All	10.7% (3)	10.7% (3)	17.9% (5)	17.9% (5)	17.9% (5)	17.9% (5)

### 3.1.8 Influence on Course of Research

The constructs students identified as easy and hard differed – particularly within control structures. How students perceive constructs may be dependent in part on the representation of those constructs – particularly whether they are presented as blocks or text – rather than an inherent feature of the construct. As the students programmed in Scratch, all constructs were blocks-based; if constructs were presented to students as text, they may have perceived them differently. In addition, it was notable that some students also differentiated between

“programming” and “coding” in interviews; for example, one participant said, “I know how to code, but I don't know how to program”. The direction of my work shifted based on the results of this study; in the next phase of my research, I began to focus on the differences between blocks and text representations, the perceptions users held of each, how they interact with learning, and how we might develop a bridge to help students move from blocks to traditional text representations used in industry.

### **3.2 Position Paper: Bridging Blocks and Text**

Early in my work I presented a position paper at a workshop on computer science education at the ACM International Conference on Interaction Design and Children [57]. In that work, I posited that, while blocks-based learning tools help facilitate the learning of computer science concepts at younger ages, students encounter challenges translating their experiences into production languages. Blocks-based learning tools and environments had made significant gains in engaging a younger audience and making programming more accessible by incorporating visual elements, drag-and-drop program construction, and media-rich environments, but while platforms were friendly for young children, they were largely built as sandboxes and at the time used languages that were environment-specific [77, 28]. I argued that, although these tools had shown great promise in exposing younger audiences to computer science and computational thinking concepts, what was still lacking was that bridge from the simplified and abstracted languages and tools to more advanced, complex environments. These more advanced environments had shown success at the high school and college levels in transitioning students to production programming languages used by programmers today such as Java and the more complex IDE tools used for these languages (e.g., Eclipse, Figure 3-1). A bridge would facilitate transfer of knowledge and skill from the early educational environments to an applied one while remaining accessible. I had further argued that this bridge could be explicit scaffolding that

facilitates movement from existing educational environments to production languages, or a completely new environment developed explicitly to grow with students as their cognitive abilities mature. Thus, in this paper [57] I proposed that the robust and effective development of such a bridge presents a key research challenge of introducing programming to younger audiences. This research challenge laid the foundation for the next phase of my research, namely, dual modality programming environments, which I anticipated might serve as that bridge from blocks to text representations.

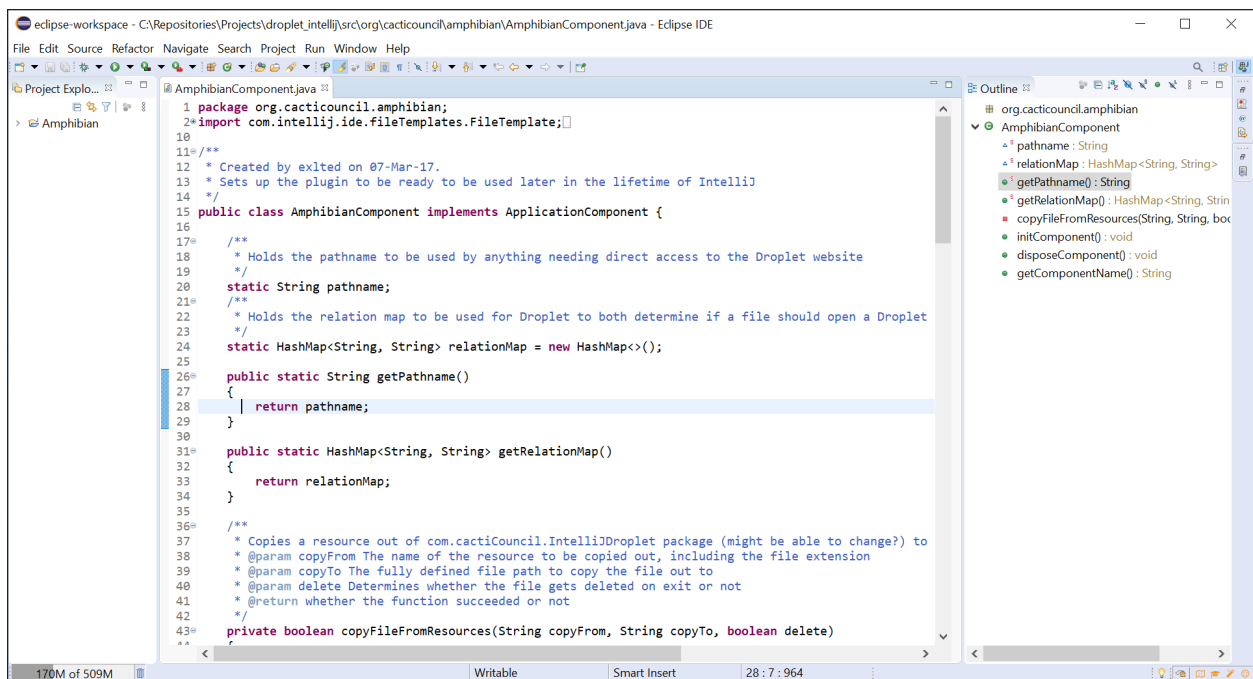


Figure 3-1. Eclipse IDE [141]

## CHAPTER 4

### STUDY OF DUAL-MODALITY PROGRAMMING ENVIRONMENTS

Based on my position paper and work during the summer camp, my work evolved to focus on dual-modality programming environments, which I anticipated could help bridge the gap between blocks and text representations. I developed dual modality representation tools in the form of Pencil Code's Python variant (Section 4.1), developed a custom assessment instrument using blocks and text representations (Section 4.2), and explored dual-modality programming environments from a perception and learning perspective using the Python variant of Pencil Code (Section 4.3). This work was principally centered on work with middle school students working in dual-modality programming environments, as by this age students have more developed reading and writing skills that provide the foundations for text-based programming. In this phase of my work, I conducted one study with middle school students (Spring 2017), presented a paper and poster at ICER's doctoral consortium (Summer 2017) [13], and presented a paper at VL/HCC (Fall 2019) [14] based on the middle school study. This work also provided the foundation for my final study at the college level.

#### **4.1 Development: Python Variant of Pencil Code**

Pencil Code, discussed earlier, is a turtle-based web application inspired by LOGO and blocks-based environments. Its Droplet Editor allows users to switch between blocks-based and text-based representations of the same program in real-time [9, 7]. The first version of Pencil Code allowed users to write programs in CoffeeScript; later, JavaScript was also added. However, JavaScript's syntax can be very complex, and CoffeeScript has limited use in industry and academia. By comparison, Python has been recognized as a language that can help students learn computer science in early courses [8] which also is in common production use. In order to explore dual-modality programming environments in the context of learning and perceptions of



programming, I needed a dual-modality programming environment that would avoid perceptions of inauthenticity of the text-based programming language. I integrated a Python runtime into Pencil Code in order to make available a language in common industry use – compared to CoffeeScript – while maintaining low-syntax threshold – compared to JavaScript. After developing this tool, I used it when running a study with middle school students (detailed later in this chapter). Pencil Code’s Python variant is available for download via GitHub:

<https://github.com/cacticouncil/pencilcode>.

#### **4.1.1 Description of Work**

The work to add Python to Pencil Code involved several steps:

Integration of a Python interpreter and runtime into Pencil Code’s web application  
Writing Python routines for all language features and functions present in Pencil Code  
Developing a “palette” that mapped Python language constructs to block representations

Existing work on Droplet (Pencil Code’s editor) provided Python parsing without integration of an additional language parser, allowing me to focus on the runtime and representation. The architecture of Pencil Code’s Python variant is shown in Figure 4-1; Python specific modules, which I developed with others as described below, are highlighted in gray.

#### **4.1.2 Development**

The development of the Python variant of Pencil Code was done by a team composed of myself as the team lead and five undergraduate students<sup>1</sup>. This section details how the work was divided and completed.

---

<sup>1</sup> Undergraduate students Stevie Magaco, Julien Gaupin, Scott Settle, Jackson Yelinek, and Kristofer Soto contributed to this project.

#### 4.1.2.1 Language interpreter runtime

Before other my work could begin on the Python variant, a Python interpreter and runtime needed to be integrated into the web application. I completed this work in Summer of 2016. The majority of this work involved integrating elements of the Brython interpreter [142], which I used to package user-generated Python scripts as web requests, and integration of the full Skulpt interpreter [143], which I used to execute the scripts after they were packaged. These interpreters were used to parse and run python text within the web-based Pencil Code runtime.

#### 4.1.2.2 Python routines

Using the integrated language runtime, our team (undergraduate students and I) created a binding layer to wrap the existing Pencil Code function calls so that they could be called from within the Python interpreter. I completed the initial subset of basic features and the binding layer design; the undergraduate students on the team worked from this basis to add additional functionality and correct issues that arose. The binding layer was composed of both JavaScript-side and Python-side elements from wrapping and unwrapping routine calls.

#### 4.1.2.3 Palette (text-to-blocks mapping)

I oversaw the palette development. The mapping was completed primarily by the undergraduate students on the team. This is part of the “**Python Blocks**” module in Figure 4-1.

#### 4.1.3 Results

The initial Python variant of Pencil Code was completed in January 2017, allowing several months of testing before the variant was used with students. Once completed, the Python variant (Figure 4-2) of Pencil Code was used to investigate the relationship between perceptions / performance and blocks / text / dual mode environments. More details of this study are outlined in Section 4.3.

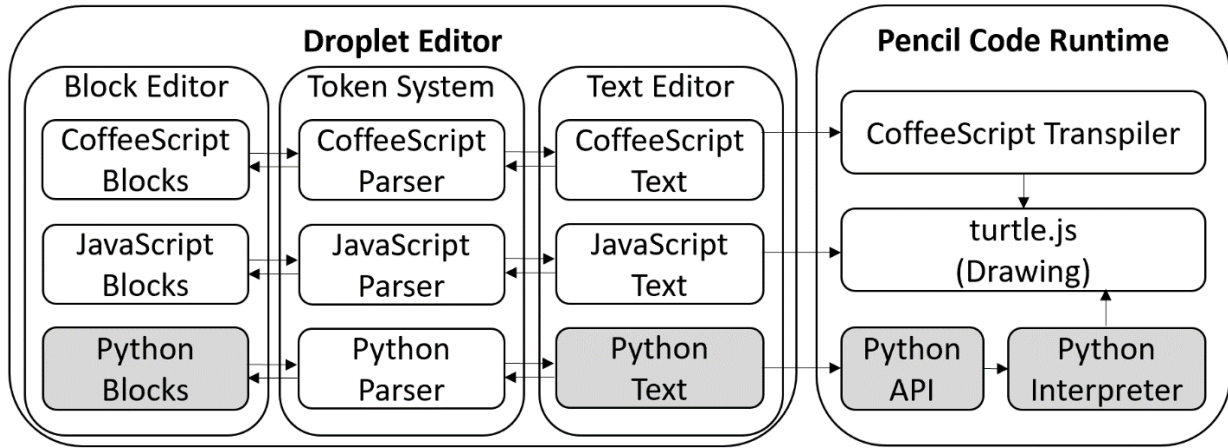


Figure 4-1. Pencil Code architecture, with added Python-variant modules highlighted in gray.

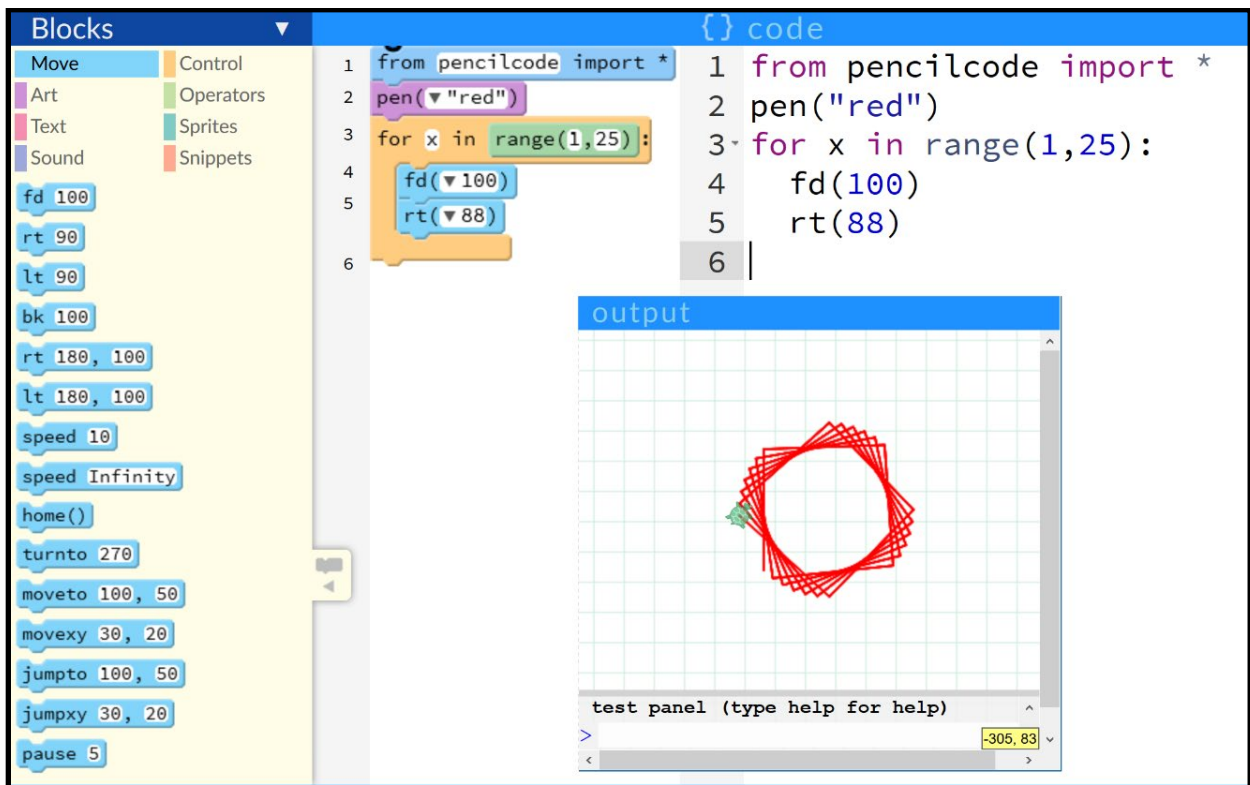


Figure 4-2. Pencil Code Python variant: Blocks-based mode, text-based mode, output window.

## **4.2 Development: Custom Dual Modality Assessment (Python Text/Blocks)**

In addition to having a suitable dual-modality programming environment to study student perceptions, an assessment was also needed to measure any notable knowledge differences in such a study. No assessment existed that provided both text and blocks representations, but I intended to study programming knowledge separately from syntax, necessitating the development of a new assessment. Some assessments, such as the FCS1 [122], are not available to the research community due to copyright limitations, while those available, such as the SCS1 [98], were only suitable for a single measurement (limiting assessment to a single point in time), and I intended to measure knowledge at multiple points with middle school students [98]. I sought to address this issue by creating an assessment with three isomorphic variants of each question so that the same concept could be tested at three separate points in time to measure change in performance over time. This custom dual modality assessment was developed in the spring of 2017.

### **4.2.1 Description of Work**

Development of the initial version of the assessment proceeded in several phases:

1. Selection / construction of questions
2. Development of multiple representations
3. Creation of isomorphic variants

Question topic, style, and in some case content were influenced by existing testing materials, including the SCS1 and AP Computer Science Principles Exam descriptions [98, 144].

### **4.2.2 Development**

I started developing the custom assessment by working from the SCS1 (Figure 4-3a). However, it was already known that the SCS1 and its predecessor, the FCS1, measure as too difficult for college students [122]; that meant that many questions were unsuitable for middle

school students. Together, my advisors and I determined that some questions should be replaced. I developed new questions using the AP CSP exam guide [144] to replace those SCS1 questions considered unsuitable as part of the assessment. Some CSP question samples use a graphical display to represent programming output (e.g., a character moving on a grid); I elected to use Pencil Code-style turtle displays to present a familiar visualization for students after they had worked in Pencil Code. I developed these questions in same multiple-choice format as the SCS1 and other concept inventories.

Once created, the questions were reviewed by myself and my advisors. Our goals included a) creating questions of appropriate difficulty for middle school students, with some questions being easy, medium and hard; b) considering the specific constructs / programming topics that should be covered; and c) developing appropriate distractors to detect student misconceptions. All questions involved one or more code snippets as part of the questions and/or answers. Over multiple passes, we refined questions using these considerations until we agreed that these objectives had been met and that the assessment was ready for use in the study.

The first version of the questions was developed in text. Once a text version of the question was developed, the Droplet editor was used to visualize the blocks-based version of the same code snippets. This was done for each question and variant (Figure 4-3a). For each question developed, two additional isomorphic variants were also developed so that the same concept could be tested at up to three points in time. This was done by changing strings, variable names, and/or code ordering; and sometimes by modifying images representing graphs to change positioning (Figure 4-3b).

I developed questions based on a) computational concept, b) level of reasoning, and c) level of difficulty (Table 4-1). Computational topics included **if-else**, **while-loops**, **for-loops**, and

**functions.** There were four (4) unique questions for each concept. Where practical, questions addressed level of reasoning by using simple **tracing** questions (for preoperational reasoning) and **code-completion** questions (for operational reasoning). These question types were selected to align with question types from the SCS1 [98]. Each concept also had questions at **easy**, **medium**, and **hard** difficulty levels. I increased the difficulty of questions by adding multiple layers of abstraction (such as nested function calls) and increasing the complexity of code blocks to be traced or completed. The assessment is included, in its entirety, in Appendix G.

### 4.2.3 Impact on Course of Research

The development of this assessment played an important role in charting the course for my research. It enabled the first study of dual-modality programming environments I conducted (detailed in Section 4.3) and served as a comparison point, via item analysis, against the SCS1 when evaluating information from the CS1 data set (detailed in Section 5.4). This was instrumental in deciding upon the form of the concept inventory for my final dissertation study.

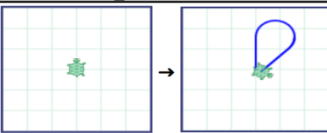
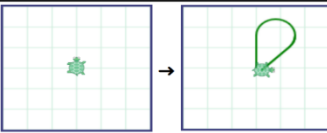
<p>Consider the following code snippet.</p> <pre> sum = 0 for i in __MISSING_CODE__:     sum = sum + i print ("The sum is",sum) </pre> <p>Which code block, replacing <code>__MISSING_CODE__</code>, would cause the program to print "The sum is 10" to the screen?</p>	<p>Consider the following function for creating a flower petal:</p> <pre> def draw_petal(color):     pen(color, 2)     fd(40)     ra(225, 20)     fd(40)     rt(255) </pre> <div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 5px auto;"> <p style="text-align: center;"><code>draw_petal('blue')</code></p>  </div> <p>Using this function, following block of code is executed:</p> <pre> draw_petal('yellow') draw_petal('orange') draw_petal('red') </pre> <p>What will be the result of running this code?</p>
<p>Consider the following code snippet.</p> <pre> sum = 0 for i in &lt;MISSING_CODE&gt;:     sum = sum + i  print("The sum is", sum) </pre> <p>Which code block, replacing <code>&lt;MISSING_CODE&gt;</code>, would cause the program to print "The sum is 10" to the screen?</p>	<p>Consider the following function for creating a flower petal:</p> <pre> def draw_petal(color):     pen(color, 2)     fd(40)     ra(225, 20)     fd(40)     rt(225) </pre> <div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 5px auto;"> <p style="text-align: center;"><code>draw_petal('green')</code></p>  </div> <p>Using this function, following block of code is executed:</p> <pre> draw_petal('blue') draw_petal('teal') draw_petal('green') </pre> <p>What will be the result of running this code?</p>

Figure 4-3. Custom assessment: a) blocks / text variants (left) and b) isomorphic variants (right)

Table 4-1. Number of Questions by Concept, Type, Difficulty

Concept	Easy	Medium	Hard
Tracing	2	5	3
· If-Else	1	1	0
· While-Loops	1	2	1
· For-Loops	0	1	1
· Functions	0	1	1
Completion	2	3	1
· If-Else	1	1	0
· For-Loops	1	1	0
· Functions	0	1	1

### 4.3 Study: Perceptions and Concept Assessment (Middle School)

Following up on my earlier work studying perceptions of programming and constructs, I wanted to examine bridging blocks-based languages to production environments. Based on my teaching experience and work by Tabet et al [119], I surmised that middle school students could grasp and work in Python. Following the development of the Python branch of the Pencil Code environment and the custom dual modality assessment, I conducted a study at a middle school in Central Florida to collect data on and identify trends in student learning and perceptions of programming and computer science when using bi-directional dual-modality programming environments. The focus of the initial analysis of the results was perceptions of programming specifically: i.e., **how do bi-directional dual-modality programming environments interact with student perceptions of programming?** In designing a study to answer this question, I was particularly interested in examining student confidence in their own ability to program, and student perceptions of text- and blocks-based environments.

#### 4.3.1 Study Context

I ran my study at a large public middle school in Central Florida in 2017. The study involved participants in a single technology course (with six class periods) under the supervision of a single instructor. Prior to participation in the study, the course instructor had planned to offer

programming instruction as part of the curriculum of the course. I partnered with the teacher to use curriculum I designed and my study framework to offer this instruction. The curriculum focused on variables, loops, selection, and functions. Of 24 school days, nine were dedicated to state standardized assessments, leaving 15 days of instruction and three days of surveys and assessments for this study. Depending on the testing schedule, participants received multiple days of CS instruction per week. Each class period was 38-46 minutes, for a total of 12 contact hours. The classroom teacher and I co-instructed the course during the instructional period.

#### **4.3.2 Participants**

I conducted my study with six classes of eighth-grade students. Before the study began, participants took home an IRB-approved letter describing the study's purpose and informing guardians of their rights to opt their child out of the study. I also asked students on the first day if they voluntarily assented to participate in the study. No compensation was provided. Of 158 students in the six classes, 129 students agreed to participate in the study. Students who did not agree to participate received the same instruction and in-class programming assignments but did not take study surveys.

I obtained demographic data by self-report. The participants ranged in age from 12 to 16 years old at the time the study was conducted: 86.0% (n=111) were 13 to 14 years old; 2.3% (n=3) were 12; and 5.4% (n=7) were 15 to 16 years old. Eight participants did not provide their age. 39.5% (n=51) of participants identified as female, and 51.9% (n=67) identified as male; one participant (0.8%) identified as gender neutral. Ten participants did not provide a gender. The classes were ethnically diverse. Of participants reporting one ethnic background, 25.6% (n=33) identified as white; 30.2% (n=39) as Hispanic/Latino; 4.7% (n=6) as black or African American; 4.7% (n=6) as Asian; and 1.6% (n=2) as Native Hawaiian or Pacific Islander. 29.5% (n=38) of participants reported multiple ethno-racial backgrounds. Five did not note a background.



### 4.3.3 Study Design

The entire study spanned the course of five weeks. Classes were taught using three tailored versions of the Pencil Code environment’s Python language variant which limited mode of use based on condition. I subdivided the six classes into three condition groups of two classes each: Blocks, Dual-Modality, and Text. Figure 4-4 summarizes the amount of time each condition spent in blocks, dual-modality, or text mode and on assessments. Participants in the Blocks condition spent eight days using a blocks-based environment, followed by seven days using text; those in the Dual-Modality condition spent four days in blocks, five days in the dual-modality programming environment, and six days in text; and participants in the Text condition used a text-only variant of Pencil Code for 15 days. Note that text syntax was available to all students at all times due to the design of the Pencil Code blocks, which presents the full text syntax of the constructs on the blocks. Three days were dedicated to assessments and surveys throughout the study.

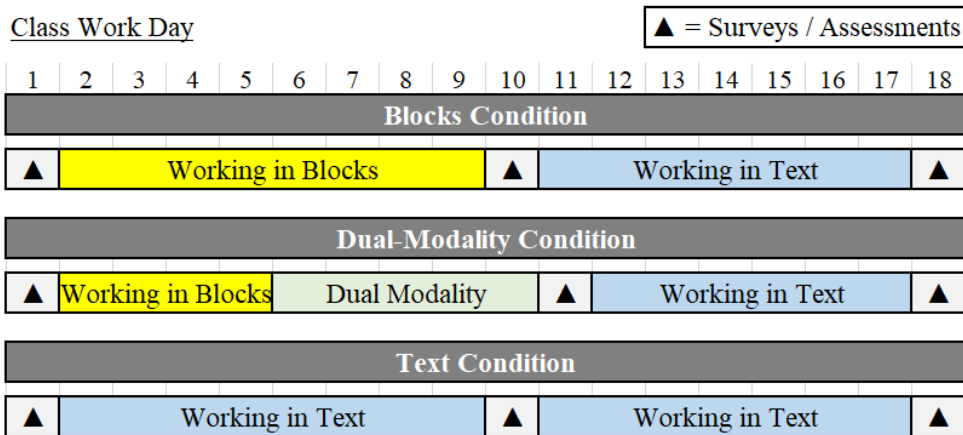


Figure 4-4. Timeline of time spent in text / dual / blocks modes by condition.

### 4.3.4 Data Collection

I collected data about participant demographics (Appendix C) as well as attitudes and programming competencies via computer-based surveys (Appendix D) and assessments

(Appendix G) on the first day, at the midpoint, and on the last day of the programming instruction period.

#### 4.3.4.1 Surveys

Participants were surveyed about their general attitudes on programming using variants of questions first proposed by Ericson and McKlin [39]. Ericson and McKlin’s questions were general computing questions, so I modified them to specifically address programming. For example, I asked students to rate agreement with this statement: “I can become good at programming.” In addition, at the midpoint and end of the study, I asked participants about their perceptions of blocks and text (see Table 4-2). Survey questions used a 7-point Likert scale to rate agreement/disagreement (“Strongly Disagree”, “Disagree”, “Somewhat Disagree”, “Neutral”, “Somewhat Agree”, “Agree”, “Strongly Agree”). Each question about perceptions of blocks and text was paired with a free response prompt: “Why do you feel this way?” All students who agreed to participate in the study (n=129) took the initial (pre-study) survey. Due to class absences, of the 129 participants, 38.8% (n=50) participated in the mid-survey, and 57.4% (n=74) participated in the post-survey.

Table 4-2. Questions Comparing Blocks & Text Programming

Num	Prompt
Q11	I think programming in text is easier than programming in blocks.
Q10	I think programming in blocks is easier than programming in text.
Q12	I think programming in blocks is frustrating or hard.
Q13	I think programming in text is frustrating or hard.
Q15	I think learning to program in text is more useful than blocks.
Q14	I think learning to program in blocks is more useful than text.
Q16	I would prefer to program using text as opposed to blocks.
Q17	I would prefer to program using blocks as opposed to text.

#### 4.3.4.2 Assessments

I assessed participants’ learning using the custom assessment I developed (Section 4.2) based on questions from the SCS1 [98] instrument and sample questions in the Computer

Science Principles AP course and exam description [144]. Each question focused on a different programming concept, with an equal number of questions assessing for-loops, while-loops, selection (if-else), and functions. Blocks-based and text-based isomorphic variants of each question were developed at multiple levels of difficulty. As most students ( $n=87$ ) had prior experience in blocks, the initial assessment used only blocks, while the final assessment used only text. The mid-assessment was dependent upon condition, with blocks condition participants receiving a blocks-only assessment, text condition participants receiving a text-only assessment, and dual-modality condition participants receiving a mixed assessment.

#### **4.3.5 Data Analysis**

To investigate the programming environment and students' perceptions of blocks and text, I analyzed the Likert responses and free response question answers. I converted all Likert responses to numeric values (1 to 7), inverted the value for blocks-preference responses, and calculated the midpoint of the two variants for question pairs. I grouped responses with midpoints of 1-3.9 as "disagree", 4.0 as "neutral", and 4.1-7 as "agree". I present the proportion of students who agreed, disagreed, or were neutral for each question in Figure 4-5. The  $n$  varies per question since not all students opted to answer all questions. I qualitatively coded the free responses to identify themes related to participants' perceptions of blocks and text programming. Using an inductive qualitative coding approach [4], I created and assigned codes to each response. If there were two or more distinct ideas addressed by a response, I assigned multiple codes to that response. Each code included an over-arching theme as well as a sub-code to identify the specific reasoning. Each response fell into one of these themes: Pro-Text, Pro-Blocks, Anti-Text, Anti-Blocks, Neutral. The sub-codes included descriptions of the mode they liked / disliked, such as Easy, Hard, Efficient, and Fun. To compute interrater reliability for each

question and response code [47], a second researcher coded 16% of the responses<sup>2</sup>. The average agreement between coders was Cohen’s kappa = 0.7845, which is characterized as substantial agreement [128].

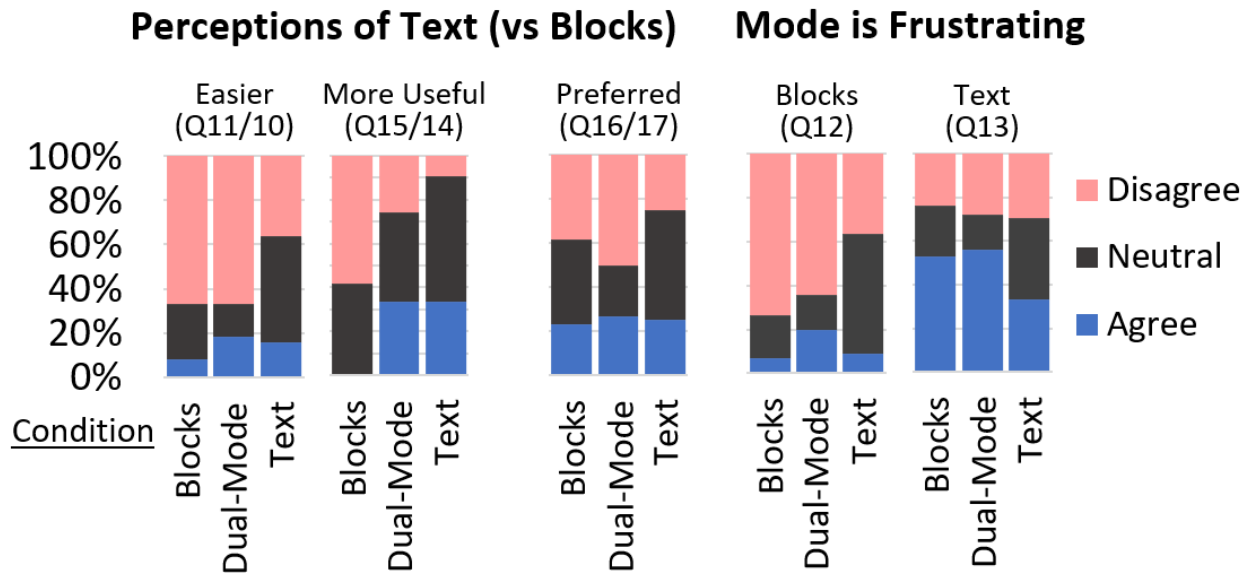


Figure 4-5. Distribution of survey Likert responses.

### 4.3.6 Findings

My findings focused on key patterns from my survey on participants’ perceptions of programming in blocks or text by condition after switching to text-only representations. I examined the distribution of participant responses to Likert scale questions regarding participants’ perceptions of text as easy (Q10/Q11) and frustrating (Q13) and also the coded responses to the accompanying free response questions.

**Dual-modality condition participants most often rated text as easier than blocks compared to blocks condition participants (Q10/Q11, Likert).** On the final survey, 18.5% (n=5) of those in the dual-modality condition identified text as easier than blocks, while 14.8%

<sup>2</sup> Pedro G. Feijóo-García coded 16% of the student responses as part of this study.

(n=4) were neutral and 66.7% (n=18) disagreed. In contrast, fewer participants in the blocks condition agreed that text was easier than blocks (agree: 8.3% (n=1); neutral: 25.0% (n=3); disagree: 66.7% (n=8)). Text condition students rated text representations as easier about as frequently as dual-modality students but disagreed less often (agree: 16.0% (n=4); neutral: 48.0% (n=12); disagree: 36.0% (n=9)).

**Dual-modality condition participants perceived text more favorably than blocks condition participants (all free response).** For every one of the text-blocks comparison questions I asked, dual-modality condition participants were 1) more often pro-text and 2) less often anti-text than their blocks condition counterparts. Dual-modality condition participants also frequently responded using comparisons between the environments when giving a neutral response.

One common reason given by dual-modality condition participants for why they liked text was that they felt it helped them make rapid progress, with students noting: "...it is a lot faster and easier to understand" [H002] and "...I think text is faster and makes it easier to change the code" [H024]. These responses suggest that dual-modality condition participants developed an appreciation for the benefits of text in terms of efficiency in programming. Another common reason cited by dual-modality condition participants for preferring text over blocks was that they found text to be more organized and easier to debug: "It's faster for me to recognize the error in my code when looking at text and it is easier to organize" [H002]. They also felt text offered more flexibility than blocks: "text is more free in what you can do while blocks have very restrictive ways of coding" [H099]. Dual-modality condition participants who perceived text less favorably than blocks cited syntax issues as their biggest challenges: "because when you [are]

programming in text there is a million ways you can mess up the coding, and it[']s not always easy remembering the codes” [H042].

Many responses given by dual-modality condition participants were comparative, noting pros and cons of a particular mode. One dual-modality student said, “because... [blocks are] easier but at the same time you need to get used to it [text]” [H126], while another noted that “I think that they both have their advantages” [H065]. The comparisons expressed in these responses are evidence of a more nuanced view of programming representations, weighing the benefits and drawbacks of blocks and text.

On the other hand, participants in the blocks condition were more negative about text programming, frequently mentioning syntax and detail issues that they felt got in their way: “it takes to[o] long to write and any little mistake can mess up the whole thing” [B047]. Another blocks condition participant said, “Any small mistake will make it say ‘script error’” [B043]. From these responses, we see that blocks participants primarily focused on the difficulties that text presented and were not able to recognize the strengths of text in terms of organization and flexibility that dual-modality participants noted.

**Dual-modality and blocks conditions participants both found text frustrating, unlike text condition participants (Q13, Likert and free response).** 56.0% (n=14) of dual-modality condition participants agreed that programming in text was frustrating or hard, while 16.0% (n=4) were neutral and 28.0% (n=7) disagreed. Similarly, the majority of blocks condition participants agreed that text was frustrating or hard (agree: 52.9% (n=9); neutral: 23.5% (n=4); disagree: 23.5% (n=3)). Meanwhile, only 33.3% (n=8) of text condition participants agreed that text was frustrating or hard (neutral: 37.5% (n=9); disagree: 29.2% (n=7)).

Many text condition participants described feeling comfortable using text despite the challenges they noted: “I feel it's really easy, I just need a little more practice” [T082]. In contrast, dual-modality and blocks condition participants mentioned similar obstacles and were more discouraged, rating text as more frustrating. One blocks condition participant noted that “[text is more frustrating than blocks because] the text has to be perfect” [B040] and one dual-modality condition participant said, “You have to beware of many errors because when you do it wrong you have to figure out where you messed up and it takes a while” [H122]. These responses show that participants in all conditions referred to experiencing obstacles in using text related to syntax. Text condition participants framed them as challenges to master, while dual-modality and blocks condition participants interpreted them as impediments that limited their progress. It is notable that text students spent the entire study within the text environment, and thus had more time to achieve a high level of comfort in text.

#### **4.3.7 Discussion**

In this study, I surveyed participants who transitioned from blocks to text directly and via a dual-modality programming environment, as well as participants learning only in text about their perceptions of blocks, text, and programming in general. Participants who used dual-modality programming environments rated text easier to use when compared with those who moved directly from blocks to text. Both dual-modality and blocks condition participants experienced more frustration in text. However, I also found that, in general, dual-modality condition participants held positive perceptions of text more frequently across questions regarding difficulty, frustration, usefulness, and preferred mode of programming, compared to blocks condition participants.

Perhaps not surprisingly, the blocks condition participants had a less favorable view of text than either dual-modality or text condition participants. Responses may reflect the

frustration of moving directly from blocks to text, suddenly losing the scaffolding on which they had come to depend, which was also supported by my classroom observations during the study. After moving to text, blocks students especially expressed frustration related to usability and increased errors.

Dual-modality participants, notably, expressed more positive views of text representation, overall, than their counterparts in the blocks condition. Many dual-modality participants expressed positive views of working in text, stating that text was easier to understand and helped support their learning, while others described it as fun – suggesting that they had developed a level of comfort in text programming. Classroom observations during the study sessions confirmed that students frequently flipped back and forth between blocks and text – taking advantage of the scaffolding that bidirectional dual-modality programming environments provide. This allowed each participant to transition at their own pace, making the transition from blocks to text less jarring and more inviting. This complements prior work showing that students in dual-modality programming environments often switch between blocks and text when new constructs are introduced [9]. The self-paced transition is particularly important as increase in confidence is one of the major motivations for creating visual (and especially blocks-based) languages [54] and suggests that dual-modality programming environments may help achieve the educational goals of blocks-based environments.

These differences between conditions provide insight that will help contextualize and facilitate further development of environments for learning programming. Perceptions of programming can impact perseverance in the field by newcomers [37]; my study demonstrates how those perceptions differ based on the tools used to transition between blocks and text. These findings suggest that educators can reduce the hurdles and frustrations students face when



moving from blocks to text by using an environment that bridges representations. By developing approaches to computer science instruction that reduce perception of difficulty and frustration, and improve perceptions of usefulness, we remove obstacles that participants face when first engaging with programming and transitioning to text-based programming.

## CHAPTER 5

### FINAL STUDY: LEARNING & DUAL-MODALITY INSTRUCTION

My early work focused on identifying how programming environments impact novice programmers in terms of learning and perception. This included understanding student perceptions of the discipline overall and of specific programming constructs. While my work began in blocks-based environments common in K-12 computing education, dual-modality programming environments presented an interesting and unexplored area of research. As discussed in Chapter 4, I began to investigate dual-modality programming environments with my study at a middle school with the Pencil Code Python variant. Building on that work, final study focuses on exploring in detail the relationship between dual-modality programming environments in production languages and learning of programming, particularly in college.

Many students receive some programming instruction as part of their K-12 education in blocks-based environments [10, 14, 16]. At the college level, however, computer science instruction is primarily in text-based languages [84, 127]. Though dual-modality representations offer students the opportunity to seamlessly transition between code representations—allowing them to build a conceptual bridge between blocks and text—they are also largely tied to sandbox environments, and few (if any) tools exist that facilitate use of dual-modality representations for general purpose programming in compiled languages (e.g., Java, C, and C++), making them more difficult to use in undergraduate instructional settings. I sought to build tools for and investigate the use of dual-modality programming environments at the college level for this reason.

To investigate the relationship between dual-modality programming environments and learning, I first conducted a study to evaluate the SCS1 in the context of the target population in UF's introductory computer science (CS1) course in the Fall of 2017. I then conducted a

comparative study of dual-modality vs text-based instruction with a baseline and intervention group that spanned two semesters. I ran this study while teaching UF's CS1 course in Spring of 2018 with a traditional text instructional approach and in Fall of 2018 using dual-modality tools and curricula developed as part of my doctoral work. My final study was composed of four distinct parts:

- a) developing a plugin for an Integrated Development Environment (IDE) providing dual-modality representation of the Java language,
- b) developing a dual-modality curriculum for a CS1 course,
- c) validating the SCS1 for use with the CS1 population at the University of Florida, and
- d) designing, conducting, and analyzing data from a study of the relationship between dual-modality representations, learning, and student perceptions in the classroom.

In this chapter, I will describe development work and studies that together constitute the capstone of my dissertation work.

- Section 5.1 describes my research questions, which explore dual-modality programming environments and their support for learning and perceptions of programming and computer science.
- Sections 5.2 – 5.4 describe the key components of my dissertation work needed to conduct the final evaluation of the connection between dual-modality programming environments and learning, with studies and analyses described in sections 5.5 – 5.6.
  - Section 5.2 discusses the software development work I completed to create a dual-modality programming environment that allows students to switch between blocks and text programming.
  - Section 5.3 discusses the curricular changes I made to the existing text-based CS1 course at UF (COP 3502) to incorporate dual-modality instruction, as well as the ethical issues considered.
  - Section 5.4 discusses the validation and evaluation of a CS concept inventory I used to evaluate student knowledge and performance.
- Sections 5.5 & 5.6 describe the final study that I conducted to evaluate how dual-modality instruction connects to student learning (5.5) and approach to analysis of data collected (5.6).

## 5.1 Research Questions & Hypotheses

My dissertation work seeks to evaluate the connection between dual-modality instruction and learning among students with and without prior programming experience in blocks and/or text environments, especially as it relates to cognitive development. My early work (Section 4.3) with middle school students has shown that students perceive text as easier when a dual-modality programming environment is used to transition between blocks and text. Additionally, prior research has suggested that students in dual-modality programming environments may use blocks when learning new constructs, but transition to text over time [9]. This is a benefit in CS1 courses that have students who come with a variety of prior experiences – including those with no prior coding experiences, block-based experience, and text-based experiences – because dual-modality programming environments allow students to self-scaffold themselves, transitioning when they are ready.

In my work, I sought to uncover how dual-modality programming environments might help novices develop computer science knowledge and skills, including chunking and abstraction. Research suggests that chunking and abstraction are important mechanisms employed by experts that may be challenging for novices to learn [71]. By learning to employ chunking and abstraction, practitioners reduce their cognitive load, allowing them to more efficiently think about information by abstracting it to tackle complex problems [71, 72]. In programming, abstraction and chunking are employed by practitioners when writing code [32], and there is evidence that chunking and abstraction aid in reading and tracing of programs [121].

I anticipated that the affordances of blocks (such as the puzzle-piece mechanism) would promote understanding of how constructs fit together (“what goes where”). By explicitly “blocking” text (associating text constructs with puzzle-piece-like blocks), dual-modality programming environments would promote chunking and abstraction via their block-

representation affordances. I further anticipated that the explicit display of text on blocks would help students learn syntax, and that dual-modality programming environments would link text constructs via their blocks with connective cues in the blocks environment (e.g., puzzle-piece connectors), which would provide scaffolding for new construct uses while students are learning them. As such, I hypothesized that dual-modality programming environments would support learning of conceptual programming knowledge by helping students overcome several challenges related to syntax, abstraction, and chunking, and in so doing, aid students' cognitive development.

In addition to the connection to conceptual programming knowledge, I hypothesized that dual-modality programming environments would promote student confidence and self-efficacy in programming. It has been shown that textual languages pose challenges to novices due to difficulties with syntax and perceptions of text languages as hard and/or intimidating [72, 54]. On the other hand, text languages have the benefit of being perceived as more authentic than blocks-based languages [134]. I expected that the association of blocks and text would reinforce authenticity of the experience, while the blocks scaffolding of the dual-modality programming environment would provide an inviting, rather than intimidating, interface.

### **5.1.1 Performance Comparison in Dual-Modality vs Text Instruction**

**RQ1.** How do students perform in code reading and writing after learning with dual-modality instruction, as compared to students learning with traditional (text-based) approaches to instruction in CS1 courses?

**H1.** Students learning using dual-modality programming environments and via dual-modality instruction will gain more knowledge and reach higher levels of cognitive development and expertise in programming as compared students learning via traditional (text-based) approaches.

**Reasoning.** Corney et al.'s work drew on the Neo-Piagetian framework for novice programmers to examine student learning in a traditional CS1 course, which used text-based instruction [32]. This work suggests that, while most students progress beyond the sensorimotor stage, in that they know a program functions but not how or why [73], the majority of students are at the preoperational stage or early concrete operational stage. In other words, they are capable of simple syntax evaluation and tracing (pre-operational) or have the ability to engage in limited abstraction and chunking when reading code (concrete operational stage). Comparatively, a minority of students show mastery of concrete operational thinking (in which they can reason routinely with abstractions about concrete situations) [32].

Dual-modality programming environments delineate programming language constructs visually, providing connective cues in addition to the text itself. These affordances scaffold code chunking and abstraction. In this way, dual-modality programming environments have the potential to reduce cognitive load which would allow students to engage in concrete- and formal-operational reasoning modes – in which they can reason with abstractions about hypothetical situations – more readily by facilitating chunking and abstraction. This in turn will help students to generalize code problems based on prior experience and develop solutions for them. As such, I expected most students learning via dual-modality instruction would show mastery of concrete operational thinking by the end of the course.

### **5.1.2 Performance Comparison by Prior Experience**

**RQ2.** How does prior programming experience affect students learning in dual-modality instruction as compared to students learning in traditional (text-based) approaches to instruction in CS1 courses?

**H2.** When comparing those learning via dual-modality vs text-based environments at the end of the course, there will be more of a difference in programming knowledge between students with

no prior experience than those with prior experience. Programming knowledge will differ the least for students with prior experience only in text when comparing those learning via dual-modality vs purely text-based environments.

**Reasoning.** As students progress from sensorimotor to concrete- and formal-operational levels of development, they must develop mental models of programming, including the ability to abstract and chunk code [71, 32]. Students with no prior experience (in the sensorimotor stage) have no mental support structures or mental models of programming; as a result, they stand to gain the most from approaches that scaffold chunking and abstraction, as I hypothesized dual-modality programming environments do.

Students with prior experience – in blocks or in text – are likely to have reached preoperational or concrete-operational stages and would have already developed some mental models of programming that aid them in abstraction and chunking. However, the literature shows that students may continue to face difficulties with syntax even after working in blocks-based environments when transitioning to text [74]. This suggests that, for students who have prior experience in blocks, some of those mental models may be tied to blocks-based representations and may not transfer to text-based environments. Dual-modality programming environments provide scaffolding in the form of a bridge between blocks and text representations. This is reinforced by the presence of text on the blocks themselves. Direct transition between blocks and text, scaffolded by dual-modality instruction and dual-modality programming environments, would reinforce the students' connections between blocks and text representations, helping students associate new text representations with block representations already familiar to them.

Students with prior experience only in text already have developed mental models of programming. As these students do not need to learn to program, they are unlikely to benefit from the blocks-based programming construct scaffolds that blocks representations provide. I felt that they might still benefit from the chunking mechanism provided by dual-modality programming environments, but if so, I anticipated that it would likely reinforce existing mental models rather than help develop new ones, so I hypothesized that the result would be a modest (if measurable) difference in ability to trace (read) and complete (write) sections of code.

### **5.1.3 Classroom Experience of Dual-Modality Instruction**

**RQ3.** What are student perceptions of dual-modality programming environments and instructional approaches, and how do they change over time, in the context of a CS1 course?

**H3.** Dual-modality programming environments will promote, strengthen, and support student confidence, motivation, and self-efficacy in programming coursework.

**Reasoning.** There are accounts in interviews in the computer science education literature that suggest some students perceive text languages as hard and intimidating [54]. In contrast, block-based environments were developed specifically to support student engagement and motivation while minimizing anxiety [54, 88]. However, some students continue to struggle with negative perceptions of text-based programming when they move from blocks to text [74]. Dual-modality programming environments provide a bridge between blocks and text representations, in effect providing the affordances and inviting context of blocks-based environments, while also providing scaffolding for learning text-based programming syntax. My findings in working with middle school students (Section 4.3) suggested that dual-modality programming environments help alleviate some negative perceptions of text. In addition, dual-modality programming environments allow switching between blocks and text in real-time, so students can switch into text easily as they come to understand constructs and integrate them into their mental models.



This self-paced nature of the dual-modality programming environment would provide students a level of control that is empowering. As such, dual-modality programming environments would alleviate the negative perceptions of text, thereby contributing to improved motivation and confidence, which have been shown to improve retention within the discipline [80].

### **5.2 Amphibian: A Dual-Modality-Representation IDE Plugin for Java**

A significant challenge to using dual-modality programming environments in instruction is that the dual-modality tools have been built into sandbox environments with functionality tailored to a specific purpose. For example, Tiled Grace [53] and Pencil Code [9] are two website-based environments that allow students to program in the browser without any additional tools, but programs are limited to turtle-graphics sandbox features; users cannot use other standard or third-party libraries and features. However, students in introductory programming classes at the college level usually use an Integrated Development Environment (IDE) which provides a suite of tools for programming support, including integration of standard language libraries. The use of IDEs is common in industry, and thus bring additional authenticity to the learning experience. As such, I could facilitate instruction and research via dual-modality representations in existing college-level curricula by integrating dual-modality tools within these general-purpose development environments.

At the time I began my work, there were no dual-modality tools for standalone IDE-based development outside of tailored sandbox environments, so I developed a plugin for IntelliJ IDEA based on Pencil Code's online open-source Droplet Editor [7, 145]. Matsuzawa et al. previously developed a blocks-text tool for a subset of the Java language, but this was also limited to a turtle-graphics environment [82]. Two undergraduate students helped develop the IDE plugin – a software component that adds functionality by “plugging into” the existing

software – to enable switching between blocks and text within a production environment<sup>1</sup>. The plugin I developed, which I dubbed Amphibian [146], enables instructors to more easily incorporate dual-modality instruction into courses and enables more rigorous investigation of dual-modality representations in classrooms by allowing researchers to reduce other potentially confounding variables, such as different languages, software systems, and development environments.

The Droplet Editor’s extensibility allowed me to integrate the language of choice into Amphibian. I noted that many introductory computer science programs at the high school and college levels, including those at my institution, use Java as the target language. To facilitate practical study of CS1 student performance in a “real-world” environment, I focused development on a Java variant. Amphibian allows users to switch back and forth between text and blocks modes, thereby enabling teachers of Java courses, including those of AP CS and many introductory college courses, to build blocks/text transitions into curricula.

### **5.2.1 Using the Amphibian Plugin**

Amphibian uses IntelliJ’s plugin API and can be installed in the same manner as other plugins. Once installed, Amphibian adds two tabs to the bottom of the editor pane of any Java file (Figure 5-1a). The tabs allow users to switch between the text of a program (Text Mode), which is the default mode upon startup, and its blocks representation (Blocks Mode), and back again.

In Text Mode, the editor retains all features of the IDE’s text editor, including syntax highlighting, prediction, error identification, recommendations, and code region identification. When the “Blocks” tab is selected, the editor switches to Blocks Mode, which uses the Droplet

---

<sup>1</sup> Undergraduate students Benjamin King and Trevor Lory contributed to this project.

Editor to present a toolbox from which blocks can be dragged to add them to the program (Figure 5-1c) as well as to display and enable editing of blocks-based constructs (Figure 5-1d). When in Blocks Mode, the program can be modified by adding new blocks to the program, with correct constructions signified by the puzzle piece style snap-together construction (Figure 5-1b) often used in blocks-based environments. Text in light-colored areas may be edited directly; in the case of variable value assignment, users may also drag-and-drop blocks representing variables / objects. At any time, a user can change modes using the same tabs.

To facilitate Java programming specifically, I added object-oriented blocks, including classes and methods (Figure 5-2a), while access modifiers such as “public” and “private” can be selected from dropdown components on the blocks themselves. Similarly, built-in variable types for parameters of variables can be selected from a dropdown menu on the blocks (Figure 5-2b), and users can enter text for custom and imported types. Whenever a block is added to the program via the drag-and-drop interface, the embedded Droplet Editor variant adds the construct to the program’s text and its blocks-based representation in real-time.

It is important to note that, as the plugin only changes the interface for editing the program, all IDE features remain available. Users can follow the typical workflow to build and run programs, including developing and running unit tests. Any Java project can be used with the plugin, including typical text-based and graphical applications, Android apps, and libraries.

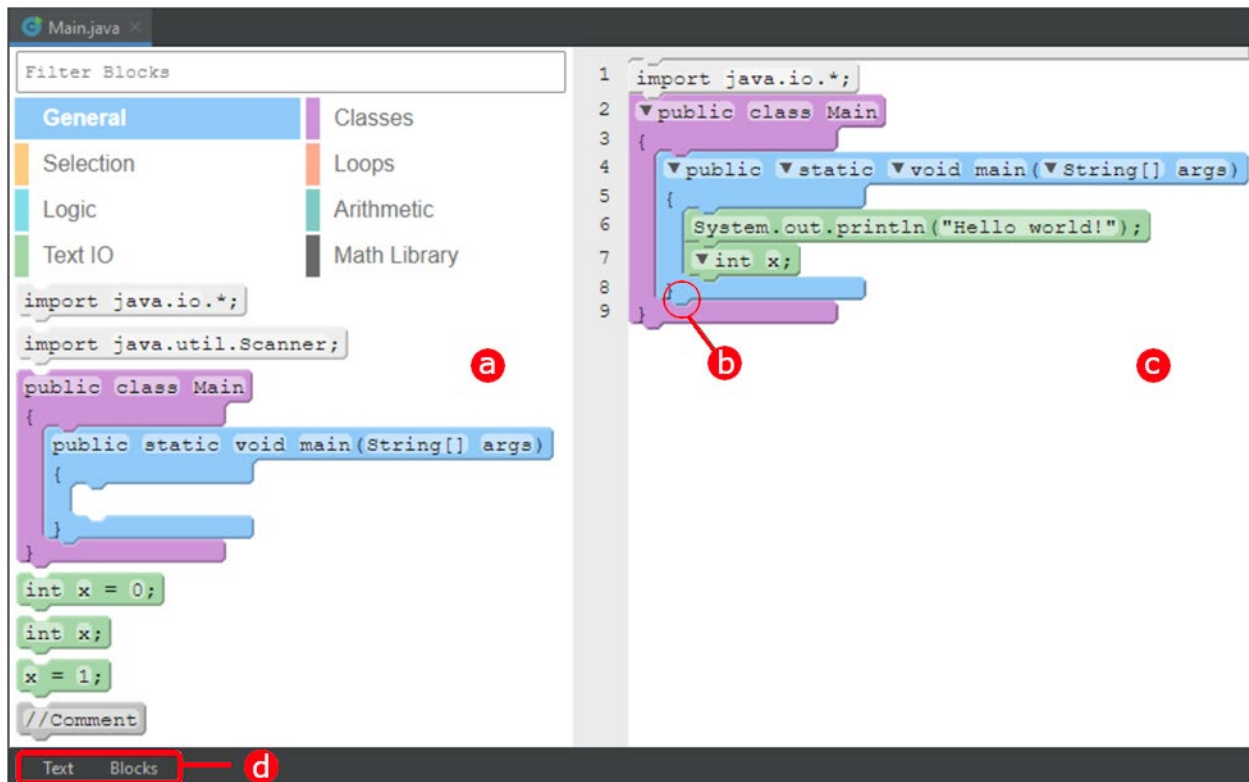


Figure 5-1. Amphibian Blocks Mode editor showing a) tabs for switching between modes, puzzle-piece connection, b) blocks representation of the current program, and c) block toolbox from which users can drag and drop constructs.

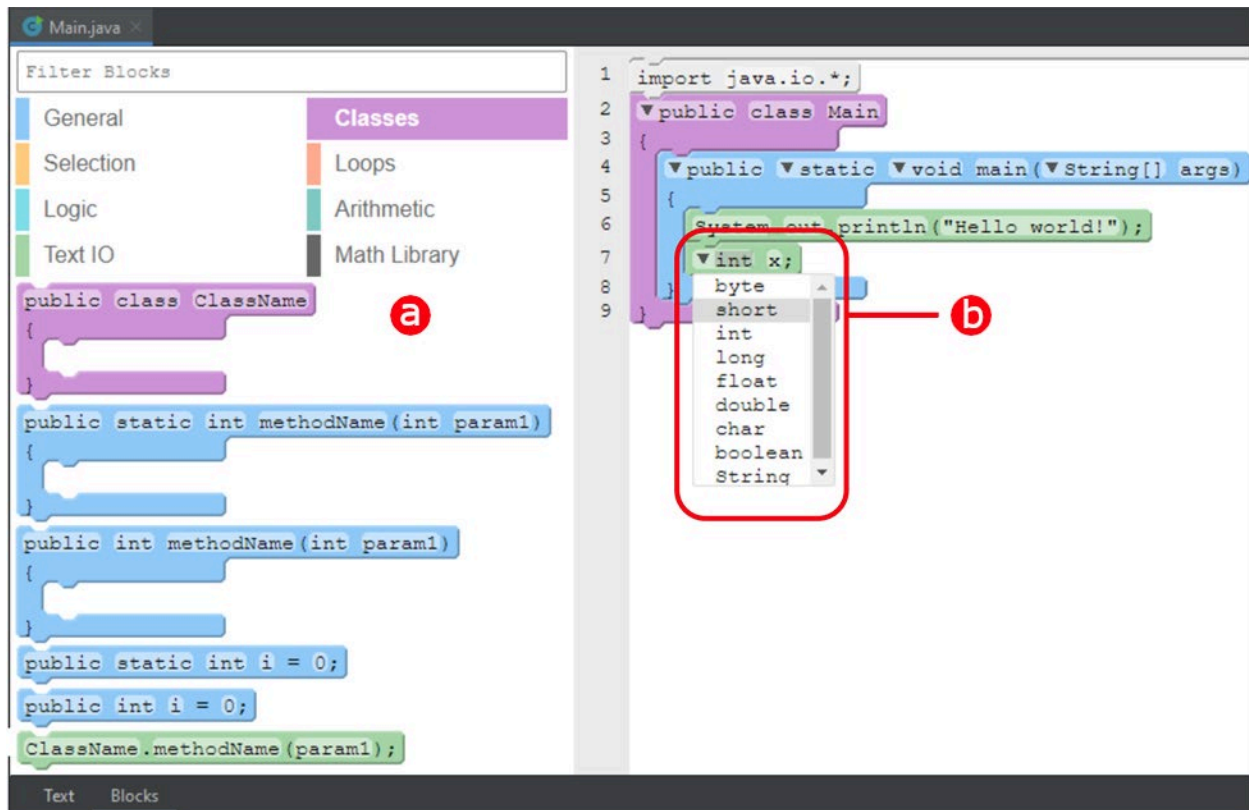


Figure 5-2. Amphibian Blocks Mode editor showing a) Java object-oriented constructs and b) drop-down menus used for types and modifiers.

## 5.2.2 Architecture

Amphibian was developed in two distinct phases. In one, I incorporated the Java language into Droplet, and in another, I developed the plugin into which I embedded Droplet.

### 5.2.2.1 The Droplet Editor

To enable Droplet to process Java language constructs, I integrated a customized Java language parser. To do so I constructed a custom variant of the Java 9 grammar specification and used ANTLR [100] to generate a parser program. Once the parser was in place, I developed a Droplet “palette” – a set of blocks-text mappings – for Java language constructs, including control structures, common statements, and object-oriented constructs such as classes and methods.

### 5.2.2.2 IntelliJ IDE Plugin Framework

The plugin connects to two major IntelliJ systems: the User Interface (UI) and the Document Manager (Figure 5-3). Whenever a Java file is opened, Amphibian adds the “Blocks” and “Text” tabs to the standard text editor. At the same time, in the background the blocks editor is loaded. This is accomplished by embedding a browser component via JxBrowser [147], which is preloaded with the Droplet Editor variant and custom JavaScript files, that can receive notifications from the plugin. When the user is in Text Mode and the “Blocks” tab is selected, an event is sent to the Droplet Editor which includes the current document text state. The text is loaded and processed, after which the embedded browser is displayed in the UI. The Java parser can interpret incomplete programs as blocks even when some constructs are missing. However, if the text syntax cannot be parsed due to irrecoverable errors, such as missing brackets, a modal dialog is shown to the user indicating the syntax error and directing the user to fix it in text mode. Otherwise, the browser editor window is shown, and user can edit the program using the blocks interface (Figure 5-4).

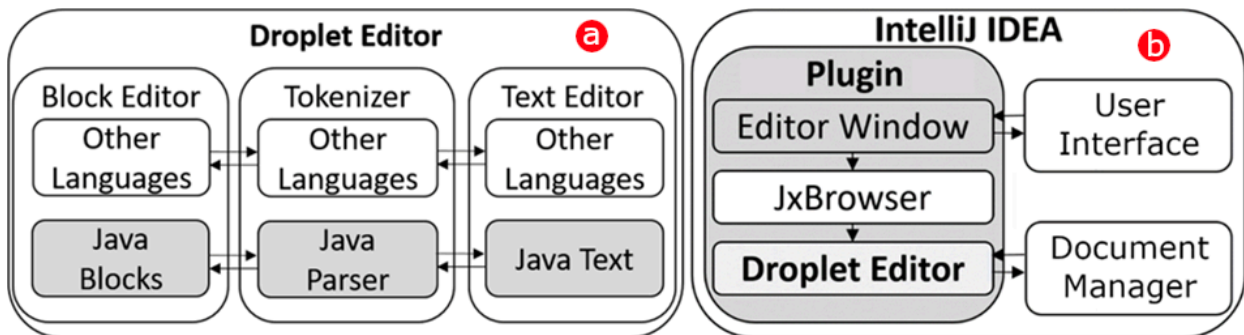


Figure 5-3. Amphibian architecture with new elements highlighted in gray: a) Modifications to the Droplet Editor and b) Architecture of the IntelliJ Plugin.

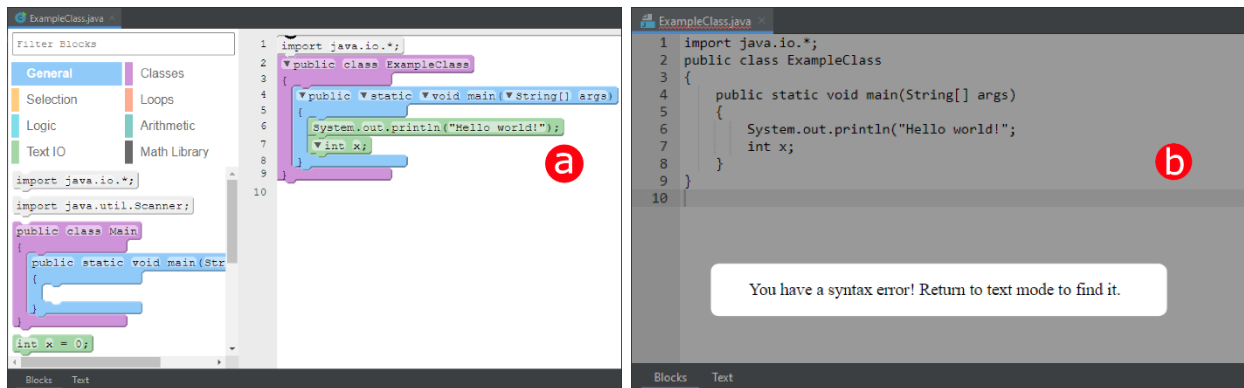


Figure 5-4. Example of switching from text to blocks mode: a) successful change to blocks mode and b) syntax error message.

### 5.2.2.3 Logging mechanism

Any time the toolbox palette changes or a block is dragged or dropped, the event is sent to the log. This log entry by default is displayed in the console, but the plugin can be configured to forward the message to a remote server so that study data can be collected from multiple users, as was done in my study. In addition, whenever the program is changed, the updated text is sent to the IntelliJ Document Manager. This ensures that the program text is synchronized between Blocks Mode and Text Mode (the standard IDE text editor). In addition, this means that there is always a text representation of the blocks; incomplete programs will not prevent conversion from blocks to text. When the text tab is selected from within Blocks Mode, the current text state is sent again to the IntelliJ Document Manager and the display is changed back to the default text editor for the IDE.

## 5.3 Dual-Modality Curriculum

To facilitate student use of and learning via the dual-modality programming environment, I updated the UF CS1 (COP3502: Programming Fundamentals I) course materials to address blocks and text representations. Previously, materials were based entirely on text representations; I added blocks-based representations to connect the classroom lectures with the dual-modality

representation IDE plugin. Based on these dual-modality-representation materials and collected data, I evaluated student perceptions and the classroom experience when using these tools during the full-semester course offering which I taught in Fall 2018.

### **5.3.1 Instruction**

I adjusted lecture materials – particularly slides and other visuals – to take advantage of the blocks-to-text dual-modality representation made available in the plugin. In lecture slides, rather than individual lines of text, code was presented in individual blocks, transitioning via animation to text to connect the representations for students (Figure 5-5). While most of the course was taught using dual-modality instruction, students will ultimately need to work in pure text environments in future coursework and their careers, so the latter part (about one-third) of the class was taught in text (Table 5-1). Aside from the addition of dual-modality representations to materials, the presentation of materials was not changed – all lecture slides and materials were otherwise the same between the two conditions. In other words, the lesson plans, lecture sequence, and assignments were the same, and the blocking mechanism was not explicitly highlighted separately in the intervention semester.

In lab sessions, teaching assistants and tutors explained and demonstrated use and function of the dual-modality IDE plugin within the IntelliJ environment. In-lab demonstrations of code and concepts were conducted directly in the plugin’s dual-modality programming environment as appropriate. In the first lab session, students were instructed on use of the plugin:

1. Installation of IntelliJ and Plugin
2. User interface for swapping between blocks and text
3. Short live-coding demonstration of “Hello World” in blocks, converted to text
4. Demonstration of changes made in text translating to blocks when mode is switched



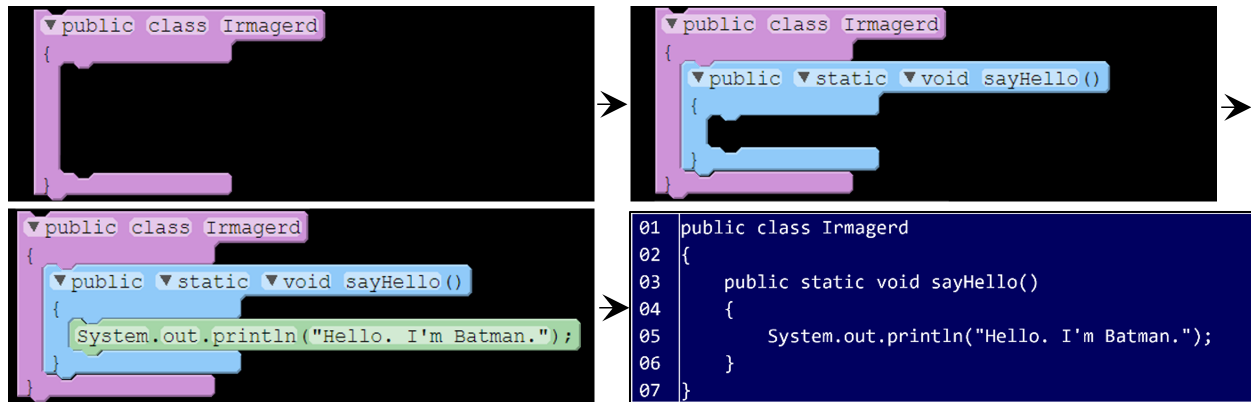


Figure 5-5. Instructional material – presentation in blocks, followed by conversion to text.

Table 5-1. Course Topics & Mode for Instructional Intervention

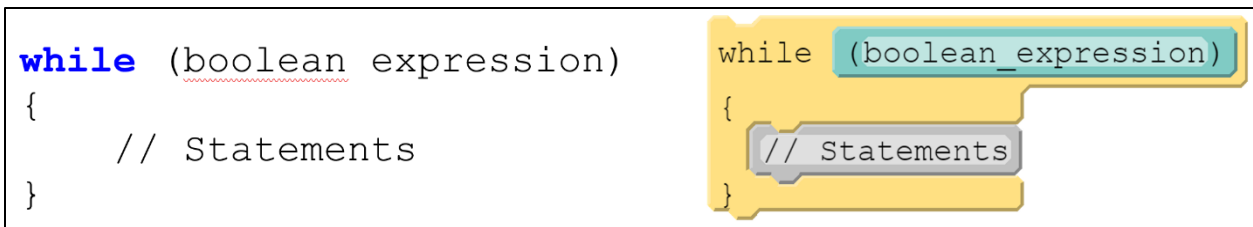
Unit Topic (Approx. 1 Week per Topic)	Instruction
Fundamentals of Computing	Dual-Modality
Variables & Arithmetic	Dual-Modality
Control Structures	Dual-Modality
Data Types & Objects	Dual-Modality
Methods & Collections	Dual-Modality
Engineering Process	Discussion
Mathematics of Computation	Discussion
Classes	Dual-Modality
Inheritance	Text
Input, Output, & Files	Text
Truth & Logic	Discussion
Programming paradigms	Text
Memory management	Text

### 5.3.2 Assignments

Assignments were updated to include text and blocks representations in Droplet style wherever sample or demonstration code was provided (Figure 5-6).

The assignments in the course served two primary functions in this study:

- They provided students with a setting in which to apply concepts learned and make use of the dual-modality IDE plugin, which facilitated the collection of log data, and
- Assignment scores were used as one of several measurements to evaluate student knowledge throughout the class.

The figure shows two side-by-side representations of a 'while' loop. On the left, the code is presented as plain text: 'while (boolean expression) { // Statements }'. The word 'while' is in blue, and 'boolean expression' is underlined in red. On the right, the same code is presented within a yellow block structure. The 'while' keyword is in a light blue box, 'boolean\_expression' is in a grey box, and the entire loop body is enclosed in a larger yellow block with a grey box for the comment '// Statements'.

```
while (boolean expression)
{
    // Statements
}
```

Figure 5-6. Curriculum assignment documentation – sample code in blocks and text.

### 5.3.3 Ethical Considerations

As a matter of caution and to ensure the integrity of the curriculum, it is important to address ethical considerations in performing studies within classrooms where they may impact student learning. The study hypotheses in Section 5.1 lay out benefits I believed students would receive from the curriculum. As this intervention took place in a core required course, a faculty review by course committees was undertaken. In this section I also address potential concerns and criticisms of using dual-modality instruction or tools in a university course and its potential impact on students, such as concerns about preparation of students for future courses.

#### 5.3.3.1 Faculty review

To ensure the curriculum adjustments were in line with expectations of the department, the proposed changes were reviewed by my advisors and other faculty members. Specifically, changes were presented to and accepted by the Undergraduate Curriculum Review and Undergraduate Curriculum committees in the CISE Department. This presentation and approval were noted in meeting logs and written correspondence (Appendix M).

#### 5.3.3.2 Delay of pure-text instruction

The use of blocks constructs as part of the dual-modality programming environment could be argued to take time away from, and therefore delay, introduction of text-based programming instruction. However, in the case of dual-modality programming environments, text is introduced along-side blocks. In other words, dual-modality programming environments

did not delay introduction of text instruction in my study. Preconceived perceptions of inauthenticity of blocks could have led students, and even some faculty, to construe dual-modality programming environments as a mental “crutch”; however, as quizzes and exams were text-based, students were incentivized to learn text representations.

### **5.3.3.3 Cognitive overload**

Dual-modality programming environments introduce two different representations of the same program. It could be argued that these dual representations require more mental effort to consider when programming, inhibiting performance. However, in Droplet’s model, text is always present, and blocks are presented as colorful highlighting of text. Thus, the text syntax and blocks constructs are presented together in blocks mode, not as separate, disconnected representations. In addition, instructions presented blocks and text representations together as a single concept in order to minimize duplication of mental effort.

## **5.4 Instrument Evaluation Study**

In order to assess knowledge in the CS1 course I studied – COP3502 at the University of Florida – I first investigated existing computer science concept instruments to find one suitable to the course’s student population. Using a computer science concept inventory instrument allowed me to evaluate student ability in the programming topics I am studying using an instrument developed by the research community for this purpose [122, 127]. I looked to the SCS1, which was available to the research community. However, the SCS1’s authors have noted that their initial results suggested its potential to discriminate by ability is limited by its high difficulty level [99]. Item Response Theory (IRT) measurements [5], taken for each question, suggest that the assessment skews toward hard difficulty, with most questions being considered fair, rather than good, discriminators [116]. As such, I also considered the custom dual-modality assessment I had developed during my middle school study. To evaluate the applicability of the

SCS1 and the custom assessment for use in future studies, I administered the SCS1 and custom assessment at the end of the Fall term of 2017 in the CS1 course, which was taught using traditional text-based instruction.

#### **5.4.1 Context & Data Collection**

I collected responses on both the custom assessment and the SCS1 from students at the end of UF's COP3502 course in Fall 2017. Student responses to each question were recorded, individually, via Qualtrics. In addition, demographic data were collected from participants at the end of the same computer-based survey. Participants completed the assessment, demographic, and attitude questions in a dedicated room with a proctor over a period of one-hour fifty-five minutes. The assessment assigned to each student was determined randomly, with half of participants being assigned to the SCS1, and the other half being assigned to the custom assessment. In all, 203 students completed the custom assessment, and 199 students completed the SCS1. This study's data collection was classified as exempt by UF's Institutional Review Board (IRB). No compensation was provided to participants, but students who participated received extra credit in the course. These data were also used to decide the direction of my final dissertation study.

#### **5.4.2 Question Analysis**

I performed an initial item analysis of student responses to the SCS1 and custom assessment using the method outlined by Sudol & Studer [115] to determine if either or both were appropriate for the CS1 student population and to make a decision on an instrument for future work. After initial item analysis of data collection in the Fall 2017 term, I determined in consultation with my advisors that the SCS1 was of appropriate difficulty and covered the correct concepts for it to be an effective assessment instrument for future work at the college

level. The methods used in this analysis and the results are described and summarized in this section.

I analyzed the responses to the Fall 2017 Custom Assessment and SCS1 results using the approaches described by Sudol and Studer [115]. As I wanted to evaluate both the difficulty of the items as well as their abilities to discriminate between students of different abilities, I used the two-parameter logistical model (2PL) approach they described. The 2PL model provides a difficulty level, which measures how difficult each item in the set is, as well as a discrimination factor, which measures how effective the item is at differentiating test takers of different ability levels. The results of these analyses are included in Appendix H and Appendix I.

The item analysis revealed that the custom assessment's questions were both too low in terms of difficulty and insufficiently discriminating according to ability with the tested population (Appendix H). Sudol noted that items typically fall in a difficulty range of -3 (easy) to +3 (hard) and discrimination values between 0 and 2 [115]. Six questions (37.5%) on the custom assessment fell outside of these ranges for difficulty or discrimination. In addition, most of the questions (9 of 16) were "easy" – i.e., having a difficulty rating of -1 or lower, and none had a difficulty of 1 or higher (i.e., none were "hard" questions). Thus, the custom assessment exhibited a ceiling effect with the students in the CS1 course, which would make it difficult to identify knowledge and cognitive differences that might manifest due to an intervention. This is likely due to the custom assessment's design for middle school students (described in Section 4.2).

By comparison, the SCS1's questions closely matched ideal ranges for ability discrimination and level of difficulty, suggesting that the SCS1 could be an effective tool with this population. All of the questions on the SCS1 in the analysis fell within the expected

difficulty range (-3 to +3) and the ideal discrimination factor range (0 to 2). In addition, for this population, the SCS1's difficulty range appeared to be an excellent match; 21 of 27 questions (77.8%) fell within a difficulty range of -1 to +1 ("medium" difficulty), with only two questions (7.4%) under -1 ("easy") and four (14.8%) over +1 ("hard") (Appendix I). As a result, I decided to use the data collected via the SCS1 to assess knowledge in the study outlined later in this chapter.

### **5.5 Study: Dual-Modality Instruction, CS Learning, and Classroom Experience (CS1)**

I investigated the use of dual-modality instruction and student learning in a study at the college level in a multi-section CS1 course (UF's COP3502), which is taught in the Java language, across two 16-week semesters (n=673). The course consisted of two large weekly lecture meetings and a weekly small lab meeting. I taught the class in both semesters. The first semester (n=248), acting as a baseline group, was taught using traditional, text-based instruction. The second semester (n=425), acting as an intervention group, was taught using dual-modality instruction and a dual-modality IDE plugin I developed. I measured participant learning via the SCS1 [98], which students took at the end of the course just before the final examination, as well as course examination questions, which I classified as either definitional / code reading or code writing [121]. The course covered all concepts tested by the SCS1 and course examinations. I also collected student responses to several surveys (Appendix F) throughout the intervention semester to help me understand the mechanisms behind any effects I might see. This included regular surveys during each module – weekly excepting exam and break weeks (Table 5-2) – as well as surveys and at the beginning, midpoint, and end of instruction, about student perceptions of blocks, text, and dual-modality instruction. Based on my hypothesis that dual-modality instruction and tools would help students better chunk and abstract sections of code, my

expectation was that students in the intervention group would score higher on exam questions and the SCS1 than those in the baseline group.

As noted previously, many K-12 curricula focus on blocks-based environments; as such, many students in CS1 courses like UF's have some prior experience in blocks-based programming environments. Students without experience could benefit from scaffolding, and students with only blocks-based experience needed to transition to programming in text, with all of the challenges and difficulties that entails. These students in particular stood to benefit from the representations provided by dual-modality programming environments, though I expected these environments to help students with prior experience as well. I hypothesized in Section 5.1 that students in the intervention group would learn more about programming compared to those in the baseline group, and that differences between the groups would be most pronounced among those with no prior programming experience.

Table 5-2. Module Survey Questions (Weekly)

Q	Prompt
1	Did you program in "Blocks" mode since the end of your previous lab (including this lab)?
2	Did you program in "Text" mode since the end of your previous lab (including this lab)?
3	What was your primary mode since the end of your previous lab (including this lab)?
4	Does instruction in dual blocks-text modes help you learn better?
5	Why do you feel this way?

### 5.5.1 Study Design

This study used a quasi-experimental design with repeated measures and two groups. Both semesters used the same lecture and lab format. The first semester, Spring 2018, acting as a baseline group, was taught using traditional, text-based instruction; the second semester, Fall 2018, acting as an intervention group, was taught using dual-modality instruction and the dual-modality IDE plugin I developed for the study:

- a) Students in the baseline group (n=248) were provided with standard development tools, including the IntelliJ IDEA, an Integrated Development Environment (IDE). All lecture slides and assignment descriptions used only text programming representations.
- b) Students in the intervention group were provided with IntelliJ IDEA and the Amphibian Dual-Modality IDE Plugin for Java I built, which presented text and blocks representations of the code they wrote and allowed them to move freely between representation modes. They were also instructed in the plugin's use in lab sessions. 66.7% of the course (8 of 12 topics) used blocks and text representations on assignment descriptions and lecture slides (Section 5.3). The remaining topics were not represented in the blocks construct models of the plugin (e.g., inheritance) or were non-programming topics (e.g., ethics and version baseline).

In general, the topic ordering between the semesters was the same, but some topics were replaced as part of typical course content adjustment in preparation for later courses. In the baseline semester, introductory Data Structure and Generics were covered, while in the intervention semester, Algorithm Complexity and Propositional Logic were covered – none of which are facilitated by dual-modality instruction.

During the first lab session, students in the intervention group completed a personal perception survey; as in middle school study, the questions were based on the work of Ericson and McKlin [39]. As in the middle school study, since Ericson and McKlin's questions were general computing questions, I modified them to specifically address programming (Appendix G). Each week, students in the intervention group completed a short survey during their laboratory period as part of the course (Figure 5-7). The survey contained questions about student use of the blocks and text modes ("Did you program in 'Blocks' mode since the end of your previous lab?") and perception of the effectiveness of the dual-modality instruction ("Does instruction in dual blocks-text modes help you learn better?"), along with a free response prompt ("Why do you feel this way?"). Students also completed three long-form perception surveys at the beginning, midpoint, and end of the course with five-point Likert-scale evaluations to



measure their comparative perceptions of blocks and text (e.g., “I think programming in blocks is easier than programming in text”). In addition, I collected logs from study participants via the Java dual-modality IDE plugin (detailed in Section 5.2), which tracked usage of blocks, mode switching, and time spent in each mode. Participants in all groups took the SCS1 at the end of the course 5-10 days before the final examination. I evaluated performance through a combination of scores on the SCS1 and score on course exams in terms of both overall scores as well as scores by question type.

This study made use of lessons learned from my early work, especially my work with middle school students (Section 4.3), to improve upon study design. The CS1 population was composed primarily of students majoring in Computer Science and/or Engineering who are personally invested in and driven to learn the material (unlike some students in middle school study who found the technology courses uninteresting or boring). The population was also be much larger (n=673) – which should reduce statistical noise and improve statistical rigor. The study was over a longer time period (16 weeks instead of 5) and provided more time between class meetings, allowing time for students to learn / iterate on content and develop skills through practice.

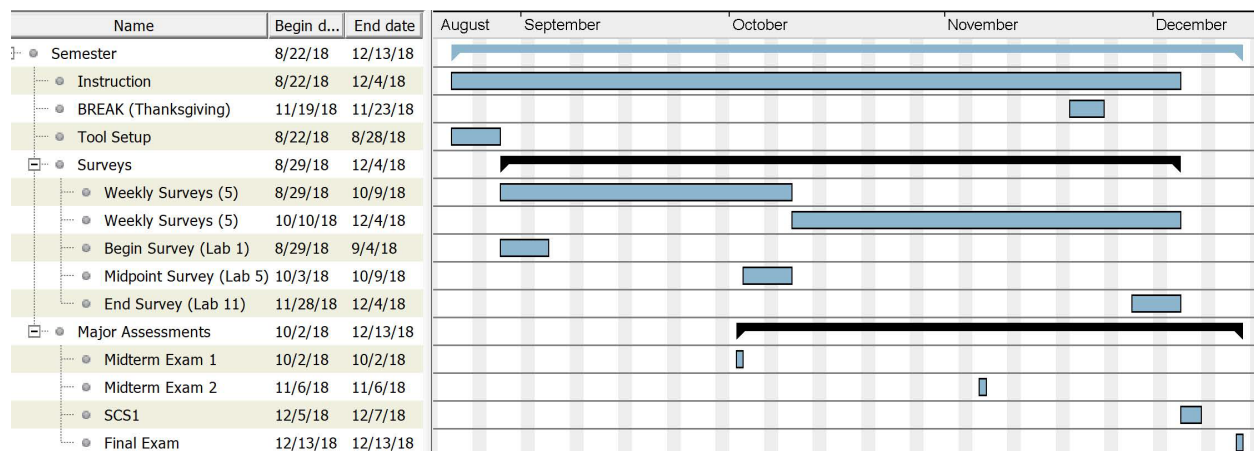


Figure 5-7. Gantt chart showing date ranges for surveys, examinations, and SCS1 assessment.

### 5.5.2 Participants

This study's data collection was classified as exempt by UF's Institutional Review Board (IRB). However, I still asked students who completed the SCS1 and demographic survey to explicitly consent to having their scores and demographic data included in the study. No compensation was provided to participants, but students who completed the SCS1 and demographic survey received extra credit in the course. Students who did not participate were offered an alternative assignment to earn the same amount of extra credit in the course. Students who did not complete the SCS1 or demographic survey received the same instruction and in-class programming assignments and took the same examinations, as these are part of ordinary classroom activity assigned by the instructor. In all, 58.1% of the baseline group (n=144) and 59.1% (n=251) of the intervention group opted-in to the demographic survey, and their exam scores were included in my analysis.

The COP3502 course is the first required course in the "Fundamentals of Programming" sequence at UF; as such, this class has a mix of students with some prior experience and those with none. Transfer students with programming coursework usually have CS1 waived as an equivalency via transfer credit. 36.8% (n=53) of students in the baseline group and 40.2% (n=101) in the intervention group had some prior programming experience; 19.9% (n=29) of the baseline and 22.1% (n=94) of the intervention groups had taken the AP Computer Science or AP Computer Science Principles courses in high school.

Participants came primarily from the young college student age range (18-22) and represented diverse ethnic, racial, and gender backgrounds in both the baseline and intervention groups due to the course's size and the university's demographics (Table 5-3). While the populations are similar, there are some notable differences. There was a higher proportion of

men in the intervention group, and more intervention students indicated they were from white, Asian, or multiple racial backgrounds.

Table 5-3. Demographic Groups by Condition

Demographic Group	Baseline	Intervention
Men	66.7%, n=96	74.9%, n=188
Women	33.3%, n=48	25.1%, n=63
Asian	27.8%, n=40	30.7%, n=77
Black / African American	11.1%, n=16	5.2%, n=13
Hispanic / Latino	20.8%, n=30	24.7%, n=62
Native American	0.0%, n=0	0.4%, n=1
Native Hawaiian / Pacific Islander	0.7%, n=1	0.4%, n=1
White, Non-Hispanic / Latino	49.3%, n=71	52.6%, n=132
Other	0.7%, n=1	1.6%, n=4
Multiple	7.6%, n=11	15.5%, n=39

### 5.5.3 Data Collection

I collected several types of data for this study which varied by condition. Initially, data were collected along with contact information in order to facilitate follow-up interviews if needed. Personally identifying information had been collected in order to link participant responses to performance in the course during the study. Once this link had been created, the data set was anonymized by removing all identifying and contact information to protect the privacy of participants as much as possible.

#### 5.5.3.1 Examinations, assessments, and demographic surveys

I collected exam question scores from each participant via the university’s learning management system (LMS). These examinations were purely text-based in both semesters, using the same framework and modeled from exams in previous terms. The midterm exams – Exam 1 and Exam 2 – were non-cumulative, and the Final Exam was cumulative. Exam 1 and Exam 2 which were broken into two sections: a multiple choice / short answer section with definitional and code reading questions, and a free response pseudocode section requiring code writing. As is typical for this course, the Final Exam had only definitional and code reading questions; since

code writing questions require significant time for students to complete and instructors to grade, it is logistically difficult to fit a hand-scored, rubric-based cumulative examination into the allotted final exam blocks of two hours and also to have grading completed in time for grade submissions.

I proctored the SCS1 at the end of the semester for both groups and recorded participant responses. In addition, I collected demographic data from participants at the end of the same computer-based survey. Participants completed the assessment and demographic questions in a dedicated room over a period of one-hour fifty-five minutes. The SCS1 was voluntary, so a subset of students opted to participate in this part of the study across the baseline and intervention semesters (58.7%, n=395).

#### **5.5.3.2 Perception surveys and usage logs**

I collected answers to Likert-scale perception survey questions in each course module (Appendix F), as well as download logs for online resources such as lecture slides. Additional surveys were given at the beginning, midpoint, and end of the semester. A secure server was also used to collect logs of how students used the plugin itself during the semester.

#### **5.5.3.3 Bias control**

In order to protect the integrity of data collection and prevent subconscious bias, neither I, as the principal investigator and instructor, nor my advisors, had access to information about who took or planned to take the SCS1 during either semester. Instead, this information was controlled by the teaching assistants until after final grade submission. Once I had submitted final grades for the course, the teaching assistants shared the assessment and participation data that were collected so that it could be analyzed.

## 5.6 Analysis Methods: Dual-Modality Instruction and Learning

In this section, I describe the analysis methods I used to evaluate the data I collected, with the findings and discussion following in Chapter 6.

To complete my dissertation work, I performed an analysis of the data collected in my final study via several methods, including analysis of student scores, survey responses, examination of student logs, coding of student responses, and review of instructor notes. This allowed me to identify differences in dependent variables, such as overall computer science knowledge and cognitive development level, according to independent variables – particularly instructional condition, demographics, and experience (Table 5-4). These analyses were then used to draw conclusions about how student knowledge differed between conditions (RQ1 & RQ2) and student / instructor perceptions of the classroom experience (RQ3).

Analyses considered independent variables (Table 5-5) including instructional condition (intervention / baseline), prior experience, and type of prior experience. Specifically, prior experience was categorized as text-only, blocks (which may include some text experience), and none.

Table 5-4. Measures by Research Question

Research Question	Independent	Dependent
RQ1 – Knowledge / Condition	Instructional Condition Demographics	Overall Knowledge Material Use Instruction Perception
RQ2 – Prior Experience	Instructional Condition Type of Prior Experience Demographics	Overall Knowledge Material Use Instruction Perception
RQ3 – Perceptions of Experience	Type of Prior Experience Demographics	Material Use Instruction Perception Blocks & Text Perceptions Instructor Perceptions

Table 5-5. Independent Variables

Measure	Instrument
Prior Experience Type	Background Survey
Demographics (Gender, Age, & Ethnicity)	Background Survey
Instructional Condition (Intervention/ Baseline)	None

### 5.6.1 Examinations and Assessments

To investigate student learning, I investigated scoring on course exam questions and the SCS1. I broke down my analysis according to question type, which I also associate with the Neo-Piagetian stages of development. This section focuses on the examination and assessment data and what it reveals about student learning of programming between these two different conditions.

#### 5.6.1.1 Hypotheses & expectations

Examination and assessment scores were used to evaluate programming ability and cognitive level in the Neo-Piagetian framework, which I used as the primary measures to identify patterns in differences between the intervention and baseline conditions (Table 5-6). My hypothesis was that dual-modality programming environments would help students develop the ability to use abstraction and chunking, so I expected students to perform better on questions that make use of them. Code reading and writing depend on chunking and abstraction and are associated with concrete- and formal-operational reasoning [71, 32]. As such, within the intervention group, and compared to the baseline group, I expected students would show higher performance on code completion and tracing questions on the SCS1 and reading and writing questions on course exams, which would suggest that they had developed expertise in chunking and abstraction, and by extension had reached the concrete-operational stage of cognitive development in the Neo-Piagetian framework. However, I believed the intervention was unlikely to play a role in performance of tasks that depended on preoperational skill sets (those without

abstractions). Definitional questions depend on understanding of construct function, but not abstraction or chunking; as a result, performance on such questions would not be significantly impacted by the hypothesized advantage granted to abstraction-based questions by dual-modality programming environments. As a result, I anticipated that students would score about the same on definitional questions.

As with the analysis of conditions, my exploration of how prior experience and dual-modality instruction interact used the examination scores when drawing conclusions. I explored the interaction of prior programming experience type – text only, blocks, and none – and condition with respect to assessment performance. I had hypothesized that, among students in the intervention group, students with no experience would see the greatest positive difference in knowledge compared to those in the baseline group, followed by students with blocks experience (Table 5-7). I expected students with text experience to show the smallest differences in scoring between conditions.

Table 5-6. RQ1 – Dual-Modality Instruction and Question Performance - Hypothesis

Question Type	Intervention	Baseline
SCS1: Definitional	No difference	No difference
SCS1: Tracing	Higher than Baseline	Lower than Intervention
SCS1: Code Completion	Higher than Baseline	Lower than Intervention
Course Exams: Definitional / Reading	Higher than Baseline	Lower than Intervention
Course Exams: Writing	Higher than Baseline	Lower than Intervention

Table 5-7. RQ2 - Dual-Modality Instruction vs. Text Instruction by Experience - Hypothesis

Question Type	No Experience	Blocks Experience	Text-Only Experience
SCS1: Definitional	No change	No change	No change
SCS1: Tracing	Much Higher	Somewhat Higher	Slightly Higher
SCS1: Code Completion	Much Higher	Somewhat Higher	Slightly Higher
Course Exams: Def. / Reading	Much Higher	Somewhat Higher	Slightly Higher
Course Exams: Writing	Much Higher	Somewhat Higher	Slightly Higher

### 5.6.1.2 SCS1 assessment questions

We collected student responses on the SCS1 assessment in order to compare student performance in the baseline and intervention semesters. Students in both semesters were offered the option to sit for the SCS1 assessment at the end of the semester for extra credit. All students in both conditions took the same assessment, whose questions are categorized by type into definitional, tracing, and code completion questions [98]. I computed overall scores on the SCS1 as well as scores by question type.

### 5.6.1.3 Course examination questions

In order to contrast code reading and code writing skills, we also collected student responses and grades from course examinations. Midterm Exam 1 and Exam 2 had two sections each – one with code reading and definitional questions (Figure 5-8) and another with a code writing question (Figure 5-9) – while the Final Exam had only definitional and code reading questions due to logistical limitations and grade deadlines.

<p>Consider the following code snippet:</p> <pre>String a = "Hello"; String b = a; int c = 47; int d = c;</pre> <p>Which of the following is true?</p> <p>I. <b>a</b> and <b>b</b> are references to the same location in memory [ T / F ]</p> <p>II. <b>a</b> and <b>b</b> store the same value [ T / F ]</p> <p>III. <b>c</b> and <b>d</b> are references to the same location in memory [ T / F ]</p> <p>IV. <b>c</b> and <b>d</b> store the same value [ T / F ]</p>	<p>What is the output of the following code snippet?</p> <pre>public static void main(String[] args) {     int matrix[][] = new int[4][3];     for (int i = 0; i &lt; 4; i++)     {         for (int j = 0; j &lt; 3; j++)             System.out.print(matrix[i][j] + " ");         System.out.println();     }     System.out.println(); }</pre> <p><input type="radio"/> 000      0      0000</p> <p><input type="radio"/> 000    <input type="radio"/> 00      <input type="radio"/> 0000    <input type="radio"/> 00000000000000</p> <p><input type="radio"/> 000      000      0000</p> <p><input type="radio"/> 000      0000      0000</p>
--	--

Figure 5-8. Definitional (left) and code reading (right) question samples from Exam 1.



You have been tasked with creating a new security system for the office. However, to throw off any scallywags who dare try to infiltrate it, you have been asked to add an additional requirement to the password - it must be a palindrome.

A palindrome is a word that is spelled the same forwards and backwards - for example, "racecar" or "madam". Additionally, in order to save space only half of the password has been saved in memory (i.e., if the password were "kayak", only "kay" would be stored). *Note that the middle letter is not repeated.*

**Your task is to write methods to read and validate new passwords for the system.**

The methods will interact with the user (as described below) to accept and test a password. The solution should consist of **at least** one primary method and one helper method. *You may assume all input is lowercase and that the password has an odd number of characters.*

Figure 5-9. Code writing question from Exam 1 (abbreviated).

For the course exam definitional and code reading questions, not all question topics and formats appeared across semesters due to exam date variation. To eliminate these differences as a confounding factor, I identified a subset of questions for Exam 1, Exam 2, and the Final Exam that were in common across semesters (Table 5-8). While Exam 1 and the Final Exam had nearly or exactly the same number of questions, Exam 2 differed in length: the baseline group's exam was shorter. For Exam 1, 10 of 16 (62.5%) questions from the baseline semester overlapped with 10 of 15 (66.7%) questions from the intervention semester, while for Exam 2, 5 of 10 (50%) questions in the baseline overlapped with 5 of 16 (33.3%) in the intervention term. Finally, on the Final Exam, 11 of 16 (68.8%) questions overlapped between the exams.

Table 5-8. List of Topics in Common by Exam

Midterm 1	Midterm 2	Final
Instructions	Classes	Instructions
Arithmetic	Encapsulation	Arithmetic
Selection	Overloading	Data Types
Data Types	Inheritance	Functions
Functions	Overriding	Arrays
Arrays		Loops
References		Versioning
		Data Streams

The code writing exam section questions were isomorphic variants of one another between semesters; that is, they required employing the same skills and tested the same concepts. For example, on the first midterm in both classes, the code writing section required students to write and invoke simple methods and engage in console I/O. By the same token, on the second midterm for both classes, the writing section required writing and extending classes, overloading, and overriding. As such, I was able to directly compare the results. As with the definitional / code reading section, I calculated percentage scores for each exam before comparison.

#### **5.6.1.4 Analysis tests**

Once I had collected the scores from all of the exams and assessments for overlapping questions, I compared the scores in the baseline group to those in the intervention group. As the scores did not follow a normal distribution, I employed the non-parametric two-tailed Mann-Whitney U test [93] to compare the groups. Further, I calculated the eta-squared ( $\eta^2$ ) value to identify the effect size and report it with my findings.

In order to identify interactions between students with different prior experience levels – those who have worked previously only in text, those who have worked in blocks, and those with no prior experience – I used Aligned Rank Transform (ART) [137] to transform the data and make it suitable for use with ANOVA. When interactions were significant, I also performed interaction contrasts to identify differences in scoring by condition dependent on differences in experience [79].

#### **5.6.2 Surveys, logs, and notes**

In this study, I used several measures to collect sets of qualitative and quantitative data via surveys (Table 5-9). These included Binary, Likert, and Free Response questions. In this section I describe the methods I used to analyze these data sets.

Table 5-9. Dependent Variables

Measure	Instrument	Data Collected	Analysis
Material Use	Plugin Logs	Sessions	Time Series
	Canvas Logs	Frequency of Use	Time Series
Instruction Perception	Module Survey	Binary Response	Time Series
		Free Response	Codes
Blocks & Text Perceptions	Mid-Post Survey	Likert Response	ANOVA
		Free Response	Codes
	Pre-Mid-Post Surveys	Likert Response	ANOVA
		Free Response	Codes
Instructor Perceptions	Notes	Text	Review

### 5.6.2.1 Qualitative data

I collected qualitative data from survey answers to free response prompts. Of the 252 students who completed the demographic survey, I qualitatively coded responses from these prompts for a sample of 63 (25.0%) students. I selected these students to maximize coverage of ethno-racial, age, educational level, gender, demographic and experience groups. I coded the responses in modules 1, 3, 4, 7, and 11. Modules 1 and 11 were included as they are the first and last module surveys, respectively; 3 and 4 were selected because they cover programming fundamentals (loops, data types, and functions) just before Exam 1; and I coded module 7 as it covered the period in which I changed the instructional approach from dual-mode to text-only instruction. I completed the coding in a four-step process. In the first step, to establish an initial set of codes for the responses, I used the qualitative coding approach described by Auerbach and Silverstein to develop a list of repeating ideas and refine them via iteration [4]. In the second step, two other researchers and I independently coded responses from three participants and discussed disagreements in order to refine the codebook. We adjusted some codes based on the discussion, and we also combined codes we determined overlapped significantly. In the third step, the other researchers and I coded an additional 8% of the samples, which I used to perform an inter-rater reliability analysis using Fleiss kappa [47]. Finally, in the fourth step, I coded the

remaining samples to complete the data set. The average agreement between coders was Fleiss kappa = 0.601, which is characterized as *moderate* agreement [128].

### **5.6.2.2 Quantitative data**

I collected quantitative data from several measures, including Likert and binary responses to survey questions, and logs from the plugin and Canvas. Likert scores were analyzed via ANOVA [44] to identify differences between the baseline and intervention groups. I plotted responses to module (weekly) questions about material use, perceptions of dual-modality instruction, and plugin and Canvas logs by module in a time series so that they could be compared time-wise for triangulation [22].

### **5.6.2.3 Surveys**

Survey responses were used to provide insight into student perceptions and the relationship between the condition of instruction and the overall classroom experience of students. I analyzed module surveys to elicit patterns in student perceptions of dual-modality instruction. For module surveys, I examined responses to detect differences in perceptions over time and also considered them in the context of differences in usage patterns for the dual-modality IDE plugin and materials.

### **5.6.2.4 Usage logs**

Plugin and Canvas resource logs were examined to identify trends in student use of scaffolding. Plugin logs were evaluated to identify programming sessions, while Canvas resource logs were examined to determine frequency of use. I sought to identify how often students used the dual-modality IDE plugin and materials in order to more clearly link the dual-modality programming environments to differences in knowledge, cognitive level, and perceptions. I also explored how often students switched between blocks and text. I examined logs in segments by module to help me identify changes that occurred over the course of the term.

To identify usage of lecture slides on Canvas, I identified a two-week timeframe for each set of slides that covered the introduction, conclusion, and first quiz or exam covering the topic (Table 5-10). If access to the slide set occurred within this coverage window, I marked the slides for that module as having been used by the student for the purposes of this analysis. For plugin usage, I identified a window covering the beginning of the module to the end of the module (corresponding to the beginning of the next module – Table 5-11), and if the plugin was used in that time window for interactive events (such as using blocks or switching modes), the plugin was marked as being used by the student. Interactive plugin events were grouped into the categories of “Block Use”, “Palette Viewing”, and “Mode Switching” for the analysis (Appendix K). **Palette Viewing** actions were those in which the student selected a category of blocks to view (such as “Control”, “Classes”, or “Variables”); **Mode Swapping** was logged whenever a student switched from blocks to text mode or vice versa; and **Block Use** actions are those in which a student selected (dragged) a programming block and/or placed (dropped) a block within the program window. For each module, I then calculated the percentage of students who used the lecture slides and plugins. I also calculated the average percentage of students using the lecture slides and plugin over all module time windows.

Table 5-10. Time Window for Lecture Slide Usage by Module

<u>Module</u>	<u>Date Range</u>
0	2018/08/22 – 2018/09/04
1	2018/08/29 – 2018/09/11
2	2018/09/05 – 2018/09/18
3	2018/09/12 – 2018/09/25
4	2018/09/19 – 2018/10/02
5	2018/10/03 – 2018/10/16
6	2018/10/10 – 2018/10/23
7	2018/10/17 – 2018/10/30
8	2018/10/24 – 2018/11/06
9	2018/11/07 – 2018/11/20
10	2018/11/14 – 2018/11/27
11	2018/11/28 – 2018/12/11

Table 5-11. Time Window for Plugin Usage by Module

Module	Date Range
0	2018/08/22 – 2018/08/28
1	2018/08/29 – 2018/09/04
2	2018/09/05 – 2018/09/11
3	2018/09/12 – 2018/09/18
4	2018/09/19 – 2018/10/02
5	2018/10/03 – 2018/10/09
6	2018/10/10 – 2018/10/16
7	2018/10/17 – 2018/10/23
8	2018/10/24 – 2018/11/06
9	2018/11/07 – 2018/11/13
10	2018/11/14 – 2018/11/27
11	2018/11/28 – 2018/12/11

### 5.6.2.5 Instructor notes

Instructor notes were used to provide insight into the instructor’s (i.e., my) perspective of the classroom experience when using dual-modality instruction. These notes helped establish relationships between instructor observations and student experience.

### 5.6.3 Summary

Through analysis of the study data, I attempted to answer questions regarding dual-modality instruction (intervention) and its connection to student learning and perceptions as compared to text-based instruction (baseline). In addition to examining the overall connection between the instructional approach a student knowledge, I also examined the role prior experience plays – including the differences between prior experience in blocks and text-only programming. When analyzing results, I examined student use of the dual-modality IDE plugin and materials to verify that students did indeed make use of them. Finally, I examined the impact of using dual-modality tools and curricula on the classroom experience and detailed student and instructor perceptions, including receptiveness, perception of effectiveness, and appropriateness for various programming topics. In Chapter 6, I discuss the findings of these analyses and discuss their implications.

## CHAPTER 6

### LEARNING & DUAL-MODALITY INSTRUCTION: FINDINGS & DISCUSSION

In this chapter I describe the findings of my study on the use of dual-modality instruction in UF's CS1 course. This chapter is organized in parts by research question. Section 6.1 describes the findings and implications of RQ1, in which I compare the performance differences between students in the baseline (traditional, text-based instruction) and intervention (dual-modality instruction) semesters. Section 6.2 describes the findings and implications of RQ2, in which I examined connections between prior programming experience and instructional condition (traditional, text-based vs dual-modality instruction). Section 6.3 describes the findings and implications of RQ3, which is focused on student and instructor perceptions when utilizing dual-modality instruction. Finally, Section 6.4 summarizes my findings from the research study.

#### **6.1 Performance Comparison in Dual-Modality vs Text Instruction**

To answer my first research question (RQ1) – *“How do students perform in code reading and writing after learning with dual-modality instruction, as compared to students learning with traditional (text-based) approaches to instruction in CS1 courses?”* – I compared student performance on exams and assessments. This included midterm Exam 1, midterm Exam 2, and the Final Exam, as well as the SCS1 assessment taken at the end of the semester. In this section I outline my findings for each examination and assessment.

##### **6.1.1 Course Exam Results**

I evaluated student performance on the course exams according to exam section question types (i.e., definitional / reading or writing). I used the Mann-Whitney U Test to analyze the exam scores due to their non-normal distribution. Course midterm exams were divided into two sections – one section included code reading and definitional questions (Section 5.6.1), while the

other section had code writing questions. The Final Exam had only code reading and definitional questions. In this section I examine these sections individually.

#### **6.1.1.1 Code reading & definitional questions**

For the code reading / definitional sections, I compared questions on topics shared between the exams (e.g., Arithmetic, Data Types, and Classes – see Table 5-8). On these questions, students in the intervention group, which learned via dual-modality instruction, scored higher on both midterm exams than the baseline group, which learned via text instruction, and the difference between the groups on both exams was statistically significant ( $\alpha=0.05$ ). On Exam 1, the intervention average ( $\mu_{\text{intervention}}=85.4\%$ ) was higher than the baseline average ( $\mu_{\text{baseline}}=58.3\%$ ) with a large effect ( $d=0.7$ ), and once more the result was significant ( $\alpha=0.05$ ) when comparing the groups ( $Z=-4.1$ ,  $p<.001$ ,  $\eta^2=0.03$ ). In summary, the students in the intervention group outperformed the students in the baseline group in every code reading / definitional section of the course exams (Table 6-1). I discuss these findings further in Section 6.1.3.

#### **6.1.1.2 Code writing questions**

The code writing sections of the exams tested the same content across semesters, with the intervention semester using isomorphic variants of questions from the baseline semester. In these sections, I saw significant differences between the conditions on Exam 1, but on Exam 2 I did not. For Exam 1, students in the intervention group ( $\mu_{\text{intervention}}=76.1\%$ ) scored significantly higher ( $\alpha=0.05$ ) than those in the baseline group ( $\mu_{\text{baseline}}=68.9\%$ ) with a small-to-medium effect size ( $Z=-4.3$ ,  $p<.001$ ,  $\eta^2=0.03$ ), while for Exam 2, means were not statistically different ( $\mu_{\text{baseline}}=68.0\%$ ,  $\mu_{\text{intervention}}=68.7\%$ ,  $Z=-0.2$ ,  $p=.826$ ,  $\eta^2=0.00$ ). In short, the intervention group outperformed the baseline group on the code-writing section of Exam 1, taken earlier in the semester, but not on the code-writing section of Exam 2, taken later in the semester (Table 6-1).



Table 6-1. Results Summary for Course Exams (Scores as Percent)

Questions	Baseline $\mu$ , $\sigma$	Intervention $\mu$ , $\sigma$	P-val.	Z	$\eta^2$
Exam 1 – Def. & Reading	58.3, 17.3	85.4, 13.9	<.001	-16.4	0.41
Exam 1 - Writing	68.9, 25.0	76.1, 24.4	<.001	-4.3	0.03
Exam 2 – Def. & Reading	72.7, 19.8	76.4, 18.6	<.001	-2.8	0.01
Exam 2 - Writing	68.0, 29.5	68.7, 25.8	0.826	-0.2	0.00
Final Exam	65.8, 18.3	72.0, 15.5	<.001	-4.1	0.03

### 6.1.2 SCS1 Results

To compare the results on the SCS1 assessment across the intervention and baseline conditions, I used the Mann-Whitney U Test as the scores did not follow a normal distribution. The results from the SCS1 assessment did not differ significantly between the baseline group, who learned via text instruction, and the intervention group, who learned via dual-modality instruction (Table 6-2). In other words, there was no meaningful difference in the scores of the baseline and intervention group on the overall SCS1 score, despite the baseline students scoring 1.5% higher than the intervention group. There was also not meaningful difference by question type. This may be related to the attributes of the specific SCS1 questions (e.g., discrimination factor and difficulty), which I discuss in Section 6.1.3.

Table 6-2. Results Summary for SCS1 (Scores as Percent)

Questions	Baseline $\mu$ , $\sigma$	Intervention $\mu$ , $\sigma$	P-val.	Z	$\eta^2$
SCS1 - All	51.6, 18.9	50.1, 18.0	0.46	-0.7	0.00
SCS1 – Definitional	58.8, 19.4	57.5, 20.1	0.64	-0.5	0.00
SCS1 – Tracing	52.1, 21.2	49.9, 21.4	0.25	-1.2	0.00
SCS1 – Completion	43.8, 26.1	43.0, 23.3	0.79	-0.3	0.00

### 6.1.3 Performance Comparison Discussion

I had hypothesized that students would learn more effectively under dual-modality instruction, as it supports and scaffolds student learning of abstraction and chunking. As abstraction and chunking are critical to later stages of cognitive development [71], I believed students would score higher on exams and assessments in the dual-modality instructional

condition. My results show that students in the intervention group outperformed students in the baseline group on questions dependent on concrete- and formal-operational reasoning for most course exam sections, but not on the SCS1. In this section I detail these results by assessment, exam, and section.

### **6.1.3.1 Course Exam performance comparison discussion**

When I compared scores from questions covering shared topics across semesters on in-class examinations, scores between the groups differed significantly on both the definitional / code reading section and writing section of Exam 1. All topics from Exam 1 (Section 5.6) were covered using dual-modality instruction. As such, I expected students in the intervention to outperform students in the baseline group, and that is what I found for both the definitional / code reading as well as the code writing sections of the exam.

On Exam 2 and the Final Exam, the intervention group outperformed the baseline group on the definitional / code reading sections, though to a lesser degree than on Exam 1. It is notable that the definitional / code reading sections of both Exam 2 and the Final Exam included a mix of topics covered in dual blocks-text instruction (using dual-modality representations) and pure-text instruction (using only text representations). As such, I would expect to see less of a difference between the groups in these sections. In line with these expectations, compared to Exam 1, the average scores on Exam 2's and the Final Exam's definitional / code reading sections were closer between the baseline and intervention groups, though the differences were still statistically significant. Further, Exam 2's definitional / code reading section had fewer questions in the baseline semester, but the exam was given in the same amount of time, giving baseline students more time per question. Despite this advantage, students in the intervention semester scored higher than students in the baseline semester.

By comparison, there was not a significant difference on the code writing section of Exam 2. The topic of the code writing section of Exam 2 was inheritance; this topic was not covered by the dual-modality instruction and instead was taught exclusively in text, because it was necessary to transition students entirely to text before the end of the course. For this reason, I did not develop visualizations for inheritance relationships in Droplet. In other words, for those exam sections which included topics covered exclusively in text instruction, there was not a difference in scores.

In summary, considering the exams over time, a consistent pattern emerges. Students in the intervention group outperformed those in the baseline group on every section of every exam that incorporated content that was covered in the dual-modality instruction; only Exam 2's code writing section, which exclusively covered material that only used text-mode instruction, did not show significant differences in scores. In addition, the performance differences extended through to the Final Exam, which covered topics from Exam 1 and Exam 2. These differences were not limited to the time period of the dual-modality instruction but persisted to the end of the course, even after the change to text-only instruction. In other words, in line with my hypothesis, when tested on topics covered by dual modality instruction, students scored better in the intervention than the baseline; when I tested on topics covered exclusively in text, the students in the two conditions scored about the same as one another. This suggests that the concepts covered in dual-modality instruction were clearly anchored in students' minds, and as a result, they retained this knowledge through to the end of the course.

When comparing scores between the baseline and intervention groups, it is useful to do so within the Neo-Piagetian framework for novice programmers [71]. This allows us to review my results in terms of cognitive stages of development. Students can trace and write simple,

individual lines of code at the pre-operational level. However, reading and writing multi-line, complex blocks of code – such as those present in the course exams – is tied to students’ abilities to recall and apply chunks [113, 45] and engage in higher-level reasoning (e.g., abstraction) [73]. This reasoning about abstract meaning is in line with the Neo-Piagetian framework’s concrete-operational stage (applying abstractions to familiar situations) and formal operational stage (applying abstractions to unfamiliar situations). On the reading and writing sections of the exams, the students in the intervention scored higher than those in the baseline group. Thus, I concluded that students in the intervention group were more often functioning at the concrete and formal operational reasoning stages when compared to the baseline group.

#### **6.1.3.2 SCS1 performance comparison discussion**

In my study, I found that the SCS1 did not help distinguish between the baseline and intervention group – as both groups scored near 50% – despite the large sample size. As a result, the SCS1 data did not help me evaluate whether student learning was different in the baseline and intervention group. The authors of the SCS1 determined in their work that the SCS1 questions overwhelmingly skewed to hard levels of difficulty, and most questions they classified as fair, but not good, in their effectiveness at discriminating between students of different ability levels [98]. Thus, in my study, the lack of difference in and general low value of the scores on the SCS1 between the baseline and intervention groups may be due in part to the SCS1’s difficulty and limited capacity to discriminate between students of different ability levels.

The results on the SCS1 contrast with my findings on the course exams. In particular, I had anticipated that the subset of questions identified as “code-completion” questions by the assessment’s authors [98] would align with the code writing sections of the exams, but they did not. This may be due in part to the fact that, while the code completion questions on the SCS1 are multiple choice and scored as either “right” or “wrong”, the course exam questions were free

response pseudocode questions. These free response questions were graded with a rubric to award partial credit, capturing more nuance in scores regarding student understanding. In addition, the course exam questions were developed to address the specific topics covered in the course, including some topics that were not covered by the SCS1 (e.g., object-oriented programming), though all SCS1 topics were covered in the course.

#### **6.1.4 Performance Comparison Summary**

In this study, I investigated student learning and dual-modality instruction using course examinations and the SCS1 assessment. I had expected to see differences in groups through both the course exams and SCS1. My analysis showed differences between the intervention and baseline groups in the course exams, but not the SCS1. When considering the course exams, students in the intervention group outperformed those in the baseline group on every section of every exam that incorporated content that was covered in the dual-modality instruction. These questions involved code reading and code writing. Code reading and writing are built on the ability to recall patterns via chunking and engage in abstraction; these skills are central to the later stages of cognitive development in the Neo-Piagetian framework – namely, concrete operational reasoning and formal operational reasoning. However, as noted, I did not see differences in the SCS1 assessments; this may be related to the SCS1’s shortcomings in discrimination ability and high difficulty, as noted by the SCS1’s authors, as well as differences in topic coverage and free response question grading in the course exams versus the exclusively multiple-choice approach of the SCS1. Thus, the SCS1 may not yield a complete picture of the knowledge and abilities of the students in this particular course.

This work has important implications regarding student learning in early course science courses such as a typical CS1 course. Prior work by Corney et al [32] found that most students who complete a typical CS1 course are at the preoperational or early concrete operational stages;

as Corney et al point out, those students at the preoperational stages are “woefully under-prepared” for traditional programming assignments – those which students are likely to encounter increasingly in coursework. The higher performance among students in the intervention group compared to the baseline group on questions focused on skills fundamental to these later Neo-Piagetian stages suggests that dual-modality instruction may help students progress beyond the preoperational stage and master concrete-operational thinking, thus preparing them for programming tasks they are likely to see in advanced courses and the industry.

The results on the SCS1 also merit consideration. Based on the results from this study, as well as the evaluations of the authors of the SCS1, there are limitations to the SCS1 (e.g., difficulty, discrimination, and binary responses). As a result, the SCS1 may not be well-suited for some populations of students. I discuss these in further detail in Section 6.2.3.

## **6.2 Performance Comparison by Prior Experience**

My second research question (RQ2) focused on the relationship between instructional condition and prior programming experience: “*How does prior programming experience affect students learning in dual-modality instruction as compared to students learning in traditional (text-based) approaches to instruction in CSI courses?*” To explore the relationship between prior experience and instructional condition, I analyzed the interaction between course examination section (code definition / reading and code writing) and the SCS1 assessment scores and type of prior experience – text only, blocks, or none. In this section I outline my findings among these interactions.

### **6.2.1 Course Exam Results**

I analyzed the exam sections (code definition / reading and code writing) separately when investigating interactions between the instructional condition and prior programming experience.

I expected to see differences between reading and writing questions, which depend on more advanced skills (e.g., chunking, recall, and abstraction) associated with later cognitive development states (e.g., concrete- and formal-operational reasoning). To identify the interaction between condition and type of prior experience, I used Aligned Rank Transform (ART) [137] to transform the non-parametric data into suitable form for use with ANOVA and performed interaction contrasts. In this section I describe these results.

### 6.2.1.1 Code reading / definitional questions

For the questions on topics shared between midterm exams (i.e., Exam 1 and Exam 2) that focused on definitions and code reading, there was no significant interaction between the instructional condition and type of prior experience (e.g., text-only, blocks, or none) with respect to the exam scores (Table 6-3). However, on the Final Exam, which was composed of only code reading and definitional questions, I found an interaction between prior experience type and condition with respect to exam score on question topics shared between the baseline and intervention semesters ( $F_{2,360}=4.4$ ,  $p=.013$ ,  $\eta^2=0.02$ ) (Table 6-3). In particular, there were differences between students **with no prior experience** and those **with prior experience** between conditions (Table 6-4, Figure 6-1):

- Students with prior text-only experience scored higher in the intervention ( $\mu=79.2$ ,  $\sigma=13.2$ ) than those in the baseline ( $\mu=66.5$ ,  $\sigma=16.7$ ) ( $p<0.001$ );
- Students with prior blocks experience scored higher in the intervention ( $\mu=74.8$ ,  $\sigma=13.9$ ) than those in the baseline ( $\mu=58.8$ ,  $\sigma=17.4$ ) ( $p<0.001$ );
- Students with no experience scored similarly between the conditions; and
- There was no significant difference between students with blocks and text experience between conditions.

In other words, on the Final Exam, students with prior experience – whether blocks or text – scored higher in the intervention group than the baseline group. However, among those with no prior experience, the scores were about the same in the two condition groups.

Table 6-3. Course Exam Interactions: Condition x Experience

Questions	F <sub>2,360</sub>	P-val.	$\eta^2$
E1 – Definitional & Reading	1.9	0.138	0.01
E1 - Writing	5.7	0.003	0.03
E2 – Definitional & Reading	1.0	0.362	0.01
E2 - Writing	3.8	0.023	0.02
Final Exam	4.4	0.013	0.02

Table 6-4. Mean & Standard Deviation, Final Exam: Condition x Experience

Prior Experience	Baseline		Intervention	
	Mean ( $\mu$ )	Std. Dev. ( $\sigma$ )	Mean ( $\mu$ )	Std. Dev. ( $\sigma$ )
None	66.4	17.4	69.1	16.6
Blocks	58.8	17.4	74.8	13.9
Text-Only	66.5	16.7	79.2	13.2

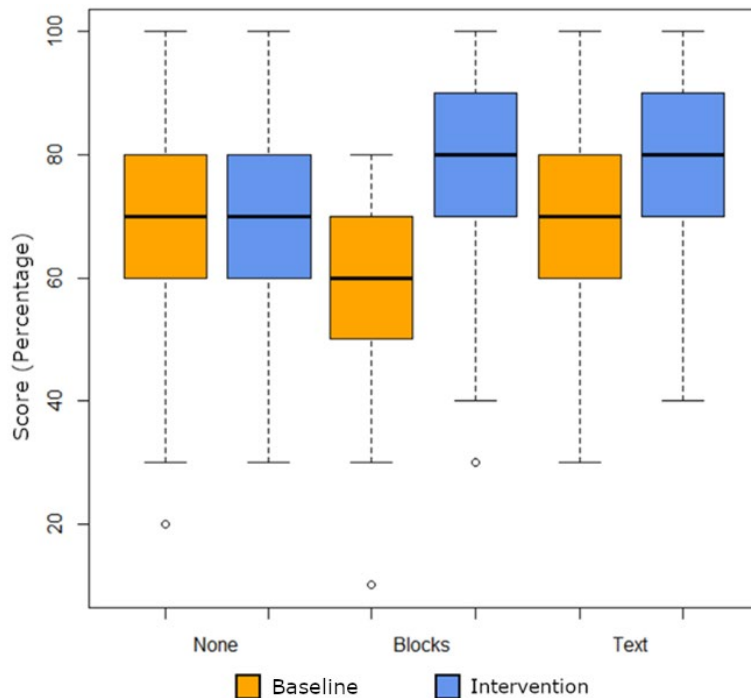


Figure 6-1. Boxplot of Final Exam scores by condition and prior programming experience.



### 6.2.1.2 Code writing questions

On the writing sections of both midterm examinations, the interaction between prior experience types and condition with respect to exam score was significant, even when a main effect was not present (Table 6-3, Appendix J) – that is, even when overall I did not see a difference between the condition groups, when I examined performance broken out by the students' prior experience (text-only, blocks, or none), there were differences.

For Exam 1's code writing section, on which there was a main effect of the condition overall, the interaction between condition and prior experience was significant ( $F_{2,360}=5.7$ ,  $p=.004$ ,  $\eta^2=0.03$ ) (Figure 6-2). The difference between conditions among students with only prior text experience was significantly different than the difference between conditions among students with blocks experience ( $p=.012$ ). Likewise, the difference between conditions among students with only prior text experience was significantly different than the difference between conditions among students with prior experience ( $p=.012$ ).

Specifically, there were differences between students **with text-only experience** and those **with no experience or experience in blocks** between conditions (Table 6-5, Figure 6-2):

- Students with prior text-only experience scored higher in the intervention ( $\mu=89.2$ ,  $\sigma=15.0$ ) than those in the baseline ( $\mu=70.6$ ,  $\sigma=23.6$ ) ( $p<0.001$ );
- Students with prior blocks experience scored similarly between conditions;
- Students with no experience scored similarly between the conditions; and
- There was no significant difference between students with no experience and those with blocks experience between conditions.

In summary, the students with only prior text experience performed better in the intervention group on Exam 1's code writing section, but there were no differences between the conditions for students with blocks experience, nor for those with no experience.

Table 6-5. Mean & Std. Deviation, Exam 1, Writing: Condition x Experience

Prior Experience	Baseline		Intervention	
	Mean ( $\mu$ )	Std. Dev. ( $\sigma$ )	Mean ( $\mu$ )	Std. Dev. ( $\sigma$ )
None	60.7	22.8	60.8	25.9
Blocks	79.7	29.6	84.6	17.5
Text-Only	70.6	23.6	89.2	15.0

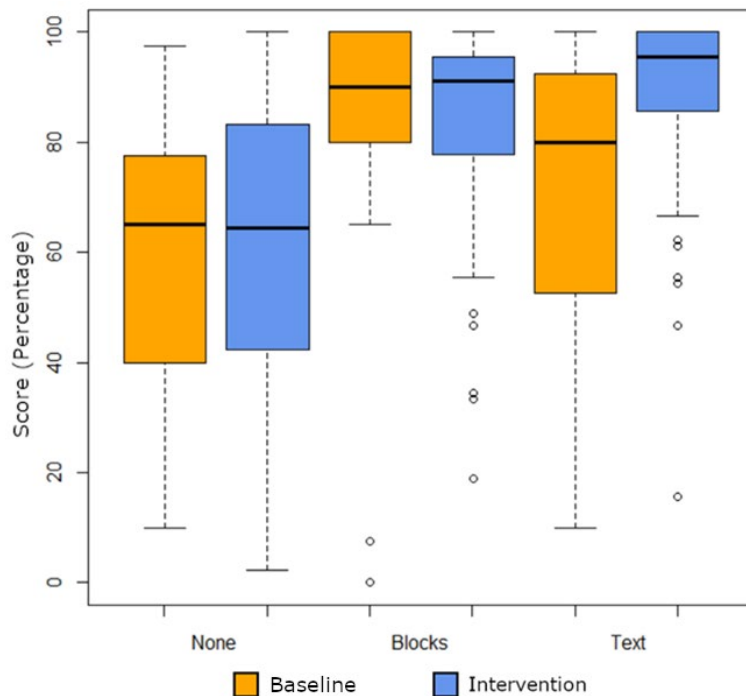


Figure 6-2. Boxplot of Exam 1 writing scores by condition and prior programming experience.

While analysis of Exam 2’s code writing section did not exhibit significance overall with respect to condition, there was nevertheless a significant interaction between condition and prior experience with respect to exam score ( $F_{2,360}=3.8$ ,  $p=.023$ ,  $\eta^2=0.02$ ) (Figure 6-3). There were differences between students with only prior text experience and those with no prior experience ( $p=.021$ ). Additionally, students with no prior experience did slightly worse in the intervention group compared to the baseline group (Table 6-6):

- Students with prior text-only experience scored similarly between conditions;
- Students with prior blocks experience scored similarly between conditions;

- Students with no experience scored slightly worse in the intervention ( $\mu=58.1$ ,  $\sigma=25.9$ ) than in those in the baseline ( $\mu=67.6$ ,  $\sigma=27.2$ ) ( $p=0.023$ ); and
- There was no significant difference between students with blocks experience and text-only experience between conditions.

In short, the students with prior experience – blocks and/or text – performed about the same in the intervention and baseline groups on Exam 2’s code writing section, but students with no prior experience performed more poorly.

Table 6-6. Mean & Std. Deviation, Exam 2, Writing: Condition x Experience

Prior Experience	Baseline		Intervention	
	Mean ( $\mu$ )	Std. Dev. ( $\sigma$ )	Mean ( $\mu$ )	Std. Dev. ( $\sigma$ )
None	67.6	27.2	58.1	25.9
Blocks	74.4	24.5	74.9	22.6
Text-Only	69.5	30.5	78.5	18.8

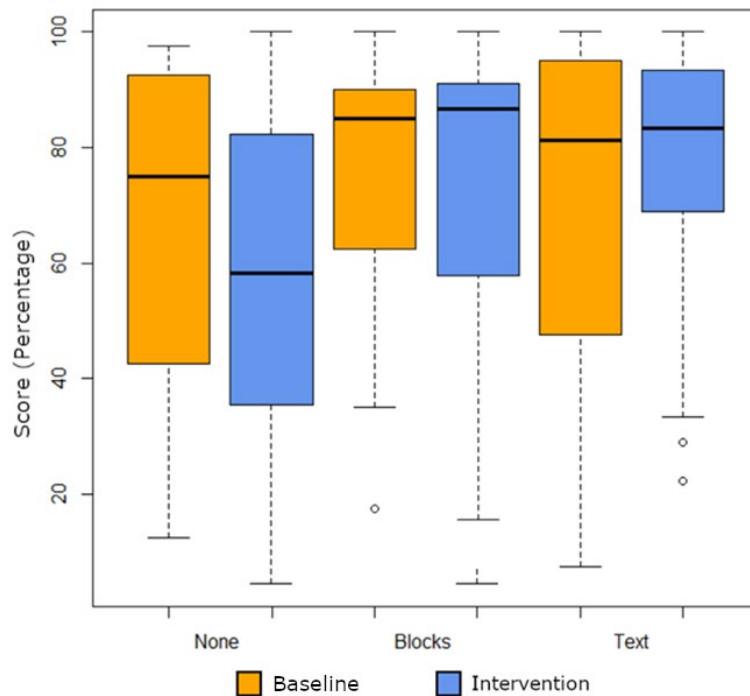


Figure 6-3. Boxplot of Exam 2 writing scores by condition and prior programming experience.

## 6.2.2 SCS1 Results

To identify interactions between condition and type of prior experience on the SCS1, I used Aligned Rank Transform (ART) [137] to transform the data and then performed interaction contrasts. There was no significant interaction between the instructional condition and type of prior experience with respect to the results from the SCS1 assessment (Table 6-7, Appendix J), mirroring the results I found when examining the main effect between the intervention and baseline conditions. This was true on the over assessment scores, as well as scores I computed by question type (definitional, tracing, and code completion) on the SCS1. In other words, I did not find any difference between students with different types of prior programming experience on the SCS1 assessment.

Table 6-7. SCS1 Interactions: Condition x Experience (See Appendix J for Means / Std. Dev.)

Questions	F <sub>2,360</sub>	P-val.
SCS1 - All	1.8	0.170
SCS1 – Definitional	0.4	0.683
SCS1 – Tracing	0.3	0.707
SCS1 – Completion	2.3	0.100

## 6.2.3 Prior Experience Discussion

I hypothesized that those students with no prior experience would have the most to gain from the scaffolding and cognitive support provided by dual-modality programming environments (e.g., construct visualization and association of blocks with text syntax), as they would have limited mental models of programming constructs and algorithms; in other words, those with no experience would be at the sensorimotor stage in the Neo-Piagetian Framework for novice programmers [71]. On the other hand, I had hypothesized that those with prior text experience would stand to gain the least, as prior to taking the class, they would already have established mental models of text-based constructs which they could depend on and recall – they

would be more likely to be at the concrete-operational or formal operational stages of the Neo-Piagetian framework.

While those in the intervention group overall scored higher on most exams than those in the baseline group, I most consistently saw significant differences among those with prior text experience, and least among those with no experience. This is a contradiction of my hypothesis: I had expected students with less experience to gain the most from the intervention, and that those with the most experience would benefit the least. I now believe that the students with prior experience performed higher in the intervention because of reinforcement of existing mental models which helped students with prior experience move from concrete-operational to formal operational stages of expertise. In this section, I will discuss the implications of my findings on the interaction of prior experience type and condition.

#### **6.2.3.1 Course exam discussion**

While those in the intervention group scored significantly higher than those in the baseline group overall, I saw greater and more frequent differences between the students with prior text experience when comparing the baseline and intervention conditions than those students with no prior experience or experience in blocks. This was contrary to my hypothesis that those with the least experience would show the greatest positive difference between the intervention and baseline group – i.e., that those students with the least prior experience in the intervention group would outperform those in the baseline group by the greatest margin. I saw higher performance differences among those with text experience on the written portion of both midterm examinations (Exam 1 and Exam 2) and the final examination (Final Exam). In other words, while student scores overall were higher on most examinations in the intervention group compared to the baseline group, the difference was most stark among those with prior text experience. While I had hypothesized that students with no experience in the intervention would

outperform those in the baseline, on the writing sections and final examination, their differences compared to the baseline group were smaller than those with prior programming experience. Additionally, I had previously hypothesized that there would be difference on code reading sections of the exams, but there was no significant interaction on the definitional and code reading sections of the midterm examinations.

Perhaps as one might expect, students with no prior experience lagged behind those with experience on the Final Exam, but at the same level in the baseline and intervention. However, my analysis provides evidence that suggests students with prior blocks experience performed on about the same level as those students with text-only experience at the Final Exam in the intervention, and better than those in the baseline. This suggests that **dual-modality instruction may provide students support in reaching the same levels of expertise as those who learn exclusively in text** by the end of a CS1 course. This also suggests that dual-modality instruction may be a viable instructional approach.

On Exam 1's code writing section, students with text-only experience performed better in the intervention. With respect to these differences, it is prudent to consider the expertise level of these students, their cognitive level, and how it might impact learning. While code reading and writing both depend on abstraction and chunking, there are differences in how they are employed. Code writing depends on chunk-based **recall** of concepts and patterns which are used to construct new code to solve a problem or build functionality [45]. By comparison, code reading questions rely on **recognition** of constructs in order to trace and understand how a program functions which can lead students to develop a mental model of a program and its function [91]. Dual-modality instruction scaffolds learning by helping students chunk code into meaningful functional pieces, which may help learners without experience develop this

recognition effectively and may also help those with prior experience reinforce their pre-existing mental models of programming. However, this scaffolding – which helps students chunk code – was not available on examinations. Students with no prior experience may not have sufficiently developed the chunk-based recall mechanisms employed by experts [45]. I believe that the more experienced learners were able to rely on their more-refined existing mental models for chunk-based recall during Exam 1.

Exam 2's results are warrant reflection, as those with prior experience performed the same in both instructional conditions, but those with the least experience performed slightly worse in the intervention than the baseline condition. The differences between those with and without experience may rise from the timing of the end of the dual-modality instruction, which coincided with this exam. Though evidence I have presented suggests that students with prior experience may benefit from the scaffolding and cognitive support provided by dual-modality instruction, due to their previous experience, they had existing mental models they could depend on in addition to the dual-modality scaffolding. However, it may be that students with no prior experience were disadvantaged by losing a scaffolding that they had come to know and use in their learning of computer science too soon.

The Final Exam went beyond the material covered in Exam 1, which was covered exclusively in dual-modality instruction, and Exam 2 which was covered partially in dual-modality instruction, to content covered only in text, as the tools and environments in my study did not have blocks representations for advanced concepts such as programming paradigms and memory management. On the Final Exam, which comes at the end of the course, I found that students with no prior experience in the control and intervention semesters were similar, as they were in Exam 1. This may be because those students with no prior experience continued their

cognitive development and adapted to text instruction in the final weeks of the class as they spent more time without the dual-modality scaffolding. Prior studies have shown that students working in dual-modality programming environments become less dependent on blocks representations over time [9], and as experienced students would have begun the term with more knowledge than those new to programming, I would have expected them to grow beyond dependence on the scaffolding provided by the dual-modality programming environments sooner – in some cases, and in larger proportion, before the Final Exam, compared to those with no prior experience. It is important to note that in the absence of dual-modality instruction, I would have expected a similar pattern – i.e., students with more experience would make more rapid progress than those without experience in a typical classroom setting. Nevertheless, students in the intervention overall outperformed those in the baseline on all code reading exam sections, suggesting that students without experience may also have also benefited from the intervention.

#### **6.2.3.2 SCS1 discussion**

Like the overall comparison between the intervention and baseline conditions (RQ1), I did not find any significant interactions in the SCS1 assessment scores. This was true even when I found interactions on the course examinations. As a result, I was not able to draw conclusions about the answer to my research questions from this analysis.

As noted in Section 6.1.3, the SCS1 has high difficulty, only fair discrimination ability, and does not allow for partial credit [98]. In addition, the SCS1 uses a pseudocode language [98] that is distinct from the language students learned in the course (Java). These aspects of the SCS1 suggest there may be room for future CS concept inventories to build upon the work of the SCS1, and the work in this study, to investigate refinement of question styles and approaches. While the SCS1’s pseudocode was intended to make the SCS1 broadly applicable in computer science instructional contexts using various programming languages [122], its syntax is based on



markup-style opening and closing keywords for code section in functions and loops (e.g., “WHILE condition ... ENDWHILE”). This is very different from Java’s C-based curly braces style (“while condition {...}”), which makes the syntax potentially challenging for student who learn using C-family languages. Future concept inventories might be more effectively tailored to language families (e.g., C-family) to reduce the cognitive load of learning and applying a new syntax while taking an assessment. The increase in cognitive load is especially problematic when attempting to measure learning of novices, as those with little or no experience are less likely to have developed nuanced chunking mechanisms that allow experts to effectively transfer knowledge from one programming language to another [118]. Additionally, some topics – such as object-oriented programming – are commonly covered in CS1 courses, like UF’s, but these constructs are absent from the SCS1 and its predecessor, the FCS1 [122]. Integration of basic, common object-oriented principles – such as methods and attributes – into concept inventories like the SCS1 would allow instructors to test a broader spectrum of generally-accepted CS1 topics.

#### **6.2.4 Performance Comparison by Prior Experience Summary**

When investigating interactions between prior programming experience and instructional condition (dual-modality or text instruction), I found interactions on the code writing sections of Exam 1 and Exam 2, as well as the Final Exam, which was limited to code reading and definitions. I found no interactions on code reading / definitional sections of Exam 1 or Exam 2.

Specifically, on the code writing sections Exam 1, there was a significant positive difference between the intervention and baseline students with only text experience – i.e., those students with prior text experience performed significantly better in the intervention semester. This was contrary to my hypothesis – I had hypothesized that students with prior text experience would have more developed mental models of programming, and as a result I anticipated that

they would have less to gain from the scaffolding. Instead, this group performed the strongest in the intervention when compared to the baseline. On the writing section of Exam 2, the students without experience performed slightly worse in the intervention compared to the baseline, suggesting that more time with the scaffolding provided by dual-modality instruction may have benefited them. On the Final Exam, among those prior experience, text-only or blocks, there was a greater positive difference between the intervention and baseline groups – i.e., students with experience performed significantly better in the intervention group – and the students with no prior experience performed about the same in the intervention and baseline.

Considering the results in the context of the Neo-Piagetian framework [71] and its connection to abstraction and chunking, we can find suggestions as to the reason for these results. While reading and tracing code requires students to be able to recognize constructs and, especially for large programs, to develop mental abstractions of sections of code [73], writing code additionally requires students to recall constructs and abstractions from memory [45]. In other words, code **reading** depends on skills associated with concrete operational reasoning (reasoning abstractly about familiar situations), including developing abstractions of code they can see. By comparison, code **writing** depends on skills associated with formal operational reasoning (reasoning abstractly about unfamiliar situations). For example, code writing requires recalling patterns stored as chunks in memory in order to apply these stored abstractions to new, unfamiliar problems – e.g., code that must be produced from a problem description [45]. This suggests that students with prior experience were reaching the stage of formal operational reasoning by the end of the course, while those without experience progressed to the point where they were demonstrating concrete operational reasoning. As such, the students in the intervention – those with and without experience – were performing in the concrete-operational to formal-

operational range, compared to those in the baseline and those in typical CS1 classrooms, which Lister noted fall in the preoperational to concrete-operational range [73].

There are important implications for this work for students with different prior experience entering a CS1 course. Students with prior experience performed better in the intervention semester and never worse, suggesting this type of instruction may be useful even with experienced students. Beyond that, it is worth considering the timing of removal of scaffolding – in this case, the end of dual-modality instruction – and how it will impact less experienced students. In my study, in the exam just after dual-modality instruction stopped – Exam 2 – students without experience performed worse in the intervention than in the baseline. It is possible that the end of dual-modality instruction was premature for these students, and that they may have benefited from a longer intervention. Though beyond the scope of this dissertation, work exploring the timing of removal of this scaffolding could help future students and instructors in CS1 courses.

### **6.3 Classroom Experience of Dual-Modality Instruction**

In my third and final research question I asked, “*What are student perceptions of dual-modality programming environments and instructional approaches, and how do they change over time, in the context of a CS1 course?*” To investigate student perceptions of dual-modality programming environments and instruction, and how they change over time, I qualitatively coded student responses to survey questions about dual-modality instruction’s usefulness and analyzed the results. I also examined trends in log files detailing student use of the plugin and lecture slides. In this section I discuss both the student perceptions revealed by these data, along with my personal experiences teaching the course using these materials. This section uses randomly generated pseudonyms for each student in the study, which are composed of adjective-animal pairs produced by the PetName module in Python [60].

### 6.3.1 Student Perceptions of Dual-Modality Instruction

During each course module, students completed a survey that included the binary response question, “Does instruction in dual blocks-text modes help you learn better?”, along with a free response “Why do you feel this way?” prompt. I first coded each free response in the sample. Then, for each code, I counted the number of participants whose responses at a given module time fit that code and divided by the total number of participants to arrive at a percentage of participants whose responses fit that code.

From answers to these module survey questions, I found that more than half students (54.6%, n=137) perceived the dual-modality instruction as helpful in learning to program at the beginning of the course. Over time, the number of students that perceived dual-modality as helpful decreased – slowly until Module 7, when instruction switched to text-only (47.2%, n=183), and then decreased more rapidly. By the final survey in Module 11, 38.7% (n=136) felt the dual-modality instruction was helpful (Figure 6-4, Table 6-8). When I analyzed the qualitatively coded sample (Section 5.6.2) of student responses (25.0%, n=63) to the open-ended prompt about their reasons for saying dual-modality instruction was helpful in modules 1, 3, 4, 7, and 11, the most commonly cited reasons were Visualization (cited by 41.3%, n=26), Structure (22.2%, n=14), Understanding (20.6%, n=13), and Introduction (19.0%, n=12) (Table 6-9). While the reasons students cited (Table 6-10, Table 6-11, Appendix L) varied according to prior programming experience, as I discuss later in this section, Visualization was consistently cited by participants from all prior programming experience backgrounds as a reason they found blocks-based instruction helpful.

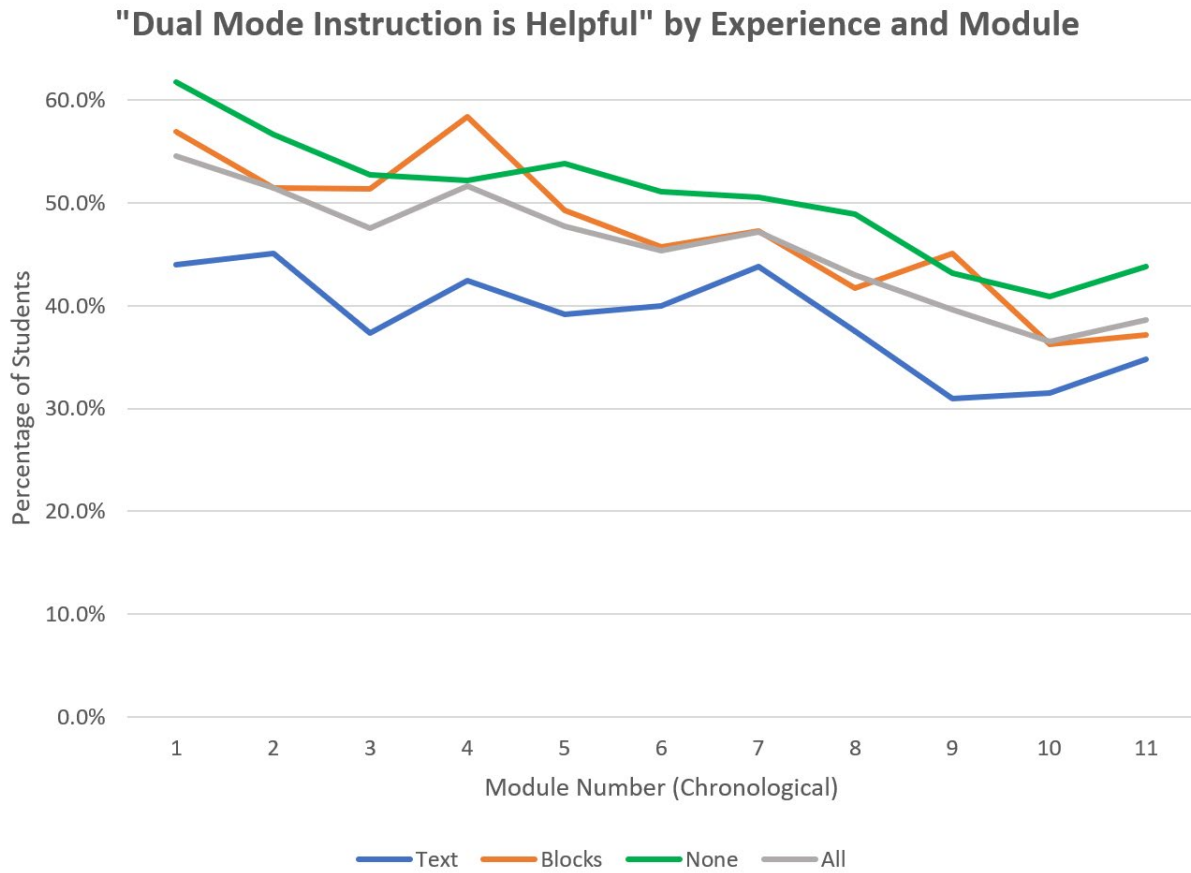


Figure 6-4. Percentage of students indicating dual-modality instruction was helpful, by module.

Table 6-8. "Dual Mode Instruction is Helpful", Range by Experience

Prior Experience	First (M1)	Highest	Lowest	Last (M11)
None	61.7%	61.7% (M1)	40.9% (M10)	43.8%
Blocks	56.9%	58.3% (M4)	36.2% (M9)	37.1%
Text-Only	44.0%	45.1% (M2)	31.0% (M9)	34.8%
All	54.6%	54.6% (M1)	36.6% (M10)	38.7%

Table 6-9. Common Codes and Examples

Code	Definition	Example
Introduction	Introducing constructs to beginners	“It should be used as a beginner introduction.”
Structure	Structuring code or understanding structure	“It... makes your work more structured”
Understanding	Understanding code / concepts generally	“For block-programming... the logic is easier to understand than text.”
Visualization	Related to visualization of the code, facilitating or inhibiting learning (function)	“I(t) helps visualize the code.”

Table 6-10. Responses: Why Dual-Modality Instruction is Helpful (n=63) (&gt;5% of Students)

Code	No Exp (n=27)	Blocks (n=19)	Text (n=17)	All (n=63)
Blocking	11.1%, n=3	21.1%, n=4	11.8%, n=2	14.3%, n=9
Colors	14.8%, n=4	10.5%, n=2	17.6%, n=3	14.3%, n=9
Formatting	3.7%, n=1	10.5%, n=2	5.9%, n=1	6.3%, n=4
Introduction	7.4%, n=2	31.6%, n=6	23.5%, n=4	19.0%, n=12
Learning (General)	7.4%, n=2	15.8%, n=3	5.9%, n=1	6.3%, n=4
Learning (Syntax)	11.1%, n=3	5.3%, n=1	0.0%, n=0	6.3%, n=4
Lectures	14.8%, n=4	10.5%, n=2	5.9%, n=1	11.1%, n=7
Organization	14.8%, n=4	5.3%, n=1	0.0%, n=0	7.9%, n=5
Reading	7.4%, n=2	10.5%, n=2	0.0%, n=0	6.3%, n=4
Scaffolding	18.5%, n=5	15.8%, n=3	17.6%, n=3	17.5%, n=11
Sequencing	7.4%, n=2	5.3%, n=1	5.9%, n=1	6.3%, n=4
Simplicity	11.1%, n=3	0.0%, n=0	5.9%, n=1	6.3%, n=4
Structure	22.2%, n=6	31.6%, n=6	11.8%, n=2	22.2%, n=14
Understanding	14.8%, n=4	36.8%, n=7	11.8%, n=2	20.6%, n=13
Visualization	<b>40.7%, n=11</b>	<b>47.4%, n=9</b>	<b>35.3%, n=6</b>	<b>41.3%, n=26</b>

Table 6-11. Responses: Why Dual-Modality Instruction is Not Helpful (n=63) (&gt;3% of Students)

Code	No Exp (n=27)	Blocks (n=19)	Text (n=17)	All (n=63)
Accustomed	<b>14.8%, n=4</b>	<b>15.8%, n=3</b>	0.0%, n=0	<b>11.1%, n=7</b>
Confusing	7.4%, n=2	0.0%, n=0	11.8%, n=2	6.3%, n=4
Dependency	7.4%, n=2	15.8%, n=3	<b>17.6%, n=3</b>	6.3%, n=8
Distraction	0.0%, n=0	0.0%, n=0	11.8%, n=2	3.2%, n=2
Experienced	3.7%, n=1	5.3%, n=1	0.0%, n=0	3.2%, n=2
Learning Syntax	3.7%, n=1	5.3%, n=1	0.0%, n=0	3.2%, n=2
Lecture	0.0%, n=0	5.3%, n=1	5.9%, n=1	3.2%, n=2
No Longer Needed	<b>14.8%, n=4</b>	0.0%, n=0	0.0%, n=0	6.3%, n=4
Speed	3.7%, n=1	10.5%, n=2	0.0%, n=0	4.8%, n=3
Unnecessary	7.4%, n=2	0.0%, n=0	5.9%, n=1	4.8%, n=3
Visualization	3.7%, n=1	0.0%, n=0	5.9%, n=1	3.2%, n=2

### 6.3.1.1 Participants with only text experience

Among students whose only prior experience was in text, 30.0%-45.1% of students said that dual-modality instruction was helpful to them throughout the semester, with the highest percent saying it was helpful in the Module 2 survey (45.1%, n=32) and the lowest in Module 9 (30.0%, n=22) (Figure 6-4). Text-experience students felt the dual-modality instruction was helpful in understanding concepts before writing text, with 35.3% (n=6) citing help in Visualization on at least one survey. One student said, “it helps me to learn the simpler way (blocks) before having to put concepts into practice (text)” (neutral-narwhal: Helpful, M3). Other text-experience students (11.8%, n=2) said that the dual-modality instruction and tools make for a handy Structure reference: “Just in case I forget something, I can see how its [sic] put together in blocks” (keen-kid: Helpful, M4).

Several students with text experience empathized with new learners, pointing out that the dual-modality instruction could help students as an Introduction to programming (23.5%, n=4). For example, one participant reflected on the experiences of new programmers, stating, “Blocks is good for new programmers” (fond-falcon: Helpful, M7), and another saying, “blocks is a nice visualization of the code, which should help to see the scope of blocks and variables” (correct-crane: Helpful, M1). Another pointed out that the dual-modality instruction was helpful in lecture when introducing new concepts: “Dual block text in class is helpful for highlighting general structured [sic] when they are introduced” (current-chipmunk: Helpful, M3).

When looking to the coded free response samples, 17.6% (n=3) of students who said dual-modality instruction was **not** helpful pointed to issues of authenticity and dependency, with students referring to the blocks scaffolding as a type of crutch: “I feel the usage of programming with blocks creates a programmer who is reliant on pre-set syntax” (pseudonym big-buzzard: Not Helpful, Module 3). Another 11.8% (n=2) said they found the blocks distracting, with one

student noting, “I feel like the different colors in block mode distracts me. I feel like I can visual(lize) [sic] what I am programming better in text mode” (fond-firefly: Not Helpful, M1).

### **6.3.1.2 Participants with only blocks or with both blocks and text experience**

More than half of students with prior blocks programming experience, on average, indicated that dual-modality instruction helped them learn in their answers to the first four module surveys (i.e., 56.9%, 51.4%, 51.4%, and 58.3%). In later modules, as students further developed their skills, the percentage declined. On the 11<sup>th</sup> and final survey, 37.1% (n=44) said they felt dual-modality instruction helped them (Figure 6-4). When asked why they indicated that dual-modality instruction helped them learn, participants with blocks experience – similar to those with only text experience – also ranked Visualization most often (47.4%, n=9), with one participant stating, “It helps to visualize more the regions of the code with the uses of colors shaped areas around the areas of the code” (ideal-ibex: Helpful, M1). Other commonly cited reasons included Understanding (36.8%, n=7), such as the student who said, “The blocks help... make the code easier to comprehend in the end” (modest-manatee: Helpful, M4). Some students indicated the dual-modality instruction helped them with Structure (31.6%, n=6); for example, one student mentioned, “it is easier to visualize the code and see which statement belongs where” (becoming-basilisk: Helpful, M3). Students with blocks experience also mentioned usefulness as an Introduction to programming (31.6%, n=6) and the effect of the Blocking mechanism (21.1%, n=4), with one student saying, “When learning a new programming language, the syntax and structure of the language can be confusing. Block code is a little easier to read and share with other beginners as well. Being able to switch back and forth between block and text, can help with identifying where a function or loop begins and ends, and what is encapsulated within it.” (equipped-emu, Helpful, M1).



In later surveys, some students indicated that they had progressed in their skills over time and used the blocks constructs less often as a result; their early survey responses (e.g., M1) described their use of blocks constructs, and their responses to later surveys (e.g., M4, M7) indicated that they no longer needed or used blocks constructs. On the first survey, for example, one participant said, “When learning a new programming language, the syntax and structure of the language can be confusing. Block code is a little easier to read... being able to switch back and forth between block and text, can help with identifying where a function or loop begins and ends, and what is encapsulated within it” (equipped-emu: Helpful, M1). In the third survey, the same participant noted “I primarily code in text mode, but sometimes it helps to jump into block mode to see the color code looping” (equipped-emu: Helpful, M3). On the fourth survey, the same participant noted that they no longer depended on blocks: “I didn't really use blocks this time around. I am just more comfortable using text” (equipped-emu: Helpful, M4).

### **6.3.1.3 Participants with no prior programming experience**

More than half of participants (50.5%-61.7%) with no prior programming experience indicated they found dual-modality instruction helpful during every module that dual-modality instruction was used, from the first survey through the seventh, in a range of 61.7% (n=58) to 50.5% (n=51) (Figure 6-4). As students progressed in the course and developed their skills, the percentage of students indicating that dual-modality instruction was helpful gradually decreased over successive modules; nevertheless, even at the end of the course (M11), nearly half (43.8%, n=39) still found dual-modality instruction helpful. This finding suggests that the scaffolding provided by dual-modality instruction continued to provide constructive support for student learning at the end of the course. Similar to the responses provided by participants with prior blocks experience, the responses from the participants with no prior experience most frequently cited Visualization (40.7%, n=11), Structure (22.2%, n=6), Understanding (18.5%, n=5), and

Scaffolding (18.5%, n=5), and similarly their responses showed growth and change over time. On the first survey, one participant said, “[it] allows for better visualization of Java language with blocks, but also allows for necessary learning of Java language through text (strings, variables, etc.)” (sure-shrimp: Helpful, M1); by the final survey, the same student responded differently: “[it] better visualizes the way the code is set up and should run, but now I feel as though I do not necessarily need it to understand the text code” (sure-shrimp: Helpful, M11).

#### **6.3.1.4 Perceptions of dual-modality instruction discussion**

The findings from this study suggest that the instruction provided affordances for students to identify meaningful chunks of code which assists students with abstraction of code functionality. Consistently, students from all experience backgrounds indicated that dual-modality instruction was helpful throughout the course. At the end of the course, 34.8% of students whose prior experience was exclusively in text said dual-modality instruction was helpful, along with 37.1% of blocks-experience students and 43.8% of students with no prior experience.

Irrespective of prior experience, students noted Visualization, Structure, and Understanding as the top reasons for their answers when they indicated that dual-modality instruction was helpful. Students with text-only experience cited visualizations on lecture slides as helpful. They also empathized with new learners, identifying ways in which they felt those new to programming might benefit from the dual-modality instruction. Similarly, students with blocks experience noted that the dual-modality instruction was helpful as an introduction to programming, and others said it helped them block out chunks of code. Students with no prior experience also brought up the role the dual-modality instruction played in providing scaffolding for chunking and abstraction. As such, the findings support my initial hypothesis that dual-modality instruction provided a blocking-mechanism support that would help students chunk to

develop code abstractions. Thus, these affordances of dual-modality instruction directly support chunking and abstraction skills characteristic of concrete and formal operational stages in the Neo-Piagetian framework for novice programmers.

Among those with blocks experience and those with no prior experience, the findings suggest students grew in their understanding of programming and relied on the scaffolding less over time – growth we would expect to see as students develop their computing skills. 56.9% (n=41) of those with prior blocks experience felt dual-modality instruction was helpful in the beginning of the class. By Module 5, just under half (49.3%, n=34) still said the dual-modality instruction helped them learn better. Students with no prior experience also showed progression toward lower use of dual-modality representations. This progression occurred over a longer range of time – i.e., they found the dual-modality representations helpful and made use of their supporting scaffolding longer. Comments at the end of the course suggest some students no longer made use of the dual-modality representations as frequently, noting that their dual-modality instruction had helped them learn and that they had reached the point where they acted independently of the scaffolding.

Despite their perceptions of dual-modality instruction – and blocks programming in particular – as an unnecessary dependency, impediment, or otherwise unhelpful instructional method, experienced students in the intervention scored more highly on course examinations than those in the baseline. By comparison, students with little or no prior experience held positive perceptions of the dual-modality instruction and tools – including the dual-modality plugin. These differences further highlight the challenges of managing classrooms with students of mixed experience levels. These findings may justify separate sections for different students based on prior experience as has been discussed in the CS Education community and

implemented at some institutions [41]. In addition, the positive response to lecture materials using blocks representations – even among those students with only text experience – suggests that this approach could be useful even in later, more advanced programming courses when presenting code.

### **6.3.2 Use of Dual-Modality Materials**

As evidence of use of dual-modality instructional materials, I focused my analysis on logs of students' access to the lecture slides on the Canvas LMS and use of the dual-modality Amphibian plugin. In both cases, I analyzed overall trends as well as trends broken up over the instructional modules, usually one-week long, except around exams, when additional time was set aside for reviews and the exams themselves. I numbered the modules 0 (introductory week and tool installation) through 11 (the last module before the final examination).

#### **6.3.2.1 Dual-modality materials results**

On average across all weeks, 58.9% of the students accessed the lecture slides for the modules while they were being covered in class (Median=58.3%;  $\sigma$ =27.3%) (Figure 6-5). The highest percentage of accesses was in the weeks immediately before Exam 1, when 70.1% of students accessed the slides (Module 4), and before and following Exam 2 (Module 5, 70.7%; Module 6, 71.8%).

88.0% of the students in the class (n=374) installed and registered the plugin and used it in at least one session, with a total of 148,931 logged events. Logs also indicated that an additional 48 unique unregistered plugin IDs were in use, but I could not determine if any of these were duplicate logins already accounted for. Logs of plugin events showed that most of the plugin-registered participants used the plugin during each module (Figure 6-5). These logs tracked events within the plugin itself – such as switching between blocks and text modes or dragging and dropping blocks. The plugin logged three major types of events: **Palette Viewing**

(selecting a category of blocks), **Mode Swapping** (switching from blocks to text or vice-versa), and **Block Use** (dragging / dropping blocks). When examining the plugin logs, I found that, aggregated across all weeks and students, the majority of participant actions when using the plugin were Palette Viewing (73.2%, n=108,674) followed by Mode Swapping (23.9%, n=35,503), with Block Use being the smallest category (2.8%, n=4,279). Examining the usage patterns by module dates, I found that the number of Palette Viewing (43.7%, n=2,450) and Mode Swapping (45.9%, n=2,577) events during the first module were comparable, with fewer Block Use events (10.4%, n=582). Over time, the percentage of Mode Swapping and Block Use events decreased while percentage of Palette Viewing events increased. By the final week, most events were Palette Viewing (81.9%, n=19,132) with a smaller number of Mode Swapping events (17.9%, n=4,182), and very few Block Use events (0.1%, n=46) (Figure 6-6). One student with only prior text experience said of the blocks mode, “it’s nice to have a simpler looking format to reference back to if I get stuck” (sound-sloth, Helpful, M2).

There is also evidence in the surveys of students identifying how they used the dual-modality materials in practice. For example, one student who used the plugin in 10 of the 11 modules said the dual-modality instruction was helpful because “it’s nice to have a simpler looking format to reference back to if I get stuck” (sound-sloth: Helpful, M2). Another student, who used the plugin during every module responded on a survey that “It allows me to see two ways of coding the same program, and sometimes blocks are more structured” (sharp-stingray: Helpful, M7).

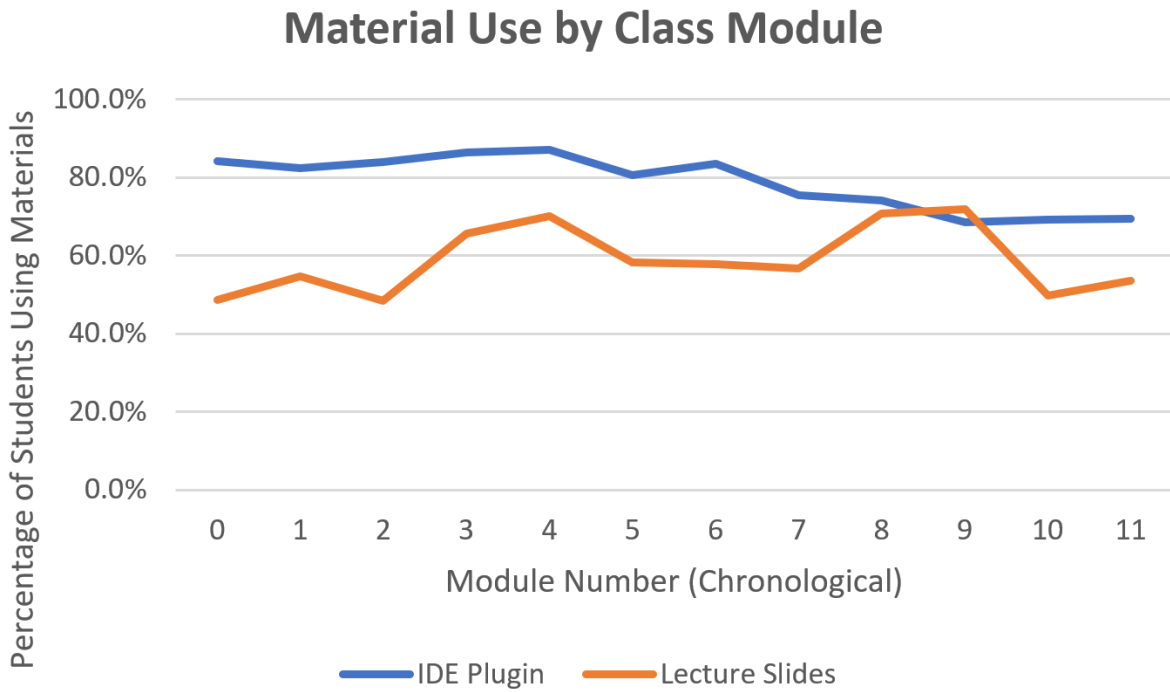


Figure 6-5. Percentage of students accessing lecture slides and using plugin, by module.

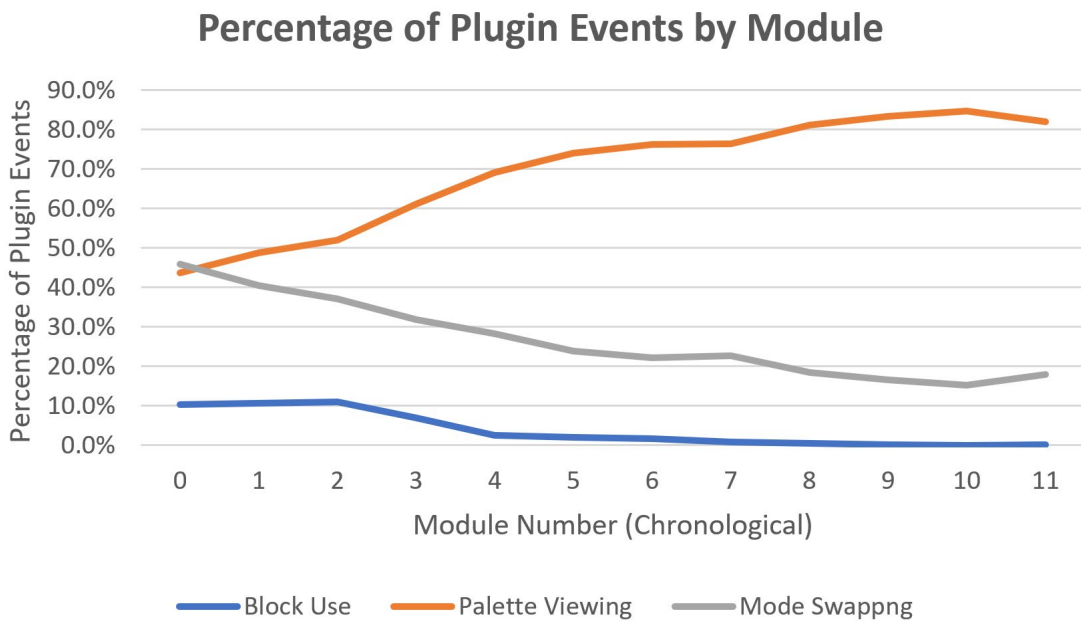


Figure 6-6. Percentage of events of each type by module.

### 6.3.2.2 Dual-modality materials discussion

These results suggest that students were using the lecture materials and plugin throughout the semester. The lecture slides being used each week, but most frequently before examinations. This is what I had expected, as students typically use lecture slides during the lecture, but also return and study from them before exams. I had expected the participants to use the plugin for a variety of tasks, including using blocks mode to write code by selecting, dragging, and dropping blocks. However, the most common event in the logs, Palette Viewing, involves students switching between displays of blocks-based code snippets. Combined with the student free responses noting the use of visualization as a reference for code structure, this suggests students are referring to the palettes as a quick-lookup mechanism for sections of code.

After Palette Viewing, the second most common events were those associated with Mode Swapping. Some students noted in surveys that swapping modes helped them visualize their code and understand the structure of the code. While previous literature studying users of dual-modality tools have noted that students swapped less often as they became accustomed to programming [9], findings from my research extend the community's knowledge to include the types actions students engaged in and how those actions changed over time. For example, students used blocks to program in the first module (10.4% of events) and did so rarely by the last module (0.2%). They engaged in Palette Viewing more often as a percentage of events. This change in behavior suggests that the scaffolding of blocks mode to write and construct programs was not a highly utilized featured and that it was utilized less often over time. Instead, students used the tool to remember how to use certain structures they had already learned in text. I further consider and discuss these plugin events, and how they differ among students based on prior experience, in the next section of this chapter.

### 6.3.3 Instructor Experience

Like many new course innovations, I found that the usage of dual-modality instruction required a significant amount of additional preparation in advance of the course launch, such as the development of tailored slides and assignment instructions, and some adjustments to instructional method – for example, extra time needed to be set aside to instruct students on the setup and use of the dual-modality plugin. However, overall, I was able to incorporate the instructional approach into my typical course structure without a complete rewrite of the lesson plans. A significant part of the preparation was modifying course materials (such as lecture slides, lab manuals, and project descriptions) to include blocks representations of code in addition to text representations. Additionally, course staff (such as graduate Teaching Assistants and undergraduate Peer Mentors<sup>1</sup>) had to be trained in the use of the dual-modality tool use, as well as how to address and navigate student problems in this space. Overall, the instructional approach was effective, and students and instructional staff responded positively to the integration of dual-modality tools and materials.

The Amphibian plugin played a key role in the study. Its addition on top of the existing technology stack (including the Java runtime environment and IntelliJ IDEA) introduced a new level of complexity, necessitating additional preparation for potential troubleshooting challenges. In previous work with middle school students, I had employed a similar instructional approach, but using a variant of the web-based Pencil Code environment. However, in that environment, we were not able to write programs that could exist outside of the Pencil Code turtle graphics sandbox. Based on this previous experience, I developed the Amphibian plugin, and in practice, I

---

<sup>1</sup> In UF's Herbert Wertheim College of Engineering, undergraduates serving as course assistants are referred to as Peer Mentors to distinguish them from Graduate Teaching Assistants.



found that the flexibility it provided to students – compared to a sandbox like Pencil Code – proved crucial to encouraging student exploration of programming in general.

The student experience and use cases were surprisingly different from what I had originally anticipated in designing the study and instruction. In particular, I had expected students to make heavy use of the blocks programming available in Amphibian, but in fact this was rare. Instead, students used it for a variety of supporting functions – such as identifying structure, helping them understand the connections between blocks, help with debugging, and as a quick-reference for common constructs and library functions. This suggested to me that the block structures were important, but rather than helping students to write new code, they were more often used by students to make sure they understood the structure of their own code and to ensure its correctness. Beyond offering a block programming environment, students found value in the visualizations in slides and materials. Expanding on these visualizations, and adding more object-oriented constructs, may further aid student conceptualization and learning of computer science. In addition, despite the voluntary nature of the blocks programming – no student was compelled to use the blocks mode in the class – some students with text experience indicated in surveys that they felt strongly that the availability of blocks-mode programming was detrimental to other less-experienced students, with many suggesting new learners should be forced to work in text and/or that it was a disservice to new programmers. When utilizing dual-modality instruction in the future, I plan to specifically address this perception by explaining the motivation and function of the dual-modality instruction, as well as by outlining the findings of this study – that those students who learned via dual-modality instruction outperformed those who learned via text on course exams.

Based on my experience in the course, in the future I would make adjustments to how I employ dual-modality instruction, and these may be instructive to other teachers considering use of the approach. In particular, developing separate paths for students with prior experience and those with no prior experience is important, as these learners reacted to – and learned from – dual-modality instruction in different ways. While more than a third of students found the plugin helpful across all experience levels, their perceptions varied according to prior experience. The students with the most prior experience largely appreciated the dual-modality presentation materials (slides, manuals, and descriptions), but some did not show the same enthusiasm for the dual-modality plugin (and especially blocks programming support). On the other hand, students with less experience more frequently said they felt that the plugin was helpful for them. This is supported by our results, which generally show students performing better in the intervention compared to the baseline. Future research could help make clear how much the dual-modality plugin, vs class presentation materials, contributed to the performance of students with and with no prior experience. Where practical, a separate instructional environment for inexperienced and experienced students [41] would allow the findings of such an investigation to be put into practice. For example, if it were found that students with prior experience benefit exclusively from the presentation materials, but not the dual-modality plugin, targeting the plugin toward students without or with little experience would allow these students to benefit from the plugin without introducing it to experienced students who may perceive it as a crutch. In this study, the dual-modality representations in presented materials had support from students at many experience levels. As such, it is worth exploring whether dual-modality instruction could be utilized in later coursework (such as a CS2 or data structures course) as well.

## 6.4 Findings & Discussion Summary

In this study, I examined dual-modality instruction and learning in a CS1 course. Overall, I did not find differences in participant scoring on the SCS1, but students performed better on most course examinations in the intervention than the baseline semester. Notably, where topics were not covered in dual-modality instruction, I did not find significant differences on exam scores. The course examinations had more nuanced grading with partial credit and were tuned to the topics of the course, unlike the SCS1, which may explain these findings.

I also examined interactions between instructional condition and students' prior experience. While I had expected those with no prior experience to show the greatest difference between intervention and baseline groups, and those with text-only experience to show the least difference, I found that the opposite was true. When I examined the interactions (where present) between prior experience and condition, they were most pronounced among those with text experience, where the intervention performed better, and the least pronounced differences were among those with no prior experience. Interestingly, this was exclusive to the code writing midterm sections and the Final Exam. Code writing uses recall (unlike code reading which relies more on recognition), so students with more experience may have been building on and strengthening pre-existing mental models. By the same token, there are indications that by the end of the class, most students – but especially those with prior experience – had dispensed with the dual-modality tools; as such, students with prior experience may have “graduated” beyond the need for the tools. Further work in this area could examine whether dual-modality tools representing advanced object-oriented concepts could be developed.

Finally, I looked at the perceptions of students and the instructor experience. Students across all types of prior experience identified Code Visualization as a key factor in helping them learn via dual-modality instruction, and those with prior blocks experience and no experience

also noted help with Code Structure and Code Understanding frequently. Even though students with prior experience exclusively in text showed the greatest difference when comparing the intervention and baseline groups, these students were least likely to say they found the dual-modality instruction helpful – with only a minority indicating as much from the very beginning. By contrast, those with prior blocks experience felt it was helpful for the first few weeks, while those with no experience found dual-modality instruction helpful for as long as dual-modality instruction was used in the classroom. This suggests that it would be valuable for further research to investigate how far the benefits of dual-modality instruction might extend – perhaps into later coursework – and identify additional ways to help students with experience learn while supporting positive perceptions among them of the efficacy of the instruction.

## CHAPTER 7 CONTRIBUTIONS

Here I enumerate the contributions to the field of computer science education that come from my dissertation work – specifically, (1) foundational work: initial studies with elementary, middle, and CS1 students on perceptions of programming, prior experience, and dual-modality code representations; (2) a technical contribution: building the Pencil Code Python variant and the Amphibian dual-modality IDE plugin for Java; (3) an empirical contribution: identification of the connection between dual-modality instruction and learning in a CS1 course; and (4) an instructional contribution: analysis of perceptions of students and instructor experience for the dual-modality instructional approach.

### **7.1 Foundational Studies (Perceptions of Programming & Dual-Modality Representations)**

My early work with K-12 students focused on how student perceptions and prior experience mold their views of programming moving forward. In working with elementary school students, I discovered that those students with prior experience held more nuanced views of programming – focusing not just on artifacts that can be created, but the role of communication and how it can be used to help others. I followed up on this work with middle school students, with whom I investigated how experience in different representations related to student perceptions of text programming. The students in this study who worked in the dual-modality programming environment held positive perceptions of text more often and negative perceptions of text programming less often than those students who moved directly from blocks to text, demonstrating the potential of dual-modality programming environments to alleviate negative feelings about text programming.

## **7.2 Technical: Python Pencil Code Variant & Amphibian Dual-Modality Java Plugin**

Java and Python are common introductory languages in K-12 and college levels. Before I began my work, dual-modality programming environments did not support Python or Java languages, nor did they support IDE-based development or development of generic programs. By integrating Python into the Pencil Code environment will support instructors who wish to introduce the Python language using a dual-modality instructional approach in K-12 schools. Further, the Java plugin I developed will enable instructors to more easily incorporate dual-modality instruction into such CS1 courses and also enable more rigorous research into these approaches by allowing researchers to reduce the impact of other variables (such as different languages, software systems, and development environments). Historically, blocks-based environments have been geared toward children and have largely been limited to sandboxed environments; the Amphibian dual-modality plugin I built makes blocks-based programming accessible broadly for any type of development. In addition, those dual-modality programming environments that existed previously are currently limited to the imperative language paradigm, even when working within languages that support object-oriented programming. The addition of a fundamentally object-oriented language to the Droplet Editor has necessitated design of object-oriented blocks-based constructs which will enable other object-oriented languages to be added more easily in the future. The plugin, along with its source code, is available on a public source repository on GitHub: <https://github.com/cacticouncil/amphibian>.

## **7.3 Empirical: Learning and Dual-Modality Approaches to CS Instruction**

Based on my analysis of the CS1 study's results, this research identified:

1. relationship between dual-modality instruction and student learning; differences correlated to prior programming experience by type; and performance differences by instructional condition on assessments (e.g., writing vs reading).

My work showed that the students in the intervention group (dual-modality instruction) outperformed those in the baseline group (text instruction) on course exams, but not on the SCS1. By using the demographic survey to account for prior experience, I found that there were differences in how students scored based on prior experience (none, blocks, or text-only) – students with prior experience outperformed those with no prior experience on the code-writing sections of the exams and the final exam. These results from the SCS1 and from the course examinations contribute to the literature by helping researchers and educators understand how dual-modality instruction connects to learning in computer science.

#### **7.4 Instructional: Perceptions in Dual-Modality Programming Environment**

Using data I collected in the CS1 study (as outlined in section 5.5), I analyzed student perceptions and summarized my experience in employing dual-modality instruction in a classroom environment. I also examined problems encountered, solutions employed, overall results, and made recommendations. I found that, even at the end of the class, more than one-third of the students still found dual-modality instruction helpful, and this was true for all prior experience groups (no experience, blocks, and text-only), suggesting that students can benefit from dual-modality instruction even at the end of a CS1 course. This analysis provides guidance to computer science educators for using dual-modality programming environments in their classrooms while providing researchers with a case study to consider in later research endeavors.

## CHAPTER 8 CONCLUSIONS

This dissertation has answered open questions within the literature of computer programming learning environments and particularly those using dual-modality representations. In this section I summarize the problem, solution, my work, and my contributions to the field.

### **8.1 Problem**

Students of programming in computer science must master several skills, among them syntax, semantics, chunking, abstraction, computational thinking, and troubleshooting [125]. Blocks-based environments showed promise in helping students develop skills [89, 31]. However, the literature suggested students may struggle when moving to text-based environments [134]. In addition, even once students have mastered syntax, they must still develop general expertise in programming – and the ability to translate their ideas into running code – while moving from the sensorimotor to preoperational to operational stages of reasoning in the Neo-Piagetian framework [71].

### **8.2 Proposed Solution**

Dual-modality block-text systems, offering both text and blocks-based representations, were developed to provide a bridge for students between learning environments and production languages [7]. Specifically, these environments offered promise in being able to allay the difficulties students face when working in text-based representations by adopting some of the scaffolding and affordances of blocks-based representations [14]. In addition, by linking textual and blocks-based modes of the same language, dual-modality blocks-text systems may facilitate chunking and abstraction by visually nesting code blocks (such as those of function or conditional constructs) [71]. My work evaluated the use of dual-modality instruction to facilitate



learning in early programming coursework and differences in performance and student perceptions that arose due to prior programming experience [16].

### **8.3 Early Work**

My early work included (1) an examination of perceptions in blocks-based and text-based programming with children, and development of a custom dual-modality programming environment variant, and (2) creation of a dual-modality curriculum and a custom dual-modality representation programming assessment for middle school students, and an analytical comparison of perceptions of blocks, text, and dual-modality representations from a study with middle school students. I summarize this work in this section.

#### **8.3.1 Perceptions of Programming Investigations**

I conducted a study of programming and specific construct perceptions with children in a summer game in 2015. I posited that, while blocks-based tools can help facilitate the learning of computer science concepts at younger ages, students encounter challenges translating their experiences into production languages; I suggested development of a bridge between blocks and text. This initial study's purpose was to identify how prior programming experience connected to overall perception of the act of programming and specific language constructs. This study's results, which showed that the children with and without prior programming experience had distinct patterns in their perceptions of programming, provided guidance for later work which focused on qualitative coding and analysis of perceptions of blocks-vs-text paired with quantitative score analysis.

#### **8.3.2 Initial Evaluation of Perceptions & Learning**

I conducted a study at a middle school in Central Florida to collect data on the use of dual-modality instruction and learning and perceptions of programming. The purpose of this study was to identify how use of bi-directional dual-modality programming environments

connects to student learning and perceptions of programming and computer science. The focus of the initial analysis was perceptions of programming specifically and learning generally.

To facilitate this work, I integrated a Python runtime environment into Pencil Code and worked with a team of undergraduate students to create a Python API for Pencil Code. This work allowed instruction using Pencil Code in the Python language. To measure learning, I developed a custom dual-modality assessment with representations in blocks and text. I created an assessment in dual text-blocks representations with three isomorphic variants of each question so that the same concept could be tested at three points in time to measure change in performance over time. I found that students who started in blocks and then worked in hybrid environments before moving into text held more positive views of text programming compared to students who moved directly from blocks to text programming. This prompted me to further investigate how dual-modality programming environments and tools could be used in college level coursework and how it might change the classroom, affect perceptions, and connect to learning.

## **8.4 Final Study**

The final work for my dissertation study is summarized here. This includes development of the Amphibian dual-modality Java language IDE plugin for IntelliJ IDEA, development of dual-modality classroom materials, analysis of responses on the custom and SCS1 assessments, and data collection during the dual-modality instructional intervention.

### **8.4.1 Amphibian Dual-Modality Java Language IDE Plugin for IntelliJ IDEA**

I developed a dual-modality plugin for the IntelliJ IDE in order to lay the groundwork for my proposed work. At the time of the plugin's development, there were no dual-modality tools for standalone IDE-based development, or development of general-purpose programs. A group of students and I developed a dual-modality IDE plugin from Pencil Code's Droplet Editor to enable switching between blocks and text within a production environment. To facilitate

practical study of CS1 student performance in a “real-world” environment, I developed a Java variant of the dual-modality IDE plugin, which I dubbed “Amphibian”. This tool will enable teachers of Java courses – including those of AP CS and many introductory college courses – to incorporate a blocks/text transition into the curriculum. The plugin was used in my study of dual-modality instruction and learning in a CS1 classroom.

#### **8.4.2 Dual-Modality Instruction & Curriculum**

I developed new materials for the CS1 (COP3502) course materials utilizing dual-modality representations in order to facilitate student use of and learning via the dual-modality programming environment in the study. These materials, along with student responses to surveys and instructor notes, were used to analyze student perceptions and the instructor experience during the intervention. This analysis can be used by future instructors of early programming courses to identify potential strategies for introduction of dual-modality programming environments into classroom instruction.

#### **8.4.3 Instrument Evaluation**

In Fall of 2017 I collected SCS1 assessment results from participants in the CS1 course at the University of Florida (COP3502) at the end of the term. Students were offered extra credit to participate in a concept inventory test just before the final examination. Both the custom assessment described earlier and the SCS1 were used during this collection phase. I used data collected during this phase to determine that the SCS1 was an appropriate instrument and to decide on the structure of the intervention in my final dissertation study.

#### **8.4.4 Study of Dual-Modality Instruction and CS Learning**

I conducted a study at the college level in a multi-section COP3502 (UF’s CS1) course in Spring of 2018 and Fall of 2018 to examine the dual-modality instruction and learning in a CS1 course. The participants from Spring 2018 learned via a traditional, text-based instructional

approach (the baseline group), while the participants from Fall of 2018 learned via dual-modality instruction (the intervention group). The intervention group used the Amphibian dual-modality IDE plugin and materials. Participants also answered survey questions throughout the term about their perceptions of dual-modality representation IDE plugin and materials. Both groups (baseline and intervention) completed the SCS1 at the end of the semester for extra credit.

#### **8.4.5 Analysis of Learning and Dual-Modality Instruction**

I analyzed the data from the intervention and baseline groups to identify differences in programming knowledge, particularly as it related to stages of cognitive development. As primary measures, I used student scores on the SCS1 and course examinations (two midterms and one final). Surveys, logs, and notes from the intervention group were used to contextualize results and provide supporting evidence for findings. Course examination sections (code reading and code writing) were analyzed separately. I also examined the role prior programming experience played in student scores. I anticipated that students in the intervention group would score more highly on tracing and code-completion questions on the SCS1, and on code reading and code writing questions on the course examinations, while students in both groups would score about the same on definitional questions. While I did not see differences in the SCS1 questions, even by type, there were significant differences in the course examinations, with students in the intervention scoring higher than those in the baseline. Digging deeper, I found that significant results were more pronounced – and had a bigger effect size – on examinations for which all topics were covered in dual-modality instruction, compared to those that were covered partly in text or only in text. I also anticipated that the largest differences between the baseline and intervention groups would be among students with no experience, followed by experience in blocks – and that students with prior text experience would differ the least. However, my results showed the opposite: students with text-only prior experience performed

better on all three course exams in the intervention group compared to the baseline group, and with a greater positive difference than the difference among those with blocks or no prior experience.

#### **8.4.6 Examination of Student Perceptions and Instructor Experience**

I also analyzed student perceptions of dual-modality instruction and reported on my experience as an instructor employing dual-modality instruction based on surveys, usage logs, and instructor notes. The analysis includes student perceptions of dual-modality instruction and how it changed over time. I had anticipated that students would find the dual-modality instruction more helpful at the beginning of the course, but that over time they would find it less useful, and that is indeed what I found when examining student survey responses. I also found that more than half of students whose prior experience was exclusively text felt the dual-modality instruction was not helpful throughout the course – especially blocks-based programming – which ran counter to the course examination results. By comparison, among those who had previous experience in blocks, or no experience, more than half said dual-modality instruction was helpful at the beginning of the course, but fewer found it helpful by the end of the course, with some students explicitly mentioning that had outgrown the support provided by the dual-modality instruction. Finally, I detailed my experience in the classroom, identifying challenges, successes, and suggestions for other instructors who are considering the employment of dual-modality instruction in early programming coursework. This analysis will also help researchers to explore when and how transitioning from dual-modality instruction to pure-text instruction is appropriate in the classroom setting.

## 8.5 Contributions

My contributions include three main elements:

- a. a technical contribution –dual-modality IDE plugin for the Java language;
- b. an instructional contribution – analysis of perceptions and experience, and materials; and
- c. an empirical contribution – analysis of dual-modality instruction and learning.

The plugin and perception analysis provide tools and guidance to educators and researchers in the classroom, while the empirical work provides insight into how and where dual-modality programming environments can have the most positive connection to learning.

## 8.6 Future Work

My work suggests several avenues for future consideration and exploration in research. I have examined dual-mode instruction as a whole, but I found that students with different experience levels reacted differently to dual-modality presentation materials (e.g., lecture slides and lab manuals) than to the dual-modality plugin that allowed programming in different modes. The benefits of dual-modality presentation materials may extend beyond early programming courses, and identifying whether such approaches help students in later courses (such as CS2 and Data Structures courses) would help instructors tailor CS course materials. In addition, the different reactions from students with and without prior experience suggest that employing the dual-modality plugin in earlier curricula – such as a “CS0” or AP CS Principles course – may help students learn even before reaching CS1 courses by helping them to link blocks and text representations. I also noted that my dual-modality instruction ended at basic object-oriented structures, and notably did not include such concepts as inheritance, interfaces, and abstract classes; integration of representations of such concepts into dual-modality tools – such as incorporating the visual inheritance modeling in BlueJ and GreenFoot [49, 51] – would allow for further investigation of the effectiveness of dual-modality instruction and tools in helping

students learn more complex computer science concepts. Finally, the benefit to even experienced programmers from the dual-modality representations in presentation materials suggests that there may be value even to experts in such a blocking mechanism. The students in the course used the blocks-mode of the plugin to check their understanding of code structure and ensure its correctness; this could be explored within professional IDEs – for example, by graphically delineating boundaries of constructs within text modes.

The data collected in these studies also provides fertile ground for future work. Though outside of the scope of this dissertation, artifacts collected during the study with middle school students could be analyzed to identify how and when students used different types of constructs, and in-depth item analysis of student performance on the custom assessment could provide insight into how that custom assessment, and potentially other concept inventory assessments, could be improved. The plugin logs from my final CS1 study could be further analyzed to identify sessions in blocks and text modes, which could then be examined over time to see if the usage patterns my expectations that students would use the blocks mode less as they gained experience in programming. Chat logs collected from instructor discussions could also yield further insight into the classroom environment and perspectives not just of the main instructor (myself) but also those of graduate Teaching Assistants and undergraduate Peer Mentors. Additionally, explicitly analyzing student perceptions of helpfulness and plugin usage patterns for interactions could provide further evidence of and directly link the plugin's connection to student perceptions and learning.

There are important limitations of my work that could be a source for exploration in future work as well. In the final CS1 study, it is possible that a selection bias played a role in student differences, as the data were collected in different semesters, and did not use a random-

control model. As I collected demographic responses from participants including educational history, I or others can use propensity score analysis [69] to address this limitation and provide supporting evidence for a causal relationship between the dual-modality instruction and improved performance and learning. I could also investigate whether there are meaningful trends in performance, tool usage, or perceptions along ethno-racial, age, or educational backgrounds. Investigation of these data could further help us craft instruction and curricula that best serve a diverse population in our community.

### **8.7 Summary**

My doctoral work was inspired by my personal experiences in teaching and focused on early programming instruction, especially blocks-based environments. I began by examining how student perceptions are shaped by prior experience and programming constructs, and I suggested in my early work that a bridge between blocks and text could facilitate novice programmers moving into production languages. Later, my work focused on the connection of dual-modality programming environments to perceptions and learning, especially Pencil Code. In order to study these dual-modality programming environments, I participated in and led the development of several instructional tools, including a Python variant of Pencil Code and a custom dual-modality representation CS assessment. As evidence of the potential role of dual-modality programming environments became more evident in the literature and my experience, my work shifted to focus on dual-modality instruction in CS1 classrooms.

To investigate the role dual-modality programming environments could play in early programming courses, I developed a plugin architecture and Java language IDE plugin, as well as dual-modality instructional materials for use in a CS1 classroom. These tools allowed me to conduct a study of dual-modality instruction in a CS1 classroom. I have analyzed that data and reported in this dissertation on how dual-modality instruction connects with student learning, and



how it varies according to prior programming experience. I have also analyzed student perceptions and analyzed them within the context of the classroom experience in order to provide a template for future instructors who wish to employ dual-modality instruction in college classrooms. My work contributes an understanding, grounded in pedagogical theory, of how dual-modality representations connect with learning and provides tools and guidance to educators and researchers in the classroom environment.

APPENDIX A  
CONFERENCES, PUBLICATIONS, & DEVELOPMENT

**Published / Completed**

Blanchard, J., Gardner-McCune, C., and Anthony, L. 2020. Dual-Modality Instruction and Learning: A Case Study in CS1. Accepted to *the 51st ACM Technical Symposium on Computer Science Education* (2020), 818-824. (Research Track, Best Paper, 2<sup>nd</sup> Place)

Blanchard, J., Gardner-McCune, C., and Anthony, L. 2019. Effects of Code Representation on Student Perceptions and Attitudes Toward Programming. *Proceedings of the IEEE Symposium on Visual Languages & Human-Centric Computing* (2019), 127–131. (Best Paper, Runner Up)

Blanchard, J., Gardner-McCune, C., and Anthony, L. 2019. Amphibian: Dual-Modality Representation in Integrated Development Environments. *2019 IEEE Blocks and Beyond Workshop* (Blocks and Beyond), 83-85.

Blanchard, J., Gardner-McCune, C. and Anthony, L. 2018. How Perceptions of Programming Differ in Children with and without Prior Experience. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (2018), 1099.

Blanchard, J. 2017. Hybrid Environments: A Bridge from Blocks to Text. *Proceedings of the 2017 ACM Conference on International Computing Education Research*. (2017), 295–296.

Blanchard, J., Gardner-McCune, C., and Anthony, L. 2015. Bridging Educational Programming and Production Languages. “*Every Child a Coder*” workshop, Position paper, *ACM SIGCHI Conference on Interaction Design and Children* (2015).

Pencil Code, Python Variant (2017). <https://github.com/cacticouncil/pencilcode>.

Droplet Variant with Java Support (2018). <https://github.com/cacticouncil/droplet>.

Droplet IntelliJ Plugin with Java support (2018). <https://github.com/cacticouncil/amphibian>.

**In Progress**

Bridging Educational Environments to Production Languages: A Survey

Effects of Dual-Modality Instruction on the Classroom Experience

APPENDIX B  
TIMELINE FOR DOCTORAL WORK

Table B-1. Doctoral Work Timeline (Chronological)

Period	Task / Activity
Spring 2015	Paper: “Every Child a Coder” workshop – position paper ( <i>accepted</i> ) [57]
Summer 2015	Data Collection: Construct Perceptions in Children
Fa. ‘15–Sp. ‘16	Data Analysis & Writing: Construct Perceptions in Children
Summer 2016	Development: Pencil Code’s Python Variant
Fall 2016	Development: Pencil Code’s Python Variant Qualifying Examination
Spring 2017	Development: Pencil Code’s Python Variant Development: Custom Blocks/Text Assessment Study Design: Middle School Study Paper Submission: ICER Doctoral Consortium ( <i>accepted</i> ) [13]
Summer 2017	Data Collection: Middle School Study Data Analysis: Middle School Study
Fall 2017	Data Analysis: Middle School Study Paper: SIGCSE – Middle School Study ( <i>reworked for VL/HCC</i> ) Poster: SIGCSE – Construct Perceptions in Children ( <i>accepted</i> ) [15] Development: Dual-Modality IDE Plugin Framework Data Collection: SCS1 & Custom Assessment in CS1 ( <b>instrument eval.</b> ) Study Design: CS1 & Dual-Modality Programming Environments
Spring 2018	Item Analysis: Custom Assessment and SCS1 Study Design: CS1 & Dual-Modality Programming Environments Data Collection: SCS1 in CS1 course ( <b>baseline</b> ) Development: Java Dual-Modality IDE Plugin
Summer 2018	Study Design: CS1 & Dual-Modality Programming Environments Development: Dual-Modality Representation Materials Development: Java Dual-Modality IDE Plugin
Fall 2018	Study Intervention: Dual-Modality IDE Plugin & Course Materials Data Collection: SCS1 in CS1 course ( <b>intervention</b> ) Data Collection: Surveys, Plugin Logs, Course Grades
Spring 2019	Dissertation Proposal Writing
Summer 2019	Proposal Defense Paper: VL/HCC – Middle School Study ( <i>accepted, 2<sup>nd</sup> Best Paper</i> ) [14] Data Analysis: CS1 & Dual-Modality Programming Environments
Fall 2019	Paper: SIGCSE – CS1 Amphibian Study ( <i>accepted, 2<sup>nd</sup> Best Paper</i> ) [16] Paper: Blocks & Beyond Workshop – Amphibian Plugin ( <i>accepted</i> ) [17]
Fa. 19–Su.’20	Complete and Defend Dissertation

APPENDIX C  
MIDDLE SCHOOL STUDY: DEMOGRAPHIC QUESTIONNAIRE

1. We want to learn if some tools help students learn to program better than others. If you decide to participate, you will be asked to fill out questionnaires on three days over the next five weeks while working with different programming tools. These questions will not be used to grade you, and there are no known risks to participation. You do not have to be in this study if you don't want to and you can quit the study at any time. Other than the researchers, no one will know your answers, including your teachers or your classmates. If you don't like a question, you don't have to answer it and, if you ask, your answers will not be used in the study. I also want you to know that whatever you decide, this will not affect your grades in class.

Would you be willing to participate in this study? [Yes] [No]

(If a participant answers "No" to this question, they will be redirected to a "thank you" page with no further questions.)

2. What is your gender identity? [Male] [Female] [Prefer to Self-Describe: ] [Decline to answer]
3. What is your age? [9] [10] [11] [12] [13] [14] [15] [16] [17] [18]  [Decline to answer]
4. Please specify your ethnicity (check all that apply): [American Indian or Alaska Native] [Asian] [Black or African American] [Hispanic or Latino] [Native Hawaiian or other Pacific Islander] [White] [Other: ] [Decline to answer]
5. Which environments/tools have you used before? Check all that apply. [Scratch] [Alice] [Pencil Code] [Hour of Code] [Others: ]
6. Which languages have you used before? Check all that apply. [Logo] [Python] [JavaScript] [HTML] [Others: ] [Blank]

APPENDIX D  
MIDDLE SCHOOL STUDY: PERCEPTION QUESTIONNAIRES

Questions used a Likert scale [Strongly Agree] [Agree] [Neither] [Disagree] [Strongly Disagree]

**Personal Perceptions (Pre, Mid, & Post)**

1. Computers are fun.
- Computer jobs are boring.
- Programming is hard.
- I want to find out more about programming.
- I can become good at programming.
- I prefer to solve my own computer problems.
- I like the challenge of computer problems.
- My family, friends, and/or classmates ask me for help with computer problems.

**Mid-Survey Only, By Condition**

**Text Condition**

1. I think programming in text is easy.
2. I think programming in text is frustrating or hard.
3. I think learning to program in blocks is more useful than text.
4. I think learning to program in text is more useful than blocks.
5. I would have preferred to program using blocks as opposed to text.

**Blocks Condition**

1. I think programming in blocks is easy.
- I think programming in blocks is frustrating or hard.
- I think learning to program in blocks is more useful than text.
- I think learning to program in text is more useful than blocks.
- I would have preferred to program using text as opposed to blocks.

**Hybrid Condition**

1. I think programming in blocks is easier than programming in text.
2. I think programming in text is easier than programming in blocks.
3. I think programming in blocks is frustrating or hard.
4. I think programming in text is frustrating or hard.
5. I think learning to program in blocks is more useful than text.
6. I think learning to program in text is more useful than blocks.
7. I would prefer to program using text as opposed to blocks.
8. I would prefer to program using blocks as opposed to text.

### **Post-Survey Only, All Conditions**

1. I think programming in blocks is easier than programming in text.
2. I think programming in text is easier than programming in blocks.
3. I think programming in blocks is frustrating or hard.
4. I think programming in text is frustrating or hard.
5. I think learning to program in blocks is more useful than text.
6. I think learning to program in text is more useful than blocks.
7. I would prefer to program using text as opposed to blocks.
8. I would prefer to program using blocks as opposed to text.

APPENDIX E  
CS1 STUDY: DEMOGRAPHIC QUESTIONNAIRE

1. We want to identify the strengths and weaknesses of computer science assessments. If you decide to participate, you will be asked to fill out a demographic questionnaire. These questions will not be used to grade you. You will also be asked to take an assessment on computer science concepts, for which you will receive extra credit. The assessment will take 60-120 minutes. You will receive 25 points of extra credit to be applied to your exam grades (out of 1000 points in the course.) You may also elect to do an alternative assignment, a course reflection essay, to earn the extra credit. Your assessment results will be connected to your class performance for research purposes only. No one other than the researchers and your teachers will know your answers or grades. There are no known risks to participation. You do not have to be in this study if you don't want to and you can quit the study at any time. If you don't like a question, you don't have to answer it and, if you ask, your answers will not be used in the study.

Would you be willing to participate in this study? [Yes] [No]

(If a participant answers "No" to this question, they will be redirected to a "thank you" page with no further questions.)

2. What is your name? (This will be used to connect your participation to the course)  
[Name]
3. What is your UFL.EDU email address? [Email]
4. May we contact you in the future about possible participation in follow up studies? [Yes] [No]
5. What is your gender identity? [Male] [Female] [Prefer to Self-Describe: \_\_\_\_\_]  
[Decline to answer]
6. What is your age? [\_\_\_\_\_] [Decline to answer]
7. Please specify your ethnicity (check all that apply): [American Indian or Alaska Native] [Asian] [Black or African American] [Hispanic or Latino] [Native Hawaiian or other Pacific Islander] [White] [Other(s): \_\_\_\_\_]  
[Decline to answer]
8. How many years of programming do you have...  
In College: [Number selector]  
In K-12 Schools: [Number selector]  
Self-taught / practice: [Number selector]
9. Have you taken any of the following courses in high school, and if so, what was your score on the AP exam?

[Choices: Did not take course; Took course but not exam; 1; 2; 3; 4; 5]

- AP Computer Science
- AP Computer Science Principles
- AP Calculus AB
- AP Calculus BC
- AP Physics

10. Have you taken any of the following courses (other than this one) at the college level?

- Calculus I
- Calculus 2
- Computer Science 0 / Computational thinking course
- Computer Science 1 / Programming class in computer science
- Physics 1

11. Which programming environments/tools have you used before? Check all that apply.

- Alice
- Code.org
- Scratch
- Pencil Code
- Python
- Java
- C#
- C++
- C
- Logo
- Other(s): [\_\_\_\_\_]



APPENDIX F  
CS1 STUDY: PERCEPTION QUESTIONNAIRES

Questions used a Likert scale [Strongly Agree] [Agree] [Neither] [Disagree] [Strongly Disagree]

**Personal Perceptions (Pre-Survey Only)**

1. Computers are fun.
2. Computer jobs are boring.
3. Programming is hard.
4. I want to find out more about programming.
5. I can become good at programming.
6. I prefer to solve my own computer problems.
7. I like the challenge of computer problems.
8. My family, friends, and/or classmates ask me for help with computer problems.

**Blocks/Text Perceptions (Pre, Mid, Post)**

1. I think programming in blocks is easier than programming in text.
2. I think programming in text is easier than programming in blocks.
3. I think programming in blocks is frustrating or hard.
4. I think programming in text is frustrating or hard.
5. I think learning to program in blocks is more useful than text.
6. I think learning to program in text is more useful than blocks.
7. I would prefer to program using text as opposed to blocks.
8. I would prefer to program using blocks as opposed to text.

**Hybrid Instruction Perceptions (Mid, Post)**

1. What benefits do you think hybrid blocks-text instruction provides, and why? [Free response]
2. What concepts or constructs do you think hybrid blocks-text instruction helps students learn or understand, and why? [Free response]
3. How frequently in a week do you refer to the lecture slides to study, prepare, and/or do assignments? [Never] [Rarely] [Sometimes] [Frequently] [Always]

### Weekly Survey

1. Did you program in “Blocks” mode since the end of your previous lab (including this lab)? [Yes/No]
2. Did you program in “Text” mode since the end of your previous lab (including this lab)? [Yes/No]
3. What was your primary mode since the end of your previous lab (including this lab)? [Blocks/Text]
4. Does instruction in dual blocks-text modes help you learn better? [Yes/No]
5. Why do you feel this way? [Free response]

APPENDIX G  
CUSTOM ASSESSMENT

Q1

Consider the following code snippet.

```
count = 0
for i in <MISSING_CODE>:
    count = count + 1

print("The count is", count)
```

Which code block, replacing <MISSING\_CODE>, would cause the program to print "The count is 7" to the screen?

range(1, 7)	Boundary
range(0, 7)	CORRECT
range(1, 6)	Fundamental misunderstanding of loop function
range(0, 6)	Boundary

Consider the following code snippet.

```
count = 0
for i in __MISSING_CODE__:
    count = count + 1
print ("The count is", count)
```

Which code block, replacing \_\_MISSING\_CODE\_\_, would cause the program to print "The count is 7" to the screen?

range(1, 7)	Boundary
range(0, 7)	CORRECT
range(1, 6)	Fundamental misunderstanding of loop function
range(0, 6)	Boundary

Q2

Consider the following function.

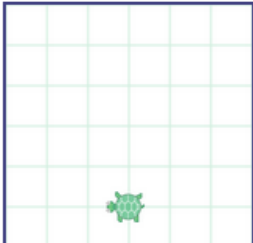


```
def draw_circle( color, radius ):
    pen(▼ color, ▼ 2)
    ra(▼ 360, ▼ radius)
```

`draw_circle('red', 25)`

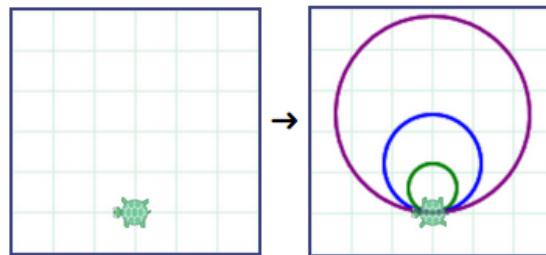
Which block of code would draw this picture of three circles from the starting point below?

<pre>draw_circle('purple', 15) draw_circle('blue', 30) draw_circle('green', 60)</pre>	Assumes last is biggest
<pre>draw_circle('purple', 30) draw_circle('blue', 15) draw_circle('green', 60)</pre>	Assumes last is biggest
<pre>draw_circle('purple', 15) draw_circle('blue', 60) draw_circle('green', 30)</pre>	Easy distractor
<pre>draw_circle('purple', 60) draw_circle('blue', 30) draw_circle('green', 15)</pre>	CORRECT

Consider the following function.

<pre>def draw_circle(color, radius):     pen(color, 2)     ra(360, radius)</pre>	<code>draw_circle('red', 25)</code>	
		

Which block of code would draw this picture of three circles from the starting point below?



<pre>draw_circle('purple', 15) draw_circle('blue', 30) draw_circle('green', 60)</pre>	Assumes last is biggest
<pre>draw_circle('purple', 30) draw_circle('blue', 15) draw_circle('green', 60)</pre>	Assumes last is biggest
<pre>draw_circle('purple', 15) draw_circle('blue', 60) draw_circle('green', 30)</pre>	Easy distractor
<pre>draw_circle('purple', 60) draw_circle('blue', 30) draw_circle('green', 15)</pre>	<b>CORRECT</b>

Q3

Consider the following code segment:

```
if doorIsLocked:  
    print ("I'll wait outside.")  
else:  
    print ("Let's relax on the couch.")  
print ("Thank you for having me over!")
```

If the variable `doorIsLocked` has the value `True`, what is displayed as a result of running the code segment?

I'll wait outside. Thank you for having me over!	CORRECT
I'll wait outside.	Tracing issue
Let's relax on the couch. Thank you for having me over!	Misunderstanding of bool values
Let's relax on the couch.	Misunderstanding of bool values & if/else behavior

Consider the following code segment:

```
if doorIsLocked:  
    print("I'll wait outside.")  
else:  
    print("Let's relax on the couch.")  
  
print("Thank you for having me over!")
```

If the variable `doorIsLocked` has the value `True`, what is displayed as a result of running the code segment?

I'll wait outside. Thank you for having me over!	CORRECT
I'll wait outside.	Tracing issue
Let's relax on the couch. Thank you for having me over!	Misunderstanding of bool values
Let's relax on the couch.	Misunderstanding of bool values & if/else behavior

Q4

Consider the following code segment:



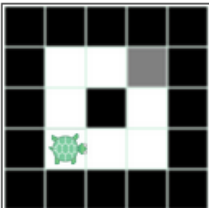

```
while not goal_reached():  
    move_forward(1)  
    move_forward(1)  
    turn_left(1)
```

`move_forward(numMoves)` moves turtle forward by specified number of blocks

`turn_left(numTurns)` turns turtle by specified number of full turns (90°) to the right

`goal_reached()` is True when the turtle is at the goal

For which terrain grid would the code correctly navigate the turtle to the grey square without entering any black squares?

	Call ordering
	Call ordering & Terminal check timing
	CORRECT
	Terminal check timing

Consider the following code segment:

```
while not goal_reached():  
    move_forward(1)  
    move_forward(1)  
    turn_left(1)
```

- `move_forward(numMoves)` moves turtle forward by specified number of blocks
- `turn_left(numTurns)` turns turtle by specified number of full turns (90°) to the right
- `goal_reached()` is `True` when the turtle is at the goal

For which terrain grid would the code correctly navigate the turtle to the grey square without entering any black squares?

	Call ordering
	Call ordering & Terminal check timing
	CORRECT
	Terminal check timing



Q5

Consider the following code snippet.

```
sum = 0
for i in __MISSING_CODE__:
    sum = sum + i
print ("The sum is",sum)
```

Which code block, replacing `__MISSING_CODE__`, would cause the program to print "The sum is 10" to the screen?

<code>range(0, 5)</code>	CORRECT
<code>range(3, 7)</code>	Adding boundaries to get result
<code>range(0, 10)</code>	Loop index value / incorrect state tracking
<code>range(1, 6)</code>	Boundary

Consider the following code snippet.

```
sum = 0
for i in <MISSING_CODE>:
    sum = sum + i

print("The sum is", sum)
```

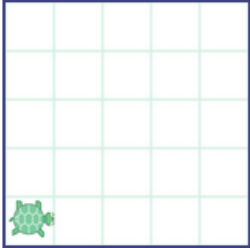
Which code block, replacing `<MISSING_CODE>`, would cause the program to print "The sum is 10" to the screen?

<code>range(0, 5)</code>	CORRECT
<code>range(3, 7)</code>	Adding boundaries to get result
<code>range(0, 10)</code>	Loop index value / incorrect state tracking
<code>range(1, 6)</code>	Boundary

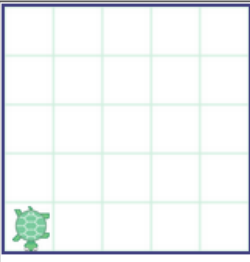
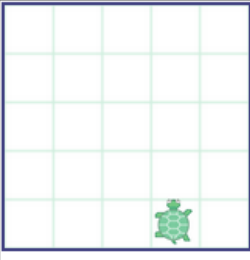
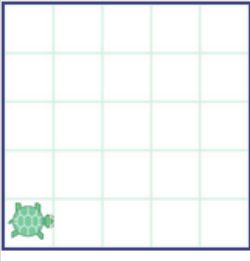

Q6

Consider the following code segment and position. See the **Code Guide** for the two helper functions `move_forward(numMoves)` and `turn_left(numTurns)`.

```
for n in range(1, 2):  
    for k in range(0, 3):  
        move_forward(1)  
    turn_left(1)
```

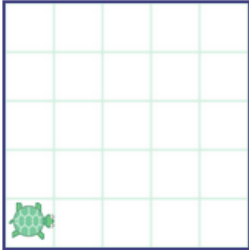


Which of the following shows the location of the turtle after running the code segment?

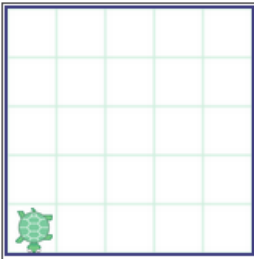
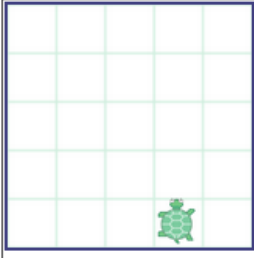
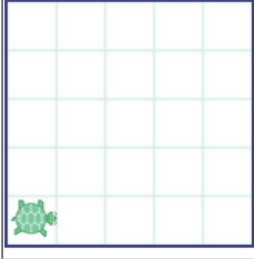
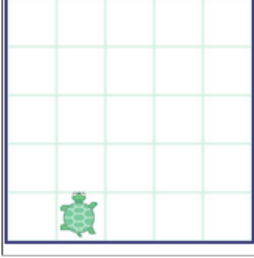
	Treat lt() as if in inner loop
	CORRECT
	add ranges, sequential: (1,5)
	Treat move_fd() as if in outer loop

Consider the following code segment and position. See the **Code Guide** for the two helper functions `move_forward(numMoves)` and `turn_left(numTurns)`.

```
for n in range(1, 2):  
    for k in range(0, 3):  
        move_forward(1)  
        turn_left(1)
```



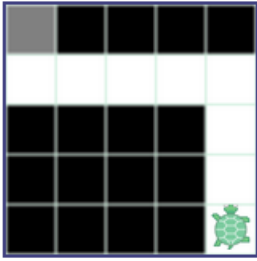
Which of the following shows the location of the turtle after running the code segment?

	Treat <code>lt()</code> as if in inner loop
	CORRECT
	add ranges, sequential: (1,5)
	Treat <code>move_fd()</code> as if in outer loop

Q7

The figure below shows a turtle in a grid of squares. The turtle can move into a white or gray square but cannot move into a black region. Consider the grid and function MoveAndTurn below.

```
def moveAndTurn( numMoves, numTurns ):
    move_forward( numMoves )
    turn_right( numTurns )
```

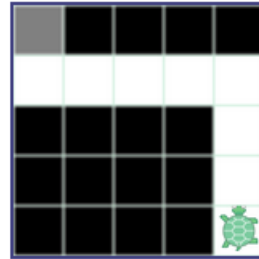


Which of the following code segments will move the turtle to the gray square?

<pre>moveAndTurn( 3, 3 ) moveAndTurn( 1, 4 ) moveAndTurn( 0, 1 )</pre>	Param order
<pre>moveAndTurn( 3, 3 ) moveAndTurn( 4, 1 ) moveAndTurn( 1, 0 )</pre>	CORRECT
<pre>moveAndTurn( 4, 3 ) moveAndTurn( 5, 1 ) moveAndTurn( 2, 0 )</pre>	Counting / state
<pre>moveAndTurn( 3, 1 ) moveAndTurn( 4, 1 ) moveAndTurn( 1, 0 )</pre>	Simplified turns

The figure below shows a turtle in a grid of squares. The turtle can move into a white or gray square but cannot move into a black region. Consider the grid and function MoveAndTurn below.

```
def MoveAndTurn(numMoves, numTurns):
    move_forward(numMoves)
    turn_right(numTurns)
```



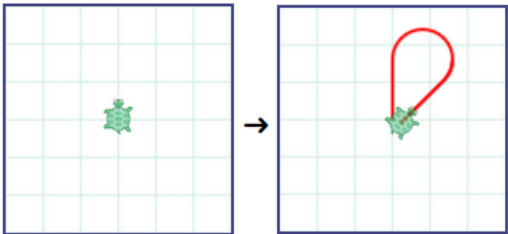
Which of the following code segments will move the turtle to the gray square?

<pre>MoveAndTurn(3, 3) MoveAndTurn(1, 4) MoveAndTurn(0, 1)</pre>	Param order
<pre>MoveAndTurn(3, 3) MoveAndTurn(4, 1) MoveAndTurn(1, 0)</pre>	CORRECT
<pre>MoveAndTurn(4, 3) MoveAndTurn(5, 1) MoveAndTurn(2, 0)</pre>	Counting / state
<pre>MoveAndTurn(3, 1) MoveAndTurn(4, 1) MoveAndTurn(1, 0)</pre>	Simplified turns

Q8

Consider the following function for creating a flower petal:

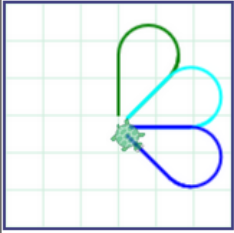
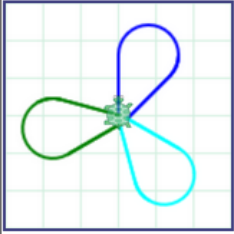
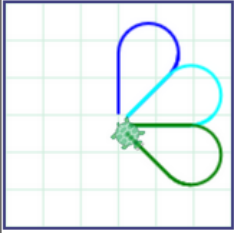
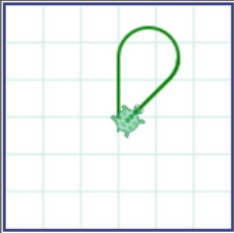
```
def draw_petal(color):  
    pen(color, 2)  
    fd(40)  
    ra(225, 20)  
    fd(40)  
    rt(180)  
draw_petal('red')
```



Using this function, following block of code is executed:

```
draw_petal('blue')  
draw_petal('teal')  
draw_petal('green')
```

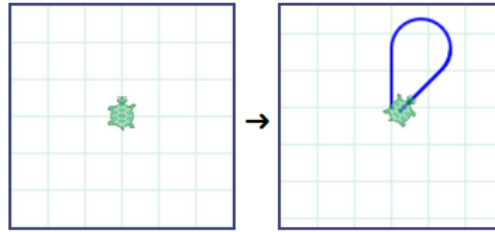
What will be the result of running this code?

	Call ordering
	Association of desired with actual outcome
	Correct
	Maintenance of state

Consider the following function for creating a flower petal:

```
def draw_petal(color):
    pen(color, 2)
    fd(40)
    ra(225, 20)
    fd(40)
    rt(180)
```

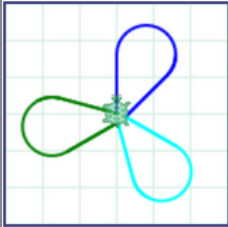
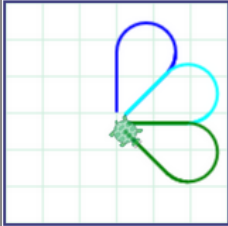

*draw\_petal('blue')*



Using this function, following block of code is executed:

```
draw_petal('blue')
draw_petal('teal')
draw_petal('green')
```

What will be the result of running this code?

	Call ordering
	Association of desired with actual outcome
	Correct
	Maintenance of state

Q9

Consider the following code segment:

```
__MISSING_CODE__  
if isPlayer:  
    print ("I'm looking for other players.")  
else:  
    if isOffline:  
        print ("I guess I'll catch you next time.")  
    else:  
        print ("Shall we play a game?")
```

Select the values for the variables `isPlayer` and `isOffline` that result in this output:

"I guess I'll catch you next time."

<code>isPlayer = True</code> <code>isOffline = True</code>	Misunderstanding of nesting
<code>isPlayer = True</code> <code>isOffline = False</code>	Nesting; Associate name truth value
<code>isPlayer = False</code> <code>isOffline = False</code>	Associate name with truth value
<code>isPlayer = False</code> <code>isOffline = True</code>	CORRECT



Consider the following code segment:

```
<MISSING_CODE>
if isPlayer:
    print("I'm looking for other players.")
else:
    if isOffline:
        print("I guess I'll catch you next time.")
    else:
        print("Shall we play a game?")
```

Select the values for the variables `isPlayer` and `isOffline` that result in this output:

"I guess I'll catch you next time."

<code>isPlayer = True</code> <code>isOffline = True</code>	Misunderstanding of nesting
<code>isPlayer = True</code> <code>isOffline = False</code>	Nesting; Associate name truth value
<code>isPlayer = False</code> <code>isOffline = False</code>	Associate name with truth value
<code>isPlayer = False</code> <code>isOffline = True</code>	CORRECT

Q10

Consider the following code segment:

```
a = 4
b = 2
c = 7
if c >= a:
    if b >= c:
        print("I see what you did there...")
    else:
        print("You can't win them all!")
print("Make sure you say bye when you go.")
```

What is printed as a result of this code's execution?

You can't win them all! Make sure you say bye when you go.	CORRECT
I see what you did there... Make sure you say bye when you go.	Misunderstanding of bool values
Make sure you say bye when you go.	Misunderstanding of if/else and/or comparison
You can't win them all!	Fails to continue after if/else

Consider the following code segment:

```
a = 4
b = 2
c = 7

if c >= a:
    if b >= c:
        print("I see what you did there...")
    else:
        print("You can't win them all!")

print("Make sure you say bye when you go.")
```

What is printed as a result of this code's execution?

You can't win them all! Make sure you say bye when you go.	CORRECT
I see what you did there... Make sure you say bye when you go.	Misunderstanding of bool values
Make sure you say bye when you go.	Misunderstanding of if/else and/or comparison
You can't win them all!	Fails to continue after if/else

Q11

A programmer is trying to move a turtle in a grid to a gray square. The turtle can move into a white or gray square but cannot move into a black region. Consider the following program:

```

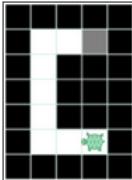
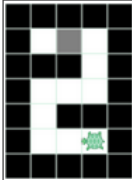
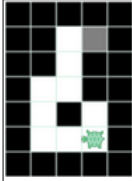
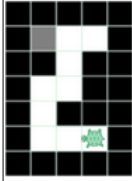
while not goal_reached():
    if can_move_left():
        turn_left(1)
    if can_move_right():
        turn_right(1)
    if can_move_forward():
        move_forward(1)

```

Refer to the *Code Guide* for these helper functions:

- goal\_reached()
- can\_move\_forward()
- can\_move\_left()
- can\_move\_right()
- move\_forward(numMoves)
- turn\_left(numTurns)
- turn\_right(numTurns)

For which of the following grids does the program NOT correctly move the turtle to the gray square?

	Easy check – clear distractor
	Condition check termination
	CORRECT
	Ordering

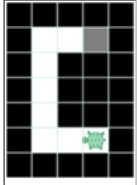
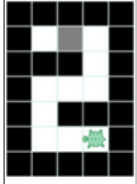
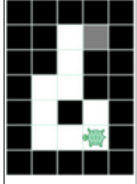
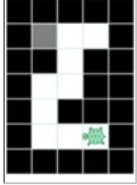
A programmer is trying to move a turtle in a grid to a gray square. The turtle can move into a white or gray square but cannot move into a black region. Consider the following program:

```
while not goal_reached():
    if can_move_left():
        turn_left(1)
    if can_move_right():
        turn_right(1)
    if can_move_forward():
        move_forward(1)
```

*Refer to the Code Guide for these helper functions:*

```
goal_reached()
can_move_forward()
can_move_left()
can_move_right()
move_forward(numMoves)
turn_left(numTurns)
turn_right(numTurns)
```

For which of the following grids does the program NOT correctly move the turtle to the gray square?

	Easy check – clear distractor
	Condition check termination
	CORRECT
	Ordering

Q12

A programmer is trying to move a turtle in a grid to a gray square. The turtle can move into a white or gray square but cannot move into a black region. Consider the following grid:



Refer to the *Code Guide* for these helper functions:

```

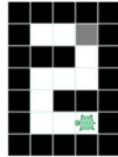
goal_reached()
can_move_forward()
can_move_left()
can_move_right()
move_forward(numMoves)
turn_left(numTurns)
turn_right(numTurns)
    
```

Which code segment works for the grid above?

<pre> while not goal_reached():     if can_move_forward():         move_forward(1)     if can_move_left():         turn_left(1)     if can_move_right():         turn_right(1)     if can_move_forward():         move_forward(1)         </pre>	<p>Terminal check</p>	<pre> while not goal_reached():     if can_move_forward():         move_forward(1)     if can_move_left():         turn_left(1)     if can_move_forward():         move_forward(1)     if can_move_right():         turn_right(1)         </pre>	<p>Ordering</p>
<pre> while not goal_reached():     if can_move_forward():         move_forward(1)     if can_move_left():         turn_left(1)     if can_move_forward():         move_forward(1)     if can_move_left():         turn_left(1)         </pre>	<p>Can't turn right</p>	<pre> while not goal_reached():     if can_move_forward():         move_forward(1)     if can_move_right():         turn_right(1)     if can_move_forward():         move_forward(1)     if can_move_left():         turn_left(1)         </pre>	<p>CORRECT</p>

A programmer is trying to move a turtle in a grid to a gray square. The turtle can move into a white or gray square but cannot move into a black region. Consider the following grid:

Refer to the **Code Guide** for these helper functions:



```

goal_reached()
can_move_forward()
can_move_left()
can_move_right()
move_forward(numMoves)
turn_left(numTurns)
turn_right(numTurns)

```

Which code segment works for the grid above?

<pre> while not goal_reached():     if can_move_forward():         move_forward(1)     if can_move_left():         turn_left(1)     if can_move_right():         turn_right(1)     if can_move_forward():         move_forward(1) </pre>	<p>Terminal check</p>	<pre> while not goal_reached():     if can_move_forward():         move_forward(1)     if can_move_left():         turn_left(1)     if can_move_forward():         move_forward(1)     if can_move_right():         turn_right(1) </pre>	<p>Ordering</p>
<pre> while not goal_reached():     if can_move_forward():         move_forward(1)     if can_move_left():         turn_left(1)     if can_move_forward():         move_forward(1)     if can_move_left():         turn_left(1) </pre>	<p>Can't turn right</p>	<pre> while not goal_reached():     if can_move_forward():         move_forward(1)     if can_move_right():         turn_right(1)     if can_move_forward():         move_forward(1)     if can_move_left():         turn_left(1) </pre>	<p>CORRECT</p>

Q13

You are writing a program that tells players if it is time to evolve their monsters in the "Go Go Monster" game. The level is stored in a variable called `level`, and the program should respond like this:

- If the level is 10 or more, display "It's time to evolve your monster!"
- If the level is 8 or 9, display "You're getting close!"
- If the level is anything else, display "Hang in there!"
- At the end, display "Thanks for stopping by!"

What code block will make the program function correctly?

<pre>if level &gt;= 10 :   print ("It's time to evolve your monster!") + if level &gt; 7 :   print ("You're getting close!") else:   print ("Hang in there!") + print ("Thanks for stopping by!")</pre>	No outer "else", not understanding need due to presence of other condition
<pre>if level &gt;= 10 :   print ("It's time to evolve your monster!") else:   if level &gt; 7 :     print ("You're getting close!")     +   print ("Hang in there!")   + print ("Thanks for stopping by!")</pre>	No inner "else"; misund. of else's purpose
<pre>if level &gt;= 10 :   print ("It's time to evolve your monster!") else:   if level &gt; 7 :     print ("You're getting close!")   else:     print ("Hang in there!")   + print ("Thanks for stopping by!") +</pre>	CORRECT
<pre>if level &gt;= 10 :   print ("It's time to evolve your monster!") else:   if level &gt; 7 :     print ("You're getting close!")   else:     print ("Hang in there!")   + print ("Thanks for stopping by!") +</pre>	Nested final statement



You are writing a program that tells players if it is time to evolve their monsters in the "Go Go Monster" game. The level is stored in a variable called `level`, and the program should respond like this:

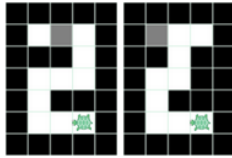
- If the level is 10 or more, display "It's time to evolve your monster!"
- If the level is 8 or 9, display "You're getting close!"
- If the level is anything else, display "Hang in there!"
- At the end, display "Thanks for stopping by!"

What code block will make the program function correctly?

<pre> if level &gt;= 10:     print("It's time to evolve your monster!") if level &gt; 7:     print("You're getting close!") else:     print("Hang in there!")  print("Thanks for stopping by!") </pre>	<p>No outer "else", not understanding need due to presence of other condition</p>
<pre> if level &gt;= 10:     print("It's time to evolve your monster!") else:     if level &gt; 7:         print("You're getting close!")     print("Hang in there!")  print("Thanks for stopping by!") </pre>	<p>No inner "else"; misund. of else's purpose</p>
<pre> if level &gt;= 10:     print("It's time to evolve your monster!") else:     if level &gt; 7:         print("You're getting close!")     else:         print("Hang in there!")  print("Thanks for stopping by!") </pre>	<p>CORRECT</p>
<pre> if level &gt;= 10:     print("It's time to evolve your monster!") else:     if level &gt; 7:         print("You're getting close!")     else:         print("Hang in there!")     print("Thanks for stopping by!") </pre>	<p>Nested final statement</p>

Q14

A programmer is trying to move a turtle in a grid to a gray square. The turtle can move into a white or gray square but cannot move into a black region. Consider the following terrain grids:



Refer to the **Code Guide** for these helper functions:

```

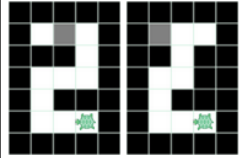
goal_reached()
can_move_forward()
can_move_left()
can_move_right()
move_forward(numMoves)
turn_left(numTurns)
turn_right(numTurns)
  
```

Which code segment will get the turtle to the goal for ALL of the grids above?

<pre> while not goal_reached():   if can_move_forward():     move_forward(1)   if can_move_right():     turn_right(1)     move_forward(1)   if can_move_left():     turn_left(1)     move_forward(1)   </pre>	<p>Ordering (2)</p>	<pre> while not goal_reached():   if can_move_right():     turn_right(1)   if can_move_forward():     move_forward(1)   if can_move_left():     turn_left(1)   if can_move_forward():     move_forward(1)   </pre>	<p>CORRECT</p>
<pre> while not goal_reached():   if can_move_left():     turn_left(1)   if can_move_forward():     move_forward(1)   if can_move_right():     turn_right(1)   if can_move_forward():     move_forward(1)   </pre>	<p>Loop condition termination timing (1)</p>	<pre> while not goal_reached():   if can_move_left():     turn_left(1)     move_forward(1)   if can_move_right():     turn_right(1)     move_forward(1)   if can_move_forward():     move_forward(1)   </pre>	<p>Both (1, 2)</p>

A programmer is trying to move a turtle in a grid to a gray square. The turtle can move into a white or gray square but cannot move into a black region. Consider the following terrain grids:

Refer to the **Code Guide** for these helper functions:



```

goal_reached()
can_move_forward()
can_move_left()
can_move_right()
move_forward(numMoves)
turn_left(numTurns)
turn_right(numTurns)

```

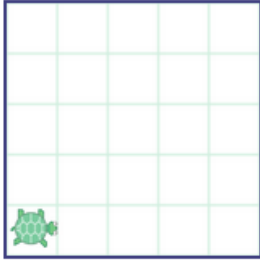
Which code segment will get the turtle to the goal for ALL of the grids above?

<pre> while not goal_reached():     if can_move_forward():         move_forward(1)     if can_move_right():         turn_right(1)         move_forward(1)     if can_move_left():         turn_left(1)         move_forward(1) </pre>	<b>Ordering (2)</b>	<pre> while not goal_reached():     if can_move_left():         turn_left(1)     if can_move_forward():         move_forward(1)     if can_move_right():         turn_right(1)     if can_move_forward():         move_forward(1) </pre>	<b>Loop condition termination timing (1)</b>
<pre> while not goal_reached():     if can_move_right():         turn_right(1)     if can_move_forward():         move_forward(1)     if can_move_left():         turn_left(1)     if can_move_forward():         move_forward(1) </pre>	<b>CORRECT</b>	<pre> while not goal_reached():     if can_move_left():         turn_left(1)         move_forward(1)     if can_move_right():         turn_right(1)         move_forward(1)     if can_move_forward():         move_forward(1) </pre>	<b>Both (1, 2)</b>

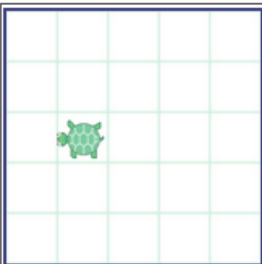
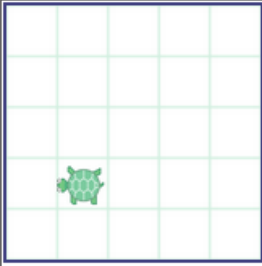


Q15

Consider the following code segment and position.

```
for n in range(1, 3):  
    for k in range(0, n):  
        move_forward(1)  
    left_turn(1)
```

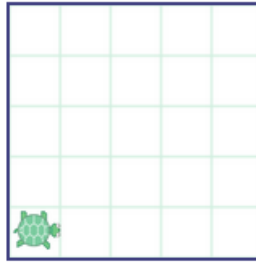


Which of the following shows the location of the turtle after running the code segment?

	CORRECT
	Inner loop limit unchanging
	Treating lt() as if not nested
	Treat lt() as if in inner loop

Consider the following code segment and position.

```
for n in range(1, 3):  
    for k in range(0, n):  
        move_forward(1)  
        turn_left(1)
```



Which of the following shows the location of the turtle after running the code segment?

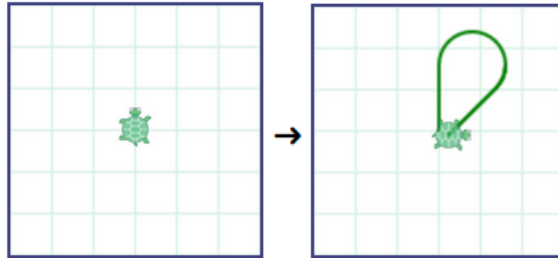
	CORRECT
	Inner loop limit unchanging
	Treating lt() as if not nested
	Treat lt() as if in inner loop

Q16

Consider the following function for creating a flower petal:

```
def draw_petal(color):
    pen(▼color, ▼2)
    fd(▼40)
    ra(▼225, ▼20)
    fd(▼40)
    rt(▼225)
```

```
draw_petal('green')
```



Using this function, the following `flower()` function is constructed:

```
def draw_flower( center, petals ):
    draw_petal( petals )
    draw_petal( petals )
    draw_petal( petals )
    draw_petal( petals )
    dot(▼center, ▼30)
```

What will be the result of running this line of code?

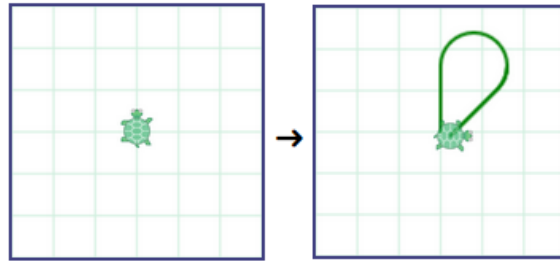
```
draw_flower('orange', 'blue')
```

	<p>Maintenance of state</p>		<p>Assoc. of desired with actual outcome</p>
	<p>Call ordering</p>		<p>Correct</p>

Consider the following function for creating a flower petal:

```
def draw_petal(color):  
    pen(color, 2)  
    fd(40)  
    ra(225, 20)  
    fd(40)  
    rt(225)
```

*draw\_petal('green')*



Using this function, the following flower() function is constructed:

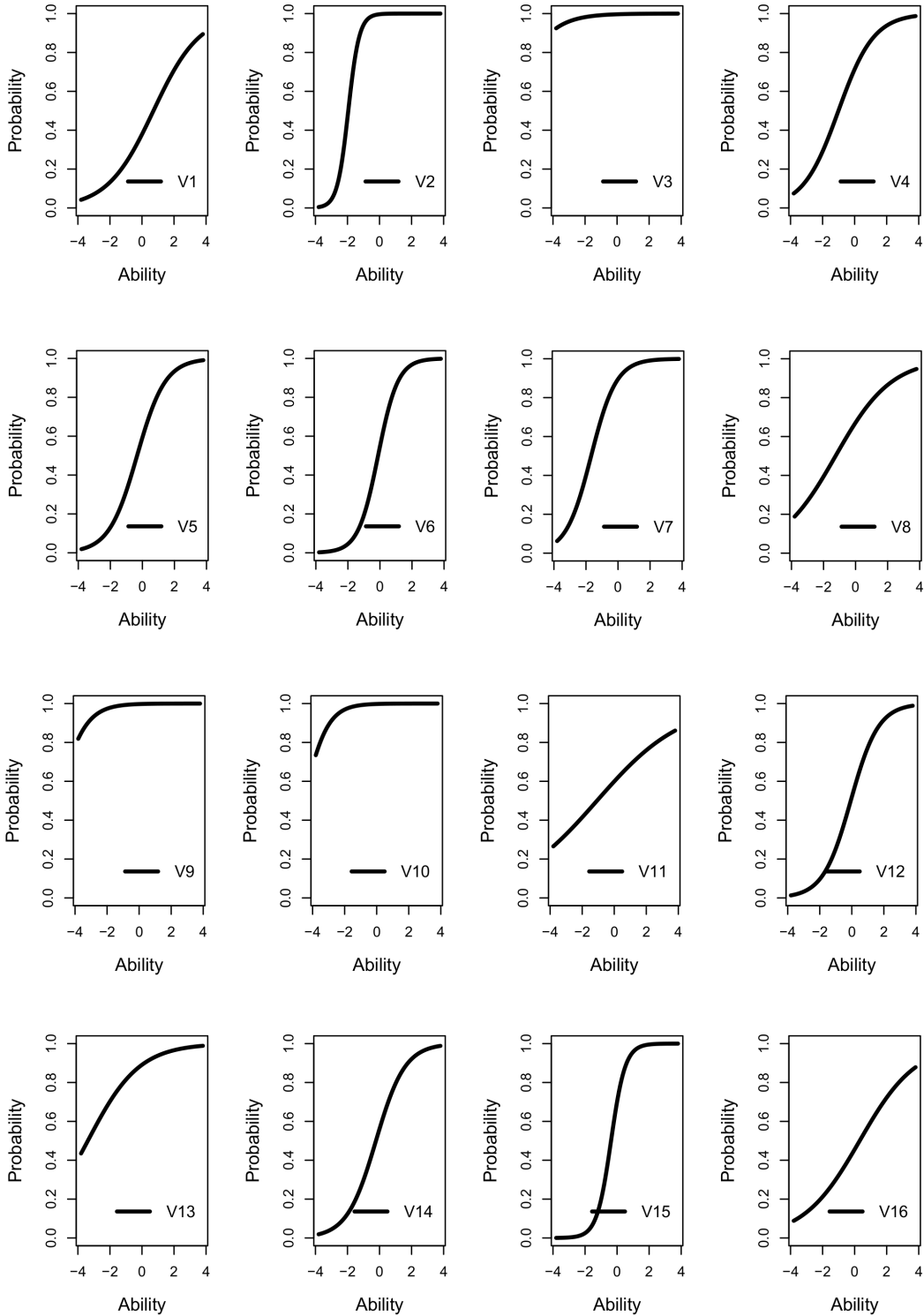
```
def draw_flower(center, petals):  
    draw_petal(petals)  
    draw_petal(petals)  
    draw_petal(petals)  
    draw_petal(petals)  
    dot(center, 30)
```

What will be the result of running this line of code?

```
draw_flower('orange', 'blue')
```

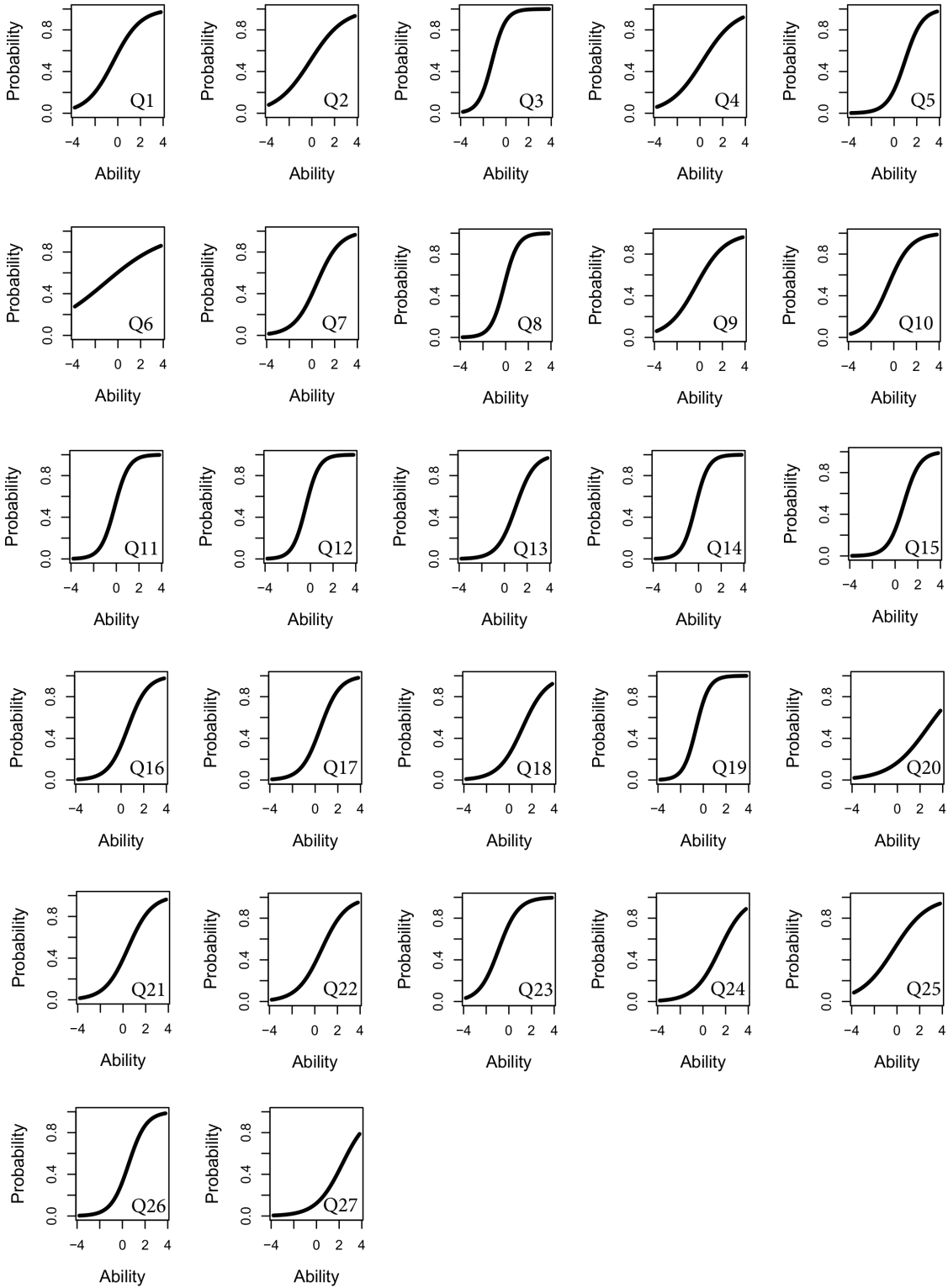
	Maintenance of state		Assoc. of desired with actual outcome
	Call ordering		Correct

APPENDIX H  
ITEM ANALYSIS: CUSTOM ASSESSMENT IN CS1 COURSE





APPENDIX I  
ITEM ANALYSIS: SCS1 IN CS1 COURSE



APPENDIX J  
CONDITION AND EXPERIENCE INTERACTIONS

Table J-1. Mean & Standard Deviation: Condition x Experience

Exams	No Experience		Blocks Experience		Text-Only	
	Basl. $\mu$ , $\sigma$	Intv. $\mu$ , $\sigma$	Basl. $\mu$ , $\sigma$	Intv. $\mu$ , $\sigma$	Basl. $\mu$ , $\sigma$	Intv. $\mu$ , $\sigma$
SCS1 All	48.3, 16.5	44.5, 16.7	55.0, 21.1	52.0, 16.3	54.4, 20.0	58.3, 18.5
SCS1 Definitional	55.4, 17.6	51.5, 20.4	60.1, 23.8	60.3, 18.4	61.7, 19.4	62.6, 18.3
SCS1 Tracing	48.0, 20.2	42.8, 19.2	52.3, 21.5	52.0, 20.4	56.8, 21.9	57.4, 22.1
SCS1 Completion	40.2, 23.4	36.5, 20.8	51.0, 27.0	43.1, 22.2	44.4, 27.4	51.1, 25.3
Ex. 1 Def. / Read.	56.7, 15.8	81.3, 14.8	63.7, 20.7	88.5, 10.1	60.0, 15.4	88.4, 12.1
Ex. 1 Writing	60.7, 22.8	60.8, 25.9	79.7, 29.6	84.6, 17.5	70.6, 23.6	89.2, 15.0
Ex. 2 Def. / Read.	70.5, 19.5	74.9, 20.1	76.2, 19.8	78.3, 14.4	72.2, 19.8	80.7, 15.8
Ex. 2 Writing	67.6, 27.2	58.1, 25.9	74.4, 24.5	74.9, 22.6	69.5, 30.5	78.5, 18.8
Final Exam	66.4, 17.4	69.1, 16.6	58.8, 17.4	74.8, 13.9	66.5, 16.7	79.2, 13.2

APPENDIX K  
 PLUGIN EVENT COUNTS AND CATEGORY MAPPING

Table K-1. Table of Event Counts and Percentages by Module (Chronological)

M	Date Range	View Palette	Get Block	Put Block	Go to Blocks	Go to Text
0	08/22–08/28	2450 (43.7%)	323 (5.8%)	259 (4.6%)	1308 (23.3%)	1269 (22.6%)
1	08/29–09/04	3543 (48.7%)	377 (5.2%)	402 (5.5%)	1488 (20.5%)	1460 (20.1%)
2	09/05–09/11	5206 (52.0%)	514 (5.1%)	584 (5.8%)	1882 (18.8%)	1826 (18.2%)
3	09/12–09/18	8159 (61.1%)	461 (3.5%)	475 (3.6%)	2159 (16.2%)	2100 (15.7%)
4	09/19–10/02	7577 (69.2%)	154 (1.4%)	124 (1.1%)	1566 (14.3%)	1527 (13.9%)
5	10/03–10/09	7048 (74.0%)	109 (1.1%)	85 (0.9%)	1146 (12.0%)	1132 (11.9%)
6	10/10–10/16	8511 (76.2%)	65 (0.6%)	116 (1.0%)	1228 (11.0%)	1251 (11.2%)
7	10/17–10/23	4798 (76.3%)	32 (0.5%)	23 (0.4%)	717 (11.4%)	715 (11.4%)
8	10/24–11/06	12066 (81.1%)	40 (0.3%)	28 (0.2%)	1380 (9.3%)	1359 (9.1%)
9	11/07–11/13	10849 (83.3%)	9 (0.1%)	7 (0.1%)	1055 (8.1%)	1099 (8.4%)
10	11/14–11/27	19029 (84.7%)	6 (0.0%)	5 (0.0%)	1704 (7.6%)	1726 (7.7%)
11	11/28–12/11	19132 (81.9%)	23 (0.1%)	12 (0.1%)	2046 (8.8%)	2136 (9.1%)

Table K-2. Mapping if Event Name to Event Category

Event	Category
View Palette	Palette Viewing
Grab Block	Block Use
Place Block	Block Use
Go to Blocks	Mode Swapping
Go to Text	Mode Swapping

APPENDIX L  
CS1 STUDY CODEBOOK AND RESULTS TABLE BY MODULE NUMBER

Table L-1. Codebook: Why Dual-Modality Instruction is Helpful / Not Helpful

Code	Definition	Example
Accustomed	Student is used to blocks/text	“I am used to programming in blocks.”
Aesthetic	Related to the look (form); visual appeal; includes style	“I like the look of text better than blocks”
Boilerplate	Used (blocks) to provide boilerplate / setup code / syntax	“I use blocks to initially set the project up, but the text is what I use for the majority of my work.”
Blocking	Separation / identification of specific constructs / grouping of code into blocks	“The colors make the grouping of code easier to follow for a beginner.”
Colors	Related to /mention of colors (usually of blocks)	“I like seeing the commands color coded in instruction.”
Confusing	Contributes to confusion / misunderstanding	“Blocks make things slightly more confusing.”
Connection	Establish / follow connections between constructs	“Blocks... shows how... blocks fit together and how the logic flows”
Correctness	Related to correctness / validity of code	“Because doing it in both blocks and text mode is very useful in showing how organized and valid my code is.”
Debugging	Finding mistakes in code	“Blocks make it very hard for one to decipher issues within the text”
Dependency	Dependency on blocks inhibits learning of programming as done in the “real world”	“Ultimately I believe this will hurt you the more you rely it, and then it's harder to switch to pure text while learning harder material.”
Distraction	More/fewer distractions in environment	“I find blocks mode to be rather distracting to the eye, it sometimes takes my focus off the content”
Enjoyability	Expression enjoyment	“I like coding as I type”
Experienced	Student indicated they had prior programming experience	“I wrote most of my codes in text because I think people who have experience with java should use that.”
Formatting	Related to formatting of code	“It allows me to see the proper way to format my code if I am confused.”
Freedom	Noted freedom of using either / both modes as a strength	“Greater freedom with my coding”
Functionality	Related to understanding / conceptualization of functionality	“I believe that it helps students understand the function of each block better than coding in text.”
Importance	Mention of general importance	“Because I feel this more important.”
Introduction	Introducing constructs to beginners	“It should be used as a beginner introduction.”

Learning (General)	Facilitates / inhibits learning in general	"I learn better in text because I program in text"
Learning (Syntax)	Facilitates / inhibits learning of new language syntax	"I think the block coding will make it easier in the beginning of the class to write code when I still am learning Java syntax."
Lectures	Help or hinder student(s) understand the lecture material	"Especially when I am sitting far back in the lecture hall."
Links Blocks & Text	Connecting concepts / ideas between blocks and text	"It allows you to see the structure of the code in blocks while gaining the understanding of every part from the text."
Organization	Related to organization of code / concepts	"Helps me organize my thoughts better."
No Longer Needed	Student no longer needed blocks (suggesting they previously found them useful)	"I have just stopped using block altogether because I have improved in my coding ability."
Perspective	Seeing things from multiple perspectives / points of view	"It just helps me see the same material twice"
Preference	Student noted a preference	"I like text more"
Puzzle-Like	Resembles puzzle-piece systems	"Blocks feels like a visual puzzle"
Reading	Reading / readability code	"I think by programming in blocks you make your code easier to read"
Scaffolding	Provides cognitive support; helping students get "unstuck", reminders, framework, enumeration, etc.	"I mainly used text, but switched if I couldn't remember the syntax for a command or function."
Scope	Related to program scoping for elements	"Blocks is a nice visualization of the code, which should help to see the scope of blocks and variables."
Sequencing	Facilitates conceptualization of sequencing / logic / execution	"Because it is easier to visualize the sequences"
Simplicity	Simpler	"I think blocks simplify the code"
Speed	Impacts how fast user can program	"It... allows you to move faster"
Structure	Structuring code or understanding structure	"It... makes your work more structured"
Transitioning	Relating to the transition between text and blocks modes	"Learning in blocks then transitioning to text is the most helpful for me."
Understanding	Understanding code / concepts generally	"For block-programming... the logic is easier to understand than text."
Unnecessary	Not needed for some reason	"The blocks seem unnecessary at times, especially if you know what to type"
Visualization	Related to visualization of the code, facilitating or inhibiting learning (function)	"I(t) helps visualize the code."

Table L-2. Table of Code Counts of Responses Indicating Instruction was Helpful, by Module

Code	M1	M3	M4	M7	M11	No Exp	Blocks	Text	All
Accustomed	1	0	0	0	0	0	1	0	1
Aesthetic	1	1	1	1	1	0	0	1	1
Boilerplate	0	0	2	1	0	1	2	9	3
Blocking	2	3	3	2	4	3	4	2	9
Colors	3	3	2	4	2	4	2	3	9
Connection	0	2	2	1	0	2	1	0	3
Correctness	1	0	0	0	0	1	0	0	1
Debugging	0	1	1	0	0	0	1	0	1
Enjoyability	1	0	0	0	0	1	0	0	1
Formatting	2	2	0	0	0	1	2	1	4
Freedom	0	1	0	0	1	1	0	0	1
Functionality	1	0	0	0	0	0	1	0	1
Importance	0	0	1	0	0	1	0	0	1
Introduction	7	5	3	4	3	2	6	4	12
Learning (General)	2	3	1	0	2	2	3	1	6
Learning (Syntax)	3	0	0	1	0	3	1	0	4
Lectures	0	5	2	1	1	4	2	1	7
Links Blocks / Text	1	0	0	0	0	0	1	0	1
Organization	2	3	2	0	1	4	1	0	5
Perspective	1	0	1	2	4	3	2	0	5
Preference	0	0	1	0	0	0	0	1	1
Reading	3	0	1	1	1	2	2	0	4
Scaffolding	4	4	1	3	1	5	3	3	11
Scope	1	0	0	0	0	0	0	1	1
Sequencing	2	0	1	0	1	2	1	1	4
Simplicity	1	2	0	1	0	3	0	1	4
Speed	0	2	0	1	0	1	1	0	2
Structure	5	4	5	6	3	6	6	2	14
Transitioning	0	0	1	0	0	0	0	1	1
Understanding	4	2	4	5	5	4	7	2	13
Visualization	13	9	7	11	10	11	9	6	26

Table L-3. Table of Code Counts of Responses Indicating Instruction Not Helpful, by Module

Code	M1	M3	M4	M7	M11	No Exp	Blocks	Text	All
Accustomed	2	4	1	3	4	4	3	0	7
Aesthetic	0	0	0	0	1	1	0	0	1
Colors	1	0	0	0	0	0	0	1	1
Confusing	1	2	0	1	2	2	0	2	4
Connection	1	0	0	0	0	0	0	1	1
Dependency	2	2	4	2	6	2	3	3	8
Details	0	0	0	1	0	0	0	1	1
Distraction	2	1	0	0	0	0	0	2	2
Experienced	0	0	1	0	2	1	1	0	2
Frustrating	0	1	0	0	0	0	1	0	1
Learning (Syntax)	0	1	1	0	1	1	1	0	2
Lectures	0	1	0	1	0	0	1	1	2
No Longer Needed	0	1	2	1	1	4	0	0	4
Practice	0	0	0	1	0	0	1	0	1
Puzzle-Like	1	0	0	1	0	1	0	0	1
Reading	0	0	0	0	1	0	0	1	1
Speed	0	2	1	1	0	1	2	0	3
Unnecessary	0	0	0	1	2	2	0	1	3
Visualization	0	0	1	1	0	1	0	1	2

APPENDIX M  
DISCUSSION WITH CURRICULUM COMMITTEE CHAIR

The following is the correspondence seeking and receiving approval to use the hybrid instructional approach in the COP3502 course. The following message was sent to the department's Curriculum Committee Chair, Arunava Banerjee:

From: Jeremiah Blanchard <jblanch@cise.ufl.edu>  
Sent: Tuesday, April 3, 2018 9:23 PM  
To: Banerjee, Arunava <arunava@ufl.edu>  
Subject: Curriculum committee

Hi Dr. Banerjee,

I am working on some updates to the Programming I course, and in hand with that I am looking into conducting a study about some of these changes and some tools I would like to bring into the course.

Since I have a research interest in the results, it was suggested that I might see if there are a few minutes that I could come into the curriculum committee meeting briefly to describe the changes. In a nutshell - a lot of our youngest students come in with prior experience. Some of those are in text, but some are in blocks languages. I'm working on a tool that is intended to help students move into text more easily, and I would present multiple modes of the same source to students to build a connection between what they've learned previously and the content. (This won't involve any change in language or topics covered.)

Regards,  
Jeremiah Blanchard

The following was the chair's response:

From: Banerjee, Arunava <arunava@ufl.edu>  
Sent: Tuesday, April 4, 2018 10:15 PM  
To: Blanchard, Jeremiah J <jblanch@cise.ufl.edu>  
Subject: Re: Curriculum committee

That is wonderful Jeremiah. We do not have regularly scheduled curriculum meetings (in fact, we do not have any physical meetings). When I need something passed by the committee, I simply run it via email.

So long as you are not changing the curriculum, I would say that you should feel free to experiment. In fact, I would encourage you to do things that you believe in your heart will help the students learn better. Not all experiments turn out for the good, but so long as you genuinely intend it for improvement, I am happy to stand behind you.

-Arunava



Arunava Banerjee  
Associate Professor  
Computer & Information Science & Engineering  
University of Florida  
[www.cise.ufl.edu/~arunava](http://www.cise.ufl.edu/~arunava)

## LIST OF REFERENCES

- [1] Agarwal, A. and Agarwal, K.K. 2003. Some deficiencies of C++ in teaching CS1 and CS2. *ACM SIGPlan Notices*. 38, 6 June (2003), 9–13.
- [2] Ahmadzadeh, M., Elliman, D. and Higgins, C. 2005. An analysis of patterns of debugging among novice Computer Science students. *Proceedings of the 10th annual SIGCSE conference on Innovation and Technology in Computer Science Education (ITiCSE '05)* (2005), 84–88.
- [3] Al-bow, M., Austin, D., Edgington, J., Fajardo, R., Fishburn, J., Lara, C., Meyer, S., History, A., Science, C. and Studies, D.M. 2008. Using Greenfoot and Games to Teach Rising 9th and 10th Grade Novice Programmers. *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games* (2008), 55–59.
- [4] Auerbach, Carl and Silverstein, L.B. 2003. Qualitative data: An introduction to coding and analysis. *Qualitative data: An introduction to coding and analysis*. NYU press. 31–87.
- [5] Baker, F.B. 2001. *The Basics of Item Response Theory*.
- [6] Barnett, S.M. and Ceci, S.J. 2002. When and where do we apply what we learn?: a taxonomy for far transfer. *Psychological Bulletin*. 128, 4 (2002), 612–637.
- [7] Bau, D. 2015. Droplet, a blocks-based editor for text code. *Journal of Computing Sciences in Colleges*. 30, 6 (2015), 138–144.
- [8] Bau, D.A. and Bau, D.A. 2014. A Preview of Pencil Code. *Proceedings of the 2nd Workshop on Programming for Mobile & Touch - PROMOTO '14* (2014), 21–24.
- [9] Bau, D., Bau, D.A., Pickens, C.S. and Dawson, M. 2015. Pencil Code: Block Code for a Text World. *Proceedings of the 14th International Conference on Interaction Design and Children* (2015), 445–448.
- [10] Bau, D., Gray, J., Kelleher, C., Sheldon, J. and Turbak, F. 2017. Learnable Programming: Blocks and Beyond. *Communications of the ACM*. 60, 6 (2017), 72–80.
- [11] Begel, A. 1996. *LogoBlocks: A graphical programming language for interacting with the world*. Ph. D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA.
- [12] Begel, A. and Klopfer, E. 2007. StarLogo TNG: An Introduction to Game Development. *Journal of E-Learning*. 53, (2007), 146.
- [13] Blanchard, J. 2017. Hybrid Environments: A Bridge from Blocks to Text. *Proceedings of the 2017 ACM Conference on International Computing Education Research* (2017), 295–296.

- [14] Blanchard, J., Gardner-McCune, C. and Anthony, L. 2019. Effects of Code Representation on Student Perceptions and Attitudes Toward Programming. *2019 IEEE Symposium on Visual Languages and Human-Centric Computing* (2019), 127–131.
- [15] Blanchard, J., Gardner-McCune, C. and Anthony, L. 2018. How Perceptions of Programming Differ in Children with and without Prior Experience. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (2018), 1099.
- [16] Blanchard, J., Gardner-McCune, C. and Anthony, L. 2020. Dual-Modality Instruction and Learning: A Case Study in CS1. *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (2020), 818–824.
- [17] Blanchard, J., Gardner-McCune, C. and Anthony, L. 2019. Amphibian: Dual-Modality Representation in Integrated Development Environments. *Proceedings of the 2019 IEEE Blocks and Beyond Workshop (Blocks and Beyond)* (2019), 83–85.
- [18] Bontá, P., Papert, A. and Silverman, B. 2010. Turtle , Art , TurtleArt Programming in TurtleArt. *Proc. of Constructionism 2010 Conference.* (2010), 1–9.
- [19] Borstler, J., Johansson, T. and Nordstroin, M. 2002. Teaching OO concepts-a case study using CRC-cards and BlueJ. *Frontiers in Education.* 32, 1 (2002), T2G-1.
- [20] Brainerd, C.J. 1978. *Piaget's Theory of Intelligence.* Prentice Hall.
- [21] Brennan, K. and Resnick, M. 2012. New frameworks for studying and assessing the development of computational thinking. *Proceedings of the 2012 annual meeting of the American Educational Research Association* (2012), 1–25.
- [22] Brockwell, P.J. and Davis, R.A. 2002. *Introduction to Time Series and Forecasting* (2nd. ed.). Springer, New York, NY.
- [23] Brown, P.H. 2008. Some field experience with Alice. *Journal of Computing Sciences in Colleges.* 24, 2 (2008), 213–219.
- [24] Buckley, M., Kershner, H., Schindler, K., Alphonse, C. and Braswell, J. 2004. Benefits of using socially-relevant projects in computer science and engineering education. *ACM SIGCSE Bulletin.* 36, 1 (2004), 482–486.
- [25] Carlson, J.E., Davier, M. von and von Davier, M. 2013. Item response theory (ETS R&D Scientific and Policy Contribution Series ETS SPC-13-05). *Princeton, NJ: Educational Testing Service.* (2013).
- [26] Conway, M., Audia, S., Burnette, T., Cosgrove, D. and Christiansen, K. 2000. Alice: lessons learned from building a 3D system for novices. *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI'00* (2000), 486–493.
- [27] Conway, M.J. 1997. *Alice: easy-to-learn 3D scripting for novices.* Ph. D. Dissertation. University of Virginia, Charlottesville, VA.

- [28] Cooper, S., Dann, W., and Pausch, R. 2000. Developing Algorithmic Thinking With Alice. *The proceedings of ISECON* (2000).
- [29] Cooper, S. 2010. The Design of Alice. *ACM Transactions on Computing Education*. 10, 4 (2010), 15.
- [30] Cooper, S., Dann, W. and Pausch, R. 2000. Alice: a 3-D tool for introductory programming concepts. *Journal of Computing Sciences in Colleges*. 15, 5 (2000), 107–116.
- [31] Cooper, S., Dann, W. and Pausch, R. 2003. Teaching Objects-first In Introductory Computer Science. *ACM SIGCSE Bulletin*. 35, 1 (2003), 191–195.
- [32] Corney, M., Teague, D., Ahadi, A. and Lister, R. 2012. Some Empirical Results for Neo-Piagetian Reasoning in Novice Programmers and the Relationship to Code Explanation Questions. *14th Australasian Computing Education Conference*. 123, February (2012), 77–86.
- [33] Cypher, A. and Smith, D.C. 1995. KidSim: End User Programming of Simulation. *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '95*. (1995), 27–34.
- [34] Dann, W., Cooper, S. and Pausch, R. 2000. Making the connection: programming with animated small world. *ACM SIGCSE Bulletin*. 32, 3 (2000), 41–44.
- [35] Dann, W., Cosgrove, D., Slater, D. and Culyba, D. 2012. Mediated transfer: Alice 3 to Java. *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (2012), 141–146.
- [36] Dillane, J. 2020. Frame-Based Novice Programming. *Proceedings of the 25th annual SIGCSE conference on Innovation and Technology in Computer Science Education (ITiCSE '20)* (2020), 583–584.
- [37] DiSalvo, B., Guzdial, M., Bruckman, A. and McKlin, T. 2014. Saving face while geeking out: Video game testing as a justification for learning computer science. *Journal of the Learning Sciences*. 23, 3 (2014), 272–315.
- [38] Enbody, R.J., Punch, W.F. and McCullen, M. 2009. Python CS1 as preparation for C++ CS2. *ACM SIGCSE Bulletin*. 41, 1 (2009), 116–120.
- [39] Ericson, B. and McKlin, T. 2012. Effective and Sustainable Computing Summer Camps. *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (2012), 289–294.
- [40] Fraser, N. 2015. Ten Things We've Learned from Blockly. *IEEE Blocks and Beyond Workshop*. (2015), 49–50.

- [41] Garcia, D.D., Harvey, B. and Segars, L. 2012. CS principles pilot at University of California, Berkeley. *ACM Inroads*. 3, 2 (2012), 66–68.
- [42] Garlick, R. and Cankaya, E. 2010. Using Alice in CS1: A quantitative experiment. *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*. (2010), 165–168.
- [43] Gindling, J., Ioannidou, A., Loh, J., Lokkebo, O. and Repenning, A. 1995. LEGOsheets: a rule-based programming, simulation and manipulation environment for the LEGO Programmable Brick. *Proceedings of Symposium on Visual Languages*. (1995), 172–179.
- [44] Girden, E.R. 1992. *ANOVA: Repeated measures*. Sage, New York, NY.
- [45] Gobet, F., Lane, P.C.R., Croker, S., Cheng, P.C.-H., Jones, G., Oliver, I. and Pine, J.M. 2001. Chunking mechanisms in human learning. *TRENDS in Cognitive Sciences*. 5, 6 (2001), 236–243.
- [46] Grover, S., Pea, R. and Cooper, S. 2014. Remedying misperceptions of computer science among middle school students. *Proceedings of the 45th ACM technical symposium on Computer science education* (2014), 343–348.
- [47] Gwet, K.L. 2008. Computing inter-rater reliability and its variance in the presence of high agreement. *British Journal of Mathematical and Statistical Psychology*. 61, 1 (2008), 29–48.
- [48] Van Haaster, K. and Hagan, D. 2004. Teaching and learning with BlueJ: an Evaluation of a pedagogical tool. *Information Science + Information Technology Education Joint Conference*. (2004), 455–470.
- [49] Hagan, D., Michael Kolling and Selby Markham 1999. *The BlueJ Experience : Implementing Educational Innovation*.
- [50] Hagan, D. and Selby Markham 2000. Teaching Java with the BlueJ Environment. *Proceedings of Australasian Society for Computers in Learning in Tertiary Education Conference* (2000).
- [51] Henriksen, P. and Kolling, M. 2004. Greenfoot: Combining Object Visualization with Interaction. *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA '04*. (2004), 73–82.
- [52] Hestenes, D., Wells, M. and Swackhamer, G. 1992. Force concept inventory. *The Physics Teacher*. 30, 3 (1992), 141–158.
- [53] Homer, M. and Noble, J. 2017. Lessons in combining block-based and textual programming. *Journal of Visual Languages and Sentient Systems*. 3, 1 (2017), 22–39.
- [54] Hsu, K.C. 1996. *Developing Programming Environments for Programmable Bricks*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA.

- [55] Huitt, W., & Hummel, J. 2003. Piaget's theory of cognitive development. <http://www.edpsycinteractive.org/topics/cognition/piaget.html>. Accessed: 2020-07-24.
- [56] Ingalls, D., Wallace, S., Chow, Y.Y.-Y., Ludolph, F. and Doyle, K. 1988. Fabrik: a visual programming environment. *Acm Sigplan*. (1988), 176–190.
- [57] Jeremiah Blanchard, Christina Gardner-McCune and Lisa Anthony 2015. Bridging Educational Programming and Production Languages. “Every Child a Coder” workshop, *ACM SIGCHI Conference on Interaction Design and Children* (2015).
- [58] Kelleher, C. and Pausch, R. 2005. Lowering the Barriers to Programming : a survey of programming environments and languages for novice programmers. *ACM Computing Surveys*. 37, 2 (2005), 83–137.
- [59] Kelleher, C., Pausch, R. and Kiesler, S. 2007. Storytelling alice motivates middle school girls to learn computer programming. *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '07* (2007), 1455–1464.
- [60] Python PetName: <https://github.com/dustinkirkland/python-petname>. Accessed: 2020-07-24.
- [61] Ko, A.J., Myers, B., Aung, H.H. and others 2004. Six learning barriers in end-user programming systems. *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on* (2004), 199–206.
- [62] Kölling, M. 1999. Teaching Object Orientation with the Blue Environment. *Journal of Object-Oriented Programming*. 12, 2 (1999), 14–23.
- [63] Kölling, M. 2008. Using BlueJ to Introduce Programming. *Reflections on the Teaching of Programming*. Springer-Verlag. 98–115.
- [64] Kölling, M. 2010. The Greenfoot Programming Environment. *ACM Transactions on Computing Education*. 10, 4 (2010).
- [65] Kölling, M., Brown, N. and Altadmri, A. 2017. Frame-Based Editing. *Journal of Visual Languages and Sentient Systems*. 3, 1 (2017), 40–67.
- [66] Kölling, M., Brown, N.C.C. and Altadmri, A. 2015. Frame-Based editing: Easing the transition from blocks to text-Based programming. *Proceedings of the 10th Workshop in Primary and Secondary Computing Education* (2015), 29–38.
- [67] Kölling, M., Quig, B., Patterson, A. and Rosenberg, J. 2003. The BlueJ system and its pedagogy. *Computer Science Education*. 13, 4 (2003), 249–268.
- [68] Kramer, J. 2007. Is abstraction the key to computing? *Communications of the ACM*. 50, 4 (2007), 36–42.
- [69] Leite, W. 2016. *Practical Propensity Score Methods Using R*. Sage, New York, NY.

- [70] Libarkin, J.C. and Anderson, S.W. 2005. Assessment of Learning in Entry-Level Geoscience Courses: Results from the Geoscience Concept Inventory. *Journal of Geoscience Education*. 53, 4 (2005), 394–401.
- [71] Lister, R. 2011. Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer. *Proceedings of the Thirteenth Australasian Computing Education Conference* (2011), 9–18.
- [72] Lister, R. 2011. COMPUTING EDUCATION RESEARCH: Programming, syntax and cognitive load. *ACM Inroads* 2,2 (August 2011).
- [73] Lister, R., Adams, E.S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J.E., Sanders, K., Seppälä, O., Simon, B. and Thomas, L. 2004. A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*. 36, 4 (2004), 119–150.
- [74] Malan, D.J. and Leitner, H.H. 2007. Scratch for budding computer scientists. *ACM SIGCSE Bulletin*. 39, 1 (2007), 223–227.
- [75] Maloney, J., Resnick, M. and Rusk, N. 2010. The Scratch programming language and environment. *ACM Transactions on Computing Education*. 10, 4 (2010), 1–15.
- [76] Maloney, J., Resnick, M., Rusk, N., Peppler, K. and Kafai, Y.B. 2008. Media Designs with Scratch What Urban Youth Can Learn about Programming in a Computer Clubhouse. *Proceedings of the 8th international conference on International conference for the learning sciences* (2008), 81–82.
- [77] Maloney, J., Rusk, N., Burd, L., Silverman, B., Kafai, Y. and Resnick, M. 2004. Scratch: A sneak preview. *Proceedings - Second International Conference on Creating, Connecting and Collaborating Through Computing*. (2004), 104–109.
- [78] Mannila, L., de Raadt, M. and Linda Mannila, M. de R. 2006. An objective comparison of languages for teaching introductory programming. *Baltic Sea '06 Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006* (2006), 32–37.
- [79] Marascuilo, L.A. and Levin, J.R. 1970. Appropriate Post Hoc Comparisons for Interaction and Nested Hypotheses in Analysis of Variance Designs: The Elimination of Type IV Errors. *American Educational Research Journal*. 7, 3 (1970), 397–421.
- [80] Margolis, J. and Fisher, A. 2002. *Unlocking the clubhouse: women in computing*. MIT Press, Cambridge, MA.
- [81] Matsuzawa, Y. 2015. Language Migration in non-CS Introductory Programming through Mutual Language Translation Environment Basic Function and Interface. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. (2015), 185–190.

- [82] Matsuzawa, Y., Ohata, T., Sugiura, M. and Sakai, S. 2015. Language Migration in non-CS Introductory Programming through Mutual Language Translation Environment Basic Function and Interface. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (2015), 185–190.
- [83] Matwin, S. and Pietrzykowski, T. 1985. Prograph: a preliminary report. *Computer Languages*. 10, 2 (1985), 91–126.
- [84] McCracken, M. et al. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*. 33, 4 (2001), 125.
- [85] McIver, L. 2000. The Effect of Programming Language on Error Rates of Novice Programmers. *12th Workshop of the Psychology of Programming Interest Group* (2000), 181–192.
- [86] McIver, L.M. and Conway, D. 1999. GRAIL: A Zeroth Programming Language. *Advanced Research in Computers and Communications in Education New Human Abilities for the Networked Society*. 43–50.
- [87] Mead, J., Gray, S., Hamer, J., James, R., Sorva, J., Clair, C. St. and Thomas, L. 2006. A cognitive approach to identifying measurable milestones for programming skill acquisition. *ACM SIGCSE Bulletin*. 38, 4 (2006), 182–194.
- [88] Meerbaum-Salant, O., Armoni, M. and Ben-Ari, M. 2011. Habits of programming in scratch. *ITiCSE*. (2011), 168–172.
- [89] Meerbaum-Salant, O. 2010. Learning computer science concepts with scratch. *ICER '10: Proceedings of the Sixth international workshop on Computing education research*. (2010), 69–76.
- [90] Miller, G.A. 1956. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*. 63, 2 (1956), 81–97.
- [91] Monig, J., Ohshima, Y. and Maloney, J. 2015. Blocks at your fingertips: Blurring the line between blocks and text in GP. *Proceedings - 2015 IEEE Blocks and Beyond Workshop, Blocks and Beyond 2015* (2015), 51–53.
- [92] Monroy-Hernández, A. and Resnick, M. 2008. Empowering kids to create and share programmable media. *Interactions*. 15, (2008), 50.
- [93] Nachar, N. 2008. The Mann-Whitney U: A Test for Assessing Whether Two Independent Samples Come from the Same Distribution. *Tutorials in Quantitative Methods for Psychology*. 4, 1 (2008), 13–20.
- [94] Nikiforos, S., Kontomaris, C. and Chorianopoulos, K. 2013. MIT Scratch : A Powerful Tool for Improving Teaching of Programming. *Conference on Informatics in Education* (2013), 11–12.



- [95] Paas, F., Tuovinen, J., Tabbers, H. and Van Gerven, P.W.M. 2003. Cognitive Load Measurement as a Means to Advance Cognitive Load Theory. *Educational Psychologist*. 38, 1 (2003), 1–4.
- [96] Papert, S. 1980. *Mindstorms: children, computers, and powerful ideas*. Basic Books, Inc, New York, NY.
- [97] Papert, S. and Harel, I. 1991. Situating Constructionism. *Constructionism*. 1–11.
- [98] Parker, M.C. and Guzdial, M. 2016. Replication, Validation, and Use of a Language Independent CS1 Knowledge Assessment. *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (2016), 93–101.
- [99] Parker, M.C. and Guzdial, M. 2016. Replication, Validation, and Use of a Language Independent CS1 Knowledge Assessment. *Proceedings of the 2016 ACM Conference on International Computing Education Research* (2016), 93–101.
- [100] Parr, T.J. and Quong, R.W. 1995. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience*. 25, 7 (1995), 789–810.
- [101] Partchev, I. 2004. A visual guide to item response theory. <http://www.metheval.uni-jena.de/irt/VisualIRT.pdf>. Accessed: 2020-07-24.
- [102] Partovi, H. and Sahami, M. 2013. The hour of code is coming! *ACM SIGCSE Bulletin*. 45, 4 (2013), 5–5.
- [103] Pattis, R., Roberts, J. and Stehlik, M. 1994. *Karel the robot: a gentle introduction to the art of programming*. Wiley, Hoboken, NJ.
- [104] Pausch, R., T. Burnette, A. C. Capehart, M. Conway, D. Cosgrove, R. DeLine, J. Durbin, R. Gossweiler, S. Koga, and J.W. 1995. A brief architectural overview of Alice, a rapid prototyping system for virtual reality. *IEEE Computer Graphics and Applications*.
- [105] Portelance, D.J., Strawhacker, A.L. and Bers, M.U. 2016. Constructing the ScratchJr programming language in the early childhood classroom. *International Journal of Technology and Design Education*. 26, (2016), 489–504.
- [106] Price, T.W., Brown, N.C.C., Lipovac, D., Barnes, T. and Kölling, M. 2016. Evaluation of a Frame-based Programming Editor. *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. (2016), 33–42.
- [107] Repenning, A. 1993. Agentsheets: A Tool for Building Domain-Oriented Visual Programming Environments. *CHI '93: Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*. (1993), 142–143.
- [108] Repenning, A. and Citrin, W. 1993. Agentsheets: Applying Grid-Based Spatial Reasoning to Human-Computer Interaction. *Proceedings 1993 IEEE Symposium on Visual Languages* (1993), 77–82.

- [109] Repenning, A. and Sumner, T. 1995. Agentsheets: A Medium for Creating Domain-Oriented Visual Languages. *Computer*. 28, 3 (1995), 17–25.
- [110] Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J. a Y., Silverman, B. and Kafai, Y. 2009. Scratch: Programming for All. *Communications of the ACM*. 52, (2009), 60–67.
- [111] Richards, B. 2003. Experiences incorporating Java into the introductory sequence. *Journal of Computing Sciences in Colleges*. 19, 2 (2003), 247–253.
- [112] Smith, D.C., Cypher, A. and Spohrer, J. 1994. KidSim: programming agents without a programming language. *Communications of the ACM*. 37, 7 (1994), 54–67.
- [113] Soloway, E., Adelson, B. and Ehrlich, K. 1988. Knowledge and processes in the comprehension of computer programs. *The nature of expertise*. 129–152.
- [114] Spencer, D. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media, New York, NY.
- [115] Sudol, L.A. and Studer, C. 2010. Analyzing test items: Using Item Response Theory to Validate Assessments. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education - SIGCSE '10*. (2010), 436–440.
- [116] Sudol, L.A. and Studer, C. 2010. Analyzing test items: Using Item Response Theory to Validate Assessments. *Proceedings of the 41st ACM Technical Symposium* (2010), 436–440.
- [117] Suskie, L. 2009. *Assessing Student Learning: A Common Sense Guide (2nd ed)*. Jossey-Bass, San Francisco, CA.
- [118] Sweller, J. 1988. Cognitive Load During Problem Solving : Effects on Learning. *Cognitive Science*. 12, 2 (1988), 257–285.
- [119] Tabet, N., Gedawy, H., Alshikhabobakr, H. and Razak, S. 2016. From Alice to Python . Introducing Text-based Programming in Middle Schools . *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (2016), 124–129.
- [120] Teague, D., Comey, M., Ahadi, A. and Lister, R. 2013. A qualitative think aloud study of the early Neo-Piagetian stages of reasoning in novice programmers. *Conferences in Research and Practice in Information Technology Series* (2013), 87–95.
- [121] Teague, D. and Lister, R. 2014. Programming: reading, writing and reversing. *Proceedings of the 2014 conference on Innovation & technology in computer science education - ITiCSE '14*. (2014), 285–290.

- [122] Tew, A.E. and Guzdial, M. 2011. The FCS1 : A Language Independent Assessment of CS1 Knowledge. *SIGCSE '11 Proceedings of the 42nd ACM technical symposium on Computer science education* (2011), 111–116.
- [123] Tew, A.E. and Guzdial, M. 2010. Developing a validated assessment of fundamental CS1 concepts. *Proceedings of the 41st ACM technical symposium on Computer science education* (2010), 97–101.
- [124] Tharp, A.L. 1982. Selecting the “right” programming language. *Proceedings of the thirteenth SIGCSE technical symposium on Computer science education* (1982), 151–155.
- [125] The Joint Task Force on Computing Curricula - ACM/IEEE-Computer Society 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*.
- [126] Thorndyke, E.L. and Woodworth, R.S. 1901. The influence of improvement in one mental function upon the efficiency of other functions. (I). *Psychological Review*. 8, 3 (1901), 247–261.
- [127] Utting, I., Tew, A.E., McCracken, M., Thomas, L., Bouvier, D., Frye, R., Paterson, J., Caspersen, M., Kolikant, Y.B.-D., Sorva, J. and Wilusz, T. 2013. A Fresh Look at Novice Programmers’ Performance and Their Teachers’ Expectations. *Proceedings of the {ITiCSE} Working Group Reports Conference on Innovation and Technology in Computer Science Education-working Group Reports*. (2013), 15–32.
- [128] Viera, A.J. and Garrett, J.M. 2005. Understanding Interobserver Agreement: The Kappa Statistic. *Family Medicine*. May (2005), 360–363.
- [129] Vilner, T., Zur, E. and Tavor, S. 2011. Integrating greenfoot into CS1 - A Case Study. *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*. (2011), 350.
- [130] Wagner, A., Gray, J., Corley, J. and Wolber, D. 2013. Using app inventor in a K-12 summer camp. *SIGCSE '13 Proceeding of the 44th ACM technical symposium on Computer science education* (2013), 621–626.
- [131] Wang, T.-C., Mei, W.-H., Lin, S.-L., Chiu, S.-K. and Lin, J.M.-C. 2009. Teaching programming concepts to high school students with Alice. *2009 39th IEEE Frontiers in Education Conference*. (Oct. 2009), 1–6.
- [132] Ward, B., Marghitu, D., Bell, T. and Lambert, L. 2010. Teaching computer science concepts in Scratch and Alice. *Journal of Computing Sciences in Colleges*. 26, (2010), 173–180.
- [133] Ward, B., Marghitu, D., Bell, T. and Lambert, L. 2010. Teaching computer science concepts in Scratch and Alice. *Journal of computing Sciences in Colleges*. 26, 2 (2010), 173–180.

- [134] Weintrop, D. 2016. *Modality Matters: Understanding the Effects of Programming Language Representation in High School Computer Science Classrooms*. Ph.D. Dissertation. Northwestern University, Evanston, IL.
- [135] Weintrop, D. and Holbert, N. 2017. From Blocks to Text and Back: Programming Patterns in a Dual-modality Environment. *Proceedings of the 48th ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. (2017).
- [136] Weintrop, D. and Wilensky, U. 2015. To block or not to block? That is the question. *Proceedings of the 14th International Conference on Interaction Design and Children*. (2015), 199–208.
- [137] Wobbrock, J.O., Findlater, L., Gergle, D. and Higgins, J.J. 2011. The Aligned Rank Transform for nonparametric factorial analyses using only ANOVA procedures. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2011), 143–146.
- [138] Xie, B., Nelson, G.L. and Ko, A.J. 2018. An Explicit Strategy to Scaffold Novice Program Tracing. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education - SIGCSE '18* (2018), 344–349.
- [139] Blockly: A library for building visual programming editors: 2016. <https://developers.google.com/blockly/>. Accessed: 2017-10-01.
- [140] Google CS-First: Game Design: <https://www.cs-first.com/course/game-design/video/241>. Accessed: 2016-01-01.
- [141] The Eclipse Foundation: [www.eclipse.org](http://www.eclipse.org). Accessed: 2020-02-03.
- [142] Brython: <https://brython.info/>. Accessed: 2020-02-03.
- [143] Skulpt: <http://www.skulpt.org/>. Accessed: 2020-02-03.
- [144] AP Computer Science Principles: 2016. <https://secure-media.collegeboard.org/digitalServices/pdf/ap/ap-computer-science-principles-course-and-exam-description.pdf>. Accessed: 2017-10-01.
- [145] IntelliJ IDEA: <https://www.jetbrains.com/idea/>. Accessed: 2020-02-03.
- [146] Amphibian Plugin: 2019. <https://github.com/cacticouncil/amphibian>. Accessed: 2020-02-03.
- [147] JxBrowser: <https://www.teamdev.com/jxbrowser>. Accessed: 2020-02-03.

## BIOGRAPHICAL SKETCH

Jeremiah Blanchard is an Assistant Engineer at the University of Florida in the Computer & Information Science & Engineering Department, where he is a full-time faculty member. Previously, he served as Program Director of Game Development at Full Sail University in Winter Park, Florida (in the greater Orlando area), where he worked for 10 years and taught the Artificial Intelligence for Games and Game Networking courses before moving into administration. Before coming to Full Sail, he worked as a freelance game and application developer and lived, worked, and studied in the Osaka region of Japan. His game development experience includes work with the National Flight Academy in Pensacola, Florida, with whom he worked to develop flight simulator scenarios to help teach at-risk middle and high school students mathematics, physics, and history. He's also worked with Design Interactive on CogGauge, a game-based cognitive battery system developed on grant funding from NASA with the intention of testing brain injuries in space.

He began his graduate work at the University of Florida in August 2005 and completed his M.S. in Computer Engineering in May 2007. He worked full time at Full Sail University from January 2007 to January 2017. From January 2017 until May 2017 he taught part-time at the University of Florida, moving to a full-time position in May 2017. His passion for the education and computer science fields drove him to return to complete his PhD in 2014 while working.