

NORTHWESTERN UNIVERSITY

Modality Matters: Understanding the Effects of Programming Language Representation in High
School Computer Science Classrooms

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Learning Sciences

By

David Weintrop

EVANSTON, ILLINOIS

September 2016

Abstract

Computation is changing our world. From how we work, to how we communicate and how we relax - few parts of our world have been left unaffected by computation and the technologies that it enables. The field of computer science and the ideas of the discipline are driving these changes, yet relatively little of it is present in contemporary K-12 education. Numerous local and national initiatives are underway to bring the powerful ideas of computing into classrooms around the world. An increasingly popular strategy being employed in this effort is the use of graphical, blocks-based programming environments like Scratch, Snap! and Alice. While these environments have been found to be effective at broadening participation with younger learners, open questions remain about their suitability in high school contexts. This dissertation uses a quasi-experimental, mixed methods design to understand the effects of blocks-based, text-based, and hybrid blocks-text programming environments in high school classrooms. Three custom-designed programming environments were created and used to understand how modality (blocks-based, text-based, and hybrid blocks/text) affects learners' emerging understandings of core computer science concepts and their attitudes towards and perceptions of the discipline. Additionally, the study investigates if and how the different introductory programming modalities support learners' transitions to more conventional text-based professional programming languages.

Findings from the study reveal that the modality matters. Differences were found with respect to students' conceptualizations of programming constructs as well as student performance on content assessments and attitudinal surveys. The data show students in the Blocks condition scoring higher on content evaluations after a five-week curriculum and reporting higher levels of confidence, enjoyment, and interest in the field relative to students

using an isomorphic text-based interface. However, these findings did not translate to greater success or better self-reported experiences upon transitioning to a professional, text-based programming language. After ten weeks of learning the Java programming language, students in the text condition showed comparable scores on content assessments and positive trends on attitudinal questions, whereas their blocks-based peers showed decreasingly levels of engagement and enjoyment. This study also demonstrates the potential of hybrid environments that blend features of blocks-based and text-based interfaces for providing the scaffolds and engagement of blocks-based tools with the perceived power and authenticity of text-based introductory environments. Collectively this work contributes to our understanding of the relationship between computational representations and learning programming, and can be used to inform the tools that will train the next generation of computationally literate citizens.

Acknowledgements

My wife would often joke that I enjoyed graduate school too much, which is why it took me so long to finish. In many ways, she was right¹. There are many reasons that I had such a positive graduate school experience, but two stand out. The first is my advisor, Uri Wilensky, and the incredible community he has built at the Center for Connected Learning and Computer-based Modeling. Thank you Uri and CCLers past and present, for helping me grow as a thinker and a learner. You have made me the researcher I am, and I am deeply grateful for your help and support. The second largest contributor to my extended stay at Northwestern was my cohort and the larger Northwestern Learning Science community. Thank you for making me excited to come to campus every day² and for challenging me to think deeply about the work I was undertaking.

At the outset of any academic pursuit, you are challenged by the adage: stand on the shoulders of the giants³. With that in mind, it is important for me to acknowledge the visionaries and great thinkers upon whose shoulders I peer out on the world from. Two in particular stand out: Seymour Papert, whose ideas and visions we are still chasing, and Andrea diSessa, for writing the book *Changing Minds*, which set me on the course that led me to the Learning Sciences and everything that followed. For a full list of giants, please refer to page 350.

It is also important to me to acknowledge two other groups of people who made this dissertation possible. First are the talented teachers and brilliant students who participated in my studies, without whom, there would be no dissertation. Secondly, are the talented, resourceful

¹ As usual.

² Or at least almost every day.

³ At least that is what Google Scholar says. If only there were some way for me to look up the origin of that expression, oh well.

and, most importantly, generous developers whose tools made this dissertation possible. In particular, the work of David Bau, Jens Mönig, and John Maloney stand out as their work contributed most directly to the programming environments used in my dissertation.

Finally, this dissertation would never have been possible without my family. To my parents, who always encouraged me to ask questions and supported me in finding answers. To my sisters, for their support and never-ending older-sibling wisdom. To my children, Maya and Jonah, whose arrival precipitated my finishing this document, and quickly replaced it as the central focus of my daily life. And finally, and most importantly, to my wife, who has been by my side offering help, support, and love on every day of this journey. Thank you for helping me become the person I am.

Dedication

This dissertation is for Jonah and Maya. It is for you and your future friends, classmates, and peers that I undertook this challenge and sought to answer the questions I did. My only hope for this work is that it will make some small difference in your lives and the lives of learners of your generation.

Table of Contents

Abstract.....	2
Acknowledgements	4
Dedication	6
Table of Contents	7
List of Figures.....	11
List of Tables	13
1. Introduction.....	14
Research Questions.....	21
Intended Outcomes.....	23
Structure of this Dissertation.....	25
2. Literature Review	27
Computers and Learning.....	27
The Computer Science Education Landscape	31
Learning Context.....	32
The Role of the Computer.....	34
Stand Alone Versus Integrated Computer Science	34
Prioritizing Inclusivity and Broadening Participation.....	35
Physical Computing & Robotics	36
Programming Languages and Environments	37
Programming Paradigm.....	38
Situating This Dissertation in the Larger Landscape	39
Representations and Learning	40
Novice Programming Environments	43
Languages and Environments from constructionist tradition.....	44
Languages and Environments from outside the Constructionist community.....	48
Visual Programming	54
Blocks-based Programming	58
From Blocks-based to Text-based Programming	63
3. Methodology	68
Study Design.....	68
Phase One: A Three-way Introduction to Programming	69
Curriculum	70
Phase Two: The To-Text Transition.....	72
Methods and Data Collection	72
Quantitative Data Sources.....	73
Qualitative Data Sources.....	81
Computational Data Sources.....	83
Data Analysis Approach	87
Setting and Participants	91
4. Design	100
Modality.....	100
Year One – Snappier!.....	102

The Three Versions of Snappier!	103
Findings from Year One with Snappier!	106
Limitations of Snappier!	117
Year Two – Pencil.cc	119
The Three Versions of Pencil.cc	123
Limitations of Pencil.cc	126
5. Attitudes and Perceptions	130
Incoming Perceptions and Initial Reactions to Introductory Environments	130
Assumption of a Text-Driven Experience	131
Why Use Non-Professional, Introductory Programming Environments	132
Perceived Affordances and Limitations of Pencil.cc and the Three Modalities	136
Perceptions of Introductory Programming Environments by Modality	142
Authenticity of the Activity by Modality	143
Learning to Program by Modalities	147
Changes in Attitudes and Perception over Time	150
Confidence in Programming Ability	151
Enjoyment of Programming	154
Programming is Hard	156
Interest in Future CS	158
Discussion	160
Students' Perceptions of Pencil.cc	161
Students' Attitudes Toward Programming	162
How Did the Hybrid Condition Fare?	164
Conclusion	165
6. Conceptual Learning Outcomes	166
Emerging Conceptual Understandings	167
Variables	168
Conditional Logic	171
Iterative Logic	176
Functions	179
The Commutative Assessment	184
Year One Concept by Modality Findings	186
Iterative Logic	188
Conditional Logic questions	190
Variables Questions	191
Function Questions	193
Comprehension Questions	195
Concept By Modality Discussion	196
Learning Outcomes by Condition	197
Condition by Modality	202
Condition by Concept	204
Perceived Ease-of-Use of Concepts by Condition	206
Discussion	209
Modality Matters	210
Blocks versus Text	211
The Case of the Hybrid Condition	214
Conclusion	215

7. Practices and Artifacts	217
Three Vignettes	217
Blocks Condition Vignette	220
Hybrid Condition Vignette	227
Text Condition Vignette.....	234
Programming Practices Across Conditions	240
Running Programs.....	241
Elapsed Time Between Consecutive Runs of Programs.....	245
Characteristics of Programs	248
Blocks-based Usage in the Hybrid Condition.....	250
Quick Reference Usage	252
Discussion	255
Using the Vignettes.....	255
Differences in Programming Practices and Artifacts	260
Conclusion	264
8. Transitioning to Java.....	266
Perceptions of Introductory Programming Environments as a Preparation for Java	267
Helpful Aspects of Introductory Programming Environment for Transitioning to Java	270
Perceptual Outcomes Discussion	276
Changes in Attitudes and Perception in Java	278
Confidence in Programming Ability.....	278
Enjoyment of Programming	281
Programming is Hard	283
Interest in Future CS.....	285
Attitudinal Changes Discussion	286
Differences in Java Programs.....	288
Frequency of Compilations Over Time	288
Types and Frequencies of Java Errors	297
Java Programs Discussion	303
Discussion	305
Conclusion	308
9. Discussion and Conclusion	310
Review of the Program of Research.....	310
Summary of Findings	313
Comparing Blocks and Text Modalities	313
The Case of the Hybrid Modality	319
Implications	324
Implications of Modality on the Learner	324
Implications of Modality on the Teacher	327
Implications of Modality Choice on Schools and Administrators	329
On Modality, Learning, and Design.....	330
Modality Matters	331
Modality is Malleable	333
Modality and Structurations	336
Modality and Milieu	341
Limitations and Future Work	343
Conclusion	348

10. References.....	350
11. Appendix A – Introductory Curriculum	383
Quilt	383
MadLibs.....	385
Tip Calculator	385
Paint by Quadrant	386
Movie Recommendation Engine.....	387
Grade Ranger.....	387
Guessing Game	388
Radial Art.....	389
Squiral.....	390
Polygoner	391
Connect 4	392
Brick Wall	393
Final Project	394
12. Appendix B – Attitudinal Survey	396
Pre Attitudinal Survey	396
Mid Attitudinal Survey	398
Post Attitudinal Survey	399
13. Appendix C – The Commutative Assessment	400
14. Appendix D – Interview Protocols	413
Pre Interview Protocol	413
Mid Interview Protocol	415
Post Interview Protocol.....	415
15. Appendix E – Coding Manuals.....	417
16. Appendix F: Java Compilation Error Parser	421

List of Figures

Figure 1.1. Comparable blocks-based and text-based programs	19
Figure 2.1. Four example blocks-based programming languages	60
Figure 3.1. The Commutative Assessment modalities.....	77
Figure 3.2. Sample Commutative Assessment question.....	79
Figure 3.3. The classroom where the study was conducted.....	94
Figure 3.4. Time spent on a computer outside of school	96
Figure 3.5. Student responses for why they enrolled in the course	98
Figure 4.1. Four examples of programming modalities.....	102
Figure 4.2. The <i>Snap!</i> interface with sections labeled	103
Figure 4.3. The <i>Snappier!</i> read-only text interface	105
Figure 4.4. The Snappier! Block Editor and read-write interface.....	106
Figure 4.5. Student reported differences between <i>Snappier!</i> and Java.....	113
Figure 4.6. Pencil Code's interface.....	120
Figure 4.7. Pencil Code's two modalities: Blocks and Text.....	120
Figure 4.8. Pencil.cc's Quick Reference feature.....	122
Figure 4.9. The Pencil.cc login page.....	123
Figure 4.10. Pencil.cc's hybrid blocks/text interface.....	125
Figure 4.11. Renderings of the same script in <i>Snap!</i> and Pencil.cc.....	127
Figure 5.1. Student reported differences between Pencil.cc and Java (Mid).....	137
Figure 5.2. Student reported differences between Pencil.cc and Java (Post)	140
Figure 5.3. Student responses to: Pencil.cc is similar to what real programmers do.....	144
Figure 5.4. Student responses to: Pencil.cc made me a better programmer	148
Figure 5.5. Students' programming confidence.....	152
Figure 5.6. Students' enjoyment of programming	155
Figure 5.7. Average responses to the Likert statement: Programming is Hard	157
Figure 5.8. Average responses to the Likert statement: I plan to take more computer science courses after this one, grouped by condition	159
Figure 6.1. Students' metaphors of variables	170
Figure 6.2. Students' descriptions of conditional logic	172
Figure 6.3. Students' descriptions of iterative logic	177
Figure 6.4. Students' metaphors of functions.....	181
Figure 6.5. Students' descriptions of functions	182
Figure 6.6. Pencil.cc's function syntax	183
Figure 6.7. The Commutative Assessment modalities.....	185
Figure 6.8. Commutative Assessment performance by modality and concept.....	187
Figure 6.9. A sample Computtative Assessment iterative logic question.....	188
Figure 6.10. Comparing blocks-based and text-based <code>for</code> loops.....	189
Figure 6.11. The Commutative Assessment variable question that students performed better in the text condition than the blocks-based condition.....	192
Figure 6.12. Two sample Commutative Assessment function questions	194
Figure 6.13. Two sample Commutative Assessment comprehension questions	195
Figure 6.14. Student Commutative Assessment scores by condition over time	199
Figure 6.15. Commutative Assessment performance by modality and condition	202

Figure 6.16. Commutative Assessment performance by condition and concept	205
Figure 6.17. Student reported ease-of-use of programming concepts	207
Figure 7.1. Completed solutions of the vignette interview question	220
Figure 7.2. Images of the Blocks condition vignette (1 st program)	222
Figure 7.3. Images of the Blocks condition vignette (2 nd program)	223
Figure 7.4. Images of the Hybrid condition vignette (1 st program)	229
Figure 7.5. Images of the Hybrid condition vignette (2 nd program)	231
Figure 7.6. Images of the Text condition vignette	236
Figure 7.8. The time elapsed between consecutive runs in Pencil.cc	246
Figure 7.9. The average size of programs by condition in Pencil.cc	249
Figure 7.10. The average number of blocks added per assignment.....	251
Figure 7.11. Quick Reference page usage	254
Figure 8.1. Perceived utility of Pencil.cc for learning Java (quantitative)	268
Figure 8.2. Perceived utility of Pencil.cc for learning Java (qualitative) - Mid	271
Figure 8.3. Perceived utility of Pencil.cc for learning Java (qualitative) - Post	274
Figure 8.4. Students' confidence in programming.....	279
Figure 8.5. Students' enjoyment of programming.....	281
Figure 8.6. Students' responses to Programming is Fun and perceived excitement	282
Figure 8.7. Students' responses to the Likert statement: Programming is Hard	284
Figure 8.8. Students' responses to the Likert statement: I plan to take more computer science courses after this one, grouped by condition	285
Figure 8.9. Compilations of Java programs by week.....	289
Figure 8.10. Compilations of Java programs by day	291
Figure 8.11. Successful compilations by week.....	292
Figure 8.12. Failing compilations by week.....	295
Figure 8.13. Percentage of syntactically correct compilations	296
Figure 8.14. The ten most frequently encountered Java errors.....	300
Figure 9.1. The three environments used in the study	313
Figure 9.2. The same concept in four modalities.....	334

List of Tables

Table 3.1. The 5-week introductory curriculum	71
Table 3.2. The 13 assignments of the introductory curriculum	71
Table 3.3. The concepts covered in the Commutative Assessment.....	78
Table 3.4. The 35 student interviews.....	82
Table 3.5. Log events captured in Pencil.cc	85
Table 3.6. Log data capatured in Pencil.cc	85
Table 3.7. Data captured by JavaSeer.....	87
Table 3.8. Student responses to why they decided to enroll in the class.....	97
Table 4.1. Mappings between <i>Snap!</i> and JavaScript	104
Table 4.2. Student responses comparing <i>Snappier!</i> and Java (quantitative)	107
Table 4.3. Student responses comparing <i>Snappier!</i> and Java (qualitative)	113
Table 6.1. Sample responses of function metaphors used by students	180
Table 7.1. Frequency of data collection events in Pencil.cc	241
Table 7.2. Run events collected for each assignment	242
Table 7.3. Frequency of consecutive runs in under five seconds	247
Table 8.1. Pencil.cc concepts perceived as useful for Java	275
Table 8.2. Successful compilations by Levenshtein distance	293
Table 8.3. High-level descriptive patterns of failing compilations and errors.....	298

1. Introduction

Computation is changing our world. From how we communicate and make decisions, to how we relax and how we shop - few aspects of our lives have been left unaffected by the long reach of computation and the technologies that it enables. Smartphones, tablets, and laptops have become the lenses through which we see, organize, and interpret the world. As such, for young learners growing up in this technological landscape, being able to recognize the capabilities and limitations of these technologies and, most critically, to be able to contribute in this technological culture is essential. Programming is the skill that enables this participation. Programming, and the critical thinking and problem solving skills that accompany it, constitute a new 21st century literacy that will need to live alongside reading, writing, and mathematics as essential competencies to empower today's students to fully engage with our technological world. These skills have far reaching benefits as they underpin and enable new forms of creative expression, support learning in diverse computational contexts across a wide range of disciplines, and provide the foundation for future careers in our increasing computationally driven economy. The importance of these skills has been documented by a number of federal agency and industry organizations. The Bureau of Labor Statistics estimates that 135,000 new computing jobs are created every year in the technology sector. Similar growth of computing jobs is projected in other fields; by 2020, one in every two jobs in the STEM disciplines will be in computing (ACM Education Policy Committee, 2014).

Despite this momentous shift happening in our world and the far-reaching benefits that accompany learning to program, very little programming education can be seen in today's schools. Computer science, the field that is driving this computational revolution, is rarely present in K-12 education. Only an estimated 10% of schools offer programming or computer

science courses (Code.org, 2014), and in schools where computer science is present, courses are often taught in ways disconnected from the computational lives of today's students, failing to instill the sense of relevance and feelings of empowerment that can and should accompany such learning experiences. Further, the students who have the opportunity to pursue programming do not reflect the racial and gender distribution of the larger population. In 2013, 14.6% of bachelor's degrees in computer science and related fields were granted to female students, with 4.5% of the graduates being African American, and 6.5% being Hispanic (Zweben & Bizot, 2014). This disturbing trend is mirrored at the high school level, where only 18.6% of students who took the 2013 AP Computer science exam were female, while 8.2% of test takers were Hispanic, and only 3.7% were African American. Research into the cause of these low numbers has identified numerous causes, including limited access to courses, a lack of support for students who express interest in the field, and cultural issues that make underrepresented populations feel unwelcome (Margolis, 2008; Margolis & Fisher, 2003).

While recent attention has focused on computer science and programming, that is just one of the many ways the computers interact with learning. Since the emergence and recognition of computers as 'Protean' devices with widespread applications major scholars have seen great potential in their use for learning and education (A Kay & Goldberg, 1977; Papert, 1980; Perlis, 1962; Suppes, 1966). The role computers were to play amongst these and other early advocates of computers as tools for education differed. Some were excited by discipline of computer science itself, while others saw it having much wider potential, spreading across disciplinary boundaries. Some focused on the power of programming as a learning activity, including the Constructionist learning approach, while others focused not on the practice of programming but the tools and environments that could be built through programming, such as cognitive tutors and

other forms of computer-aided instruction. Others viewed computers as supports for standardized content-based instruction while others viewed computers as a medium for supporting personal expression and fostering creativity, art, and the creation of personally meaningful artifacts. Others still argued for learning computer science ideas is broadly applicable both when working on a computer, as well as for being more efficient and success at non-computer-based activities. Collectively, these various perspectives on the role of computation in education and their work, along with others in the decades since, have produced a plethora of technologies, curricula, pedagogies, and learning experiences designed to leverage the power of computers. Having recognizing the larger picture of the intersection of computers and computational technologies and learning, we bring the discussion back to programming and computer science, the focus of this dissertation. A longer discussion of this broader computer and learning landscape will be discussed in the next chapter, along with situating the work being presented within this larger orientation.

Numerous local and national efforts are underway to address the lack of computational learning opportunities for both underrepresented minorities and the student body at large. These efforts are utilizing innovative materials, engaging pedagogy, and new tools and environments for students to learn the powerful ideas of computing. This includes initiatives and learning environments designed for informal settings like computer clubhouses (Kafai, Peppler, & Chapman, 2009; Resnick & Rusk, 1996) and game-based learning environments (Berland & Lee, 2011; Holbert & Wilensky, 2011; T. Y. Lee, Mauriello, Ahn, & Bederson, 2014; Weintrop & Wilensky, 2014a), as well as more formal school-based contexts. Within the initiatives designed to teach computer science in formal contexts, a number of strategies are being used, including creating new Advanced Placement (AP) courses that can be adopted nationally (Astrachan &

Briggs, 2012) and designing new and engaging computer science courses organized around more appealing topics like media-based computing (Guzdial & Ericson, 2009) and game design (Papastergiou, 2009). There is also a thrust of work exploring and studying the approach of integrating computing and computational thinking across the K-12 curriculum (I. Lee, Martin, & Apone, 2014; Settle, Goldberg, & Barr, 2013; Weintrop, Beheshti, Horn, Orton, Jona, Trouille, & Wilensky, 2016). Other computing outreach programs are looking at moving away from the screen towards physical computing (Brady, Weintrop, Gracey, Anton, & Wilensky, 2015; M. S Horn, Solovey, Crouser, & Jacob, 2009; Jamieson, 2010), Making (P Blikstein, 2013; Eisenberg, 2003; Vossoughi & Bevan, 2014), robotics (Fagin & Merkle, 2003; T. R. Flowers & Gossett, 2002; Goldman, Eguchi, & Sklar, 2004), and even fully off-line curricula like the CS Unplugged design approach (Nishida et al., 2009). Central to many of these initiatives is the use of new, more inviting and accessible approaches to programming that emphasize personal expression, foreground ease-of-use, align with current youth culture, and draw on prior student knowledge and values.

While there is need for research looking across the full spectrum of computing learning opportunities mentioned above, this dissertation focuses on formal high school computer science classrooms as the learning context of interest. This decision is motivated by a number of factors, including the growing importance of computational thinking and computer science in society, the ability to reach learners in meaningful ways in formal classrooms settings, and as an attempt to inform the increasing number of efforts to bring computer science coursework into high schools happening at the local and national scale. These efforts includes Chicago's CS4All initiative that is making computer science a graduation requirement and the New York City school districts plans to bring computer science into every school across the city over the next ten years. These

and other large-scale initiatives are being implemented in the midst of a blossoming of new approaches to programming and the development of new and engaging programming tools. Given this co-occurrence, it is not surprising that many of the new curricula that are being designed and widely adopted rely on recently designed programming tools that have little empirical evidence with respect to their effectiveness for high school aged learners in formal settings. As such, there is great need and opportunity for investigating the effectiveness of design features of the current generation of introductory programming tools in formal settings. This dissertation will answer three sets of interrelated research questions all of which address different facets of the guiding question: How best can we design high school computer science learning environments to educate the next generation of computationally literate citizens?

The first set of questions investigates the relationship between the modality students use while learning to program and the resulting attitudinal and conceptual outcomes. By modality we mean to capture the representational infrastructure used to depict the program, as well as the various forms of composition supported by the representation and the affordances it provides for each. Understanding the impact of modality is important because, as previously mentioned, new environments for teaching programming are emerging and becoming increasingly used in formal educational settings, but we lack a clear understanding of the relationship between these new tools and the resulting conceptual gains, attitudinal outcomes, and programming practices they promote. Prominent among the features of these new environments is the introduction of graphical, blocks-based interfaces (Figure 1.1) that allow learners to use only a mouse to drag-and-drop commands together to form functioning programming. Led by the popularity of environments such as Scratch (Resnick et al., 2009), Alice (Cooper, Dann, & Pausch, 2000), and Pencil Code (Bau, Bau, Dawson, & Pickens, 2015), a growing number of formal curricula are

now utilizing blocks-based programming. Examples include the Exploring Computer Science (Goode, Chapman, & Margolis, 2012), at least two curricula being designed for the AP CS Principles Course (Astrachan & Briggs, 2012; Garcia, Harvey, & Barnes, 2015), and many of the curricular materials being developed and disseminated by Code.org (*Code.org Curricula*, 2013). Despite its growing popularity, open questions remain surrounding the effectiveness of blocks-based programming for helping high school aged students learn basic programming concepts and the overall effectiveness of the approach for preparing learners for future computer science learning opportunities that rely on text-based languages. Research towards this end has identified that representational tools greatly affect the learning process and outcomes (diSessa, 2000; Green & Petre, 1996; Sherin, 2001; Wilensky & Papert, 2010), but little work has been done on the current generation of programming environments with respect to these questions.

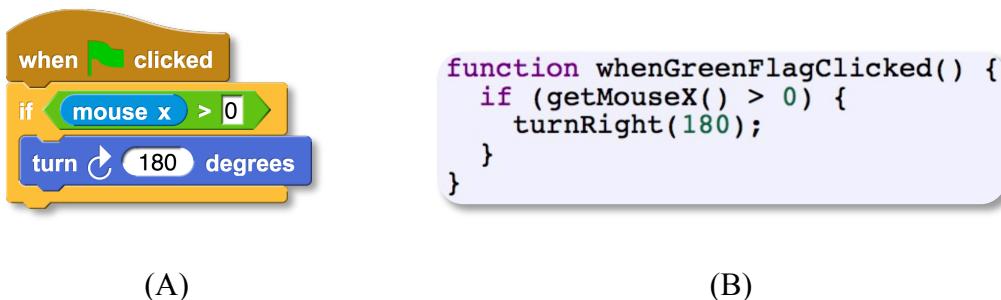


Figure 1.1. Comparable blocks-based (A) and text-based (B) programs

The second set of research questions looks at the suitability of these new introductory programming approaches for preparing learners for future computer science learning opportunities. Research is emerging that suggests that blocks-based programming environments, while successful in changing attitudes and engaging learners, do not adequately prepare them to transition to more conventional programming languages, thus imposing an artificial ceiling on

how far learners can progress with these tools (Cliburn, 2008; Garlick & Cankaya, 2010; Parsons & Haden, 2007; Powers et al., 2007). This finding is consequential as it calls into question the utility of such introductory tools in formal learning contexts in the first place. The work done to date has largely provided descriptive accounts of learners failing to transfer knowledge and practices from introductory environments to more sophisticated, powerful tools. This dissertation will contribute detailed accounts of students transitioning from introductory to professional programming environments, and provide mechanistic, theoretically sound cognitive explanations of how and why gains made in introductory environments do or do not transfer to more sophisticated programming tools.

The third set of research questions surround the evaluation of a new hybrid introductory programming environment that was designed and constructed as part of this dissertation. The new environment is intended to blend the strengths of various existing programming tools in an effort to create a tool that provides the low-threshold to entry and high level of engagement of existing introductory approaches, with the high-ceiling and powerful expressivity of more fully featured programming tools. Based on findings from the first two sets of questions, the newly designed hybrid environment will serve as an empirically grounded example that can be used to evaluate whether or not it is possible to blend the strengths of introductory and professional environments. In creating and evaluating a hybrid environment, this work will serve as one possible example of what a blended environment can look like and be evaluated alongside complementary introductory and professional analogous environments.

This dissertation is built around a three-condition, quasi-experimental study comparing three isomorphic introductory programming tools – two environments are exemplars of common modalities currently used in introductory programming contexts, and the third is the newly

developed hybrid environment developed as part of the third research question. The term isomorphic used here and throughout the dissertation is meant to capture the fact that the capabilities and expressive power of the environments are equivalent; anything that can be done in one environment can also be achieved in the other two. Justification for this isomorphism will be made in Chapter 4, which focuses on the design of the programming environments. The study took place over 15 weeks in three sections of the same class at a diverse urban public high school. This comparative study design in an ecologically valid setting makes this work one of the “rare” studies investigating actual learning benefits in a scientifically rigorous way (Kölling & McKay, 2016). Beginning on the first day of school, students spent five weeks working through a custom designed curriculum using one of the three introductory programming environments (the three conditions of the study). At the conclusion of the fifth week of school, all three classes transitioned to the Java programming language and followed the same curriculum for the remainder of the year. This study design allows for a direct, side-by-side comparison of the three introductory environments, as well as, providing data to answer questions about their suitability for preparing students for future learning with more conventional, professional programming languages. The study uses a mixed-methods approach and will include qualitative, quantitative, and computational data collection and analysis techniques. During the 15-week study, a variety of data were collected including weekly classroom observations, one-on-one student interviews, automated collection of student-authored programs, and pre, mid and post content assessments and attitudinal surveys. Collectively these data were used to answer the research questions being pursued.

Research Questions

As stated earlier, this dissertation seeks to answer three sets of interrelated research questions. The first two are empirical questions on the relationship between the modality used to introduce learners to programming and the conceptual understandings, programming practices, and attitudes and confidence they engender. The third is a design question exploring and evaluating programming language, environment, and modality designs. The three sets of research questions are:

1. (a) For text-based, blocks-based, and hybrid blocks/text programming tools, what is the relationship between the programming modality used and learners' perceptions of programming with respect to confidence, authenticity, enjoyment, and to their larger attitudes towards the field of computer science? (b) How does the representational infrastructure used affect learners' emerging understandings of programming concepts?
c) What programming practices do learners develop when working in each of these three modalities? And for each of these questions, how do the answers differ across blocks-based, text-based, and hybrid blocks/text environments?
2. (a) How do understandings and practices developed while working in different introductory programming modalities support or hinder the transition to conventional text-based programming languages? (b) How do learners' understanding of and attitudes towards programming change as learners shift from introductory environments to more widely used, professional programming languages? How is this different between text-based, blocks-based and hybrid blocks/text introductory modalities?
3. Can we design hybrid introductory programming environments that blend features of blocks-based and text-based programming that effectively introduce novices to programming and computer science more broadly? How does such an environment

perform relative to blocks-based and text-based programming tools with respect to conceptual understanding, development of productive programming practices, and attitudinal, motivational, and engagement outcomes for learners?

The first set of research questions explores what effects blocks-based, text-based and hybrid blocks/text representational systems have on learners while they are learning with them. This includes how each tool influences learners' emerging conceptual understanding, the programming practices they develop with the tool, and how the tool affects learners' attitudes and perceptions. Each of these questions will be analyzed using isomorphic blocks-based, text-based, and hybrid blocks/text tools. The second set of questions explores the suitability of each of these modalities used in an introductory programming context for preparing learners for future text-based programming learning experiences. Answering these questions involves looking at students' perceptions of the representation when used in introductory programming environments and follows them as they move from them to conventional text-based languages. Like with the first set of questions, our data sources for these questions will be gathered from students' work with blocks-based, text-based, and hybrid blocks/text introductory tools. The final research question is a design question intended to explore ways to draw on the strengths of both blocks-based and text-based programming to see if it is possible to effectively create tools that blend the two modalities.

Intended Outcomes

There are three overarching goals for this dissertation, all with an eye towards taking the findings of this work to make positive changes in classrooms and computer science learning opportunities around the world. The first is to better understand the relationship between

representations of programming concepts and learners' emerging understandings, practices and attitudes towards programming in the early stages of learning to program. The first research question is designed to achieve this goal. The second goal for this dissertation is to produce evidence-based recommendations on features to look for and features to avoid when choosing introductory programming environments and languages based on their effects on students learning as well as the suitability for preparing students for future programming and computer science learning opportunities. With this goal, the hope is to make use of the findings from the first two research questions to provide guidance to computer science educators who are eager for empirically grounded recommendations. The final goal for this dissertation is to provide a proof-of-concept introductory programming environment that blends the strengths of textual languages with the affordances of the blocks-based programming approach to create a potentially powerful new interface for novice programmers. The hypothesis here is that it is possible to pair the low-threshold aspects of graphical programming tools that research has identified with the high-ceiling, text-based programming approach used in higher education and professional contexts. Having created this environment, and should the research bear out that such a hybrid interface is capable of achieving this balance, the dissertation will contribute a new, evidence-backed approach to teaching beginner to program.

We are at a critical juncture in the history of computer science education in this country. The ability to program is a central skill all students should develop, but it is currently absent from the coursework of today's students. To address this gap, educators, school administrators, and state and national legislators are all taking action to bring computer science into the classroom. The practices, tools, and curricula that are being developed today, will become the standards used for years to come. Therefore, it is critical that we are confident that the curricula

and environments that we advocate for today are effective at teaching the core concepts, engaging learners from diverse backgrounds, and successful in preparing students for the computational endeavors they will face in the future. The findings from this dissertation will advance our understanding of how best to introduce students to these core 21st century skills and contribute new tools that will prepare students to be successful in the computational futures that await them.

Structure of this Dissertation

The remainder of this dissertation is broken down into 5 main sections. The first of these sections is a comprehensive literature review that covers the history of programming languages and environments designed for learners and prior work on outcomes of using various introductory programming environments with learners. Care is taken in this chapter to lay the theoretical and empirical foundation for the questions being pursued in this dissertation as well as identifying the gaps in the literature this study is addressing.

The next section, chapter three, presents the methodological approach used in this study. This includes the study design, the instruments and procedures used, a description of the various data collected, and information about the participants of the study and the setting in which the work was conducted. The third section (chapter four) of the dissertation is an extension of the methods section focusing specifically on the design of the three programming environments that lie at the heart of this work. In this chapter, the three environments used in the first year of the study are described with a brief analysis of the outcomes from this first year. In particular, this analysis focuses on what was learned in the first year and how it informed the design of the three versions of the programming environment used in year two of the study.

The fourth section, which covers four chapters (chapters six – eight), is the analysis and findings from the study. Each of these chapters addresses a separate part of the stated research questions. Chapter five focuses on students' attitudes and perceptions of programming and how modality differentially influenced these aspects of the learners (RQ 1a). The next chapter, chapter six, looks at conceptual understanding by modality (RQ 1b), specifically trying to understand the role of modality in facilitating learners' emerging understandings of foundational programming concepts. Chapter seven focuses specifically on the differential practices that form across the three variations of the introductory environment (RQ 1c). The fourth and final analysis chapter (chapter eight) looks at if and how gains, both attitudinal and conceptual, made in through the use of the introductory programming environments carry over to the Java programming language (RQ 2).

The ninth and final chapter of this dissertation constitutes the summative discussion and conclusions. In this chapter, the findings are summarized and contributions of this work are discussed in greater detail, synthesizing what was learned across the various analyses that were conducted. As part of this section, the third research question, trying to understand the affordances and drawbacks of the newly developed hybrid programming interface are discussed. The dissertation concludes with limitations of this study, future work that is still to be done, and potential implications of this work.

2. Literature Review

A large body of research has informed this dissertation including work on the challenges beginners face when learning programming concepts, the design of languages and environments for novice programmers, and research on the relationship between representations and the understandings they engender. Before diving into the literature most directly tied to this dissertation, this chapter begins with a discussion of the historical relationship between computer and learning and a high-level mapping of different approaches taken for teaching computer science, discussing various dimensions along which the challenge of teaching programming has been approached. From there we continue with a review of literature on the relationship between representations and learning as it is this relationship that underpins the discussion of the design of novice programming languages and environments and studies evaluating their strengths and weaknesses that follows. We then review the history of the design of programming environments for novices, giving special attention to the constructionist tradition from which the environments used in the study emerged. From there, we broaden our lens to look at the larger class of visual programming tools and empirical work that has been done evaluating them, with a particular emphasis on the blocks-based programming paradigm. We conclude the chapter by reviewing work that looks at the relationship between blocks-based and text-based programming and research on students transitioning across those environments.

Computers and Learning

Early on in the history of digital computers, their utility for learning was recognized. The earliest advocates for computers as tools for learning came from university faculty members with experience working with computers, teaching students, and access to the computers of the day.

Early advocates saw different pedagogical uses for these new digital machines. In the early 1960s, Alan Perlis, the director of the Computation Center at what was then called the Carnegie Institute of Technology⁴, argued for the inclusion of programming in the curriculum for all learners as part of as working with computers would help develop well-rounded students who would be ready for whatever challenges the world put before them (Perlis, 1962). Perlis envisioned students writing programs to solve real world problems, in doing so, he argued they would improve their abilities to abstract, organize, plan, and use information from diffuse and abstract environments. While Perlis was thinking about programming as a pedagogical strategy at the undergraduate level, others saw the potential of learning as a powerful learning strategy for younger learners. Papert and his colleagues at MIT and BBN Labs developed the Logo programming language as a way to allow younger learners to engage in programming and the various metacognitive practices that accompany it (Feurzeig, Papert, Bloom, Grant, & Solomon, 1969; Papert, 1980). In foregrounding the act of authorship and construction of programming, this type of learning experience, which Papert called “Constructionism”, also granted autonomy to learners, using the computer as a medium for creativity and personal expression. The view of the computer as an expressive and powerful medium for learning has shaped decades of designs of computational learning environments (diSessa & Abelson, 1986; A Kay & Goldberg, 1977; Resnick et al., 2009; Wilensky, 1999). The type of learning enabled by computers and advocated by Papert and colleagues went beyond just the act of programming to include other aspects of the potential of computer for learning, which will be returned to below.

Other early proponents of computers as tools for learning saw their utility in creating computer-aided instruction. This support could take a number of forms. One argument was that

⁴ Later renamed Carnegie Melon University.

computers could use their “information-processing capacities [to] adapt mechanical teaching routines to the needs and the past performance of the individual student” (Suppes, 1966, p. 207). This thinking led to the development of intelligent tutoring systems, which are software packages that are designed to replicate the personalized and customized supports that human tutors can provide (Derek Sleeman & Brown, 1982). Later iterations of these types of computer-aided instruction systems integrated findings from cognitive science, creating cognitive tutors, that used computational models of cognition to try and diagnose learners misconceptions and provide customized feedback and carefully curated questions to facilitate the learning process (Anderson, Corbett, Koedinger, & Pelletier, 1995). In this role, the computers are providing instructional supports and thus serving a much different role than the one previously mentioned that saw the act of programming as a the central learning activity. Papert summarized the extreme versions of these two approaches thusly: in computer-aided instruction “the computer is being used to program the child” whereas in Constructionism, “the child programs the computer” (Papert, 1980, p. 5). It is important to mention both of these approaches to computers in education have been shown to support positive learning outcomes.

As the importance of computing and computational technologies in society has grown, so too have the ways computers have been brought into educational spaces and the arguments made for what should be taught with respect to computers and why. With the rise in importance for learners to be comfortable with keyboards, computers and standard software packages (notably the Microsoft Office Suite), one thread of computing education has looked at what is often called “Computer Literacy.” This is meant to connate a basic familiarity with computers and was once a focus of vocational and high school classes, but has since fallen in prevalence in K-12

classrooms given the increased presence of computers outside of classrooms. This dissertation is not concerned with this portion of the computers and education landscape.

A second more recent case made for bringing computers into education is the idea that computers and ideas from computer science can help deepen learners' understandings of content beyond computer science. This idea began with Logo, which was designed as a language to allow learners to engage with powerful mathematical ideas (Feurzeig et al., 1969; Papert, 1972, 1980). Other examples followed, showing computation and programming to be a powerful context to engage learners with ideas ranging from physics (Sherin, diSessa, & Hammer, 1993), to complex systems (Wilensky, 2001), to language and grammar (Goldenberg & Feurzeig, 1987).

Taking this view one step further, it has been argued that computers can serve as a medium for exploring new ideas and fields (Wolfram, 2002) as well as be tools for creating new types of representations with which to express ideas and explore and understand aspects of our world (Wilensky & Papert, 2010). In this view, computers and the ideas and skills from computer science can serve as a foundational new literacy to express and communicate ideas (diSessa, 2000). In this framing, the case for bringing computers into formal education spaces goes beyond the improving of learning and instruction of existing subjects to now include teaching ideas and skills that otherwise would not be possible.

In the last decade, a growing number of people have been making the argument that the ideas from computer science are broadly useful across diverse setting, both on and away from a computer. Collected under the umbrella term "Computational Thinking", it has (and is) being argued that these skills constitute core 21st century skills and deserves a place alongside reading, writing, and arithmetic as essential content that all learners should be taught (S. Grover & Pea, 2013; Wing, 2006). While this movement has been successful in gaining momentum, attention,

and commitment from decision makers at various levels of education and government, the concept itself remains under-specified, resulting in many looking to the other literature mentioned in this chapter for guidance.

The last major intersection of computer and education, and the focus of this dissertation, is formal instruction in the field of computer science. The case for the importance of teaching students computer science has taken a number of forms in recent years, including economic motivation and job opportunities, reasons of equity and empowerment, and responding to the growing prevalence of computers and technology in society. These arguments are on top of those listed in previous sections, as in many cases; formal computer science education accomplishes the goals laid out by other motivations and approaches of bringing computers and education together. It is important to note there are still challenges associated with teaching computer science in formal contexts. Some of which are infrastructural (like a lack of qualified teachers and school resources) while others stem from the quickly changing nature of the disciplines and the large open questions that remain with respect to how best to teach computer science to all students. It is these last issues that this dissertation is seeking to contribute to solving. Having laid out the high-level relationship between computers and education, this literature review now begins to narrow its focus, first looking at different dimensions of computer science education and then further narrowing its focus to cover the literature most directly tied to the questions being answered in this work.

The Computer Science Education Landscape

Since the emergence of the field of computer science, there has been work looking at how best to introduce learners to the discipline. Given the growing scope and nature of the field, new and diverse approaches are constantly being developed while tried-and-true strategies are being

reimagined and reconstructed with new technologies. In deciding how to teach computer science, or computing more broadly, there are a number of decisions that need to be made. These decisions exist along various dimensions of the space associated with learning the discipline. Dimensions include whether instruction is going to be online or offline, in a formal learning context (i.e. classroom) or an informal space, whether or not to prioritize inclusivity and the goal of broadening participation, if computer science is going to be a stand alone topic or integrated with other disciplines, deciding which pedagogical strategy should be taken, selecting a programming language and environment to be used, whether to use physical or virtual learning tools, and choosing the programming paradigm that will be used (turtle graphics, object oriented, functional, etc.). Any given designed learning experience makes decisions along most (or all) of these dimensions, either explicitly or implicitly, and thus, when talking about a given approach to learning computer science these various dimensions factor into the discussion. To start this literature review, we briefly discuss each of these dimensions and highlight important work in that space. At the conclusion of this section, we situate the approaching taken in this dissertation, taking care to be clear about what is held constant and what is being varied and investigated. Throughout this high-level review, when encountering work that is closely tied to this dissertation, reference is made to later portions of this literature review chapter where the ideas or approach are more systematically reviewed. Before presenting the landscape, it is important to note that any given learning experience is a mix of these various dimensions. In some cases specific features (like the language or goals of the activities) are foregrounded, but decision are made with respect to all of the dimensions discussed below.

Learning Context

As technology pervades our daily lives, there are growing opportunities to have learners engage with ideas from computer science across diverse contexts both in formal and informal contexts. A growing body of work is looking at ways to teach computing in informal settings, be they at home or in shared communal spaces like libraries or museums. At the same time, new initiatives to bringing computer science into formal spaces are also underway. In many ways, these two approaches complement each other and have a different goals and contain a different set of features that can draw diverse populations of learners into the field.

In the formal space, long standing courses like AP Computer Science are being supplemented with new curricula like the AP Computer Science Principles course (Astrachan & Briggs, 2012) and other curricula leveraging new technologies and programming environment like the Exploring Computer Science (Goode et al., 2012) project and a Taste of Computing course (Reed, Wilkerson, Yanek, Dettori, & Solin, 2015). Formal learning opportunities provide infrastructure, access to teachers and extended amounts of time for learners to engage with core ideas in computing, but are often limited by resources available to schools and other constraints that come from being situated inside existing educational infrastructure.

Informal learning environments provide a much more open canvas upon which to create learning opportunities and allow for types of engagement that are not possible with the constraints of schools. Projects like the Computer Clubhouse initiative (Kafai et al., 2009; Resnick & Rusk, 1996) give learners an open space for learners to pursue projects of their own interest, grant greater flexibility to learners in terms of the types of activities they engage in, and are not subject to the same constraints or expectations that accompany school-based learning. Other after school projects offer more structure, but still take advantage of the freedom that accompanies informal learning, such as the FUSE project (Jona, Penney, & Stevens, 2015).

Other efforts look at contexts such as video games (Holbert & Wilensky, 2011; T. Y. Lee et al., 2014; Weintrop & Wilensky, 2014a), board games (Berland & Lee, 2011; M.S. Horn, Weintrop, Beheshti, & Olson, 2012), and online communities (Fields, Giang, & Kafai, 2014; Resnick et al., 2009) as informal contexts to engage learning in computer science. Museums offer another out-of-school space for introducing learners to computer science (M. S Horn et al., 2009; M. S Horn, Weintrop, & Routman, 2014). Again, these approaches bring the ideas of computing in to the lives and practices of the learners they are trying to reach. It is important to note there have been initiatives to bring productive aspects of informal learning into formal spaces with varying degrees of success (M.S. Horn & Jacob, 2007; Malan & Leitner, 2007; Squire, 2005).

The Role of the Computer

While one might initially assume that learning about computer science requires the use of a computer, a growing body of work is showing that learners can engage with core ideas from the field without sitting in front of a screen. This approach is especially productive when working with younger learners, where tasks like using a mouse or typing in commands are not trivial. The growing set of activities in the Computer Science Unplugged catalog serve as an exemplar of what it looks like to learn computer science away from the computer. Other non-computer based computer science learning work includes board games (Berland & Lee, 2011), sticker books (Michael S. Horn, AlSulaiman, & Koh, 2013), and embodied motion (often called “playing turtle”) (Papert, 1980) as ways to allow learners to explore computer science away from a computer.

Stand Alone Versus Integrated Computer Science

Another dimension along with computer science educational work has differed is on its relation to other fields and courses. Traditionally, computer science has been treated as a stand-alone course akin to a mathematics or foreign language class. An alternative to this approach that is growing in popularity is to bring computer science, or computational thinking more broadly, into other courses including mathematics, sciences, and the arts (I. Lee et al., 2014; Settle et al., 2013; Weintrop et al., 2016). A number of arguments have been made in advocating for this approach including addressing issues of a lack of teachers and resources for stand-alone computing course, pedagogical and conceptual advantages to blending computing with other disciplines, providing a more realistic perspective of increasingly computational fields like biology and chemistry, and finally, using other disciplines as a meaningful context in which to situating learning core ideas from computer science. Another, related, form of this approach comes through the use of computational modeling as a way to blend content (often science related, but not always), with central ideas of computer science and computational thinking (I. Lee et al., 2011; Repenning, Ioannidou, & Zola, 2000; Stonedahl, Wilkerson-Jerde, & Wilensky, 2010; Wilensky, 2001; Wilensky, Brady, & Horn, 2014).

Prioritizing Inclusivity and Broadening Participation

The field of computer science has historically struggled with both gender and racial diversity. The most recent numbers from the annual Taulbee Survey that tracks enrollment in computer science related disciplines find that only 14.5% of bachelor's degrees awarded in 2013 went to women, while 6.5% went to Hispanic students and only 4.5% were to Black or African American students (Zweben & Bizot, 2014). The male-dominated nature of the computing field and the culture that has emerged are well documented and many interventions have been proposed to try to address it (American Association of University Women, 1994; Fisher,

Margolis, & Miller, 1997; Margolis & Fisher, 2003). Similarly that lack of racial diversity in the field has also been the focus of much scholarship (Margolis, 2008). To address these issues of underrepresentation, a growing number of tools, curricula and initiatives specifically designed to attract and engage learners from these underrepresented populations have been developed.

A number of strategies for accomplishing this have been pursued. One avenue is the creation of activities and larger curricula that draw on areas of interest and cultural relevance (Bruckman, Jensen, & DeBonte, 2002; DiSalvo, Guzdial, Bruckman, & McKlin, 2014). A related approach that has found success and been employed in a number of projects leverages the practice of storytelling (Burke & Kafai, 2010; Papadimitriou, 2003). One successful tool that builds of storytelling is Storytelling Alice. As the name suggests, Storytelling Alice is a version of Alice, a widely used graphical programming tool, that has been altered to support storytelling as its central activity. Studies comparing Storytelling Alice to conventional Alice (which lacks some storytelling support features) found that girls using Storytelling Alice were more motivated to program and spent longer working on their programming projects (Kelleher, Pausch, & Kiesler, 2007).

Another avenue for promoting inclusivity in computer science education has been the creation of curricula designed to directly confront existing stereotypes. One version of this approach takes the form of courses that paint a richer, more diverse (and accurate) view of what computing is (i.e. it's more than just programming). The Exploring Computer Science (ECS) course is one notable example of this strategy. The creators of the ECS curriculum took care to build their course and train their teachers to emphasize “inquiry, culturally relevant curriculum, and equity-oriented pedagogy” (Ryoo, Margolis, Lee, Sandoval, & Goode, 2013, p. 1).

Physical Computing & Robotics

While introductory programming and computer science learning activities have historically taken place in the virtual realm with programming featured prominently, a growing number of toolkits and technologies are making physical computing another avenue for learning about computing. There is a long history of technologically enhanced physical devices serving as contexts for meaningful computational learning experiences (Paulo Blikstein, 2013; Eisenberg, 2003; McNerney, 2004). A growing ecosystem of microcontroller, like the Arduino (Jamieson, 2010), GoGo board (Arnan Sipitakiat, Blikstein, & Cavallo, 2004), Lego Mindstorms kits (Lego Systems Inc, 2008), and the CCL-Parallax Programmable Badge (Brady et al., 2015) are allowing learners to engage with foundational computer science ideas through physical devices. Physical computing devices have been designed to appeal to diverse ranges of learners. For example, the Lilypad Arduino (Buechley & Eisenberg, 2008) is a fabric based construction kit that enables novices to design and build their own e-textiles and bring crafting and fabric-work into the computing educational space. Likewise, robotics has served as a compelling context in which to engage learning with ideas from computing while grounding the learning in the construction, manipulation, and controlling of physical devices (Fagin & Merkle, 2003; Martin, Mikhak, Resnick, Silverman, & Berg, 2000). Thus, yet another dimension along which computing education varies is the incorporation of computationally enhanced physical devices and artifacts.

Programming Languages and Environments

The decision of what program language to use and when environment learners will program in has a long been a subject of vigorous debate in the computing education research community (Stefik & Hanenberg, 2014). As the design of programming languages and introductory environments is closely related to the questions being pursued in this dissertation,

this portion of the literature is presented in much greater detail later in this chapter. The chapter includes discussion of textual versus graphical approaches to programming interfaces as well as the design of languages and environment design for novice programmers.

Programming Paradigm

Along with differing languages and tools, there are also bigger picture differences around types of languages and programming paradigms. An early paradigm for introductory programming activities that gained lots of followers and has been replicated in various ways across countless environments is the use of “body-syntonicity” (Papert, 1980) as a way to ground programming understanding. Started by Papert and his colleagues in the creation of Logo, a large family of environments leverage an embodied motion component to early programming experiences, as can be seen in environments ranging from Scratch (Resnick et al., 2009) to Karel the Robot (Pattis, 1981) and all its descendants. Common across these languages are primitives that relate to egocentric motion as well as a graphical execution of programs where programs can be visually executed. These environments are in contrast to conventional programming environment that are entirely text-based, both in the form the programs take as well as what output looks like. Many early environments (and vestiges of them that remain in use) constrain the learner to a command line interface where all input and output must come from the keyboard.

Along with the types of programs that can be authored with the language, there are also a number of different programming paradigms that have been advocated for in terms of how students should be introduced to the subject. A growing number of tools and educational researchers advocate an object-oriented approach to programming as being the best suited for beginners. As such, environments like BlueJ (Kölling, Quig, Patterson, & Rosenberg, 2003) and Alice (Cooper, Dann, & Pausch, 2003) have been created that follow an object-oriented

approach. In a separate, but equally active camp are educators and computer scientists who advocate for functional languages as being the best suited for introductory programming contexts (Felleisen, Findler, Flatt, & Krishnamurthi, 2004). At the same time researchers are arguing for paradigm-based instruction, there are many instructional approaches that rely on general purpose, multi-paradigm languages like Java and Python that usually begin with an imperative programming orientation, saving object-oriented features or functional strategies for later in the learner's trajectory. A final paradigm for teaching computing that moves even further from language features advocates teaching not a programming language, but instead grounding computer science instruction in predicate calculus and Boolean algebra (Dijkstra, 1989).

Situating This Dissertation in the Larger Landscape

As mentioned in the introduction of this high-level overview section, any given learning environment puts a stake in the ground at some point along each of these dimensions. While work can prioritize one facet over another, all must be accounted for. In this dissertation, the focus is on the design of programming languages and interfaces, but does fall at specific points along the other dimensions. The work takes place in high school classrooms, so falls at the formal end of the context spectrum and does not try and integrate the material with other coursework. The curriculum designed for the students to follow is geared toward open-ended activities and encourages students to incorporate various aspects of their own personality and interests into the final project, in this way, it tries to be appealing to a diverse array of learning and create an inclusive, participatory context. As the dissertation focuses on features of programming languages and environments, it resides entirely in the virtual world, not utilizing offline or physical aspects that are increasingly included in introductory computer science courses. With respect to programming paradigm, the environment includes a turtle graphics

interface, so follows in the Logo tradition, but only half of the assignments take advantage of that feature, the other half use only text. Finally, as the questions being pursued relate to the design of languages, the dissertation does not take a specific stand with respect to the design of introductory programming languages, but instead uses multiple conditions that live at different points along the spectrum to provide the data that will allow us to better understand the impact of this decision on various aspects of learning computer science.

Representations and Learning

“The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.” (Dijkstra, 1982)

As stated by the Turing Award winning computer scientist Edsger Dijkstra in the quote above, the tools we use, in this case the programming languages and development environments, have a profound, and often unforeseen, impact on how and what we think. diSessa (2000) calls this material intelligence, arguing for close ties between the internal cognitive process and the external representations that support them: “we don’t always have ideas and then express them in the medium. We have ideas *with* the medium” (diSessa, 2000, p. 116 emphasis in the original). He continues: “thinking in the presence of a medium that is manipulated to support your thought is simply different from unsupported thinking” (diSessa, 2000, p. 115). These symbolic systems, provide a representational infrastructure upon which knowledge is built and communicated (Kaput, Noss, & Hoyles, 2002). Adopting this perspective informs why it is so crucial to understand the relationship between a growing family of programming representations and the understandings and practices they promote. This literature informs the dissertation as we are fundamentally investigating the relationship between programming representations and learners.

We begin our literature review of representational infrastructure by first investigating the role of external representational systems broadly to understand the various purposes they serve, which in turn will inform the role they place in cognition and learning. Palmer (1978), in his early work forming a cognitive framework for understanding representations, states: "a representation is, first and foremost, something that stands for something else" (p. 262-3). His framework makes a division between the *represented world* and the *representing world* with a correspondence existing between the two and argues that using representations involves processing these two worlds to determine the relations held between the two. This is categorized as a "symbol-systems" perspective (Nemirovsky, 1994; Sherin, 2000). Drawing on the work of Bhaktin (1981) and his distinction between a sentence and an utterance, Nemirovsky (1994) developed a framework that makes a distinction between the symbol-system perspective of Palmer and what he calls *symbol-use*, shifting focus from the rules of the representational system to an emphasis on their use and the meaning they carry within a particular context. This emphasis on symbols-in-use and a recognition that learners' own knowledge and experience should influence the representations used and how they are studied and evaluated has been a recurring idea within the Learning Sciences (Confrey & Smith, 1994; diSessa, Hammer, Sherin, & Kolpakowski, 1991; Lave, 1988; Noss & Hoyles, 1996; Sherin, 2000; Weintrop & Wilensky, 2014b; Wilensky, 1995). Bringing this perspective to the study of the design of programming representation broadens our focus from the representations in isolation, to a broader analytic lens that incorporates the contexts, activity, and learners themselves to understand the relationship between modality and learning.

The role of representations on cognition has been studied across a variety of representational infrastructures and their influence on various cognitive tasks. One large body of

work that has emerged from studying this question is identifying the relationship between language, literacy and thought (Boroditsky, 2001; Luria, 1982; Ong, 1982; Scribner & Cole, 1981; Vygotsky, 1986; Whorf, Carroll, & Chase, 1956). As we are less interested in thought and natural language broadly but instead logical thinking coupled with symbolic formalisms, we focus our review on scholarship within the mathematics domains as the structure imposed by mathematical symbolic forms and the concepts they express are more closely related to our domain of computer science.

As our interest is in student learning with representations, taking a perspective that moves past the symbol-system in isolation is essential as “a symbol systems analysis does not, in its raw form, provide a theory of the knowledge or capabilities possessed by students. Instead, it describes knowledge only by the function that it must perform” (Sherin, 2000, pp. 405–6). Sherin, having identified this gap between the symbolic systems view and the understanding they promote and usages they enable, pursued a research course to better understand this relationship that is similar to our own. Focusing on concepts from physics and investigating the use of conventional algebraic representations as compared to programmatic representations, Sherin (2001) found that different representational forms have different affordances with respect to students learning physics concepts and, as result, affects their conceptualization of the material learned. “Algebra physics trains students to seek out equilibria in the world. Programming encourages students to look for time-varying phenomena, and supports certain types of causal explanations, as well as the segmenting of the world into processes” (Sherin, 2001, p. 54).

Wilensky and Papert (2006, 2010) give the name structuration to describe this relationship between the representational infrastructure used within the domain and the understanding that infrastructure enables and promotes. While often assumed to be static,

Wilensky and Papert show that the structurations that underpin a discipline can, and sometime should, change as new technologies and ideas emerge. In their formulation of the idea of structurations, Wilensky and Papert (*ibid*) document a number of restructurations, shifts from one representational infrastructure to another, and provide a set of criteria with which to evaluate them. Shifts including the move from Roman numerals to Hindu-Arabic numerals (Swetz, 1989), the use of the Logo programming language to serve as the representational system to explore geometry (Abelson & DiSessa, 1986), and the use of agent-based modeling to representation various biological, physical, and social systems (Blikstein & Wilensky, 2009; Wilensky, 1999; 2001; Wilensky & Reisman, 2006). This work highlights the importance of studying representational systems, as restructurations can profoundly change the expressiveness, learnability, and communicability of ideas within a domain. As we will see in the next sections, the rise of new programming modalities, representations, and tools demand that such analyses be conducted to better understand the effects of these emerging approaches to teaching, learning, and using ideas within the domain of computer science.

Novice Programming Environments

The previous section highlighted the critical importance of representations and their influence on cognition and learning, with that as a backdrop, we now proceed with a review of various design efforts intended to improve a learner's introduction to the field of computer science and the practice of programming. A great deal of work has been done on the design and implementation of programming languages and environments for beginners (for reviews of this work, see: Duncan, Bell, & Tanimoto, 2014; Guzdial, 2004; Kelleher & Pausch, 2005). In this section we discuss some of the more influential languages and environments and theoretical contributions that informed the environments and designs being investigated in this dissertation.

Also, it is important to note that this section does not include a detailed review on all aspects of learning to program, instead it more narrowly focuses on the design of learning environments and the relationship between modality and learning. For larger reviews of the computer science education literature see (Guzdial, 2015; Pears et al., 2007; Anthony Robins, Rountree, & Rountree, 2003).

Languages and Environments from constructionist tradition

Constructionism has a long history of producing computer-based learning environments that empower learners and make computational and mathematical ideas accessible (Harel & Papert, 1991; Papert, 1980, 1993). This work laid much of the important theoretical and design groundwork upon which current movements to promote programming and computer science are built. Languages and environments from the constructionist tradition have successfully enabled children (as well as adults) to construct their own, personally meaningful computational artifacts, often with little (or no) formal instruction. In this section we provide a brief history of the more influential constructionist programming environments, beginning with Logo, the language that started it all.

Logo

The Logo programming language was iteratively developed by Seymour Papert and colleagues in Boston in the 1960's. Logo was the earliest programming language designed explicitly for children (an early report is given in Feurzeig et al., 1969). Based on the Lisp programming language, “Logo was designed to provide a conceptual foundation for teaching mathematical and logical ways of thinking in terms of programming ideas and activities” (Feurzeig, Papert, & Lawler, 2011, p. 487). In *Mindstorms*, Papert (1980) dedicates a chapter to

discussing the theoretical roots that most directly informed the design and creation Logo. The first stems from Piaget and his work on epistemology, recognizing that to study how a child comes to understand a concept is to study the concept itself. Logo, in its design to teach mathematics in a fundamentally different way, reimagines what mathematics looks like and how the learner interacts with and thinks with mathematical concepts. This can be seen in the way Logo shifts learners away from viewing mathematics as a domain of calculations and towards envisioning mathematics as series of processes (Papert, 1972). The second theoretical influence to Logo was from the field of artificial intelligence (AI). As one of the goals of AI is to build machines that can perform intelligent behavior, such an endeavor needs to study the nature of intelligence. An appeal of this work was that its methodology relies heavily on computation and computational environments that force theoreticians to concretize and explicitly represent their ideas and theories of learning by computationally reifying them. Papert saw in this line of work the potential for giving children the opportunity to similarly think concretely about mental processes and what it means to learn.

Logo was designed with the principle of “low threshold, no ceiling” and saw early and widespread international adoption and influence in the mathematics community especially among forward thinking educators. A central contribution of Logo to introductory programming design was the invention of the turtle – an entity (either physical or virtual) that the user controls in the form of movement instructions, then watches the turtle carries them out. The turtle leveraged what Papert (1980) called “body syntonicity” which enabled learners to use their own experiences in the world as a productive resource for generating programming instructions. As we will see throughout this review, this design feature shows up repeatedly as an accessible way for learners to engage in, and have early successes with, programming.

Naïve Realism and Spatial Metaphors with Boxer

Boxer (diSessa & Abelson, 1986) was an early descendant of Logo that added to the environment the feature that every object in the system had an on-screen graphical representation that could be inspected, modified and extended. This design feature was based on the naïve realism theory of mind and was intended to create a programming interface where “users should be able to pretend that what they see on the screen is their computational world in its entirety” (diSessa & Abelson, 1986, p. 861). As such, the environment presents the user with a complete visual model of what is happening in the machine. This resulted in a design where all computational objects in Boxer are displayed as two-dimensional boxes (hence the name). These boxes can each be unpacked, giving the users access to its contents, creating a “glass-box” environment where nothing is hidden from the learner. A second major design feature of Boxer was the use of a spatial metaphor as a way to display information about the entities within a program and their relation to each other. As such, boxes visually rendered inside other boxes spatially depict a hierarchy of boxes. This use of visual layout of commands as a means of communicating information about the program and the effect of such an interface will resurface again in later environments and is at the heart of the questions being pursued in this study.

Many-Turtled Logo Environments

A second that the Logo language was built upon was to relax the constraint that there can be only a single entity being controlled in the environment. By allowing users to introduce as many turtles as they want and providing tools that allow them to give instructions to only a subset of the turtles, learners could create programs that support the investigation of emergent and decentralized complex systems (Resnick, 1997; Wilensky & Rand, 2014; Wilensky & Resnick, 1999). Two early version of these environments were StarLogo (Resnick & Wilensky,

1993) and StartLogoT (Wilensky, 1997). The successor to StartLogoT, NetLogo (Wilensky, 1999b), has since become a very widely used implementation of a Logo-based multi-agent programming environment.

Building on the finding that programming and construction are effective ways for learners to develop mathematical understandings, these multi-turtle environment environments (subsequently named agent-based models) have extended this work beyond mathematics to include a wide range of STEM topics. NetLogo was designed as a modeling environment that captures emergent phenomena (Wilensky, 2001). NetLogo enables learners to use, modify, and create models of real-world phenomena as a means to develop deep understandings of the underlying mechanisms and properties of the topic under investigation. A core constructionist design principle of NetLogo is that by programming models of scientific phenomena, students will learn science more deeply while also learning programming and modeling. This approach has been found to be effective for teaching students in a wide variety of domains including biology (Wilensky & Reisman, 2006), electromagnetism (Sengupta & Wilensky, 2009), chemistry (S. T. Levy & Wilensky, 2011; Stieff & Wilensky, 2003), evolution (Wilensky & Novak, 2010) and material sciences (Paulo Blikstein & Wilensky, 2009). By situating the programming activity within a larger goal of learner specific content introduces another oft-replicated feature of introductory programming environment – the importance of context surrounding the programming activity (Cooper & Cunningham, 2010; Guzdial, 2010).

Blocks-based and Graphical Logo Environments

The last Logo descendants we include in this review are environments that use a blocks-based or other graphical programming approach. Scratch (Resnick et al., 2009) is the most well known of the group, but other blocks-based constructionist tools include LogoBlocks (Begel,

1996), StarLogo TNG (Begel & Klopfer, 2007), and Snap! (Harvey & Mönig, 2010). These environments replace the textual interface of Logo with a programming-primitive-as-puzzle-piece metaphor that allows users to drag commands into place and snap them together to assemble their programs. Other approaches that leverage similar graphical approaches include ToonTalk (Kahn, 1999), which relies on a much more direct visual metaphor for defining instructions, and Squeak (now e-toys) (Alan Kay, 2005), which uses a rules-as-tiles programming mechanism. We only briefly mention these environments here, as more time will be dedicated to graphical and blocks-based programming tools later in this chapter.

Languages and Environments from outside the Constructionist community

While many early programming languages emerged from the constructionist research community, the computer science education community also developed a number of programming languages and environments designed for novices. Here we review some of the more influential efforts that helped inform this dissertation. This includes languages designed explicitly for educational contexts as well as software authoring tools for novices.

Beginner Programming Languages

Early on it was recognized that the design of the language itself can support or hinder students in their quest to master programming, which resulted in early efforts to develop more accessible programming languages (Mendelsohn, Green, & Brna, 1990). Along with Logo, which was explicitly designed with mathematics learning in mind, other languages emerged with the goal being to serve as an introduction to the field of computer science. An early, influential language designed for novices was BASIC (Kemeny & Kurtz, 1980), whose acronym stands for Beginner's All-purpose Symbolic Instruction Code. BASIC included a relatively small

instruction set, removed all unnecessary syntax, and was designed to support short turn around times between composition and execution of programs, which collectively made it more accessible to novices. BASIC experienced a great deal of success and was a popular language throughout the early era of personal computing from the mid 1970's through the 1980s.

As the field of computer science education matured, new languages and strategies emerged that were designed to serve as introductory tools and prepare learners for more industrial, fully featured languages. Languages such as Blue (Kölling & Rosenberg, 1996) and JJ (Motil & Epstein, 1998) simplified syntax and provided tools to allow learners to focus on programming fundamentals before progressing to fully featured languages. Other languages tried to blend the best features of various languages in hopes of developing new languages that were both powerful and easy to learn and use (Holt & Cordy, 1988). Another direction introductory programming languages took was to create more declarative languages in which programming was a more direct, rule defining activity. Languages such as Prolog (Colmerauer, 1985), and later graphical environments such as Agentsheets (Repenning et al., 2000), ToonTalk (Kahn, 1999) and StageCast Creator (D. C. Smith, Cypher, & Tesler, 2000) utilize this strategy.

A third approach was the use of mini-languages, which are small, simple languages designed to support the first steps in learning to program (Brusilovsky et al., 1997). Mini-languages often centered around specific activities and provided only the commands necessary to accomplish the immediate task, such as Karel the Robot, which has learners write short programs to control an on-screen robot (Pattis, 1981). These mini-languages share features with domain-specific languages, which are not intended for general purpose programming, but instead tailor a smaller language around specific tasks, narrowing the gap between the objective and the representations in which intentions are encoded (Van Deursen, Klint, & Visser, 2000). In doing

so, the designer of the language can leverage the existing knowledge of the user to provide a set of tools tailored for the task at hand, narrowing what Norman (1991) calls the gulf of execution.

Another approach taken in the design of programming languages for novices is to bring the programming language closer to natural language. The first language that tried to draw on natural language grew out of an effort to create a “Common Business Language” (COBOL), which intentionally tried to maximize the use of English in its syntax (Sammet, 1981). Another early language that took this approach was Hypercard. When asked about Hypercard’s ancestors, designer Bill Atkinson responded: “The first one is English. I really tried to make it English-like” (*Goodman*, 1988 as cited in Bruckman & Edwards, 1999, p. 208). A more recent language that adopted this strategy, that was designed explicitly for younger learners is the MOOSE programming language designed to enable kids to create places, creatures and other objects in a text-based virtual game (Bruckman, 1997). This desire for a more readable, natural language-like aesthetical can also be seen in the blocks-based visual programming languages we will review in the next section, as these tools use other mechanisms to facilitate the computational parsing of programs, thus allowing the language itself to be more expressive with it’s commands. This feature has been found to influence learners’ perceived ease of use of the language (Weintrop & Wilensky, 2015b).

A final strategy that is important to include in this review of approaches to designing programming languages for novices is the creation of languages that try to address the documented issues that novices have with the syntax of programming languages. Research has found language syntax, the seemingly esoteric punctuation and formatting rules that must be followed when composing programs, to be a serious barrier for novice programmers (Denny et

al.; Stefik & Siebert, 2013). Through a series of controlled experiments that had novices use one of a variety of languages that demonstrated various syntactic features, Stefik and Siebert (2013) found that characteristics of syntax do directly influence a language's learnability. One solution to the syntactic problem is the creation of programming tools that prevent syntax errors through the use of visual cues and graphical composition tools. This feature is especially relevant to the proposed study, as graphical programming proponents boast that the lack of syntax is a key feature that contributes to its appropriateness for young learners (Maloney et al., 2010; Resnick et al., 2009), but research is finding this approach does not solve the syntax problem, but only delays it (Parsons & Haden, 2007; Powers et al., 2007). This issue will be discussed in more detail later in our literature review.

Integrated Development Environments for Novices

Along with recognizing that features of the language can support or hinder learnability, it was realized that software used to author programs (called Integrated Development Environments or IDEs) themselves could provide a large number of supports to help the novice overcome challenges including syntax errors, deciphering compilation errors, and problematic sections of programs. This recognition coincided with a larger shift in the computing space that was advocating for a shift away from users conforming to computer requirements towards a world where the computer conformed to the user (Donald A. Norman, 1993). These efforts initially focused on supporting experts, but educational technology designers soon realized that what is good for the expert is often not the same as what is best for the novice and when designing educational tools, you should proceed with the learner in mind (Soloway, Guzdial, & Hay, 1994). As such, a growing number of IDEs have been developed explicitly with novice programmers in mind.

An early and influential development environment designed to facility novices

specifically through a reduction on potential syntax errors was the Cornell Program Synthesizer, which built on the fact that programs are not flat text, but instead “hierarchical compositions of computational structures and should be edited, executed, and debugged in an environment that consistently acknowledges and reinforces this viewpoint” (Teitelbaum & Reps, 1981, p. 563).

The Cornell Program Synthesizer provided users with templates that followed the syntactic structure of the language and thus, when filled in, would result in valid statements that could be added to the program. These templates were constructed by following the grammar defined by the language’s abstract syntax tree (AST). While a number of different project and research groups followed the lead set by the Cornell Program Synthesizer, Carnegie Mellon University emerged as a leader in the development of programming tools that used features of the language to support novices. Over the course of a number of projects, the CMU group iteratively developed a family of programming environments including GNOME (Garlan & Miller, 1984), Genie (Chandhok & Miller, 1989), and ACSE (Pane & Miller, 1993). These environments progressively introduced features including code layout based on the language’s AST, incremental parsing and feedback for syntax errors while editing, supporting multiple views of the same program simultaneously including high-level design views and code navigation views, and runtime visualization tools (a history of these environments can be found in Miller, Pane, Meter, & Vorthmann, 1994). Differentiating these tools from the efforts in the previous section is the fact that being environments, these tools were not necessarily tied to a specific language. For example, GNOME environments were created for various languages including Karel the Robot, Pascal, Fortran, and Lisp (Miller et al., 1994). The inclusion of a language’s AST as part of what

determines how programs are composed is central to the blocks-based visual programming tools of interest to this dissertation and an idea we will return to later in this literature review.

More recently, a new generation of IDEs have been developed that are designed with a specific language in mind and include features unique to that language and even to specific pedagogies for learning that language. Environments such as DrScheme developed for the scheme programming language (Findler et al., 2002), and DrJava, a similar tool developed for Java (Allen, Cartwright, & Stoler, 2002), present an integrated, graphics-rich editor and use a functional read-evaluate-print development cycle to assist novices in their early programming endeavors. The BlueJ environment is a popular Java IDE designed to support an object-first approach to learning to program in Java (Kölling et al., 2003). BlueJ was intentionally designed to keep the interface simple as to not overwhelm the learner and emphasize the features of the language deemed most important, which in BlueJ is the object-oriented nature of Java. Building off of the successes of BlueJ and remaining faithful to the learner-focused design, the BlueJ team released a second IDE called Greenfoot, designed for younger learners that adds visual program execution to the IDE to further support younger learners and their developing understanding of programming concepts (Henriksen & Kölling, 2004). A particular feature of Greenfoot that is of relevance to this study is the decision to share many interface features between BlueJ and Greenfoot to make transition between the environments easier for learners as they progress. As we discuss below, transition from introductory to more sophisticated programming environments and language is rarely a consideration for designers of IDEs for novices. More recently, the Greenfoot team has released a new language called Stride, that is based off Java and supports a new form of hybrid text-graphical editing the call Frame-based Editing (Kölling, Brown, &

Altadmri, 2015), we will return to this new environment and programming approach later in the chapter.

Visual Programming

As the development of programming languages and environments evolved, it was found that shifting from an all-text representation of programs to an approach that leverages spatial and graphical features could be productive for learning and understanding (D. C. Smith, 1977). Collected under the label “Visual Programming”, these environments are broadly defined as “any system that allows the user to specify a program in a two (or more) dimensional fashion” (Myers, 1990, p. 98). While this definition is intentionally broad, it excludes text-based programming (which is considered to constrain composition of programs to a single, horizontal dimension), software used to produce visualizations (like animation and drawing programs), and tools that visually depict the execution of programs (like environments that visually render memory contents or algorithmic flow of a running program). To provide a framework for evaluating visual programming environments, Green and Petre (1996) developed a cognitive dimensions framework that characterizes features of these environments and maps out the trade-offs that exist between different visual design choices. These cognitive dimensions include Abstraction Gradient (various granularities of abstraction supported), Consistency (how formulaic is the language), Progressive Evaluation (what feedback is available from partially-complete programs), and Visibility (how easy is it see and read the code) among others. This framework proved to be productive and is widely used to evaluate visual programming environments.

Direct manipulation was an early and widely implemented approach to graphical programming that touches on a number of strengths of a graphical authoring modality. Hutchins et al. (1985) explain the concept:

The promise of direct manipulation is that instead of an abstract computational medium, all the "programming" is done graphically, in a form that matches the way one thinks about the problem. The desired operations are performed simply by moving the appropriate icons onto the screen and connecting them together. Connecting the icons is the equivalent of writing a program or calling on a set of statistical subroutines, but with the advantage of being able to directly manipulate and interact with the data and the connections. There are no hidden operations, no syntax or command names to learn.

What you see is what you get. (p. 314)

Included in this definition are a number of key features of graphical programming: the presence of onscreen icons that carry some computational or programmatic meaning that can be controlled directly, a two (or more) dimensional space to work within, a minimization (or absence) of syntax or commands, and a general transparency that permeates the environment and how it is meant to be used. Bruner (1973) distinguishes between this "transparent" use of a representational medium, where actions are guided by reasoning about the entities being represented, and an "opaque" use of symbols, where attention is focused on the inscriptions themselves.

Based on the promise of easier to learn, easier to use, programming systems, many visual programming languages have been designed and evaluated. Direct manipulation tools often rely on flow-chart or data-flow diagrams that map logical flows through instructions (Hils, 1992). LabVIEW (Johnson, 1997; Santori, 1990), a circuit diagram program built on an electronic block

wiring metaphor, often serves as an exemplar direct manipulation environment in studies of this programming modality, with the results of these studies generally being mixed (Whitley, 1997). Flogo (Hancock, 2003), a graphical programming environment designed to facilitate learners programming robot behaviors, used a visual dataflow view of information intended to make programs more understandable, accessible, and modifiable. A more contemporary version of direct manipulation software is the Lego Mindstorms NXT-G programming tool (Lego Systems Inc, 2008), which allows children to program robots by dragging iconic representations of program commands and robot components from a palette onto a workspace, where they can be wired together.

Another family of programming tools that emerged from this tradition use images and a graphical stage rendered as a grid to allow users to program rules the system can follow. Building on the idea of programming-by-demonstration, KidSim, later renamed Stagecast Creator (D. C. Smith, Cypher, & Spohrer, 1994; D. C. Smith et al., 2000), was developed to allow users to create games by defining rules using symbols and graphics, removing the need for syntax or text-based programming commands. These graphical rules govern the behavior of the entities (called agents) in the world being programmed making it easy to create playable video games. Repenning and colleagues took a similar approach (although they choose to call it programming-by-problem-solving) in the development of Agentsheets, an environment in which users program sets of agents that move via graphical rules (Repenning & Sumner, 1995). Unlike Stagecast Creator, Agentsheets moves beyond game-making to include the creation and exploration of scientific models and simulations as part of its uses (Repenning et al., 2000). With the release of Agentcubes, the two dimensional restriction of the stage has been lifted, enabling

learners to create three-dimensional games and simulations (Ioannidou, Repenning, & Webb, 2009).

Two other approaches to visual programming are important to mention. The first is the use of tangibles as representations of programming statements. Tools such as AlgoBlocks (Suzuki & Kato, 1995), the Digital Construction Set for Lego Bricks (McNerney, 2004), and Tern (M.S. Horn & Jacob, 2007) all explored different ways to program with physical devices. In a comparative study with Tern, Horn and colleagues found the tangible programming approach to be more approachable and resulted in longer engagement by visitors in a museum setting (M. S Horn et al., 2009). The second consist of languages that do not rely on visual metaphors of rules or objects, but visual metaphors of programming statements and abstractions directly. We call these blocks-based programming environments and review them in more detail later in the chapter as they are a focus of this dissertation.

Evaluating Visual Programming Environments

In the early 1990s a thread of research, lead by Green and Petre among others, systematically compared text-based and visual programming to understand which was superior. While some studies showed promise in the use of visual programming tools (Baroth & Hartsough, 1995), other studies conducted in more controlled environments found contradictory evidence. Green and collaborators found that visual programming environments required longer amounts of time to develop solutions and provided less guidance on strategic approaches, resulting in more difficulty among novice programmers (Green & Petre, 1992; Green, Petre, & Bellamy, 1991; Petre, 1995). They attributed these findings to unfamiliarity with available secondary notations of the languages (a dimension from the Green and Petre's cognitive dimensions framework that captures the use of layouts and other informal cues to express

structure) and the match-mismatch hypothesis (Green, 1977), which links difficulty in generating function solutions to misalignment between the structure of a problem with the structures supported by the language. These findings were reproduced in a later set of studies with a larger set of visual programming tools, in which it was found that various visual representations were at best on-par with their textual equivalents (Moher et al., 1993). For a longer review of this work, see Blackwell et al. (2001). While much of this comparative work was conducted over twenty years ago, the field is still active with studies being conducted with new tools (for example Hundhausen, Farley, & Brown, 2009). We will return to these more contemporary studies later as they focus not just on comparisons between tools, but also transitioning between representations.

Blocks-based Programming

The blocks-based approach of visual programming, while not a recent innovation, has become widespread in recent years with the emergence of a new generation of tools, lead by the popularity of Scratch (Resnick et al., 2009), Snap! (Harvey & Mönig, 2010), and Blockly (Fraser, 2013). These programming tools are a subset of the larger group of editors called *structured editors* (Donzeau-Gouge, Huet, Lang, & Kahn, 1984) that make the atomic unit of the composition tool a node in the abstract syntax tree (AST) of the program, as opposed to a smaller piece (i.e. a character) or a larger piece (a fully formed functional unit). In making these AST elements the building blocks, then providing constraints to ensure a node can only be added to the program's AST in valid ways, the environment can protect against syntax errors. The constraints can be provided in a number of ways. Blocks-based programming environments leverage a programming-primitive-as-puzzle-piece metaphor that provides visual cues to the user about how and where commands can be used as their means of constraining program

composition. Programming in these environments takes the form of dragging blocks into a scripting area and snapping them together to form scripts. If two blocks cannot be joined to form a valid syntactic statement, the environment prevents them from snapping together, thus preventing syntax errors but retaining the practice of assembling programs instruction-by-instruction. Along with using block shape to denote usage, there are other visual cues to help programmers, including color coding by conceptual use, and nesting of blocks to denote scope (Maloney et al., 2010; Tempel, 2013).

Early versions of this interlocking approach include LogoBlocks (Begel, 1996) and BridgeTalk (Bonar & Liffick, 1987) which helped formulate the programming approach which has since grown to be used in dozens of applications. Alice (Cooper et al., 2000), an influential and widely used environment used in introductory programming classes, uses a very similar interface and has been the focus of much scholarship evaluating the merits of the approach.

Figure 2.1 shows programs written in a number of blocks based programming tools.

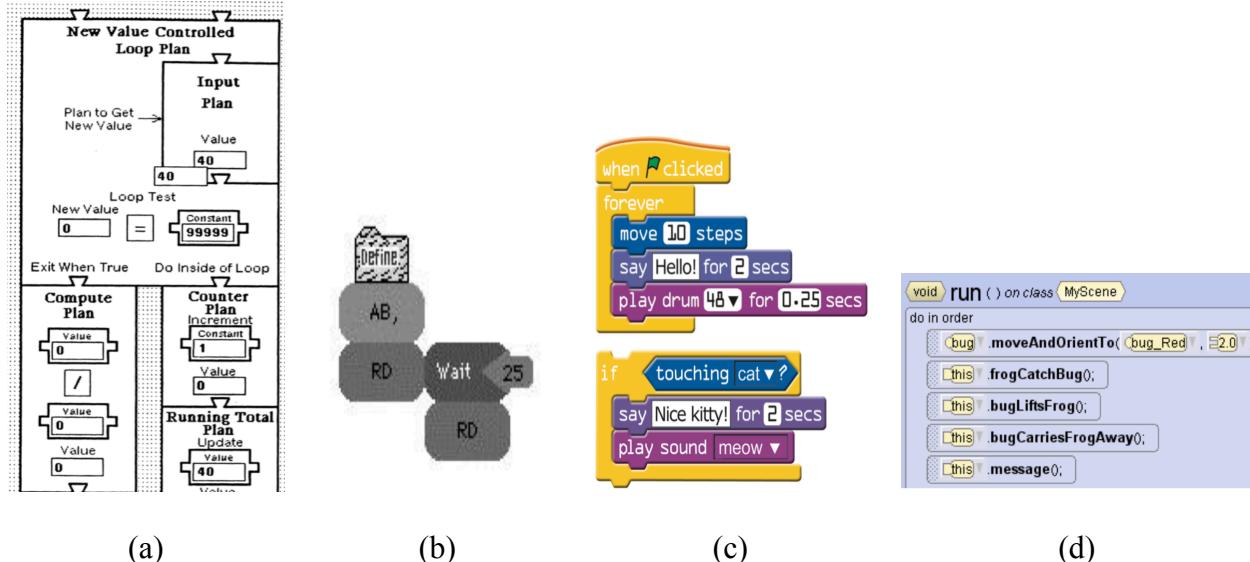


Figure 2.1. Four example blocks-based programming languages: (a) BridgeTalk,

(b) LogoBlocks, (c) Scratch, and (d) Alice.

In addition to being used in more conventional computer science contexts, a growing number of environments have adopted the blocks-based programming approach to lower the barrier to programming across a variety of domains. These include: mobile app development with MIT App Inventor (Wolber, Abelson, Spertus, & Looney, 2011) and Pocket Code (Slany, 2014), modeling and simulation tools including StarLogo TNG (Begel & Klopfer, 2007), DeltaTick (Wilkerson-Jerde & Wilensky, 2010), NetTango, and EvoBuild (Wagh & Wilensky, 2012), creative and artistic tools like Turtle Art (Bontá, Papert, & Silverman, 2010), and PicoBlocks (*PicoBlocks*, 2008), commercial educational programming applications like Tynker (*Tynker*, 2014) and Hopscotch (*Hopscotch*, 2014), and game-based learning environments like RoboBuilder (Weintrop & Wilensky, 2012), Lightbot (Yaroslavski, 2014) and the activities included in Code.org's Hour of Code (*Hour of Code*, 2013) and Google's Made with Code initiative (*Made with Code*, 2014). Further, a growing number of libraries are being developed that make it easy to develop application or task specific blocks-based languages (Fraser, 2013; R. V. Roque, 2007). This diverse set of tools and the ways the modality is being used highlights its recent popularity and speaks to the need for more critical research around the affordances and drawbacks of the approach (Shapiro & Ahrens, 2016; Weintrop & Wilensky, 2015a). There is also a growing number of environments they blend blocks-based and text-based programming approaches, including Pencil Code (Bau et al., 2015) and Tiled Grace (Homer & Noble, 2014).

Evaluating Blocks-based Programming Environments

In our review of literature focusing on the educational efficacy of blocks-based languages, we focus on Scratch and Alice, as these two tools have the widest use in contemporary computer science education of the blocks-based environments listed above. While both Alice and Scratch have been used in formal education environments, it is important to keep in mind that the two projects initially had different goals and different target age groups. Scratch from its inception, was focused on younger learners and informal environments (Resnick et al., 2009), while Alice was targeted at more conventional computer science educational contexts and, as such, has been the focus of more initiatives to evaluate student learning of programming concepts (Cooper et al., 2000).

We begin by reviewing literature on Scratch investigating its use as the language of choice in formal computer science environments. Ben-Ari and colleagues have conducted a number of studies of the use of Scratch for teaching computer science. Using activities of their own design (Armoni & Ben-Ari, 2010), Meerbaum-Salant et al. (2010) concluded that Scratch could successfully be used to introduce learners to central computer science concepts including variables, conditional and iterative logic, and concurrency. While students did perform well on the post-test evaluation from this project, a closer look at the programming practices learners developed while working in Scratch gave pause to the excitement around the results. The researchers found that students developed unfavorable habits of programming, including a totally bottom-up programming approach, a tendency for extremely fine-grained programming, and often incorrect usages of programming structures as a result of learning programming in the Scratch environment (Meerbaum-Salant, Armoni, & Ben-Ari, 2011). Other work looking at comparing blocks-based to text-based programming using Scratch has similarly found that Scratch can be an effective way to introduce learners to programming concepts, although it is not

universally more effective than comparable text languages (C. M. Lewis, 2010). Given Scratch's intention on being used in informal spaces and its emphasis on introducing diverse learners to programming, it is important to highlight Scratch's success in generating excitement and engagement with programming among novice programmers (Malan & Leitner, 2007; Maloney et al., 2008; Tangney et al., 2010; Wilson & Moffat, 2010).

Compared to Scratch, the Alice programming environment has a longer history of serving as the focal programming tool in introductory programming courses. Much of the motivation for using Alice in courses is based on findings that Alice is more inviting and engaging than text-based alternatives, and improves student retention in CS departments (Johnsgard & McDonald, 2008; Moskal, Lurie, & Cooper, 2004; Mullins, Whitfield, & Conlon, 2009). Alice has also effectively been used by instructors who adopt an object-first approach to programming as it provides an intuitive and accessible way to engage with objects with little additional programming knowledge needed. Part of Alice's success and relatively widespread use is due to the fact that the creators of Alice have authored a number of textbooks and curricula that can serve as texts for an introductory programming course (Dann, Cooper, & Ericson, 2009; Dann, Cooper, & Pausch, 2011). It is also important to mention here the growing body of work looking at blocks-based programming as an introductory tool used for preparing students for learning more conventional text-based programming, which is discussed in the next section. Until recently more research had been conducted around the transition from Alice to Java, as it is more frequently featured in conventional CS contexts, but recently this line of research has expanded to include Scratch and other blocks-based tools. As this goal is at the heart of this dissertation, we devote a full section to reviewing efforts towards this end.

A small but growing body of research is conducting systematic comparisons of blocks-based and text-based environments. In a pilot study of this dissertation, we found that students perform differentially on questions asked in blocks-based form compared to the isomorphic text alternative (Weintrop & Wilensky, 2015d). These differences were not universal however, but instead were influenced by the concept under question, with students performing better on blocks-based questions related to conditional logic, function calls, and definite loops, and finding non statistically significant differences on questions related to variables, indefinite loops, and program comprehension questions. Other studies investigating learning outcomes in blocks versus text environments found little difference in learning outcomes, but did report that students completed activities in blocks-based environments at a faster rate. This suggests that while the same learning can be achieved, it happens more quickly in blocks-based environments (Price & Barnes, 2015).

From Blocks-based to Text-based Programming

With the rise in popularity of the blocks-based approach to programming, a question of growing importance is how well these tools prepare students for future, text-based programming languages. Do students develop understandings that can serve as a foundation for future learning or do students struggle to apply what they have learned in new programming contexts with more powerful, text-based programming languages. Recent studies have begun to explore this question of transfer of programming knowledge between blocks-based and text-based programming. Before reviewing this literature, it is important to note that not everyone is in agreement that blocks-based programming is indeed only to be used as a stepping-stone for text-based programming. Some argue that a blocks-based modality is a sufficient end-point for those who are not intent on pursing a career in programming (Modrow, Mönig, & Strecker, 2011).

While we see merit to this position, as we are focused on high-school aged students, and all widely adopted computer science assessments (notably the AP Computer Science exam) are conducted with text-based programming languages and a vast majority of libraries and programming tools are text-based, we see this as a question of great importance. We begin this review by looking at studies attempting to bridge Alice and Java as it has the longest, and most well documented history.

From the outset, Alice was intended for formal educational contexts and was widely shared and discussed in computer education circles. As a result, it has become a popular tool for use in introductory computing courses at the university level, thus the challenge of transition from Alice to Java has become an active area of research. As is often the case, results have been mixed in studies looking at the transition of students from Alice to Java. Powers et al. (2007), in their study following students from a semester in Alice to a semester in Java (using BlueJ) documented a number of challenges faced by their students including issues with syntax, frustration with the lack of progress at a similar pace as in Alice, and a feeling that programming in Alice was not authentic due to its graphical nature. This position has been taken to various degrees (Cliburn, 2008), with some researchers going so far as to state “based on our classroom experience, we question its real pedagogical value for programming education at the tertiary level. Students do not seem to naturally make a strong connection between the formal coding process and what they are doing with Alice” (Parsons & Haden, 2007, p. 213). Another study compared students spending time using Alice compared to students working through the same activities in pseudo-code and found that students in the pseudo-code condition performed better on standard performance measures (Garlick & Cankaya, 2010). In contrast, other researchers have found Alice to be an effective way to introduce learners to programming and had success

using it as a tool for transition to Java. Notably, the authors of Alice developed a tool that allowed students to move back and forth between Java and Alice which was found to be effective at bridging this gap (Dann et al., 2012). Citing this work, classes have adopted this strategy of mixing Java and Alice as a way to leverage the strengths of the visual programming approach while mitigating issues cited above (Dann et al., 2009). A number of textbooks have also been written to bridge the gap (Adams, 2007; Dann et al., 2009; J. Lewis & DePasquale, 2008), but research evaluating the effectiveness of these texts is relatively sparse.

A few studies have been conducted looking at the transition from Scratch to other text-based programming languages. While many of these report only anecdotal evidence, Armoni, Merrbaum-Salant & Ben-Ari (2015) conducted a longitudinal study looking at whether students who had taken Scratch programming classes in middle school performed better in a high school, text-based programming course. Overall, the researchers found little quantitative difference in performance on assessments between students who had previous worked with Scratch and those who had not, but were able to find some areas where the Scratch students out-performed their peers (specifically on the concept of looping). Additionally, the authors found qualitative differences between the two populations, with students who had prior Scratch experience reporting high levels of motivation and self efficacy.

Beyond Alice, a growing number of tools are being designed to address the blocks-to-text gap, either as new stand-alone tools or add-ons to existing tools. Such efforts including the ScratchBlocks (*ScratchBlocks*, 2014), Pencil Code (Bau et al., 2015), Tiled Grace (Homer & Noble, 2014), PyBlocks (Bart, Tilevich, Shaffer, & Kafura, 2015) and SLASH (Behnke, 2013) add-ons to Scratch, the TAIL plugin (Chadha, 2014) and the App Inventor Java Bridge (*App Inventory Java Bridge*, 2014) for MIT App Inventor. PicoBlocks (*PicoBlocks*, 2008), TurtleArt

(Bontá et al., 2010), and recently the Tynker platform (*Tynker*, 2014) all come with the ability to view text-based equivalents to programs constructed with the blocks-based interface. Other tools provide native language translation, for example, the Blockly toolkit comes with built-in language generators that allow you to convert graphical scripts to equivalent JavaScript, Python, or XML files (Fraser, 2013). Additionally, Blockly is architected in such a way as to make it easy to add additional generators to the library making it extensible for future blocks to text transformations. The DrawBridge project is noteworthy in it's effort to bridge blocks-based and text-based programming by introducing pen-and-paper drawing and program-by-demonstration features into its larger pedagogical strategy (Stead & Blackwell, 2014). Game authoring has also been used as a context to motivate blocks-to-text programming as demonstrated by the Flip project, although this environment's text-programming uses natural language expressions over conventional text-based programming syntax (Howland & Good, 2014).

While these environments provide a one-way transition from a blocks-based interface to the textual form, a growing number of tools are providing bi-directional support for new and established languages. Pencil Code (Bau et al., 2015) provides a two-way transition between blocks and Coffee Script, JavaScript, and HTML, while tools have also been built for Java (Matsuzawa, Ohata, Sugiura, & Sakai, 2015), Python (Bart et al., 2015), and Grace (Homer & Noble, 2014). Little work has been published on these hybrid environments. One notable exception is Matsuzawa et al.'s (2015) study in which they taught a semester long introductory programming course using an environment that allowed users to program with either a blocks-based or text-based Java interface. The authors found that over the course of the class, students systematically transitioned from blocks to text on their own, and also found a correlation

between learners' initial confidence and the modality they chose to work in (Matsuzawa et al., 2015).

It is also important to note the difficulty in transferring between graphical and text-based programming environments should not be surprising as researchers have documented difficulties in novices transferring knowledge between two text-based programming languages (Scholtz & Wiedenbeck, 1990; Wiedenbeck, 1993), so seeing similar difficulties across modalities is unsurprising.

This concludes our review of the various literatures that have informed the design of this study. In the next chapter, we layout the study design used to answer the stated research questions, including the data sources used, settings in which the research was conducted, the population we recruited, and the analytical methodology used to evaluated the collected data.

3. Methodology

The study that makes up the heart of this dissertation is a three-condition, quasi-experimental study designed to understand the effects of using blocks-based, text-based, and hybrid blocks/text programming tools in formal introductory computer science classrooms. The study is broken up into two phases: *A Three-way Introduction to Programming* followed by *The To-text Transition*. During the three-way introduction, we followed three sections of an introductory programming class at the high school level for the first five weeks of the school year. The *To-text Transition* follows those same three classrooms as the students transition from the introductory tools to more a conventional text-based programming environment. The second phase commences immediately following the conclusion of the three-way introduction phase and will last for 10 weeks. This is a mixed-methodology study, so a variety of data sources were used, including classroom observations, written assessments, student and teacher interviews, and the collection of student-generated artifacts. This chapter presents the various methodological aspects of the study, including a detailed description of the study design, information about the setting and participants, and a discussion of the data that were collected. We also present the analytic approach taken for each set of data collected.

Study Design

Like many studies in the field of the Learning Sciences, this study is inspired by the design-based research methodology (Design-based Research Collective, 2003; Collins, Joseph, & Bielaczyc, 2004.). In design-based research, designed artifacts are used to inform our understanding and advance our theory of how students come to understand the topic under investigation. In this study, the three variations of our introductory programming tool serve as

the central design component with which to investigate student understanding. A central characteristic of design-based research studies is their iterative nature, where earlier trials with a given tool are used to inform and revise the designed artifacts. This study was conducted over two consecutive school years. The first year was used as a pilot study for the programming environments, the curriculum, and the data collection methodologies. The data collected in the first year were used to inform and revise the materials and procedures for the second iteration of the study⁵. Both years of the study followed the same study design and were conducted in the same setting. As the findings presented in this dissertation focus on data collected in the second year, we will focus our methodological discussion on that iteration of the study.

Phase One: A Three-way Introduction to Programming

The first phase of the study was designed to examine the use of three different versions of the same introductory programming tools in a high school level introductory programming classroom. This phase of the study follows three classrooms during their first 5 weeks of a yearlong introduction to programming course. Each of the three classes used a different variant of the same programming environment called Pencil.cc (a customized version of the Pencil Code environment). The difference between the three versions of the environment is in how programs are represented and authored. One class used a blocks-based interface, the second used a text-based authoring interface, and the third version of the tool used a hybrid blocks/text approach. While there are many differences that exist between the introductory environment and conventional Java programming environments, the focus in this work is on the role modality plays in influence novice programmers' experiences. The three environments are discussed in

⁵ The findings from the pilot study and a discussion of how they informed the design of the tools used in the second study are discussed in the next chapter, which focuses on the design of the introductory programming environments.

greater detail in the next chapter. These three classrooms make up the three conditions of our quasi-experimental design. All three classes worked through the same set of activities and engage in the same classroom discussions. One teacher taught all three classes, allowing us to control for teacher effects, though some overall results will be influenced by the specific teacher. The design of the three environments, which is central to this dissertation are discussed in detail in the next chapter.

Curriculum

The five-week curriculum for the introductory course is based on the Beauty and Joy of Computing course designed by Daniel Garcia and Brian Harvey at UC Berkeley (Garcia et al., 2015), along with an assortment of other introductory computing activities grounded in the Constructionist programming tradition pioneered by Papert and others around the Logo programming language (Harvey, 1997; Papert, 1980). An emphasis of this design was to allow students creative freedom in each assignment, confronting the traditional approach to programming assignments where each assignment is the same. This approach resonated with students, as one student said at the conclusion of the five-week curriculum: “*I liked the fact that we were able to, like, we were given a prompt and we were able to go from there, for most projects. That was cool, I found that fun. It kind of let me go off, it let me tinker a bit, but it also let me stay focused. I really liked that.*”

Over the course of the five weeks, four major conceptual topics are covered: variables, conditional logic, looping logic (including both definite and indefinite loops), and procedures. Throughout the activities, care was taken to blend visually executing programs (like traditional Logo graphics drawing assignments) and number or text processing activities. Table 3.1 and

Table 3.2 provide a high-level outline of the curriculum, a more detailed description of the curriculum, including detailed descriptions of the specific activities can be found in Appendix A.

Table 3.1. A high-level summary of the 5-week introductory curriculum.

Week 1: Introduction to Pencil.cc and Variables The goal of the first week is to acclimate students to programming in Pencil.cc. This includes introducing them to the environment, the quick reference menu where additional information can be found, and various basic commands (mostly associated with moving the on-screen turtle and basic I/O). Activities: Quilt, Mad-Libs, Tip Calculator
Week 2: Conditionals In this second week we introduce students to conditional logic and predicates. Activities: Color-by-Quadrant, Movie Recommendation Engine, Grade Ranger
Week 3: Iterative Logic In week three, we introduce looping logic. This includes an assignment having students draw repeating figures using definite loops and concentric shapes with indefinite loops. Activities: Guessing Game, Radial Art, Spiraling
Week 4: Procedures The fourth week begins with an activity showing students how to define new procedures and how to pass parameters into them. The second half of the week includes an assignment that asks students create procedures and also use conditional and looping logic. Activities: PolyGoner, Connect Four, Brick Wall
Week 5: Summative project The final week of the curriculum has students develop their own projects. The only requirement is that projects must include all of the concepts students have encountered (variables, iterative logic, conditional statements, and define functions). Students spent four days working on their projects then presented them to their peers on the final day of the initial phase of the study.

Table 3.2. A high level description of the 13 assignments given during the 5 week introductory portion of the course.

#	Assignment	Graphical or Text-based?	Concept
1	Quilt	Graphical	Introduction
2	Madlibs	Text-based	Variables
3	Tip Calculator	Text-based	Variables
4	Paint by Quadrant	Graphical	Conditional Logic
5	Movie Recommendation Engine	Text-based	Conditional Logic
6	Grade Ranger	Text-based	Conditional Logic
7	Guessing Game	Text-based	Iterative Logic
8	Radial Art	Graphical	Iterative Logic

9	Squiral	Graphical	Iterative Logic
10	Polygoner	Graphical	Procedures
11	Connect 4	Graphical	Procedures
12	Brick Wall	Graphical	Procedures
13	Final Project	Graphical	Summative Project

Phase Two: The To-Text Transition

The second phase of this study is intended to shed light on the question of how well each of the three introductory tools used in phase one prepared students for future computer science instruction in a more conventional text-based language. In this phase we followed students as they transition from the introductory programming environments used in the first five-weeks of class to the Java programming language, which is the language they will use for the remainder of the school year and also the language the school's AP Computer Science class. All students learned Java and used the same basic text editor for their programming assignments, regardless of what condition they were in during the first phase of the study. The choice for using a basic text editor that does not include common programming editor features like syntax highlighting or auto-formatting was the teacher's.

We followed the students through their first ten weeks in Java. The course follows the Java Concepts: Early Objects text book (Horstmann, 2012) which, as the name suggests, uses an objects-first approach. This means students encounter the concepts of objects and classes before conditional logic and loops. During the ten weeks of the Java portion of the study, students encountered basic input/output, variables, data types, creating objects, and calling functions. While there is not complete content overlap between the introductory curriculum and the first ten weeks in Java, there are concepts that were encountered in both, notably variables and procedures.

Methods and Data Collection

This study utilizes a mixed-method approach to answer the stated research questions.

This includes quantitative and qualitative methodologies as well as computational methods to analyze the large dataset of learner-authored materials. In this section, we breakdown each data source used, present our data collection schedule and describe what the data is and how and when it was gathered. We begin by discussing our quantitative data sources then continue with our qualitative and then computational data.

Quantitative Data Sources

The main quantitative data sources for this study are a series of attitudinal surveys and content assessments. The surveys were administered three times over the course of the study: (1) at the outset in week 1, (2) between phase 1 and phase 2 which is after students have completed the portion of the study where they will be using the introductory programming environments, and (3) after the conclusion of the first Java unit, which was roughly 10 weeks into the course. Each administering of the surveys helps us answer a different question. The initial set of data gives us a base line for each student and the classes as a whole. The second set allows us to measure the impact of the introductory programming environments, both within each condition as well as how they do relative to one another. The third data set will allow us to measure students' initial experiences learning to program in Java. This data set gives us insight into students' attitudes and whether or not confidence has shifted since moving to text-based programming as well as information about if and how concepts learned with the introductory tools are still salient after leaving those tools behind. Administering these two instruments multiple times over the course of the study gives us power to speak to the effects of the tools immediately, comparatively, as well as their lasting effects.

The surveys were administered online during class time on consecutive days so as to minimize testing fatigue. The attitudinal survey took students around 20 minutes and the content assessment took close to 25 minutes. The assessments were given at the same time for all three classes. The surveys were hosted on Google Forms and the responses were recorded in a Google spreadsheet. Students who were absent the day it was given were asked to take the survey outside of class time, although not all did as course credit could not be given for completion of the surveys under the agreement made with the school district.

Attitudinal Surveys

The attitudinal surveys are loosely based on materials used as part of the Georgia Computes initiative (Bruckman et al., 2009). The questions in this survey are designed to gain insight into a number of attitudinal and perceptual facets. The three versions of the survey were largely the same, with a few additional questions being added with the second and third administrations asking students to reflect on their experience in the class. The survey begins with 17 Likert scale (1-10) questions asking students about their confidence in taking the course, their excitement for the course, and their general perception of the field of computer science. The survey then continues with 9 short answer questions about motivations for taking the course, prior experience with programming and computer science culture, things they are excited to learn as part of the class, and some open-ended prompts about how they view the field of computer science. In the first administration of the assessment, we ask a number of questions about students' prior experience with computer science and technology. A full copy of the attitudinal survey can be found in Appendix B.

The Commutative Assessment

For this dissertation a customized content assessment was designed.

Across educational research broadly there is a recognized need for high quality and validated assessments, a position echoed in computer science education circles (Tew & Dorn, 2013).

Towards this end, a number of assessments have been developed and validated with the goal of improving our ability to evaluate and measure student learning across a variety of languages, environments, and contexts (Shuchi Grover, Cooper, & Pea, 2014). Related work has sought to define the process one follows to develop quality computer science assessments, beginning with identifying the goals of the assessment and the material to cover, through validating, piloting, and refining the instrument (Buffum et al., 2015). Additionally, new techniques are being developed and applied to programming assessments to improve accuracy and build confidence in new assessments (Sudol & Studer, 2010). One notable example of a rigorous, validated assessment is the Foundational CS1 assessment (FCS1) (Tew & Guzdial, 2011), which is a language independent instrument designed to decouple concepts from the language used to represent them. This makes it useful to learners regardless of the language used during instruction. This is in contrast to most validated programming assessments developed by testing boards, like the Advanced Placement (AP) CS exam and the A-level General Certificate of Education in Computing, both of which are currently designed for the Java language.

There are a growing number of projects working towards developing assessments for the blocks-based approach to programming that we are investigating herein. Much of this work looks to assess not programming specifically, but computational thinking more broadly (Shuchi Grover et al., 2014). For example, the Fairy assessment (Werner, Denner, Campe, & Kawamoto, 2012), designed for middle school aged learners, uses Alice and presents learners with partially completed, or buggy, programs that need to be fixed in order for in-world characters to

accomplish a specific task. In taking this approach, the Fairy assessment evaluates both comprehension (learners understanding of what a written program does) as well as gives learners a chance to problem solve, design and implement algorithmic solutions to assessment tasks. This design addresses the critique that process is often lost in conventional assessments of programming knowledge (Piech, Sahami, Koller, Cooper, & Blikstein, 2012). Another innovative assessment approach to computational thinking comes out of the Scalable Game Design group that developed an automated way to measure the frequency of computational thinking patterns in student-authored programs as a way to assess learning (Koh, Basawapatna, Nickerson, & Repenning, 2014). Despite this growing number of assessments that incorporate the blocks-based modality, prior to this dissertation study, there did not exist an assessment that could be used to comparatively evaluate student understanding across blocks-based and text-based modalities. In response to this, the Commutative Assessment was developed.

The central feature of the Commutative Assessment is that every question can be asked using one of three isomorphic programs. One version of the question presents the program in a textual form, the second uses the Pencil Code blocks representation, and the third is how the program would be written in Snap! (Harvey & Mönig, 2010), a widely used blocks-based programming environment that includes a larger set of visual cues in rendering its programs. Figure 3.1 shows the three different forms that a single question's program may take on the assessment. The assessment is called The Commutative Assessment to reflect the fact that the modality of the program included in each question can be swapped between administrations of the assessment. Details of the assessment are provided later in this section, after a discussion of the content areas covered by the Commutative Assessment.

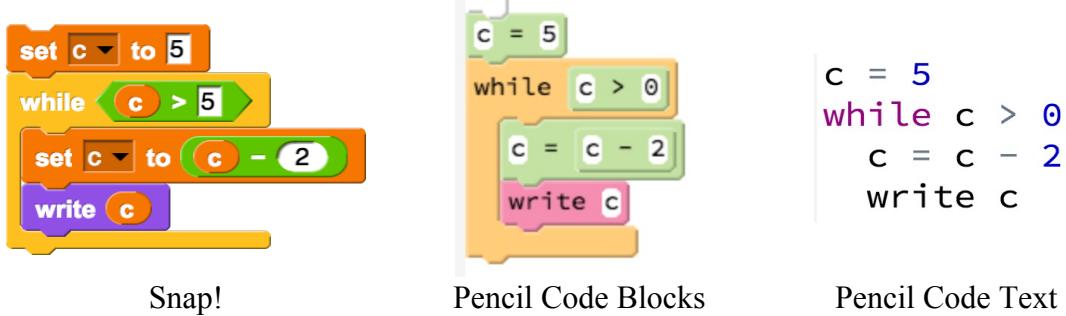


Figure 3.1. The three forms programs may take in the Commutative Assessment.

While a standard concept inventory for introductory computer science has yet to be established (Taylor et al., 2014), there are a set of concepts that are recognized as being central for such courses. The content assessments primarily draw on two resources in deciding what concepts to include in the assessments. The first is the recently released 2013 CS Curriculum (ACM/IEEE-CS Joint Task Force on Computing Curricula, 2013) that is meant to provide guidelines for university computer science departments. This curriculum breaks down the field of computer science into broad categories and recommends how much time should be committed to each category and at what point the material should be covered. The assessment focuses on concepts that were emphasized as being foundational early in a students' career. The second resource the assessment draws on is the work of Tew and Guzdial (2010, 2011) and their effort to create a validated CS 1 assessment. As part of this effort, they reviewed the contents of 12 introductory computer science textbooks along with other published curricula to establish a list of core CS1 concepts. Their final list consists of:

- Fundamentals (variables, assignment, etc.)
- Logical Operators
- Selection Statements (if/else)
- Definite Loops (for)
- Indefinite Loops (while)
- Arrays
- Function/method parameters
- Function/method return values

- Recursion
- Object-oriented Basics (class definition, method calls)

Informed by these two resources, the Commutative Assessment focuses on the subset of these concepts that could fit within our five week curriculum. This included the following five categories from Tew and Guzdial's work: fundamentals, selection statements, definite loops, indefinite loops⁶, and function/method parameters. Based on the review of the CS2013 Curriculum and what it emphasizes for introductory courses, as well as the desire to broaden the assessment beyond programming specifics, the Commutative Assessment also includes two additional content categories: program comprehension (interpreting the behavior of short programs), and algorithms (natural language descriptions of steps to be followed to solve a problem). Table 3.3 shows the final list of concepts included in the newly designed assessment and how it maps on to Tew & Guzdial (2010) and the CS2013 curriculum.

Table 3.3. The computer science concepts covered in our content assessment.

Commutative Assessment	Mapping to Tew & Guzdial (2010) Categorization	Mapping to CS 2013 Curriculum Category
Variables	Fundamentals	Fundamental Programming Concepts
Conditional Logic	Selection Statements	Fundamental Programming Concepts
Iterative Logic	Definite Loops; Indefinite Loops	Fundamental Programming Concepts
Functions	Function/method parameters; Function/method return values	Fundamental Programming Concepts
Program Comprehension	-	Development Methods
Algorithms	-	Algorithms and Design

⁶ Definite and indefinite loops were collapsed and taught together in our curriculum and at times grouped together in our analysis. In this document, references to iterative logic refer to both of these forms of looping structures.

The Commutative Assessment includes 30 questions, five in each content area. All of the questions are multiple choice or true/false and, with the exception of the algorithm questions, which take the form of a short piece of code that students are asked to interpret. The algorithm questions have plain text descriptions of a problem then ask the students questions about steps that need to be taken to solve that problem. Each question on the assessment has three possible programs that could be displayed. For each administration of the content assessment, there is roughly an equal number of questions asked in each modality, additionally, within each conceptual category, every modality is present at least once. As the assessment was given three times during the study, it ensured that every student answered every question in every modality. This means the questions in the second administration of the Commutative Assessment included the same set of questions, but used different modality than the version the student saw in the first administration. Likewise, the final time the learner took the assessment, each question was presented in the format he or she had not yet seen. Figure 3.2 shows a sample variable question from the assessment. When taking the assessment, students see only one version of the program.

What will be the value of x and y after this script is run?

(or)

(or)

$x = 10$
 $y = 5$
 $y = x$
 $x = x + 5$

A) x is equal to 15 and y is equal to 15
B) x is equal to 5 and y is equal to 10
C) x is equal to 15 and y is equal to 10
D) x is equal to “x + 5” and y is equal to “x”
E) x is equal to 10, 15 and y is equal to 10

Figure 3.2. A question from the Commutative Assessment.

The multiple choice answers were informed by misconceptions that have been identified in the literature (see appendix A of (Sorva, 2012) for a summary of misconceptions). The set of available choices includes the correct answer as well as responses drawn from the literature on misconceptions around variable assignment. Option A would be chosen by a student that holds the misconception that when one variable is assigned to another, the two values are linked and that whatever happens to one, happens to the other (du Boulay, 1986). If a student incorrectly thinks that a value gets passed from one variable to another (i.e. the variable does not retain its value if another variable is set to it), then the student would choose option B. Option D would be chosen by a student who thinks expressions do not get evaluated during assignment (Bayman & Mayer, 1983; Sorva, 2008). Finally, option E would be chosen by students who think that variables “remember” prior values (Doukakis, Grigoriadou, & Tsaganou, 2007; du Boulay, 1986). We also choose to write out “is equal to” instead of using an equals sign to be explicit about the meaning of the choices. Throughout the assessment we tried to follow this approach as much as possible to shed light on potential misconceptions conveyed or supported by the different modalities. Including responses drawn from the misconceptions literature is intended to help provide evidence for linking certain modalities with misconceptions about the concepts that are being demonstrated. The full Commutative Assessment can be found in Appendix C. Basic validity measures were run on the responses collected in the second year of the study and showed the assessment to have an acceptable reliability score across all items (Cronbach’s $\alpha = .80$).

It is important to note that while the goal of this assessment is to understand the effect of programming modality on learning, there are other factors complicating the issue, most notably, differences in the language itself. For example, in Figure 3.2, the syntax and keywords used in variable declaration and assignment are different between the two modalities, making the

difference between the two forms of the question more than just a shift in modality. This is a constant challenge with this work as a feature of the blocks-based modality is the ability to support more conversational and readable commands (Weintrop & Wilensky, 2015a, 2015c).

The design of the Commutative Assessments makes it possible to group the responses along a number of dimensions that can yield insight into the relationship between modality, tools, and emerging understanding. The dimensions include grouping responses by: condition (what tools the students used), representation used in the question (graphical vs. textual), concept (conditional logic, iterative logic, etc.), prior computer science experience), and various combinations of those groupings. The details of this analysis are presented later in the Findings chapters.

Qualitative Data Sources

A number of qualitative data sources were gathered as part of this study to compliment the quantitative data just discussed. These data sources included: classroom observations, semi-structured student and teacher interviews outside of class, and the collection of non-computational student generated artifacts.

The major qualitative data source for this study is semi-structured clinical interviews with students and the teacher. These interviews occurred outside of class time throughout the fifteen weeks of the study. For the student interviews, the researcher sat alongside the student either asking the student questions about their experiences in the class or having them think-aloud as they work activities designed to illicit specific types of thinking around computer science concepts. The goal of these interviews was to more deeply probe students' emerging understandings and identify if and how understandings are bound up with the representations they are using in the classroom. Three rounds of interviews with students were conducted. Each

set of interviews included students from all three conditions of the study. Three protocols and sets of activities were designed, one for each wave of the interviews. The interview protocols can be found in Appendix D. The first wave of interviews began at the start of the course, the second at the conclusion of the introductory tools portion of the study (weeks 5 and 6), and the third and final wave occur during the last two weeks of the study, after students have been programming in Java for 10 weeks. The interviews were recorded with screen capture software and the computer's on-board camera and microphone and serve as the primary data source for understanding the relationship between the representation used and students' emerging understandings. A total of 35 interviews were conducted. Table 3.4 provides information on the students that were interviewed.

Table 3.4. The 35 student interviews conducted during the study.

	Pre Interviews	Mid Interviews	Post Interviews
Blocks Condition	4 Interviews (2M, 2F)	4 Interviews (3M, 1F)	4 Interviews (3M, 1F)
Hybrid Condition	4 Interviews (4M)	3 Interviews (3M)	4 Interviews (4M)
Text Condition	4 Interviews (3M, 1F)	3 Interviews (2M, 1F)	5 Interviews (5M)

Two teacher interviews were also conducted: once after the conclusion of the first phase, and again at the end of the study. The teacher who taught all three sections of the class also participated in the pilot study, so an additional three interviews were conducted the previous year, which covered topics including background and personal pedagogy beliefs. The two teacher interviews were conducted outside of class during the teachers' preparation period. The interviews asked the teacher to reflect on the tools being used in her classes and to compare these classes to each other and to prior years when other introductory tools were used. We also discussed specific assignments and students in each of the classes. The goal of these interviews

was to draw on the teacher's expertise to gain insight into how she perceived the students were reacting to the different tools.

Over the course of the study, the classrooms were regularly observed. Each observation included field notes capturing what was happening during the course of the period. These observations were focused on student questions, teacher strategies, and characteristics of the class dynamic that are influenced by the programming environment. The final form of qualitative data we collected was non-computational student created artifacts. In particular, the teacher has students keep a journal over the course of the school year. In these journals the teacher has students write pseudo-code, respond to class prompts about different computer science concepts, and reflect on in-class activities. We gathered and recorded the content of these journals to serve as a further data source for understanding the relationship between how students understand concepts and how they compose solutions (in the form of pseudo-code) and the programming modality they have used in class.

Computational Data Sources

The third component of the data collection plan involved collecting snapshots of the programs students wrote throughout their time participating in the 15-week study. In taking this approach we developed computational data collection strategies and employed various computational methods to identify trends in how students developed programs and the unique characteristics of the programs they produced, looking for patterns across the three conditions.

Conducting this analysis was dependent on the ability to record every iteration of every program written by students participating in this study. Over the course of the 15-weeks, we recorded a snapshot of a student's program each time they compiled and/or ran it. As two different environments (and thus languages) were used during the study (Pencil.cc and Java), two

different sets of data collection tools were deployed. Both tools were built on top of the same data collection infrastructure. We will begin with a brief description of our data collection and storage backend before discussing our two unique logging instruments.

To gather and store information on student authored programs, we instrumented both the way learners compiled their Java programs as well as the Pencil.cc programming environment to push data about student actions and student programs to a remote web-server. In both Java and Pencil.cc, each logged event produced a call to a RESTful webservice that would handle the inbound request and parse and store the data it received in a server-side database. The data logging web service is build using the django framework and hosted on the Heroku platform. The service has a number of endpoints that our learning environments push data to depending on the learner-triggered event, which we will discuss in more detail in the next section. The collected data was then exported from the databases using custom actions written for django's administrative interface, which makes it easy to manipulate, organize, and selectively export data based on custom defined goals. To date, over 220,000 programming events have been logged using this system. In our analysis section, we will provide more specifics about the frequency and content of these logged events.

Pencil.cc Data Collection

All three versions of our introductory tools were built on top of the Pencil Code programming environment use the same data collection system. The customized version of Pencil Code used for this study (called Pencil.cc) was instrumented to log data about the state of the learners program at various times during the learners use of the tool. This was achieved by issuing an asynchronous HTTP request in the background of the programming environment (the browser) that posts the contents of the composed program along with other pertinent information

to a server that we control. Table 3.5 describes the different events that triggered a log message to be sent to our server. The last three events in Table 3.5 (all starting with the Block-drop prefix) were not logged in all conditions; Block-drop-addition was logged for Blocks and Hybrid while the last two were only logged in the Blocks condition. These provide additional insight into programming patterns but were not included in the text condition of the study as there are no blocks provided and thus, the events could not be triggered.

Table 3.5. The events that trigger a log event to be captured in Pencil.cc

Event Type	How the Event is Triggered
New	A new program is created
Load	An existing program is opened
Save	A program was saved by the user
Logout	The user exits Pencil.cc
Run	The user ran a program
Block-drop-addition	A block is added to the program
Block-drop-deletion	A block is removed from the program
Block-drop-floating	A block is dropped outside of the current program

Each log event included information about the user, the state of the environment and the contents of the program. Table 3.6 describes the content of each entry in our Pencil.cc log event table.

Table 3.6. The values stored for each logged event in the Pencil.cc environment.

Column Name	Description of data being stored
StudentID	A unique identifier for the author of the program
Assignment	The assignment the student is working on
Hostname	The url where the assignment can be found
ProjectName	The student defined name of the current project
TimeStamp	The time (to the millisecond) that the logging request was recorded
Condition	The condition of the student the student is in (blocks, text, hybrid)
EditorMode	If the program is rendered in blocks or text
PaletteVisible	If the blocks palette is visible to the user
EventType	The cause of the event to be logged (see Table 3.5 for possible values)
Program	The full contents of the program at the time of the log event

FloatingBlo cks	The value of any floating blocks ⁷ if any are present
ProjectHTML	Any user defined HTML in the program
ProjectCSS	Any user defined CSS in the program

Java Data Collection

As with Pencil.cc, the learners' Java programming environment was instrumented to log the contents of each program the learner wrote, along with additional environmental and learner data. Like with the introductory tools, this logging happens in the background of the environment and is transparent to the learner and thus does not interfere with classroom practice or the programming experience. Student programs are stored on the same remote web server as the Pencil.cc logs, although in a different table as a slightly different set of data is collected. For the Java portion of the study, students compile their programs using the command line `javac` call. To record these compilation events we developed a tool called JavaSeer that replaces the students' `javac` command with a script that wraps the compilation call with additional logic to capture the student program and compiler feedback. JavaSeer accomplishes this by creating an alias to the `javac` command line call within the terminal on the computers the students use. When students run `javac`, JavaSeer reads in the arguments the student passed to `javac`, which includes the list of files the student intends on compiling. Inside JavaSeer, `javac` is called with the same commands the student passed in, then records the output from `javac` and passes it back to the user as output. Additionally, JavaSeer reads in the contents of the files being compiled and sends them, along with the compilation output and other information to the JavaSeer server via an HTTP request. This whole process is invisible to the student. This

⁷ Floating blocks are blocks that have been added to the canvas but are not connected to any part of the program. This is the blocks-based equivalent to commenting out lines of a program, so they are present but not executed.

approach to logging student programs was informed by the Git Data Collection project and the work built as part of that effort (Danielak, 2014). Table 3.7 describes the data that we collect for each program via JavaSeer.

Table 3.7. Data recorded via JavaSeer, our automated data collection tool for command line compilation.

Column Name	Description of data being stored
StudentID	A unique identifier for the author of the program
JavacCall	The argument(s) passed to <code>javac</code> (which will be the list of file names)
TimeStamp	The time (to the millisecond) that the logging request was recorded
JavaProgram	The contents of the java files that are being complied
JavaCompilerOutput	The compiler output from the call to <code>javac</code>

Data Analysis Approach

A central component of this dissertation is the use of a mixed-methodological approach, which uses findings from one methodology to support and validate the findings of another. Having described the various types of data that were collected and the larger shape of the study, this chapter continues with a high level description of the analytic approach for each type of data collected. All of the methods pursued will be further discussed in more detail later in the findings chapter.

Quantitative

The design of the Commutative Assessment and the attitudinal surveys provide the ability to speak to a number of the research questions posed in this dissertation. By looking at student performance on different conceptual questions based on the modality they were answered in, we advance our understanding of the relationship between modality and conceptual understanding. By looking at differential performance on the assessment based on the introductory tool used by the participant, we can start to understand how those modality support learning and

understanding of different concepts. Additionally, because the surveys were given at three distinct points, we can look for learning gains over the introductory tool period, as well as, over the first ten weeks in Java. These content assessment findings are complemented by the attitudinal surveys, which allow for shifts in perceptions, confidence, and attitudes towards programming and computer science to be tracked. By looking across these two quantitative sources, we can see correlations across the various dimensions of the study, including attitudes, learning, and tool use.

When calculating differences across conditions or modalities, an analysis of variance (ANOVA) test is run, revealing if there is a statically significant difference among the groups. When significance is identified by the ANOVA test, a Tukey HSD post-hoc test is run, which reports the differences between each pair of groups that were included in the initial ANOVA test. In cases where the test is looking within a group across time periods on Likert scale questions (i.e. differences in responses to a specific questions between the Pre and Mid surveys), a Wilcoxon Signed Rank test is used. This test is appropriate given the ordinal nature of the Likert responses and because it is a non-parametric test used to compare paired samples. As these data are within the same population over time and there is no guarantee of an underlying normal distribution, this test is a better fit than alternatives like a t-test (which is parametric) or a Mann-Whitney U-test (which is appropriate but does not take advantage of the paired nature of the data being analyzed). When comparing across time periods on continuous data sets (i.e. student performances on the Commutative Assessment), paired t-tests are used given the structure of the data and the fact that the same students participated at each point in time. Similarly, on the content assessments, when looking for changes over time, an ANCOVA calculation is used, which allows for the comparison across the three groups while also adding the ability to control

from covariates. On these analyses, the covariate is students Pre scores when looking at Mid differences and Mid scores when looking at Post outcomes. The reason for this is to better attribute changes in outcomes to the tools, as opposed to prior differences.

Qualitative

The primary qualitative data source of the study are the pre, mid, and post interviews conducted with students from all three conditions and the free-response questions asked on the attitudinal surveys. The analysis of these data follows an iterative approach, beginning with content coding to understand the breadth of what is present and describe the various aspects of the topic being discussed that are attended to. Once this is complete, we then do a second pass over the interviews taking a grounded theory approach, employing both open and axial coding (Strauss & Corbin, 1994). The goal of open coding is to gain a sense of the overall features of the phenomenon under study, while axial coding is a more focused approach to coding qualitative data as it is informed by theory and previous findings, trying to evaluate if expected patterns or concepts are encountered. Given the relatively small number of interviews conducted (35) and responses to interview questions (~80), codes that emerged from these analyses were applied by a second coder to the complete dataset to calculate inter-rater reliability (IRR) measures. The procedure for calculating IRR was to calculate the Cohen's Kappa for each individual code then average those values across the given dataset. This approach was necessary as many of the codes applied were not mutually exclusive, so Kappa could not be calculated for the full set of codes together. IRR scores are reported in the text when the data is presented. After the data had been coded by the secondary coder, the two researchers met to resolve any differences found and the coding manuals were updated to reflect the agreed upon resolutions. The coding manuals for every systematic qualitative analysis conducted can be found in Appendix E.

Computational

A growing body of research is looking at the use of big data for studying student learning in increasingly nuanced and sophisticated ways (Baker & Yacef, 2009). In particular, this approach is productive for open-ended, constructionist, learning experiences (Berland, Baker, & Blikstein, 2014). Computational data analysis has been especially active in the domain of computer science education as the incremental building of programs by students lends itself well to the gathering of time series data and the imposing a relative uniformity to data making it possible to computationally compare across large data sets (Jadud & Henriksen, 2009). Recent work has looked at trends in how students write programs (specifically looking at frequency and size of incremental updates), intermediate program states the students visit as they work toward the final project, and the predictive nature of programming features on students' final course grades (Berland, Martin, Benton, Petrick Smith, & Davis, 2013; P Blikstein et al., 2014; Werner, McDowell, & Denner, 2013). The strategy for collecting and analyzing student projects in this dissertation follows this line of work.

One of the questions this dissertation seeks to answer is how programming differs between different modalities and if and how practices developed in one modality persist or fade when moving to another. As the dissertation will collect data on students programming in introductory graphical, hybrid, and textual introductory environments along with data from those same students programming in a conventional text-based programming environment, the data set will allow us to begin to answer these questions. The analytic approach that used in this work is similar to the approach used in Blikstein et al. (2014). The contribution to this approach is the inclusion of three different modalities and two programming languages that grant the ability to look across modality and language to compare various aspects of resulting programs. In

particular, analyses of the content of programs, the nature of how they were composed, and meta-information about the programs (like frequency complications) will all be included in our computational analysis. A more detailed account of the computational methodologies employed will be discussed in the findings section.

Setting and Participants

In this section we present information about the setting of the study and information about the various participants involved.

School information

This study was conducted at a large, urban, public high school in a Midwestern city, serving almost 4,000 students. The school is a selective enrollment institution, meaning students have to take an exam and qualify to attend. In this school district, students are selected based on their performance on the admissions test relative to other students from their school (as opposed to all other applicants). As a result, students attend this school from across the city and there is an equal representation of students from under-resourced middle schools and from schools in more affluent parts of the city. The student body is relatively diverse 44% Hispanic students, 33% White, 10% Asian, 9% Black, and 4% multiracial/other⁸. A majority of the students in the school (58.6%) come from economically disadvantaged households, with the student body also including a small number of second language learners (0.6%) and diverse learners⁹ (4.5%). Being a selective enrollment school, students are academically high achieving; reporting an average growth in student performance on the ACT in the 66th percentile national and a average

⁸ These numbers are reported annually by the district. Similar breakdowns for the study participants are given later in this section.

⁹ The diverse learners designation is often referred to as special education students or differentially-abled students.

ACT scores that was in the 95th percentile nationally. Ninety-five percent of freshmen at the school are on track to graduate at the end of their first year and reports 86.1% of graduates continued on to college.

The school was selected for a number of reasons. First and foremost is the fact that school has a well established and well supported computer science department. This had a number of beneficial aspects with respect to the study including the existence of three sections of the same class for the three-condition experimental design, sufficient technological capabilities to conduct the study, exceptional teachers willing to take on the challenges the accompanied teaching a class in the study, and a faculty and administration invested in this work and thus willing to commit class time and teacher resources to the project. The administration in this school has shown a great deal of support both for computer science education and for pursuing innovative educational programs making it especially well suited as a research site for this work. Second, an important aspect of the project was working with a diverse student population, including students from backgrounds that have not historically excelled in computer science, which meant working in a diverse, urban setting. Finally, the faculty of the computer science department at the school are active in the local computer science teachers association (CSTA) chapter and national educator-oriented computer science education research communities, thus providing an easy way to develop a relationship with the researchers as well as being well connected and well informed with respect to the latest trends, technologies, and curricula in the computer science education space.

Class and Classroom Information

The experiment was conducted in an existing Introduction to Programming course. Historically, the class spent the entire year teaching students the Java programming language.

The course did not follow a specific published curriculum, but instead was a combination of materials designed by the teachers and following the structure provided by the course textbook, Java Concept, Early Objects (Horstmann, 2012). In participating in the study, the class schedule was shifted back five weeks to allow the classes to go through the five-week introductory curriculum before moving to Java. The class is an elective and open to students from all four years of high school.

The classroom the study was conducted in was a recently renovated space designed to have an open flooring plan and a collaborative studio feel. Each class had around 30 students and each student was assigned a laptop computer which they used everyday for the duration of the study. Students sat in individual desks that were on wheels, that allowing them to move their desks around the room. As such, there were no assigned seats, no fixed seating arrangement, and students were given freedom to sit where they wanted. Most instruction was project based, with the teacher doing some lecturing from the front of the class using an interactive white board to display programs and lead instruction. Figure 3.3 is a picture of the classroom in which the study was conducted.



Figure 3.3. The classroom in which the study was conducted.

Teacher Participant Information

The design of the study put the researcher in an observer role and relied on the course teacher to lead all instruction, including during the experimental portion of the course. The same teacher taught all three sections of the course, allowing us to control for teacher effects. The teacher holds an undergraduate degree in technical education and corporate training. The year she participated in the study was her eighth year of teaching (third at this school). Along with the Introduction to Programming course, she has also taught Exploring Computer Science, AP Computer Science, Android Application Development, and Web Development. She was also one of two teachers that participated in the pilot study for this work, so was familiar with both the goals of the project as well as the high level course and tool design before agreeing to participate. The curriculum taught in the class was largely designed by the lead researcher but the lead teacher did contribute ideas, lessons, and customize the activities while teaching them.

Student Participant Information

The computer science course used for the study is an elective class but historically has attracted students from a variety of racial background and been taken by both male and female students. A total of 90 students participated in the study. The self reported racial breakdown of the participants was: 41% White, 27% Hispanic, 11% Asian, 11% Multiracial, and 10% Black. Relative to the larger student body, White students were overrepresented and Hispanic students were slightly underrepresented, with the other racial groups roughly matching the larger school demographics. The three classes in the study were comprised of 15 female students and 75 male students. This gender disparity is problematic, but as recruitment for the courses was out of the control of the researchers, there was little that could be done to address this¹⁰. Of the students participating in the study, almost half (47%) speak a language other than English in their households. Throughout this dissertation, where vignettes are presented, details about the participants will be shared, otherwise, all data reflects the full set of participants in this study.

At the outset of the study a pre survey was given to understand students existing computer science knowledge, their prior computing experience, and to gain insight into the motivations and goals of the students and why they chose to enroll in an introductory programming class. An analysis of early responses shows the large role the computing plays in the lives of learners. Figure 3.4 shows student responses to the prompt: How much time do you spend on a computer at home each day? The fact that this chart skews towards students spending more time versus less on computers outside of school (only 10 of the 87 respondents use a

¹⁰ It is important to note that this gender disparity is an issue the computer science department and the school administration are working to address through a number of initiatives.

computer for less than 1 hours a day outside of school), speaks the relatively large role of computation in their lives and their comfort with computers.

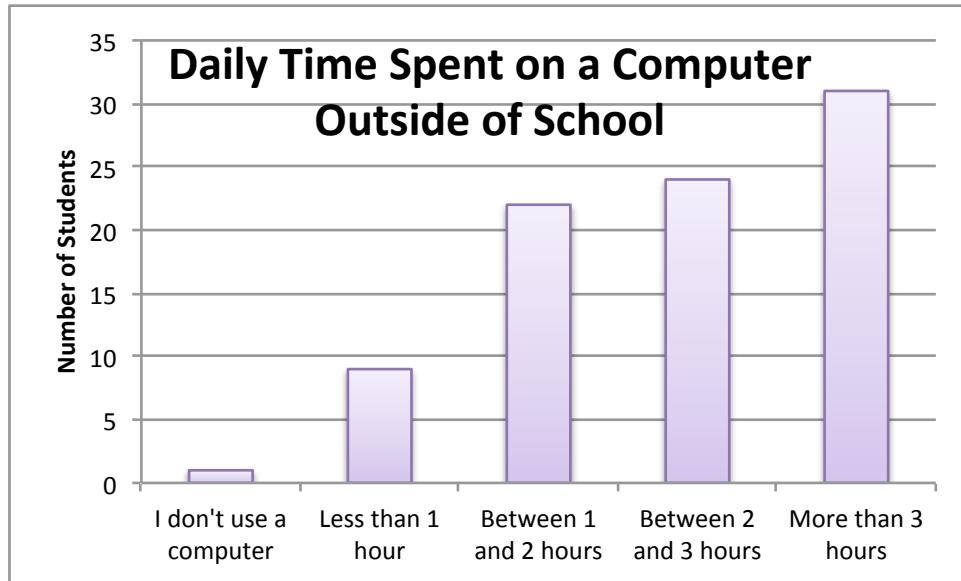


Figure 3.4. Time spent on a computer outside of school.

A similar predisposition and prior experience with programming can also be seen in the set of participants. Of the 88 students that filled out the pre attitudinal survey, just under half of the students (40) responded saying they had some prior computer science experience, ranging from taking a non-programming oriented introductory computer science class (30 students), to spending their free time over the summer learning a trendy web application framework (1 student), with only 17 of the 89 respondents saying they had never used any programming language before (including languages like HTML and Scratch). Thirty-two students reported some experience with a text-based programming language, with that number growing to 63 by including HTML, CSS and JavaScript (which was a single category). Students were also asked if they knew any professional programmers, of the 88 respondents, just under one-third of the students (29) knew a programmer personally, often a member of their family. This speaks to the number of first generation programmers in this learning community.

To get a sense of students initial motivations for taking the course, students were asked the following open-ended question on the pre-survey: Why are you taking this course? A coding scheme was developed based on student responses to form a larger categorization to accurately describe students' motivations for taking the course. Five major reasons were identified from the data, which are described in Table 3.8.

Table 3.8. Student responses to why they decided to enroll in the class.

Code	Description
Personal & Enjoyment	Response alludes to enjoyment (<i>I want to take it</i>), specific creative or personal goals (<i>I want to make video games</i>) or enjoyment (<i>It seems fun/I like computer science</i>)
Learn to Program	Response speaks specifically to learning to program being the goal itself (<i>I want to learn to program</i>)
Future Job/Major	Response specifically refers to getting a job or being able to pursue a specific career or major in university
Broadly Useful	Response alludes to the broad applicability of computer science (<i>Computer science opens a lot of doors</i>)
General Interest	Responses suggest a general interest in computers - but not a specific reference to wanting to learn to program

As students could write as much or little as they wanted, responses could be coded for multiple reasons. For example, one student responded: "*I am taking the course because it is a good skill to learn and I may want to get a job involving programming in the future.*" This response was coded for both Future Job/Major and Broadly Useful. Figure 3.5 shows the result from coding the full set of student responses. Unsurprisingly, given the course is called Introduction to Programming, the desire to learn to program was the most frequently cited reason for taking the class. Slightly more surprising is the fact that more students saw computer science as broadly useful than saw the course as a stepping-stone towards a specific job or pursuing a degree in computer science. The final thing important to point out in this data is the prevalence of student taking the course because of their enjoyment of programming or desire to learn to

program for personal reasons (like making video games, which was cited by four students).

Other reasons students gave for enrolling in the class include things like them wanting a challenge, they needed to add one more class in their schedule, or to try something new.

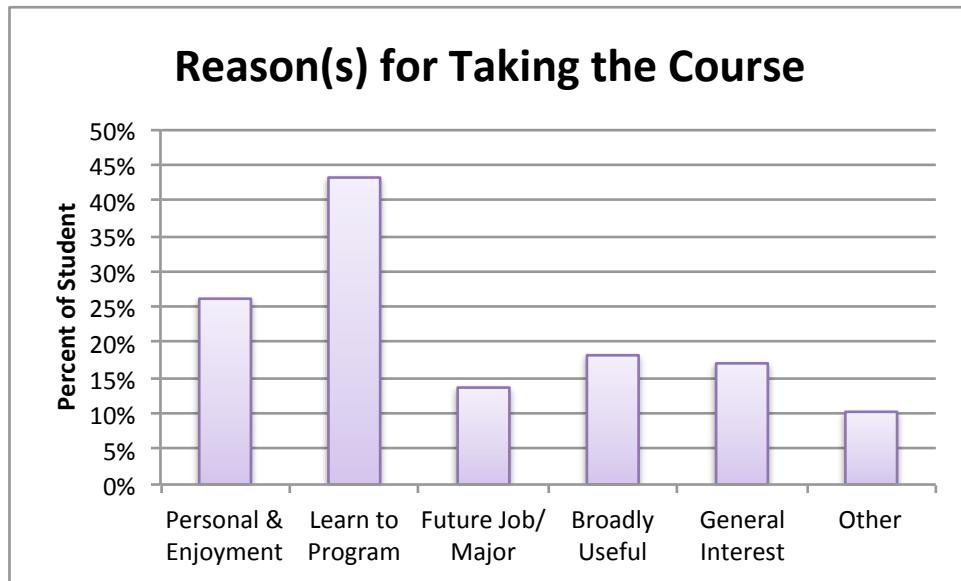


Figure 3.5. Student responses for why they enrolled in the course.

These free-response reactions given were echoed in the pre-interviews conducted at the outset of the study. For example, when asked why they were taking the course, students said things like: "*I think it will be helpful for college and some things I was looking into said that, to get in, it'd be helpful to get some CS background*" and "*I like learning things that could help me later, like maybe with a job that would use some of this stuff*". The responses weren't always so grounded in specifics, for example, another student made the general comment "*Computers are becoming the future, I wanted to take at least one computer course before going to college and I figured programming was a decent one.*" The opening of this quote nicely captures how many students viewed this course and the importance of computers, and technology more broadly, in their futures.

This chapter presented the various dimensions of the design of the study, including the settings in which it took place, the materials used, the data sources and assessment instruments, and details about the participants of the study. The one component of the study that was not given adequate description is the focus of the next chapter: the introductory programming environments that are at the heart of this dissertation.

4. Design

At the heart of this dissertation lies the design of introductory programming environments and an investigation into the impact of modality on novice programmers. The two iterations of this study relied on two different programming environments, with the second being informed by the design of, and findings from, the first. This chapter presents the two programming environments: Snappier! and Pencil.cc. Each environment is discussed with respect to the environment upon which it was based (Snap! in the case of Snappier! and Pencil Code for Pencil.cc), as well as the various features that were added to it in order to create the three conditions used for the studies. As Snappier! was used in the first year of the study, findings from that year are briefly reported with a focus on aspects of the environment that informed designed aspects of Pencil.cc. Limitations and potential future extensions for each environment are also discussed. But first, the chapter begins by laying the theoretical groundwork for our conceptualization of modality upon which this dissertation is built.

Modality

Before diving into the environments and their designs, it is important to be explicit about what is meant by the term modality. Given a semantics, a modality is a way of composing within that semantics. In this way, modality is not a characteristic of the representation alone, but also captures the relationship between the representation and how one uses it. In this way, a modality can be thought of as similar to the notion of affordance (Gibson, 1986; D. A Norman, 1990) in that it captures a characteristic of the interaction between an actor and the thing being acted upon. A named modality (like text-based or blocks-based) is a label given to the set of affordances provided by a given representation. Thus, it is possible for different representations

to be of the same modality (like Java and C++ both being text-based modality) or the same representation to support different modalities (like Pencil Code providing both a text-based and blocks-based modalities). While modalities are characteristics of representational systems, the environment in which the representation is situated also shapes the modality. For example, writing a Java program in a basic text editor is different than writing a Java program in an Integrated Development Environment designed for Java, which can also include code completion features, predefined code templates, code refactoring tools, and code navigation options. While the underlying representation is shared (both text-based Java code), the way the user interacts with the representation and the set of possible operations that can be executed differ. Colloquially, the term modality is often used to describe the representation itself (i.e. “the blocks-based modality”). This usage is consistent with the previous definition in so much as it serves as a proxy for the more nuanced and robust articulation of what modality means given above.

This dissertation is primarily concerned with two programming modalities, blocks-based programming and basic text-based programming¹¹. The basic text-based modality allows character-by-character interactions through the use of a keyboard, while the blocks-based modality provides a drag-and-drop form of composition. These two modalities can be seen in Figure 1.1 and will be discussed in much greater detail throughout this chapter. While text-based programming is most common programming modality and blocks-based programming is becoming increasingly popular, they are but two of a larger set of programming modalities. Looking across the history of computer science we can see a diversity of other programming

¹¹ The qualifier “basic” is added here to refer to a text-based modality that does not include advanced coding interactions like autocomplete or code refactoring. The text-based modality referred to throughout this manuscript requires that every character be manually entered.

modalities. Early programming with punch cards provides one example of a different modality (Figure 4.1a). In this form of programming, the author wrote instructions by using tools to physically punch holes in paper cards for the computer to read. The programming-by-demonstration paradigm is another alternative modality (D. C. Smith, Cypher, & Tesler, 2001). In environments that use this approach, such as KidSim (later renamed Cocoa then Creator) (D. C. Smith et al., 1994) and Agentsheets (Repenning et al., 2000), users program instructions by defining rules that are created by acting out the desired behavior, in this way programs are authored without interacting with an underlying textual grammar (Figure 4.1b). There are also more recent examples of new programming modalities. For example, Horn et al.'s (2013) computational literacy sticker books allow children to write programs by placing stickers in a specific order (Figure 4.1c). A second example of a recently emerging programming modality is the Ozobot (<http://ozobot.com>), which is a robot that can be programmed by drawing colored lines on a sheet of paper. The robot can either follow the pen-trails defined, or can roll over a sequence of different colored pen trails and interpret the colors as a set of instructions (Figure 4.1d).



Figure 4.1. Four examples of programming modalities beyond blocks-based and text-based: (a) punch cards, (b) programming by demonstration, (c) computational literacy sticker books, and (d) path-following.

Year One – Snappier!

Having laid the theoretical groundwork for thinking about modality, we move on to presenting the two sets of programming environments used in this dissertation. The three

environments for the first year of the study were all derived from the *Snap!* programming environment (Harvey & Möning, 2010). *Snap!* is a JavaScript-based implementation of a blocks-based programming language that can run in any modern web browser. Programs in *Snap!*, like other graphical blocks-based programming environments (such as *Scratch* and *Alice*), center around controlling on-screen avatars, called sprites, as they move around a stage. Programming in these environments largely entails giving behavior to sets of interacting sprites to create animations, games, and stories. The *Snap!* user interface (Figure 4.2) is broken down into five main sections: the Palette where language commands are organized, the Scripting Area where programs are composed, the Stage where programs are visually carried out, the Sprite Corral where the available set of sprites are displayed, and the Tool Bar which provides menus for additional capabilities (like saving and loading programs).

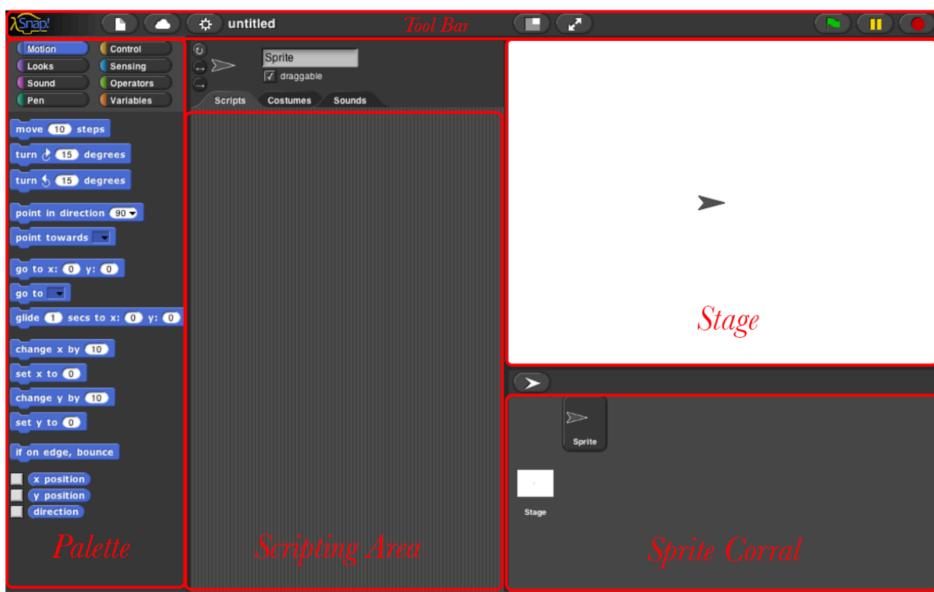


Figure 4.2. The *Snap!* interface with sections labeled.

The Three Versions of Snappier!

The *Snappier!* environment was built on top of the base *Snap!* functionality, thus has the same visual execution environment and, as much as possible, shares language semantics,

behavioral properties, and design aesthetics with *Snap!*. This means in *Snappier!* programs were visually executed on the Stage and largely centered around defining behaviors for sprites. *Snappier!* was designed to introduce text-based programming aspects into an otherwise completely blocks-based interface. To accomplish this, *Snappier!* added a mapping from each block in the *Snap!* palette to an equivalent and valid JavaScript call. This mapping was accomplished using *Snap!*'s Code Mapping capability, which allows the user to programmatically associate a block with a piece of code. For *Snappier!*, a full set of JavaScript mappings were defined and the user interface was restricted to prevent the user from overwriting these mappings. To make the mapping more linguistically direct, a series of helper functions were written to introduce more legible commands. Table 4.1 shows a subset of the mappings between the *Snap!* blocks and the functional JavaScript equivalent.

Table 4.1. A subset of the mappings between *Snap!* blocks and the JavaScript equivalent¹².

<i>Snap!</i> Block	JavaScript Equivalent
	<code>this.forward(10);</code>
	<code>this.turnRight(15);</code>
	<code>this.sayFor('Hello!', 2);</code>
	<code>for (var i = 0; i < 10; i++) { }</code>
	<code>if (x < 10) { } else { }</code>
	<code>function whenGreenFlagClicked() { } }</code>

This mapping served as the main addition to the *Snap!* codebase, along with supporting functionality that enabled learners to view, edit, and run code written in JavaScript. Using this

¹² In both columns, the images shown are taken directly from the *Snappier!* environment.

newly added functionality, three versions of *Snappier!* were created. The first version of Snappier! did not expose any of the new text-based functionality, making it equivalent to the Snap! environment; this version constituted the graphical condition of Snappier!. The second version of Snappier! gave the user the ability to right click on any blocks or script and see the JavaScript equivalent of the blocks-based program that had been written. This was called the Read-Only version of *Snappier!*. Figure 4.3 shows the interface for the Read-Only condition as well as a short program that can be viewed by the user. It is important to note that, in this version of Snappier!, it was not possible for the user to write any JavaScript.

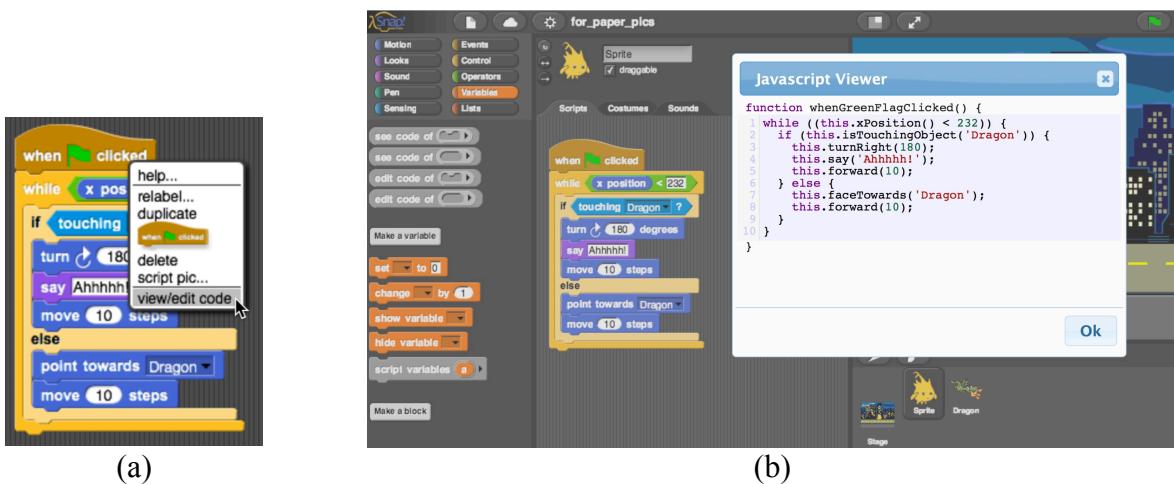


Figure 4.3. The context menu that users could use to open the JavaScript Viewer (a) and a picture showing the *Snappier!* environment with the JavaScript Viewer open (b).

The third version of Snappier! included the ability for the user to right-click and view the JavaScript behind any block or script and added the ability for the user to write their own JavaScript into new blocks. This was called the Read-Write version of Snappier! To support the user in writing new JavaScript, the environment was modified so that when a user defined a new block (akin to creating a new function in a conventional programming language), *Snappier!* presents the user with a code editing window (called the JavaScript Editor) for writing their own JavaScript. This is in contrast to defining the new block's behavior using blocks, as would

normally be the case. The JavaScript Editor and Viewer is an embedded instance of the CodeMirror library, which provides is a JavaScript-based code editor that supports various code editing features including basic error detection, syntax highlighting, and auto-formatting. Once the user was done writing the new JavaScript code, the block was saved and could then be incorporated into a script alongside any other block. When the execution thread came across a custom defined JavaScript block, the JavaScript inside the block would execute. In this way, users would write JavaScript code inside the blocks-based environment. Inside the JavaScript editor, users would have access to arguments that were passed in as well as globally scoped objects in the execution space. Figure 4.4 showing a picture of what it looks like to define your own custom block in *Snappier!*

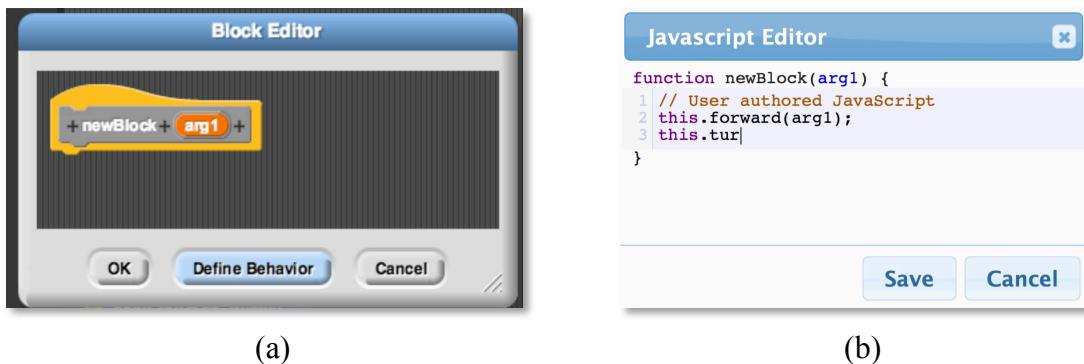


Figure 4.4. The Block Editor that allows the user to define the name and arguments of the new blocks (a) and the JavaScript Editor where the behavior of the block is defined (b).

These two modes of Snappier! (Read-Only and Read-Write), along with the default Graphical mode, constituted the three conditions used in the first iteration of the study. The next section of this chapter briefly reports on some of the findings from the pilot study, focusing specifically on the findings that informed the design of the programming tools in the second iteration of the study.

Findings from Year One with Snappier!

One of the analyses conducted on the data collected in the first year sought to understand how students perceived the blocks-based programming modality. Do they find it easy to use? If so, what do they attributed that ease-of-use to? What, if any, drawbacks do they see with respect to using blocks-based programming tools in a high school computer science class? These findings have been published elsewhere (Weintrop & Wilensky, 2015b), but are recounted below as the outcomes from this analysis informed the design of the hybrid environment used in the second year of the study. For an extended discussion of these findings and methodological details, please refer to the previously published paper.

Ease-of-Use

The first research question pursued was to understand if students thought blocks-based programming was easier than text-based programming, and if so why. To answer this question, data from the surveys administered at the midpoint and conclusion were analyzed. The surveys asked students to compare *Snappier!* with Java (either based on experience or expectation), with questions specifically asking what they viewed as the major difference between the two. The responses were then analyzed, identifying which answers attended to ease-of-use as contributing to the difference between *Snappier!* and Java. Of the 84 responses collected, more than half of students (58%) included ease-of-use as a major difference between the graphical and text-based environments. Table 4.2 shows the outcome of the coding of the responses that attended to ease-of-use as a difference. The subscript numbers in Table 4.2 show the breakdown by the three *Snappier!* conditions (Graphical, Read-Only, Read-Write).

Table 4.2. Student responses comparing Java to *Snappier!* - coded for ease-of-use of the environment.

Perception	Count (Graph/Read-only/Read-write)
Text-based Programming is Easier	4 (0/1/3)

Blocks-based Programming is Easier	42 (14/15/13)
Comparable Difficulty	2 (0/1/1)
Did not attend to Difficulty	41 (13/13/14)

In this analysis, care was taken to only include responses that clearly attended to a difference in difficulty between the two environments. For example, the response “[In Java] *there are no blocks to help out, it is basically done from scratch*” was coded as attending to ease-of-use, since the blocks “*help out*”, while the response: “*Java is more writing as if it was a language, while Snap! you use logic to put blocks together*” was not coded as attending to ease-of-use because the student did not make it clear that this difference made one environment easier than the other. While many responses required some interpretation, others were very clear on which environment they found easier, giving responses like: “*Learning Java is more complicated than Snap!*” and “*Java is much easier for me than Snap!*” Additionally, two students attended to ease-of-use, but specifically said the two modalities were comparable: “*one is hard and the other is equally as hard.*” These data show that students found the blocks-based programming approach of *Snappier!* to be easier than Java, thus supporting the general view of blocks-based tools being easier for novice programmers.

Reasons for Ease-of-Use

Having established that students perceive blocks-based interfaces as easier than text-based programming tools, the follow-up analysis sought to understand what features of blocks-based tools contribute to this perception. The reason for pursuing this question was to distill design principles for future hybrid tools. To answer this question, a first analysis was conducted using the pre and mid interviews to see what students initial impressions were of blocks-based interfaces compared to text-based alternatives. A secondary analysis to supplement these first findings was conducted using the short answer responses from the post survey. The emphasis of

this analysis was on how and when students attended to features of the blocks-based modality contributing to its ease-of-use.

The first aspect of the blocks-based tools that students identified as helpful was the descriptive, easy-to-read labels on the blocks. “*Well, I mean, if you can read it...for humans this looks better, it's easier to understand.*” Despite its looking less like a text editor when compared with the text-based code, a number of students viewed the blocks-based representation as closer to English than its text-based counterpart. “*With blocks, it's in English, it's like pretty, like, more easier to understand and read,*” a second student highlighted this difference, saying: “*Java is not in English it's in Java language, and the blocks are in English, it's easier to understand.*” A third student explained: “[the blocks] are basically a translation of what [the JavaScript] is doing, in, I guess, English for lack of better words. It is describing what [the JavaScript] is doing, but it's describing it in an English form...like a conversion.”

The second feature students identified that makes blocks-based programming easy is the visual nature of the blocks and the graphical cues that each block provides for how and where they can be used. Four of the nine students interviewed explicitly mentioned the shape of the blocks as being useful. For example, when an eighth grade student was asked why some blocks have rounded edges and others have diamond shaped edges, she explained that it was so “*the user knows that...they have a limited choice so that you don't make the mistake, because if all of [the blocks] were the same, it might not work. If [the block is] rounded or diagonal, they'll know the difference; they'll know that you can't put [a diamond block] in [an oval slot], it's like a puzzle.*” A second student echoed this fact when asked how he knew that Boolean blocks could be used with control structure blocks and numbers and that mathematical operators worked with motion blocks he explained: “*it's because of their outline; [the Boolean blocks shape] is the*

same as [the control blocks inputs] and then in motion, the [oval input] is the same as [the mathematical blocks]." The shape was identified as being useful to see how blocks fit inside each other, as well as how sequences of blocks could be built, which was helpful for making sense of the resulting behavior. "*When [the blocks] are attached to each other, you know that the first one is going to affect the ones underneath it...everything is connected and it's easier to understand what is going on...I guess it's more intuitive too, because you can see how they all connect.*"

Students said that these shape cues helped not only to see where blocks could be used, but also the larger idea of the importance of the sequence of commands, "*[the environment] teaches you that order is important.*"

A third advantage identified by students was how the act of composing a program was easier with blocks. This is in part due to the shape of the blocks discussed above, but also a product of a number of other features of the blocks-based modality. The first is that the act of dragging-and-dropping commands is easier and less error prone than having to type in commands character-by-character: "*If you type it, with like one word or one period or one something that's wrong it's going to mess everything up...it's just harder to write with the codes.*" Another student put it slightly differently saying: "*I like visualizing things more so with Snap!; it's a lot easier than having to type everything in,*" The student continued by saying how with text-based programming "*you have to be pretty precise with your punctuation, you have to type everything in.*" A third student succinctly put it, with blocks "*you don't end up making as much mistakes.*" Along with the ease of composing valid programs, a number of students highlight how blocks make it easier to tinker with a program. "*You get to play around with [blocks]...because if you do it with writing, you like, have to erase everything or like start all over. It's not as easy to change and make new things. With blocks, you can just drag them and*

change what it's going to do." This benefit can be seen when watching students compose programs, often taking a block or sets of blocks and putting them off to the side while trying new blocks in their script, only to ultimately reintroduce the removed blocks back into the script.

The final feature of blocks-based programming that emerged from the interviews was identified by four of the nine students and stems from the ease of finding blocks and understanding what they do through their organization within the programming environment. More specifically, how the blocks themselves alleviate the memorization that is required in text-based programming. "[The blocks] *kind of jog your memory, so you can see something and be like 'oh, I remember how to do that now', but with [text-based programming] you don't really have anything there to help you remember how to code something.*" As a second student put it: "[In JavaScript] *you need to like, know all the code words to draw something. Let's say you want to draw something, you need to type in a certain word to do that when in scratch you could just like, find the pen down block¹³ or something.*" This last point is critical; blocks-based environments provide an easy and organized way to browse all the available blocks, making it possible to use the blocks themselves as a source of ideas, as one student put it: "*everything is here that you can do.*" Another student focused on how easy it was to browse the available set of blocks as being a key reason blocks-based programming was easier, saying "*it's just because of the blocks and how they're separated into categories...so it's just much simpler to find the blocks and put them in to the pane.*" The utility of the organization and ease of browsing of the blocks was evident throughout the interviews. For example, during an interview with a grade ten student, when asked if he could draw a square on the screen, he successfully did so, but relied on the forever block in his program. When asked how he would change his program so it would

¹³ The pen down block is a block that asks the sprite to leave a trail behind it as it moves.

be possible to draw a second square next to the first, he opened, the Control category where looping blocks were stored, read through the blocks, and said “*I’m not really sure, I think it’s in the tab somewhere though,*” showing how the organization of the blocks within the environment can support novices in constructing programs.

After analyzing the interviews, two questions from the mid and post survey were coded for additional sources of ease-of-use. On the mid-study survey the question: “The thing that will be the most different about programming in Java compared to programming in Snap! is” was analyzed. Students answered this question after using Snap! for five weeks but before they had started working in Java. Five weeks later, after students had been working in Java, the same question was asked, shifting from the future tense to the present tense. A total of 85 students took the mid-study survey with one fewer student taking the final survey, resulting in a total of 169 responses. These two sets of responses were coded and categorized by what students chose to identify as the largest difference between the two modalities. Figure 4.5 shows student responses to these questions grouped by the difference identified, the point-in-time, and the version of *Snap!* the respondents used.

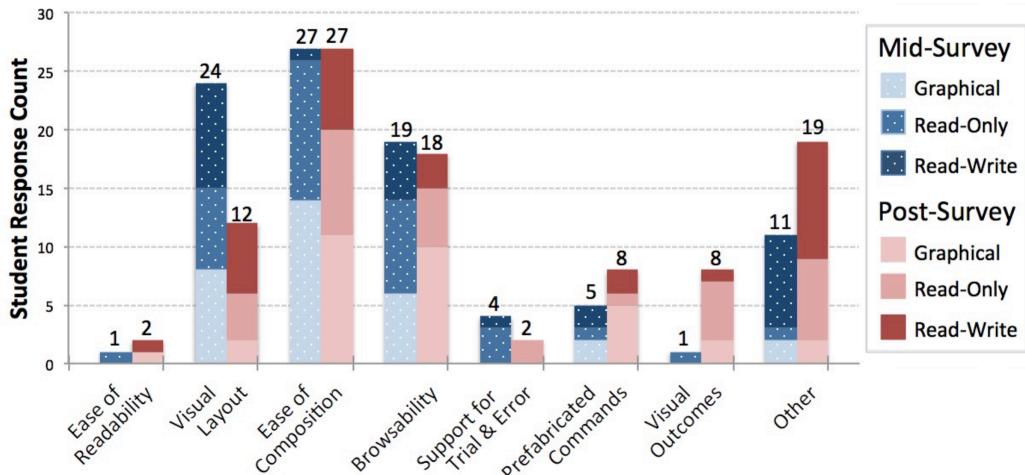


Figure 4.5. Student reported differences between *Snappier!* and Java given on the mid and post surveys.

This analysis revealed three new categories on top of the four themes that emerged during the interviews about what makes blocks-based programming easier. The new categories include the presence of prefabricated commands, the ease of trial-and-error programming in *Snappier!*, and the different types of programs authored in *Snappier!* versus Java. Table 4.3 provides examples of student responses for each category identified.

Table 4.3. Sample responses to the question having students compare Snappier! and Java

Category	Example Responses
Ease of Readability	<i>"The programming language will no longer be translated to English completely for a user to easily understand what is going on."</i> <i>"Snap! was easy to read."</i>
Visual Layout	<i>"There aren't going to be anymore colorful blocks."</i> <i>"I will have to code without having help from blocks."</i>
Ease of Composition	<i>"Actually having to type everything out instead of dragging and dropping."</i> <i>"Java is all hand typed while in Snap! you grab and drop blocks."</i>
Browsability	<i>"You will not have the blocks to aid you anymore and you will have to memorize and learn the Java script for everything you are trying to do."</i> <i>"Not feeling as restricted and having to think more because you don't have all the options in front of you."</i>
Support for Trial & Error	<i>"Java is not a trial-and-error program. If I make a mistake, then I must fix it on my own. There is no guessing involved, and I think I will have a really difficult time adapting to this process."</i>

	<i>"In Java, I will not be able to test out blocks and incorporate them and see if they work."</i>
Prefabricated Commands	<i>"There will be no set blocks that will provide you with pre made functions."</i> <i>"You do everything on your own without the help of preset blocks for the code, and you have to compile the file."</i>
Visual Outcomes	<i>"Java is more about having things such as text be displayed while Snap! was more about making sprites do things such as move or complete a goal etc."</i>

The first new category identified was how Java was not as conducive to the use of trial-and-error programming. This is particularly interesting as the trial-and-error approach is as valuable in text-based programming as in blocks-based, and nothing about text-based programming prevents the programmer from using the strategy. There are also potential consequences to thinking trial-and-error is not possible or not acceptable in text-based programming. Papert (1980) addresses this in his discussion of the difference between learners perceiving errors as wrong versus errors as fixable and how the errors-as-fixable orientation is a much more productive learning strategy. If the shift from blocks-based to text-based programming also carries with it a shift from the trial-and-error strategy being supported to it being viewed as impractical or even not possible, it is important that we as designers and educators be aware of this misconception and try and address it.

The second new category to emerge was the lack of pre-fabricated commands in text-based programming. Whereas a single block can do something in *Snappier!*, like move a sprite or ask a question, students thought that with text-based programming, the individual commands were more fine-grained, requiring more commands to be used to accomplish comparable behavior. While this is not necessarily true when calling APIs or other pre-defined functions, this reported difference highlights the perceived contrast in the size of atomic block commands and text-based language primitives. The final new category captures students identifying the visual

enactment of programs as being a major difference between Java and *Snappier!* This difference speaks less to the blocks versus textual nature of the languages themselves and more to the larger environments in which the programming is occurring. Interestingly, this was only identified by one student as a difference before the Java portion of the course, but was highlighted by eight students at the end of the study.

Drawbacks to Blocks-based Tools

Over the course of the ten-week pilot study, students identified a number of drawbacks to blocks-based programming. The data presented below were drawn from the same data sources as the previous section. Across this dataset, three drawbacks to programming in a block-based environment were raised. The first drawback to blocks-based programming students cited was that block-based programming was viewed as a less powerful programming technique compared to the text-based alternative. Power in this case refers to the set of things that are possible with the language. As one student said, with text-based programming “*you can do a lot more.*” A second student reiterated this point, saying: “*blocks are limiting, like you can't do everything you can with Java, I guess. There is not a block for everything.*” This comment is interesting as one could rebut that there is not a command for everything in Java either. The student who made this comment did not know how to program in Java, but nonetheless held the belief that the two representations were not equally powerful or expressive. Another student made these same points saying: “*In Java you can make it more complex than something you make in Snap! or Scratch.*” She then continued: “*I'm pretty sure there are going to be some things that are too big to put in blocks...too complex.*” This student viewed the blocks-based interface as a simplified version of Java, saying: “*I think what Snap! does it just takes the simpler things in Java and then turns them into blocks.*” This last statement is particularly interesting given that the available set

of primitives provided by *Snappier!* is largely a superset of the keywords reserved in Java, not the other way around. When asked why we chose to start the course with *Snappier!* before moving to Java, a grade ten student responded: “*to increase understanding of programming. I mean like, Snap! is an awesome program, but there is only so much you can learn in it. But in Java, you can like figure out how to do like, all the other stuff.*” When pressed, the student was unable to articulate what “*other stuff*” consisted of, but still, this reveals a perceived limitation of what can be accomplished with blocks-based programming environments. In the post survey, one student summed up the difference between Java and Snap! succinctly by saying of Java: “*there are more possibilities.*”

The second drawback brought up by a number of students was the time and number of blocks it takes to compose a program in the blocks-based interface compared to the text-based alternative. For example, when comparing Snap! to her previous experience making web pages, a 9th grade interviewee said: “*I know you have the variables [in Snap!] that you can edit and mess around with but sometimes that takes a lot of time, but HTML and CSS you can kind of get creative and quickly just type something in to do something different*”. This was reiterated by a second student who said: “*if you want a specific block and it's not there, you're going to have to put a lot of blocks together to make it do what you want it to do, and I think with JavaScript, it's just, like, one sentence I guess.*” While it is unclear what is meant by a “*sentence*” in JavaScript, this comment provides insight into how the student perceived text-based programming to be advantageous. Text being more concise was identified as not only useful for composing programs, but students also thought that the resulting shorter text-based programs could be easier to understand. “*It seems like when there is more blocks it's more confusing...when we did the games, we did a lot of, like a whole bunch of blocks, it was really hard to find where mistakes*

were. [Text-based programming] *seems easier when there is like a lot.*" During the five-week study, programs rarely exceeded the size of the screen the students were working on, but in this case, the students experience with longer blocks-based programs lead to the recognition that longer blocks-based programs can be difficult to manage.

The third and final drawback identified about the use of blocks-based tools is potentially the most damaging with respect to the effectiveness of their effectiveness when used in introductory programming courses for older learners. Some of the students we interviewed expressed concerns over the authenticity of blocks-based programming. Authenticity here refers to how closely the programming tool and practices adhere to conventional, non-educational programming contexts. As one student said: "*Java is actual code, while Snap! is something nobody will let you code in.*" This same point was made by another student who said: "*if we actually want to program something, we wouldn't have blocks.*" It is important to note that this view was not universally held. As part of the interview protocol, students were asked if they thought what they were doing in Snappier! constituted programming, to which every student answered in the affirmative. A number of students recognized blocks-based programming as being an introductory tool, giving responses like "*I think [blocks-based programming] is the same thing, just easier*" and "*I would say [blocks-based programming] is like beginners programming*". This suggests that even when perceived as potentially inauthentic, students still recognize the pedagogical utility of blocks-based tools. This drawback in particular seems like it is more likely to affect older learners who are eager to develop skills that can be used beyond the classroom, be it for a job or further computer science coursework.

Limitations of Snappier!

While *Snappier!* served as an informative first iteration, there are some limitations of the environment with respect to answering the stated research questions. First and foremost being that *Snappier!* lacked a full text condition. Even in the read-write condition, students did most of their program authoring using a drag-and-drop interaction, with the text-programming being limited to defining custom block behavior. Additionally, when students were doing the text-based programming, they could (and often did) first write the script with the blocks, before viewing the text version of that script and then copy/pasting that code into the custom block text-editor. This further limited the amount of text-based programming and supported an approach to text-based programming quite different than conventional text-based programming (defining the program in blocks then copy/pasting versus writing in text from scratch).

A second limitation of Snappier was that by restricting the text-based programming to be inside new block definitions, there are some programming activities that rarely, or never happened. For example, students in the text condition would never define new functions, since they were already inside a new function definition. Similarly, the act of defining new event-driven actions conceptually wouldn't make sense inside of a function as the event definition itself was already playing the role of defining how/when the code would be executed.

A third limitation of the *Snappier!* condition is that the hybrid form of programming that was supported still kept the two modalities separate. Yes, students were writing text programs in a blocks-based environment, but the act of composing the text instructions was essentially no different than using a normal text editor. This approach situates text-programming in a blocks-based world, but is a very limited exploration of the potential hybrid space.

A final limitation of *Snappier!* as an environment for answering the stated research question is the nature of the types of programs that are best supported by the environment.

Snappier!, following the lead set by Scratch and Alice, primarily supports writing programs with graphical outcomes in the form of on-screen sprites moving around and interacting with each other. This is quite different than the types of outputs normally supporting by textual languages, which often have outputs that are textual or numerical (like printing words to the screen or doing mathematical calculations). The lack of these types of outcomes is important as students often blur the lines between the language, the modality, and the larger programming environment in which the programming is situated (Weintrop & Wilensky, 2015b). This means that comparing the textual programming of sprites moving around in the world to the textual programming of number and text manipulations results in a larger gap than just the specifics of the programming language. Despite this set of limitations, the *Snappier!* pilot study proved to be a fruitful first iteration and lessons learned from the study helped inform the design of the programming environment used in the second iteration of the study.

Year Two – Pencil.cc

For the second year of the study, a new programming environment was developed. Starting with the *Pencil Code* environment (Bau et al., 2015), *Pencil.cc* was created. *Pencil.cc* defined three distinct modes of interaction: text only, blocks-only, and a hybrid blocks-text interface. The three distinct versions of *Pencil.cc* will be discussed below, but first is a description of *Pencil Code*. All of the aspects of the environment discussed in this section were present for all three versions of *Pencil.cc* unless otherwise stated.

Pencil Code is an online tool for learning to program. Its interface (Figure 4.6) is split into two panes: on the left is a dual-modality programming editor that supports both visual blocks and textual code, while the right side is a webpage that can visually run the program the

learner creates. The dual modality feature was the primary reason Pencil Code was chosen as the base environment for this study.

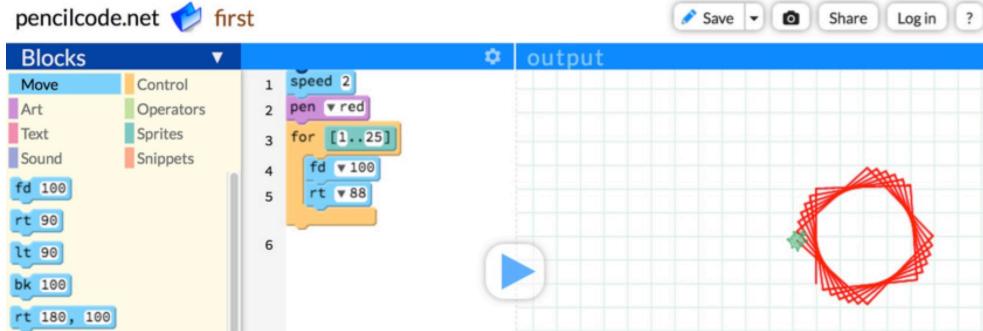


Figure 4.6. Pencil Code's Interface with the coding area on the left and program output on the right.

Students can click a button (Figure 4.7b) and see their programs transition between the blocks-based (Figure 4.7a) and text-based modality (Figure 4.7c). The two modalities are completely isomorphic, meaning any program written in one modality can be rendered in the other, and the user can freely move back and forth between the two modalities as they choose. The ability for the user to shift between the two modalities was suppressed in Pencil.cc as part of the study design (this will be further described in the sections that follow).

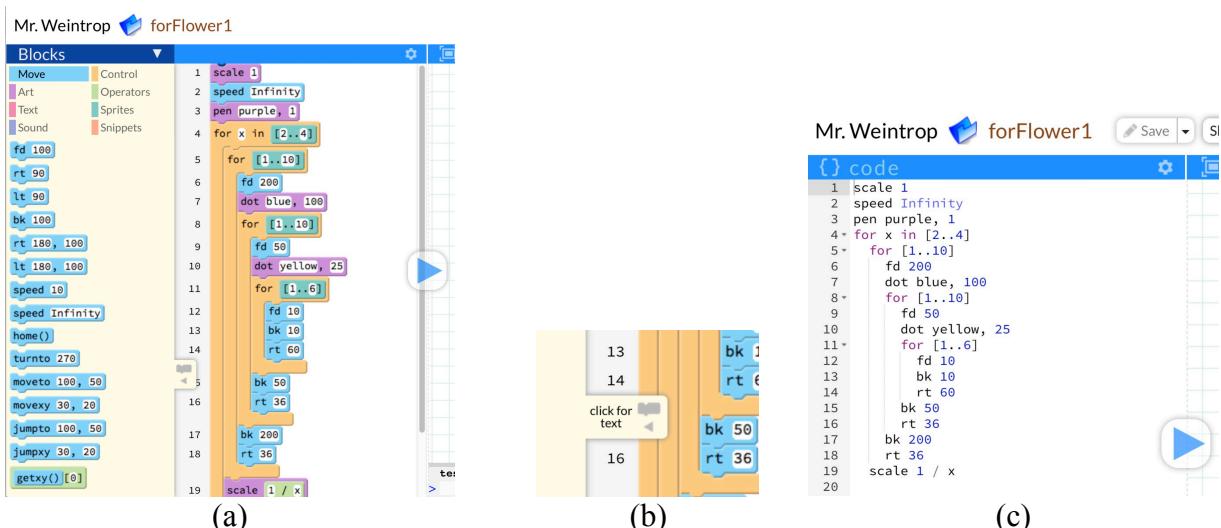


Figure 4.7. Pencil Code's two modalities: (a) Blocks and (c) Text, with a button presented to the user (b) that allows her to move back and forth between the two.

Pencil Code embeds all student work in the Web: every student project is actually JavaScript on an HTML page, with an accessible URL that can be linked to, run, and embedded on all modern browsers. Pencil Code is a "high ceiling" learning environment that it is careful to avoid placing artificial barriers around the learner.

Pencil Code as an environment supports a number of programming languages. For this study, the basic language of choice was CoffeeScript. This was chosen as it is syntactically lightweight and also sufficiently different from Java; thus students in the text condition will still experience some transition difficulty when moving from Pencil.cc to Java in the sixth week of the study. Pencil Code was designed to encourage two main types of programming activities. In the spirit of the Logo language, traditional coding concepts such as loops, conditionals and functions can be exercised by creating turtle graphics drawing programs starting from a single line of code such as `fd 100`. At the same time, real-world applications can be created by building webpages with HTML images, buttons, animation and music, that will appear no different to a visitor to the page than any other website online. This means programs can be written that output text or numbers onto the screen in a form that is akin to writing programs that output characters in a terminal.

Pencil Code differs from similar introductory coding environments in three main ways. First, unlike offline programming tools such as Python, Java, C, or Alice, it is a fully cloud-based online environment that does not tie the student to a specific device. The editor runs in a browser, and students save, edit, share, and publish their work online, incorporating the Web as a resource. Second, unlike traditional learn-to-code online courses such as those offered by Codecademy, it is designed to be welcoming to the timid beginner. Pencil Code draws design

lessons from block-based environments and provides visual primitives that give concrete and immediate feedback. Finally, unlike limited sandboxes such as *Scratch*, *Snap!* or *code.org*, it is an open-ended high-ceiling environment that allows unrestricted use of CoffeeScript, jQuery, and web resources. Collectively, these characteristics create a compelling introductory programming environment while also supporting key features at the heart of this dissertation, namely the ability to support both a fully textual and a fully graphical programming interface.

Pencil.cc adds a few additional features to the Pencil Code interface. The first, and most prominent to the user, is the addition of the Quick Reference menu. When a user hovers over the Quick Reference menu (Figure 4.8a), they are shown a series of topics related to programming in Pencil Code, grouped in the same high level categories as the blocks (i.e. Move has Curves and Speed options, Art has Colors and Pens). When a learner clicks on a menu option, an overlay appears (Figure 4.8b) giving instructions on how to use that aspect of Pencil Code, including examples that can be run. The Quick Reference was added specifically to provide embedded scaffolds for the text-only condition, so those students wouldn't be fully reliant on the teacher for guidance.

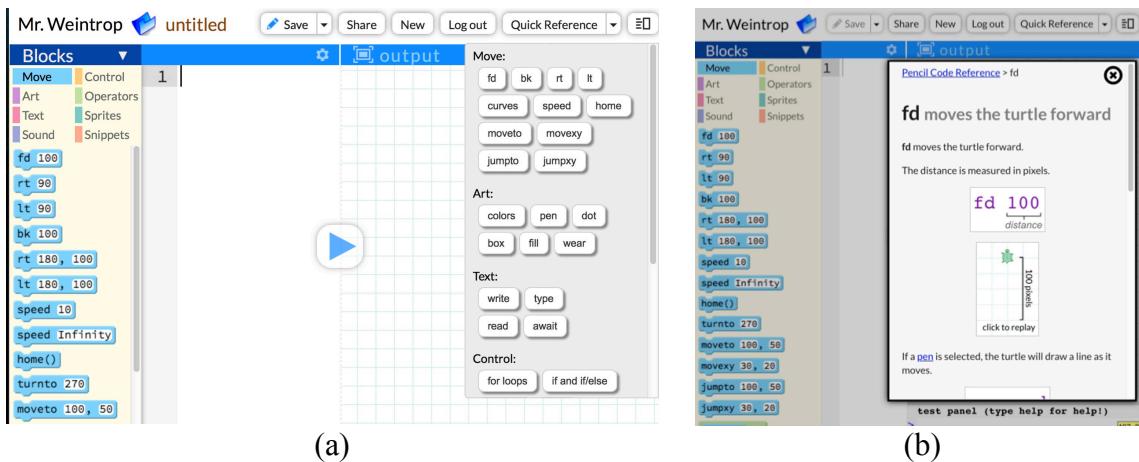


Figure 4.8. Pencil.cc's Quick Reference feature.

Other added features to Pencil.cc include added instrumentation in support of data collection as the learner interacts with the environment and the removal of the buttons that allow the users to switch between modalities. This was added as part of the experimental design as participants are only able to see and use one modality. The hybrid version of pencil code, which is discussed in more detail in the next section, was also a new addition, although much of the implementation work for that feature was done by developers at Code.org as part of their App Lab environment. The final new feature to Pencil.cc was a login page (Figure 4.9) that was used to ensure learners saw the correct version of the environment, as well as to serve as a place to put other study-related materials like surveys and consent materials.

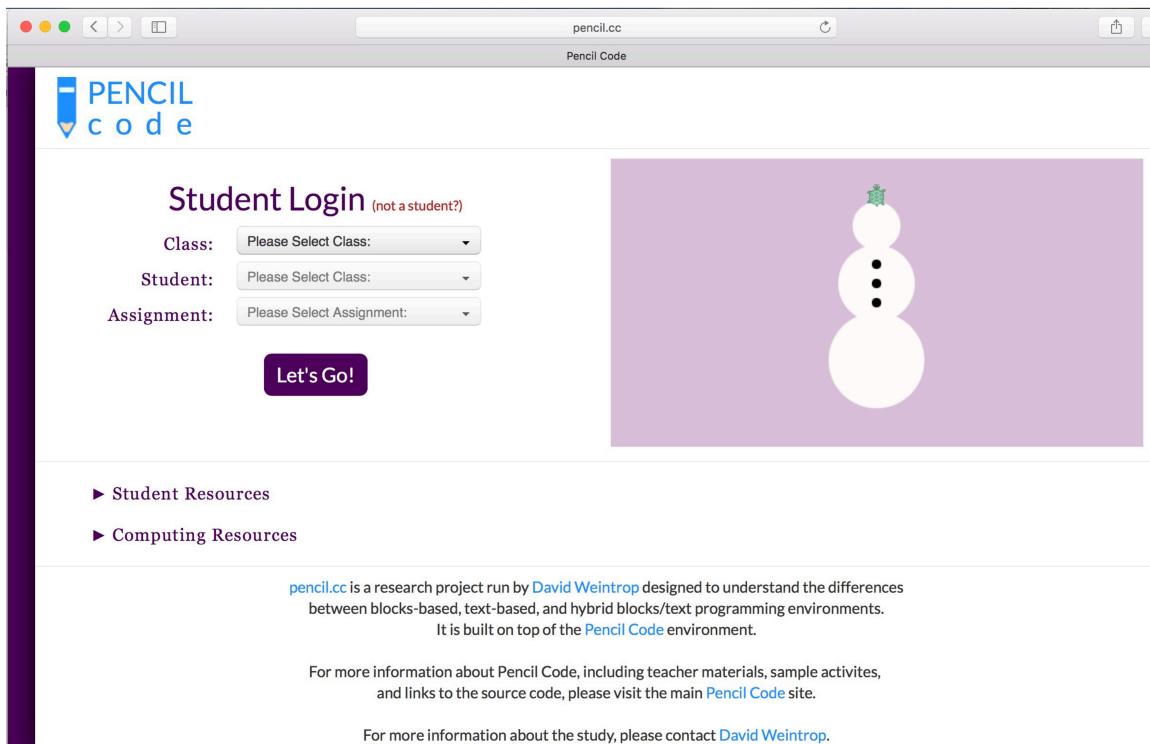


Figure 4.9. The Pencil.cc login page.

The Three Versions of Pencil.cc

The previous chapter laid out the three-condition experimental design used in this study. These three conditions are based on the three versions of Pencil.cc used across the three classes: blocks-only, text-only, and hybrid blocks/text.

Blocks-Only Pencil.cc

As the name suggests, in the blocks-only version of Pencil.cc, students were only able to view and compose programs in the blocks-based modality (Figure 4.7a). It is still possible to click on a block and type commands into the editor (either as arguments, like how far you want your turtle to move for a `forward` command, or by hitting return and starting to type, which upon completion of your typing, the editor will parse into blocks). The blocks-only version of Pencil.cc includes many of the features identified by learners in year one of the study, such as the browsability of blocks in the palette and the ease of composition through the drag-and-drop interface. It is also important to mention that, in the blocks-interface, users can hover over the blocks to get a short description of their behavior thus providing additional in-editor scaffolds to go along with the previously mentioned Quick Reference feature.

Text-Only Pencil.cc

The text-only version of Pencil.cc had students exclusively use the text interface (Figure 4.7c) and, thus, never saw the blocks-based feature of the programming environment. In this condition, students had to type all of their commands in manually and had to rely on the Quick Reference for any embedded help with respect to the command available and the syntax for them. The text editor does include syntax highlighting as well as basic compile-time error checking (this took the form of a red X to the left of the line number when students typed invalid commands). There has been some research on how novices parse compiler warning and error

messages (Hartmann, MacDougall, Brandt, & Klemmer, 2010; Nienaltowski, Pedroni, & Meyer, 2008), but Pencil Code does not follow these recommendation, instead using a relatively standard approach to displaying error messages taken by many editors. While such additions could be useful for Pencil Code's text editor, it is beyond the scope of the proposed study.

Hybrid Blocks-Text Pencil.cc

The third condition is a hybrid blocks/text interface that is a first attempt to answer the third stated research question on the design space between blocks-based and text-based introductory programming environments. The approach is to have learners still program using the text editor while providing the blocks-palette. Figure 4.10 shows Pencil.cc's hybrid interface.

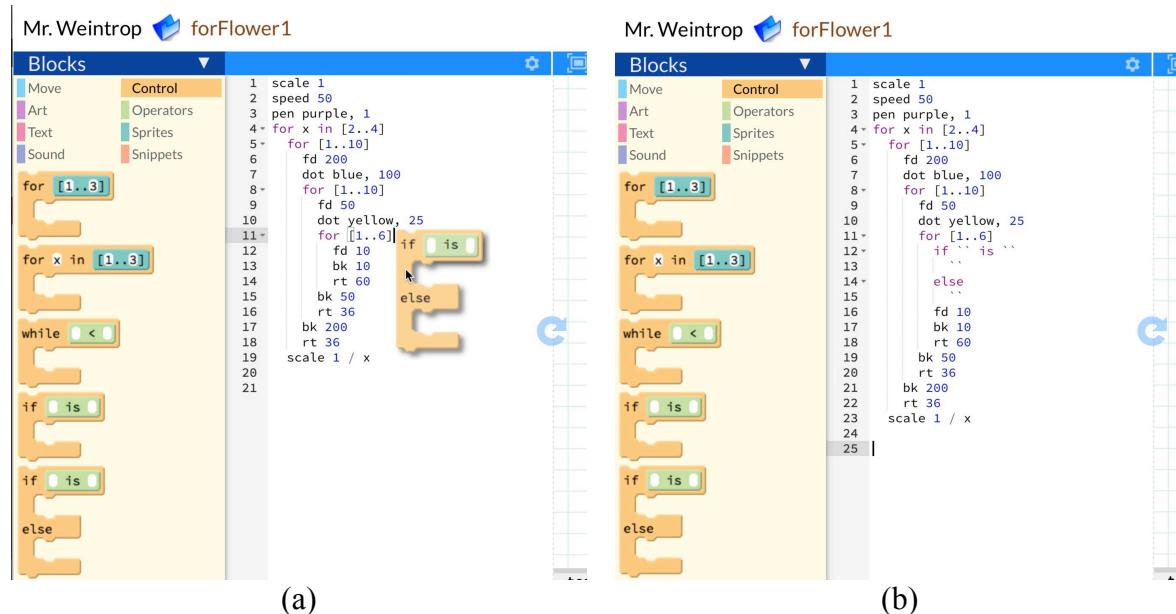


Figure 4.10. Pencil.cc's hybrid blocks/text interface. The left image (a) shows how learners can drag-drop blocks into the text editor; the right image (b) shows the results.

This hybrid approach was informed by findings from the first year of the study. In asking students to reflect on why Snappier! was easier-to-use than Java, high school students attended to a number of aspects of the blocks-based interface that are preserved in this hybrid approach. This

includes features such as browsability, drag-and-drop composition, and pre-fabricated commands. Similarly, the conceptual grouping of commands and the ability to hover-over blocks to see how they are used are other supports that have been identified as helping novice users. At the same time, the text-editor interface tries to address some of the drawbacks identified by learners in blocks-based tools, such as perceived inauthenticity and issues with blocks-based environment being less powerful or slower than text-based alternatives. As will be discussed in the findings chapters, users took advantage of the hybrid interface in a various ways, sometime relying on the drag-and-drop approach, other times typing in instructions with the keyboard.

Limitations of Pencil.cc

While the intention of the design of the three versions of Pencil.cc was to understand the various affordances and drawbacks associated with blocks-based and text-based programming, each of the three interfaces has limitations that narrow the scope of the claims that can be made from this study.

Blocks-Only Pencil.cc Limitations

A challenge of studying the affordances of blocks-based languages is the fact that there are so many features of blocks-based tools that contribute to users successful interactions with the tool (Weintrop & Wilensky, 2015a). While Pencil.cc captures many of those aspects (like the drag-and-drop composition mechanism, and the visual representation of blocks), there are some common features of blocks-based environments not captured by Pencil.cc. Figure 4.11 shows the same script implemented in Snap! and Pencil.cc, demonstrating a number of these differences.



Figure 4.11. Renderings of the same script in Snap! and Pencil.cc.

One difference between Pencil.cc and other blocks-based environments is that Pencil.cc does not provide the same level of visual cues that other blocks-based tools do. For example, in Snap!, predicates have a diamond shape and slots that expect predicates to match that shape, while in Pencil.cc predicates have the same shape as numerical or string blocks. This same difference exists with defining new functions and the shape the function call takes, and is potentially problematic as this feature has been linked to supporting conceptual understanding of function calls (Weintrop & Wilensky, 2015b). *Snap!* and *Scratch* also use a more diverse color palette and take better advantage of the natural language capabilities of blocks. Due to the isomorphic text-to-block relationship, Pencil.cc is constrained by the compiler in terms of how it presents commands.

A second difference between Pencil.cc and other tools is that in Pencil.cc, only a single script executes while other tools offer the ability to execute scripts placed anywhere on the two dimensional canvas, a feature found to be productive for learners (Weintrop & Wilensky, 2016b). This constrains the user to programming in a single, vertical dimension, which is true of text-based programming and some blocks-based tools (like *Alice*), but not true in other blocks-based tools like *Scratch* or *Snap!*. This feature is a central mechanism in the event-based model used by tools like *Scratch*, where “hat” blocks can be used to link up scripts with various events like user inputs, sprite events, or broadcast messages.

A final limitation of the blocks interface is that it is still possible to compose commands in text. If a user presses the return key when the cursor is inside one of the text slots, a new blank block will appear where the user can type in a command that is parsed into a block afterwards. While this is a powerful feature of the environment, it starts to blur the lines between the fully blocks-based condition the study had intended and the hybrid block/text condition.

Hybrid Blocks/Text Pencil.cc Limitations

While the hybrid mode of Pencil.cc brings together a number of productive aspects of the blocks-based interface with the text-editor, there are some limitations to this mode that leave room for improvement. One major drawback is that, once a command has been added to the program (either by drag-and-drop or by typing), from that point forward it loses its blocks-based affordances, and thus is seen as text and can no longer be drag-and-dropped the way blocks can in fully blocks-based environments. This is a general feature of text-based editors and only recently have new approaches been introduced to address this, with BlueJ 3's frame-based editor being a prominent example of this approach (Kölling et al., 2015).

A second limitation of this hybrid implementation is that there are some design issues that have yet to be completely ironed out. Notably, when a block is dragged out that does not have an obvious default value (like the terms being compared in an `if` statement or the body of a `while` loop), it is unclear how the editor should depict these empty ‘slots’. Pencil.cc’s solution was to display two tic marks (‘`’) in place of the empty slot, but this introduced confusion around where arguments should go inside of these tic marks or if they should be replaced (which is the correct behavior). This was a source of confusion for many learners and will be revisited in future versions of this hybrid editor.

The final limitation of the hybrid interface is that there are many possible design directions that can be pursued, with this design being but one of those. Other directions include frame-based editing (Kölling et al., 2015), editors that allow users to move between different modalities (Bau et al., 2015; Matsuzawa et al., 2015), or provide text-based inputs for blocks-based editors (Mönig, Ohshima, & Maloney, 2015).

Text-Only Pencil.cc Limitations

Just like with the blocks and hybrid interfaces, there are also limitations to the text-only condition of this study. The first is that, while there are some basic complication and runtime error supports in the form of red Xs in the margins and some user-friendly messages in the case of runtime errors, the error handling left much to be desired relative to other more fully-featured development environments. This is left as a possible direction for future improvements in the text mode. A second drawback of the text condition is that the Quick Reference menu, which provides the in-editor scaffolds for helping users learn what is possible and provides syntax supports, did not have the exact same coverage as did the blocks palette. For example, the blocks palette includes a *Snippets* category that provides short scripts to do basic things like have the turtle follow the mouse, or are a keyboard listener to respond to user input. While all these things can be implemented in the text mode, they were not included in the Quick Reference menu as a block of code that could easily be copied and pasted into the editor in the same way that could be done in the blocks or hybrid conditions.

This chapter presented the various design aspects of the environments used in this dissertation. Having laid out the research questions, reviewed relevant literature, and presented the study and environments designs, the next chapter finally gets to the good stuff: the findings.

5. Attitudes and Perceptions

The first research question this dissertation is investigating is the comparative affordances and drawbacks of the different programming modalities: Blocks, Text, and the Hybrid blocks/text interface. This chapter focuses explicitly on the attitudinal and perceptual differences across the three conditions (blocks-based, text-based, and hybrid blocks/text) in this dissertation study. It begins with an analysis of students' initial perceptions of the introductory programming environment and the modalities used. The goal of this portion of the chapter is to understand high school learners' perceptions of the three modalities before and after they use them. This includes a discussion of perceived design affordances and drawbacks of the three modalities. An analysis of students' perceptions of the different programming interfaces with respect to authenticity, enjoyment, and usefulness is presented next, followed by an analysis of the Pre, Mid, and Post attitudinal survey responses given. This section includes an analysis of within-student shifts over time on Likert questions, trying to understand how confidence, enjoyment, perceptions of programming, and interest in computer science change based on modality. It also looks at comparative change between the three modalities. The chapter concludes with a discussion of the attitudinal and perceptual findings presented. As a reminder the three classes were taught by the same teacher, followed the same curriculum, spent the same time-on-task, and had roughly the same number of students (30 students in the Blocks class, 31 in the Hybrid section, and 32 in Text condition).

Incoming Perceptions and Initial Reactions to Introductory Environments

This section looks at students' incoming perceptions of modalities and what it means to program. It begins by exploring students' initial expectations for the class and the assumption of

a text-based programming experience. It then reports findings into why students think the classes chose to use the various modalities it did. In asking this question, we fill in another aspect of student perceptions of different modalities as it relates to learning and pedagogy. Finally, this section presents students reactions to learning to programming in a given modality; looking across the three conditions to understand what students attend to based on different modalities.

During the first week of the study, four students from each condition were interviewed. As part of this interview, students were shown the version of Pencil.cc they were going to be using for the next five weeks. This sections presents data from these pre-interviews, revealing students' incoming perceptions of programming as well as their initial reactions to the modality they would be using for the next five weeks of class.

Assumption of a Text-Driven Experience

One thing that became clear early in the interviews was that students entered the study perceiving programming as a text-based activity -- and that Pencil.cc was not exactly what they had expected. This was true of students across all three conditions. For example, one student from the Text condition said: "*I watch a lot of CSI and Criminal Minds and I thought it would be more characters and underscores and very intense coding, instead of just like making the turtle move. But this is cool.*" This gap between the initial perception of what the course would be like and CoffeeScript, the language used in Pencil.cc, could also be seen from students in the Hybrid condition: "*[Pencil.cc] is a little different, I thought we'd mess more with brackets but I do like it because it's helping me.*" This comment is interesting because the Hybrid condition uses a text-based editor, so this perceived difference is not due to any environmental factors, but instead due to the choice of using the syntactically light CoffeeScript programming language, which does not have brackets. Students in the Blocks condition picked up on this difference, and unsurprisingly

cited the drag-and-drop feature as part of what contributed to the gap between what they expected in the class and what they were doing: "*I wasn't really thinking about dragging things to make something*", when this student was asked what she did expect, she continued: "*like typing something to do something, but not having set things already there.*"

It is important to note that not all students were surprised to see a non-text-only first programming environment. As one student from the Hybrid condition said: "*I knew we'd probably start with something simple, like this or Scratch, because, from what I hear, we'd probably start with one of these to get into the language first, so yeah, I figured it'd be something relatively simple like this.*" It is not too surprising that at least some students knew to expect an introductory environment that incorporated visual component, in part due to the frequency of prior computer science experience across the classes, and also due to the growth in popularity and awareness of blocks-based programming environments like Scratch and Code.org's Hour of Code activities. From these quotes we see that students at the beginning of the school year already have some set of expectations about what programming looks like and what to expect with respect to language features (like brackets) and visual presentation/modality.

Why Use Non-Professional, Introductory Programming Environments

In open coding students' initial reactions to the use of Pencil.cc in an Introductory programming course that teaches Java, two main themes emerged: that they thought that introductory tools can lay the foundation upon which Java can build and that introductory environments are easier and friendlier than their fully-featured professional counterparts. This analysis includes students attending to features specific to modality as well as other, more general, aspects of Pencil.cc and its use as a programming environment designed for beginners. This broader lens is included to help gain a fuller understanding of the how modality is situated

within the larger *webbing* (Noss & Hoyles, 1996) of the programming environment.

Additionally, this discussion is included in response to the challenges of isolating modality from the larger programming context, which at times is possible, but in the eyes of the learner is often blurred with the larger programming environment.

Laying a Foundation for Future Learning

One of the more frequently cited reasons for starting with an introductory environment like Pencil.cc given by students across the three conditions was the ability for the environments to lay an effective foundation for future learning. Numerous students verbalized this view, for example, one student said: “*it's a good foundation for us rookies to start. This is a beginners class and this helps teach me the very basics.*” In this view, students also cited how Pencil.cc would prepare them for shifting to Java, for example students also said things like: “*I think it's because it's easier to learn on something that is a little less advanced and more like, it's a good start, then once we know the commands and everything, we can move on to Java*” and “[Pencil.cc] is kind of like practice, it gets you ready for [Java], because I'm pretty sure this is way easier than what we're going to be doing later in the year, just getting us ready for what we're going to be doing.” Another student said that Pencil.cc has the “*basic structure to help you learn other codes in the future, like c++.*” With these quotes we see students recognizing the temporary nature of the introductory environment and its use as laying a foundation for future learning. In viewing introductory programming environments in this way, some students also distinguished them as something different from the tools and languages they would later be using. For example, another student said: “[Pencil.cc] could form the basis of programming, but it's just basic stuff, not like professional or anything.” So even in praising the environment, this

student saw a potential drawback with it in the form of its inauthenticity with respect to what was viewed as “*professional*” programming.

The recognition of Pencil.cc serving as a launch pad for future learning shows some sophistication on the part of the students in that they recognized similarities across languages and that some concepts and practices are universal across programming languages. Likewise, the different modalities did not interfere with students making this connection. This can be seen in one student’s response from the Hybrid condition who responded to the question of why the class was starting with this specific programming interface by saying “*probably just to build up those fundamental things, you got to know, like variables, that's always going to be in any language or like, algorithms, you need that no matter what language you transition to. Just like basic stuff, even though each language probably has it's own pros and cons, these are just stuff that are always going to be a constant.*”

There is also some evidence that the Hybrid condition further supported this perception of laying the foundation but in a more accessible way. For example, one student, while looking at the hybrid interface, responded to the question of why the class started with Pencil.cc by saying “*possibly so that kids can get a feel of the syntax and understand like how to put things together, [it] helps me understand like how specific I should be, or how exactly what I want to type.*” The expression “*get a feel for*” and “*put things together*”, suggests that the learner sees the relationship between the dragging-and-dropping that can happen in the hybrid interface and the long-term goal of programming in an all text environment.

Ease and Friendliness

The second reason cited for the use of programming environments designed for novices at the start of the year was due to the perception that it would provide an easier entry into the

world of programming. When a student in the Blocks condition was asked if Pencil.cc was what he expected, he responded: “*I had no idea what to expect. This is definitely more, I'd say this is friendlier, than what I expected it to be.*” A number of factors are cited for this perceived friendliness and ease-of-use. For example, one student in the Text condition attended to the visual outcomes of Pencil.cc, saying “[Pencil.cc] is easier...it's more clear what is happening...you get to see the immediate action of your code on the screen.” Another students in the text condition echoed the importance of the immediate visual outcome: “*you can see the results immediately by pressing the play button.*” As will be shown later in this chapter, visual outcomes was a salient feature of the three versions of Pencil.cc used, but was cited far less frequently than other features, including those related to modality.

Students in the graphical conditions keyed in on different features of the introductory environment. For example, when a student in the Blocks condition was asked why we chose to start with *Pencil.cc*, she responded: “[Pencil.cc] is easier, if you want to go forward, that's already there for you, so you don't have to type it out. Everything is kind of already there, so you just, guess and check sort of, so if something doesn't work out, you need to try something else, so like if the number 100 doesn't work how you want it to, you could do like, 200.” When asked if this same approach could be used in Java, she responded “yes, but maybe not as easy though.” Here a number of aspects of the blocks modality she was looking at are cited as contributing to the ease of use, including the pre-fabricated blocks and the ease of guess-and-check, two features identified in year-one of the study that were intentionally retained in the Hybrid condition.

Students in conditions where the blocks palette was present also cited the visual and graphical aspects of the blocks-based modality as contributing to the ease of getting started, saying things like “*it's simple, easier to understand, maybe to get people engaged because it's*

colorful, and got the game aspect" and "You can just mouse over and it'll tell you what you can do. You know the commands 'cause it's right there, for the most part." Collectively, these aspects echo the previous analysis in year one, where a variety of reasons were given for the use of specially designed introductory environments.

Perceived Affordances and Limitations of Pencil.cc and the Three Modalities

In the first year of the study, students were asked to compare the three versions of the introductory programming environment used with the Java language. This question was asked at the midpoint and conclusion of the study. The results of this analysis were presented in the Chapter 4 and summarized in Figure 4.5. In the first year of the study, little attention in the analysis was paid to differences between the three conditions (Graphical, Read-only, and Read-write) due to the overall similarity between the three environments.

In the second iteration of the study, there was significantly more difference between the Blocks, Text, and Hybrid conditions, making a side-by-side comparison by condition more fruitful and relevant for the research questions being pursued in this dissertation. For the second iteration of this analysis, student responses to open-ended survey questions were open coded looking for students attending to various features of Pencil.cc. Figure 5.1 shows a summary of student responses to the question: "The thing that will be the most different about programming in Java compared to programming in Pencil.cc is." It is important to remember that when students were asked this question, the students will envision either the blocks-based, text-based, or hybrid version of Pencil.cc based on the version they used for the first five weeks of the school year. This was asked on the mid-survey, after students had spent five weeks working in Pencil.cc, but had not yet used Java. The finalized coding manual used to code these responses, along with an example of a response from each category can be found in Appendix E. The

responses in this and the next figure were coded by two researchers. Cohen's κ was run to determine agreement and consistency of the application of these codes, and found there to be agreement between the coders, $\kappa = .80$, all differences were resolved through discussion¹⁴.

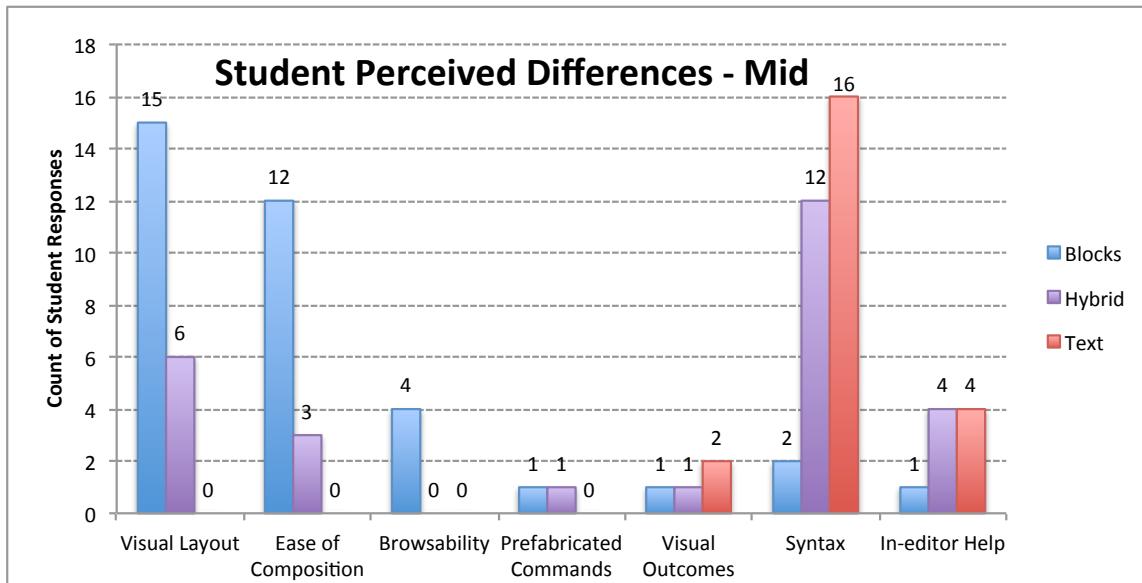


Figure 5.1. Student reported differences between Pencil.cc and Java at the midpoint of the study.

There are a few interesting things that stand out in this chart. First is the difference in features identified by the Blocks condition (blue columns) compared to the Text condition (red columns), and how the Hybrid condition (purple columns) frequencies often land between them. Students in the Blocks condition identified the Visual Layout and the Ease of Composition as the two most salient differences, with Browsability and Syntax sharing the position of third most frequently cited differences. Students in the Text condition, on the other hand, overwhelming identified Syntax as the most distinct difference, with In-editor Help being the second most oft-cited difference. The focus on syntax was particularly common among more

¹⁴ Note some of the Cohen's κ 's that are reported in this dissertation are below the conventional .80 threshold. This is due to the relative infrequency of some of the codes and the fact that not all codes are mutually exclusive, thus providing smaller distribution of codes, which were then aggregated together. Cohen's κ is known to not handle skewed and sparse datasets particularly well (Feinstein & Cicchetti, 1990), so the relatively low values are not viewed as problematic.

advanced students, as one student said in a later interview: “*In my personal agenda i was focusing on syntax, Making sure everything would work well and then I got to see how each code works.*” The fact that a student came in with a personal agenda to learn syntax speaks to where his attention lay early in the course. No students in the Text condition cited either of the two most popular differences from the Blocks condition. This serves as evidence for the salience of modality in learners’ perceptions of introductory environments. The Hybrid condition, seeing both the blocks palette and the text editor, cited both blocks-centric features (like the Ease of Composition) and more text-centric differences (Syntax) in their responses, never identifying a feature more frequently than either the Blocks or Text groups. Interestingly, no students in the Hybrid condition cited Browsability as a major difference between Pencil.cc and Java, this is surprising given that part of the motivation for the specific form of hybrid interface chosen was based on the utility of the blocks palette to support browsing and relax the need for the user to memorize the set of available commands. This does not mean that students did not use this feature, as we can see some evidence of its utility in responses coded for other categories like In-editor Help, but that the blocks palette provided other supports or was just not the most salient difference. An example of non-Browsability support provided by the blocks palette can be seen in this student response: “[Java] *will not have blocks and captions that can help me identify my codes and what errors I made in my program.*” In this response, the student is attending to the fact that she can hover the cursor over a block in the palette and get a brief description of what the command does. In this way, she is highlighting an affordance of the blocks-palette that is not related to the ability to browse the full set of commands available in the language.

Returning to Figure 4.5 and comparing it to Figure 5.1, it is also interesting to note the disappearance of two categories that were identified in the first year of the study: Ease of

Readability and Support for Trial & Error. The disappearance of the Readability category is not that surprising given the semantics of Pencil.cc are taken from CoffeeScript, so do not have the natural language feel that *Snappier!* had (e.g., the `set x to 10` *Snappier!* command becomes `var x = 10` in *Pencil.cc*). The fact that Support for Trial & Error was not cited is a little more surprising and not as easily explained. One possible explanation could be that unlike Snappier!, in Pencil.cc, the user cannot click on a block or a subscript to run it independently from the main program. This ability to run smaller scripts or commands directly from the blocks palette contributed to the larger trial-and-error approach and was not supported in Pencil.cc. It is important to note that this explanation draws not on a feature of the modality, but instead a characteristics of the environment in which the blocks were situated.

The second year also saw the emergence of two new categories, Syntax, which dominated responses from students in the Text condition, and In-editor Help, which was used to capture student responses that attended to the Quick Reference menu or the ability to hover over a block to get information about a block's behavior (two features that were not present in *Snappier!*). The inclusion of features of the editor is interesting as it blurs the line between a language (like Java) and a larger programming environment (like Pencil.cc), disentangling these two and how students conceptualize this relationship was a challenge throughout this analysis and is a planned avenue of future research.

At the conclusion of the 15-week study, after students had worked in Java for 10 weeks, they were again asked to reflect on the differences between Java and Pencil.cc, this time, the open response question that the students responded to was: “The thing that is the most different between Pencil.cc and Java is:”, the results of which are presented in Figure 5.2 below. Again, the responses were coded by two researchers and Cohen’s κ was run to determine agreement and

consistency of the application of the codes. There was found to be moderate agreement

between the coders, $\kappa = .68$, all differences were resolved through discussion.

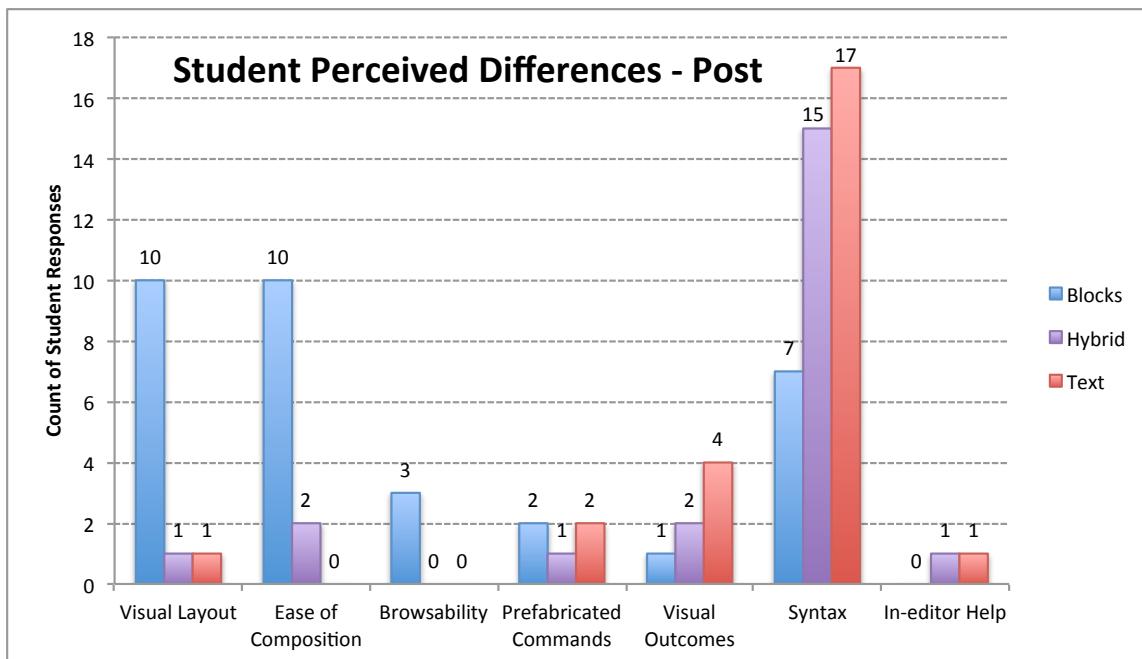


Figure 5.2. Student reported differences between Pencil.cc and Java at the conclusion of the study.

There are a few things to note about these results relative to the responses from the midpoint survey shown in Figure 5.1. First, is that the pattern is largely the same across the three condition, with Blocks responses trending toward features of the Blocks themselves (the left-most categories) while the Text condition mainly cited syntactic differences between the environments, with the Hybrid student responses again living between the two. A second shift to note is the significant decrease in the number of students that cited In-editor Help as being the most salient difference between Java and the their experiences in Pencil.cc. On the Mid survey, 10% of responses referenced this difference, on the Post, that number fell to only two students. There are a number of potential explanations for this including: the increased salience of other differences between the environments, the fact that ten-weeks had elapsed since using Pencil.cc

so students may have forgotten these features of the environment, or a growing recognition of the difference between a programming language (like Java) and the larger programming environment in which the language is used (Pencil.cc in this case). In responding to this question, learners no longer attended to features peripheral to the language itself. The comparison between Mid and Post responses also shows a shift for students in the Hybrid condition away from visual features as being the salient difference towards Syntax, the difference most frequently cited by students in the Text condition. In other words, after students spent time working in Java the salience of Syntax as a difference grew among learners from the Hybrid condition. One thing that is important to mention is that in some cases, when students are referring to syntax, they mean more than just semicolons and keywords (i.e. what is conventionally covered by the term syntax). In a Mid interview, one student explained the importance of syntax by saying: “*So it's not just knowing how to make the syntax correct, but knowing what your syntax is.*” This prompted the interviewer to ask the student what he meant when he said syntax, to which he responded: “*knowing when to use an if/else condition, using a for loop, a while loop.*” This lead the interviewer to respond: “*oh, so by syntax you mean more than just semi colons and curly braces*” to which the student responded with a nod of his head. This is important to note as it introduces a layer of complexity to the notion of syntax and that it cannot be assumed that the learner is talking only about punctuation. A final thing to note in this analysis is the appearance of responses from the Text condition in the Prefabricated Commands category. It is only two responses, but it is interesting given that none of the 30 respondents cited this difference on the Mid survey.

In the first year of the study, there were a number of students who cited drawbacks and limitations of the *Snappier!* environment. These critiques fell into three broad categories:

Inauthenticity, Less Powerful, and Slower Authoring. In conducting the same analysis in year two, we found less evidence of students taking issue with Pencil.cc. In analyzing the responses to the differences between the environments (which was part of the data corpus for the year one analysis), we find only three students who attended to drawbacks of Pencil.cc. Two of these responses came from students in the hybrid condition with one coming from the Text condition, which means no students in the Blocks conditions raised concerns. Two of the three responses talk about how Java is more authentic, saying “*There will be more actual coding involved as opposed to using predetermined blocks of code.*” The other limitation cited for Pencil.cc is that the environment is only used for drawing, saying: “*for Pencil.cc, all you can really do is draw.*” This limited list is in stark contrast to the longer, and more elaborated drawbacks identified in year one. There are a number of possible explanations for this, including the Pencil.cc interface being seen as more authentic and having a higher ceiling, the fact that all of the Pencil.cc commands are valid CoffeeScript, so have the feel of more conventional programming languages (i.e. are not natural language), or that the activity of writing programs that produce actual websites that can be linked to and shared engendered a sense of authenticity that was lacking in *Snappier!* A more careful analysis of these perceptions of Pencil.cc is explored in the next section.

Perceptions of Introductory Programming Environments by Modality

Trying to understand students’ perceptions of the three versions of *Pencil.cc* used in the study requires looking to a number of data sources as there are many facets to how the tool can be perceived. For example, students had perceptions of Pencil.cc with respect to utility, enjoyment, authenticity, and effectiveness. As this dissertation is trying to broadly understand the impact of the modality used to introduce learners to programming, the analysis looks across

these different dimensions. This section looks specifically at students' perceptions of the introductory programming environment as it relates to their own learning and if and how programming in the modality they used matches their view of authentic programming practices. In later chapters, a similar analysis will be presented looking at whether or not students viewed their time with the introductory environment as productive with respect to the goal of learning to program in Java. After looking at perceptions of the introductory environment by modality specifically, the next section will investigate students' perceptions towards programming and computer science more broadly.

Authenticity of the Activity by Modality

One drawback identified in using blocks-based programming environments with high-school aged learners is the perceived lack of authenticity and a recognized difference between what it looks like to program in blocks-based languages versus text-based professional languages (Weintrop & Wilensky, 2015b). In the first year of this study, the analysis of this question found that students did raise concerns over the authenticity of the *Snappier!* environment, but it was unclear where the source of that inauthenticity lay. A number of factors could have contributed to this view, including the blocks themselves, the drag-and-drop programming mechanism, or the larger context of giving instructions to an on-screen sprite being a very different type of program output than what typically accompanies programming. In the second iteration of the design, questions were designed to tease apart the role of the modality specifically in contributing to this perception. On the Mid and Post attitudinal surveys, students were asked if what they did in the first five weeks of the course was similar to what "real programmers" do. Responses were given

on a ten-point Likert scale, with a higher score meaning students agreed more strongly.

Figure 5.3 shows student responses by condition to this prompt given on the Mid and Post surveys¹⁵.

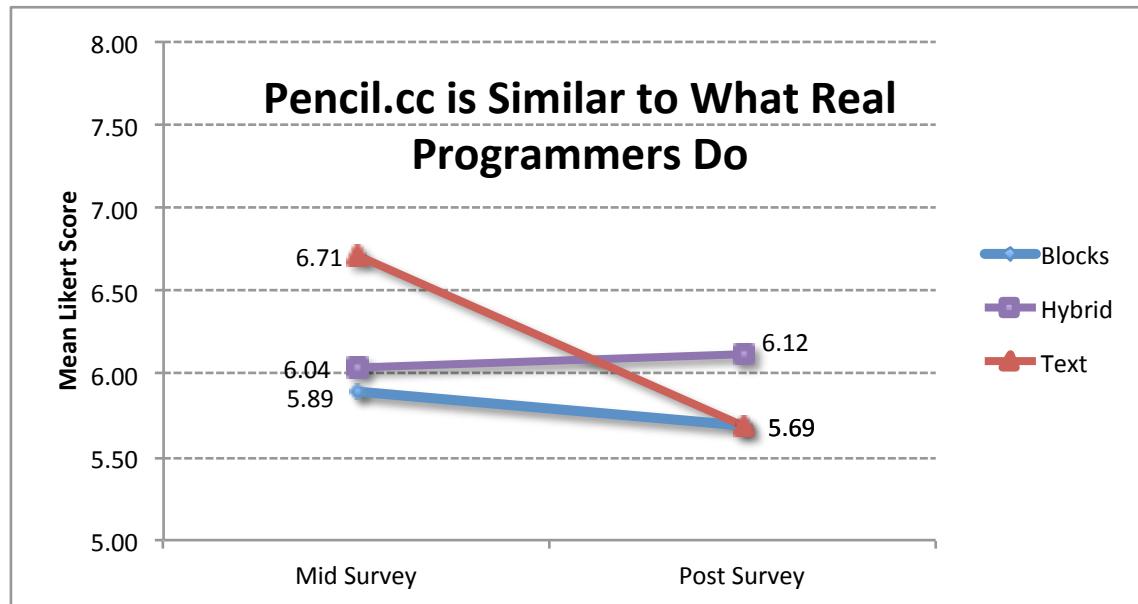


Figure 5.3. Student responses to the prompt: Pencil.cc is similar to what real programmers do.

Overall, the mean student response on the Mid survey was 6.22 ($SD = 2.15$), while on the post survey, the mean was 5.9 ($SD = 2.22$). This means that overall, students tended to agree with the statement that what they did in the introductory environment was similar to what real programmers do. A Wilcoxon signed rank test comparing the aggregated Mid to Post scores shows the two time points to be different from each other at a $p = .05$ level ($Z = 1143, p = .05$), meaning an overall shift did occur, although with the statistical power these data provide, the change was only just reached the conventional statistical significance threshold of $p = .05$ ¹⁶.

¹⁵ Note: the y-axis scale for this and all figures in this chapter do not start at zero, but all are on the same scale so can be compared relatively

¹⁶ An alternative test that could be used here is a pair-wise t-test, which gives a comparable result of $t(76) = 1.89, p = .06$. The Wilcoxon signed rank test is preferred due to the ordinal nature of the underlying Likert data, so will be used throughout the dissertation.

Running an ANOVA test on normalized Z-scores for the two time points shows there is not a statistically significant difference between the three conditions at the Mid point $F(2, 78) = 1.15, p = .32$, or at the conclusion of the study $F(2, 80) = .32, p = .76$. The convergence of scores on the Post survey relative to the Mid suggests differences may exist at the Mid point, but that the data in this study does not have the statistical power to make that claim. To understand whether there was a significant shift in perceived authenticity between the two surveys for each of the conditions a Wilcoxon signed ranks test was run. The test shows a significant change in the perceived authenticity of the introductory environment for students who were in Text condition ($Z = 166.5, p = .02$) in the negative direction. This means that students in the Text condition found Pencil.cc to be less similar to “real programming” after ten weeks of working in Java. Non-significant shifts in the positive direction for the Hybrid condition ($Z = 64.0, p = .35$) and a negative direction for the Blocks condition ($Z = 135.5, p = .49$) were observed but neither were significant.

To understand if the changes between the Mid and Post surveys were significant across the three conditions, normalized Z-scores were calculated for each condition, then an ANOVA was run on the deltas of students’ reported responses across the three conditions. In other words, this test is looking to see if the slopes of the three conditions are different from each other, and if so, where the statistical significance lies. The ANOVA calculation on the changes in perception of authenticity across the three conditions was $F (2, 74) = 3.5, p = .03$, thus a statistically significant difference does exist across these groups. A post hoc Tukey HSD test shows the change in attitudes between the Text and Hybrid conditions was significant, $p = .03$, and that no statistical significance was found for the other pairings (Text/Blocks, $p = .24$, Hybrid/Blocks $p = .54$).

Taken together, this analysis reveals a few conclusions about how modality affects students' perceptions of the authenticity of a given programming environment. Students in the Text condition initially saw Pencil.cc as the most similar to what real programmers do, but this perception shifted downward after working in Java for 10 weeks, suggesting that in gaining experience with Java, that initial perception changed. As can be seen from Figure 5.1 and Figure 5.2, a lot of this difference is driven by syntax, which is an immediately visible difference between the Pencil.cc and Java experiences. As one student put it in his interview after starting Java, "*I just like Java, the syntax makes me feel more complete. I'm actually coding.*" This perception can be seen across the data in various places, and will be explored more deeply in the chapter looking at the transition from Pencil.cc to Java. The issue of long-term utility was also raised in interviews with students after working with Pencil.cc. For example, one student, in reflecting back on his time with the introductory environment said: "*I feel like Java will be more useful in the long run than what [Pencil.cc] could offer me*". This view was echoed by another student, who in his post-Pencil.cc interview said "*[Pencil.cc] is a bit too limiting for someone who goes into this class thinking I'm going to make something that is going to be used in industry.*" In these quotes, the students long terms plans with programming can be seen and how Pencil.cc does not fit into them. Like in other places, the cause of these views include features of modality along with other aspects of the programming environment, but as is shown in Figure 5.1 and Figure 5.2, a number of aspects of modality play a significant role in shaping students perceptions of the differences.

Looking across the three conditions, students who worked in the Blocks condition had the lowest average response on the perceived similarity of Pencil.cc to professional programming after the introductory portion of the study, and, like Text, saw their perceptions drop over the 10

weeks working in Java. In contrast to the other two conditions, the Hybrid condition saw the authenticity of their experience during the first five weeks increase after working in Java for 10 weeks (although not significantly). In comparison to the other two conditions, this shift was statistically significant. There are a number of possible explanations for these collective outcomes. One theory is that students in the Text condition had an easier time doing a direct comparison between Pencil.cc and Java, since they were in the same modality, and thus, the shared modality made the differences more salient. The Blocks condition had the opposite problem, from the beginning, students perceived the blocks interface to be different and unlike what real programmers do, and the shift to Java reinforced this. The Hybrid condition however, potentially benefitted from a best of both worlds effect. The underlying text editor makes clear that what they are doing is the same type of activity (i.e. manipulating text) but was different enough, thanks to the presence of the blocks, to not provoke a direct comparison with Java. Java and the Hybrid form of Pencil.cc are both programming (i.e. both manipulating text to give instructions to a computer), but are also different from each other, but not in a way that necessarily delegitimizes Pencil.cc, which is recognized as being a useful introductory approach (a finding that will discussed below). Another possible explanation for the Hybrid condition's different outcome stems from the fact that only in that Condition do students interact with more than one modality (graphical blocks and text side-by-side), thus possibly showing students that programming is not a uniform activity, but instead, that the act of programming and programming languages and environments can take many shapes and rely on many modalities, interfaces, and technologies.

Learning to Program by Modalities

A second dimension of students' perceptions of different programming modalities

that is of interest in this dissertation is whether or not they felt that using a given modality made them better at programming. This analysis looks at the three modalities used in Pencil.cc in isolation to see if they were viewed as a productive with respect to the goal of learning to program, not whether it was effective for preparing them for something else (i.e. Java). Figure 5.4 shows students' responses to the following prompt: Pencil.cc made me a better programmer. Like with the last question, students experience of Pencil.cc will refer to different modalities depending on which condition they were in. This question was asked on the same Mid and Post surveys and on the same 10-point Likert scale as the question in the previous section. The mean score on the Mid survey was 7.5 ($SD = 2.1$) and the mean on the Post survey was 7.0 ($SD = 2.2$). This suggests that overall, students felt that all three modalities improved their programming, but did not hold particularly strong feelings about this statement.

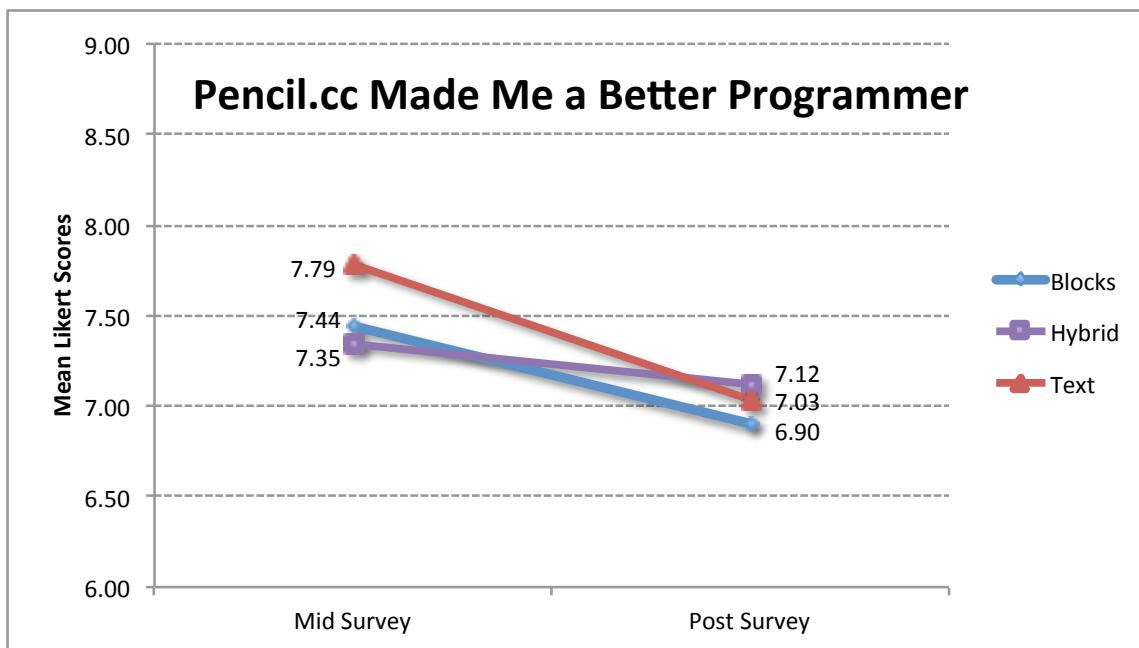


Figure 5.4. Student responses to the prompt: Pencil.cc made me a better programmer.

The first thing to notice from this chart is the negative slope for all three conditions, meaning overall, students' view of how helpful the introductory environment was with respect to learning to programming decreased after spending 10 weeks learning Java, regardless of modality used. The mean response on the Mid survey was 7.5 ($SD = 2.14$), while the mean Post response was 7.01 ($SD = 2.21$). Running a Wilcoxon signed rank test on for the whole set of responses shows a significant difference between student responses on the Mid survey and the Post survey $Z = 1100$, $p = .01$. This provides evidence that there was in fact a significant decline in students' perceptions of whether or not Pencil.cc made the students better programmers. Running this same test on each condition individually shows a significant effect for the Text condition ($Z = 163$, $p = .03$), and smaller, non-significant effects for the Blocks ($Z = 121.5$, $p = .12$) and Hybrid ($Z = 95$, $p = .38$) conditions. An analysis looking across the three conditions found no signification differences between the three conditions either on the mid survey ($F(2, 78) = .14$, $p = .87$), the post survey ($F(2, 80) = .07$, $p = .93$), or on the difference in performance by condition (i.e. the slopes) ($F(2, 74) = .949$, $p = .39$).

These quantitative findings are supported by data from the student interviews conducted after students finished the introductory portion of the course, which reveal that students found Pencil.cc productive for learning to program. For example, when asked about this topic, students gave responses like "*It has definitely given me the basis of like, a computer follows everything, with like, total logic. Like if you say, write something in quotes and then don't end that quote, it's not going to work. You have to be very specific with your code.* [Pencil.cc] just kind of taught me how much syntax and semantics matters" and "*It was good to learn basic concepts.*" Students also had similar constructive comments about the introductory environments across modality when talking about it preparing them to learn Java, a topic that will be explored in further detail

in chapter 7. It is important to note that not all students felt this way. Some students, especially students with prior programming experience did not find Pencil.cc to be as productive. “*Pencil.cc helped a little bit, it helped other people too, it didn't help me too much because I knew some of these things already.*”

These findings begin to show one of the features of this study design. The data reveal that students’ perceived utility of working with Pencil.cc dropped after spending 10 weeks learning to program Java, independent of the modality used. However, there is no difference in students’ perceptions across the different conditions. This can be interpreted to mean that modality was not a significant factor contributing to this perceptual shift. This leads to the explanation that other aspects of the Pencil.cc environment or the larger intervention are potential causes of this decline. This may include the CoffeeScript language, the setting of creating interactive drawings and webpages, or the curriculum that the students followed during the five weeks spent working in Pencil.cc. It is also important to mention that with a larger dataset and thus more statistical power, significant differences may emerge. For example, the fact that the Hybrid condition was initially seen as the least effective modality for learning programming, but after 10 weeks in Java became the most effective with respect to students’ perceptions suggests there may in fact be some interaction between modality (or rather mixed-modalities) and perceived utility.

Changes in Attitudes and Perception over Time

Along with perceptions of the three modalities used in the study, this dissertation seeks to understand how modality affects students’ attitudes toward programming and computer science more broadly. This includes questions of how much they like the field, if they think they will be successful in their programming endeavors, and whether or not they plan to enroll in future computer science learning opportunities. This section looks specifically at attitudinal and

perceptual changes between the start of the school year and the midpoint of the study, after students completed the introductory portion of the course but had not yet begun working in Java. Shifts after the transition to Java, and how they relate to changes that occurred during the first five weeks of the study, are discussed later in the chapter looking at the transition from Pencil.cc to Java. Four attitudinal dimensions are investigated: confidence, enjoyment, perceived difficulty, and interest in continuing with more computer science learning opportunities.

Confidence in Programming Ability

The first attitudinal dimension discussed is students' perceived confidence in their own programming ability. To calculate a reliable measure of confidence, student responses to the following two Likert scale statements were averaged together: I will be good at programming (or I am good at programming on the Post test) and I will do well in this course. These questions show an acceptable level of correlation, having Cronbach's α scores of .79 on the PRE survey, .80 on the Mid survey, and .88 on the Post survey, which are all near or surpass the .8 threshold commonly used to define an acceptable level of reliability. The aggregated confidence measure at the Pre, Mid and Post points in time are shown in Figure 5.5. In this section, the figures show all three time points even though only the Pre and Mid values are discussed here, the Post scores and the Mid to Post differences are discussed in a chapter 8 which looks at the transition to Java. Also note that all figures in this section are on the same scale, but do not cover the same range, so can be compared relatively, but not absolutely, and like the previous sections, the y-axis in these figures do not start at zero in order to make the trends more clear.

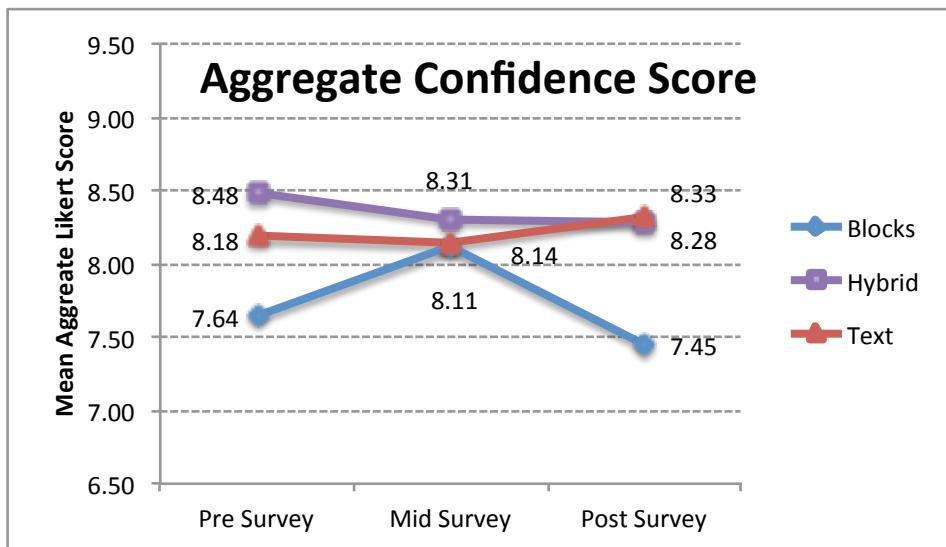


Figure 5.5. Calculated levels of students' confidence in programming at three points in the study.

The mean confidence scores at the outset of the study was 8.11 ($SD = 1.47$) on a 10-point Likert scale, which is rather high. After spending the first five weeks of school working in Pencil.cc, students' confidence scores inched up to 8.19 ($SD = 1.67$), a change that is not statistically significant ($Z = 3296, p = .48$) after running a Wilcoxon signed rank test. This lack of change is possibly explained by the fact that the students came in extremely confident (this is a selective enrollment school, so the students have historically been successful in academic contexts), so there is a possible ceiling effect on this measure. An alternative explanation is that five weeks in the introductory environments did not have any effect on students' confidence, possibly because they did not get any better at programming or that what the students were doing was found to be too easy or different from the type of activity that would increase their confidence with respect to programming.

The next step in this analysis is to look at differences by modality. In calculating the ANOVA on the Pre scores, the results show that there is a slightly significant difference between the three starting data points, $F(2, 84) = 2.46, p = .09$, meaning that the three samples were not

the same with respect to their initial confidence levels. A Tukey HSD post hoc calculation shows there to be a difference between the Blocks and Hybrid conditions ($p = .08$), but no difference between the other two conditions ($p = .33$ and $p = .71$). This difference in confidence cannot be explained by other factors of the class that were collected (like grade, gender, or prior computer science experience), suggesting that a prior difference does exist. For analytical purposes, this leads us to be less interested in absolute comparisons of values at the mid points or the changes across conditions (since they are not the same initially), but does allow us to look at within-group differences.

Running a Wilcoxon signed ranks test on the three conditions and their changes between the Pre and Mid surveys shows a significant change for the Blocks condition ($Z = 46$, $p = .05$), but not a significant change for either Hybrid ($Z = 108$, $p = .61$) or the Text condition ($Z = 98.5$, $p = .82$). Given the positive slope of the change in the Blocks condition, this difference can be interpreted as showing that students in the Blocks condition saw a significant increase in their confidence in their own programming abilities. This finding correlates with how useful students found the blocks-based modality to be in preparing them for Java, which found that students in the Blocks condition found Pencil.cc to be the most useful at the five-week point of the study. This data will be presented and discussed in a later chapter. This outcome is consistent with other less quantitative studies suggesting that the blocks-based programming interface is effective at increasing students confidence in their own programming ability (Maloney et al., 2008; N. Smith, Sutcliffe, & Sandvik, 2014). Further, this positive increase in confidence supports one of the arguments made in favor of blocks-based language and their affective strengths, although, as will be discussed in the next section, not all such claims are supported by this study. The lack of positive trends for the Hybrid and Text conditions can be interpreted in a few ways. One

explanation is that these modalities do not improve students' confidence in programming, for which a number of possible explanations could be given (e.g. they find it difficult or did not feel successful in their time with it). A second plausible explanation for these data is that there was a ceiling effect, meaning the students started with a high level of confidence, so there was little room for them to become more confident, which was not the case in the Blocks condition.

Enjoyment of Programming

The second attitudinal dimension is whether or not students' enjoyment of programming differed based on the modality they used. To calculate a measure of enjoyment, responses to the following three Likert statements from the Pre, Mid, and Post surveys were combined: I like programming, Programming is Fun, and I am excited about this course. These three questions were found to reliably report the same underlying disposition at all three time points (Pre Cronbach's $\alpha = .79$, Mid Cronbach's $\alpha = .84$, Post Cronbach's $\alpha = .89$). Figure 5.6 shows the aggregated enjoyment scores for students across the three conditions at all three time points, although, again, in this section only the Pre and Mid scores will be discussed.

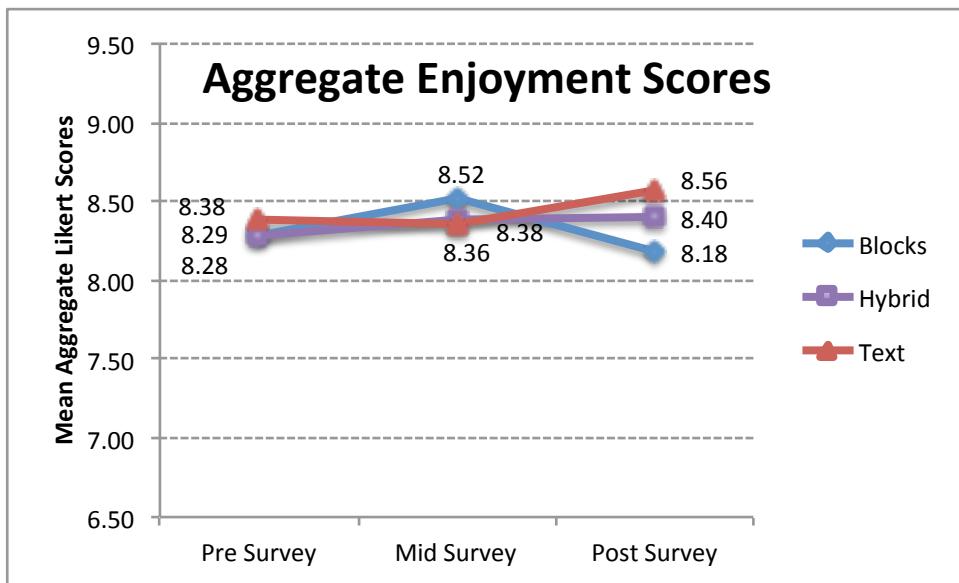


Figure 5.6. Calculated levels of students' enjoyment of programming by condition at three points in the study.

At the outset of the study, the overall average mean enjoyment response 8.31 ($SD = 1.39$) on the 10-point Likert scale. After five weeks in the course, the mean enjoyment response increased slightly to 8.42 ($SD = 1.60$). A Wilcoxon signed rank test shows that these two scores are not statistically significant from each other ($Z = 3227$, $p = .34$), meaning on average, students did not like programming any more or less after using Pencil.cc than they did before. Unlike the responses for initial confidence, there is no difference between the three condition at the outset of the study ($F(2, 84) = .047$, $p = .95$). After working in either the blocks, text, or hybrid interface of Pencil.cc for five weeks, the survey did not reveal a difference in enjoyment across the three conditions ($F(2, 78) = .08$, $p = .93$). Given these two results, it is not surprising to find that there was no difference in the changes between the three groups ($F(2, 75) = .14$, $p = .87$). Looking within each condition, a Wilcoxon signed ranks test did not show any significant changes for the three conditions (Blocks: $Z = 70.5$, $p = .20$, Hybrid: $Z = 72$, $p = .22$, Text: $Z = 93.5$, $p = .68$). This lack of significant finding by condition suggests that modality plays a relative small role

with respect to perceived enjoyment of programming. Also, the high scores for all three conditions at all three points speaks to a potential limitation due to the fact that this is an elective course which students have self-selected into. Qualitatively, this graph does show a positive slope for both the Blocks and Hybrid conditions and a slightly negative slope for the Text condition, which suggests there may be differences here, but these data do not have the statistical significance to support such claims.

Digging in a little deeper, looking at the three underlying questions individually does reveal a significant finding, on the question “I like programming”, the Hybrid group saw a statistically significant increase ($Z = 36, p = .05$). This provides a little piece of evidence that in fact that there may be a difference based on modality that a larger study with more statistical power might be able to reveal. It is interesting to note that the only significant gain for any of these questions between the Pre and Mid time points came from the Hybrid condition, as opposed to the Blocks condition, which suggests that the Hybrid condition may be tapping into the enjoyment that comes from blocks, but that the authenticity of the hybrid condition reported earlier in the chapter may also contribute to the sense of enjoyment, meaning that the Hybrid condition may be benefiting from the best-of-both-worlds.

Programming is Hard

The attitudinal survey included the Likert statement: Programming is Hard. Initially this was intended by be part of the Confidence aggregate score, but ended up not correlating with the other two confidence questions (Pre Cronbach’s $\alpha = 0.48$, Mid Cronbach’s $\alpha = 0.57$, Post Cronbach’s $\alpha = 0.67$), as all three time periods fall well below the .8 level generally agreed to be threshold for adequate correlation. As such, this question is treated independently. Figure 5.7 shows the Pre, Mid, and Post scores for students grouped by Condition for this question.

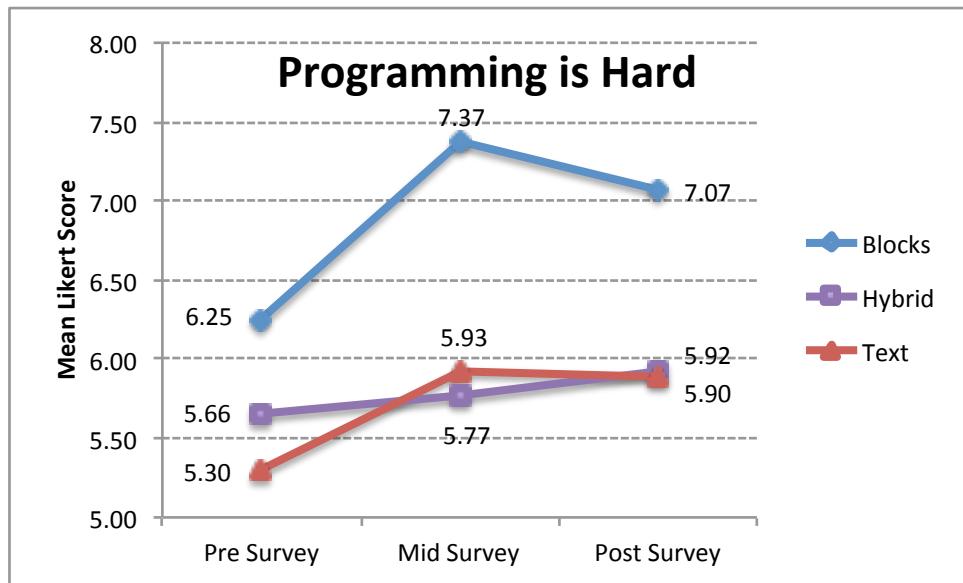


Figure 5.7. Average responses to the Likert statement: Programming is Hard.

The average response to this statement across the three classes during the first week of school was 5.72 ($SD = 2.29$). By the end of the fifth week, the mean response had shifted to 6.36 ($SD = 2.28$), resulting in a moderately significant increase ($Z = 3.009.5$, $p = .10$), meaning students thought programming was harder as a subject after spending five weeks using one of the modalities of Pencil.cc. This shift is largely driven by the steep increase in the responses among students in the Blocks condition. An analysis of variance (ANOVA) calculation on the Pre survey responses shows no significant difference between the three conditions ($F(2, 84) = 1.28$, $p = .28$) at the outset of the study and a significant difference emerging by the Mid survey $F(2, 78) = 4.36$, $p = .02$. A Tukey HSD post hoc analysis shows a significant difference between the Blocks and the Hybrid condition ($p = .03$) and the Blocks and Text condition ($p = .04$), while the Text and Hybrid conditions were comparable ($p = .96$). This means the Blocks condition was a significant outlier with respect to perceived difficulty of programming. As will be shown in the next chapter, the Blocks condition did not perform significantly differently than the other two

conditions on the content assessments. In fact, on the total score, the Blocks condition performed the best. This means that this perceived difficulty of programming does not match the Blocks students' performance on the assessments. One possible explanation for this outcome is that it serves as another data point suggesting that students did not view what they were doing in the Blocks-based interface as being the same as "real" programming. Comparing the Pre to Mid changes across the three groups shows that while there are different levels of change (i.e. different slopes), those differences are not significantly different from each other ($F(2, 74) = .76$, $p = .47$).

Looking within the three conditions, only the Blocks group saw a significant change in responses. A Wilcoxon signed ranks test returned $Z = 42$, $p = .02$ for the Blocks condition, but nothing significant for the Hybrid condition ($Z = 72.5$, $p = .57$) or the Text condition ($Z = 96$, $p = .51$).

Interest in Future CS

The last attitudinal category is looking at whether or not the modality used in the introductory programming environment affected students' interest in enrolling in future computer science courses. More specifically, students were asked, on a ten-point scale, how much they agreed (10) or disagreed (1) with the following statement: I plan to take more computer science courses after this one. Figure 5.8 shows the average response for students grouped by condition.

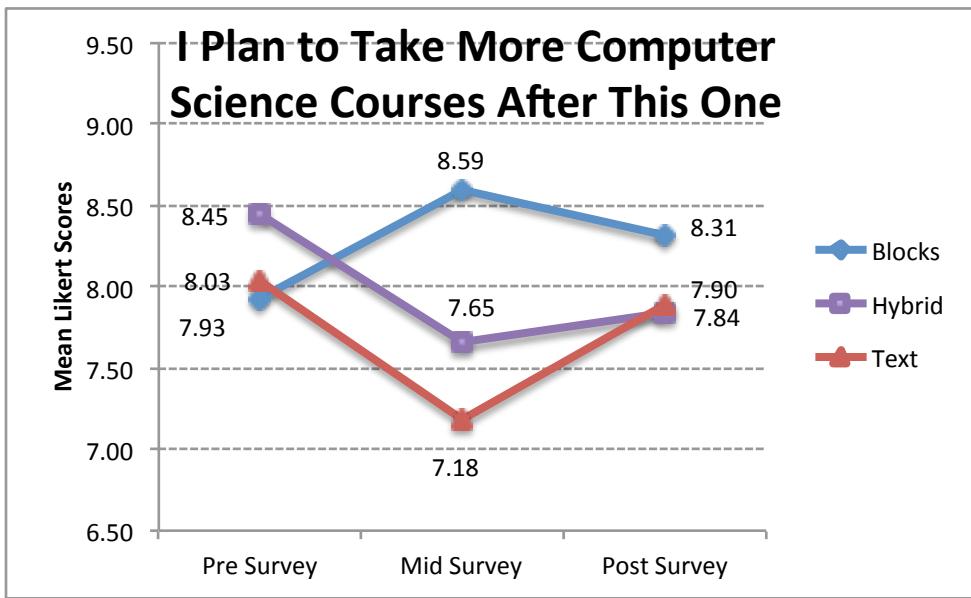


Figure 5.8. Average responses to the Likert statement: I plan to take more computer science courses after this one, grouped by condition.

The mean scores of the Pre (8.14 , $SD = 2.40$) and Mid (7.80 , $SD = 2.57$) surveys show a slight decrease in overall interest in wanting to take future computer science courses, however not at a significant level ($Z = 442$, $p = .28$). Although there was no overall trend between Pre and Mid when all students scores were aggregated together, Figure 5.8 shows a rather different story. The survey administered at the start of the school year showed no significant difference between the three conditions ($F(2, 84) = .37$, $p = .69$). After five weeks working in the three modalities, a numerical difference emerged, although in this case did not reach a level of statistical significance ($F(2, 78)=2.22$, $p = .12$). At the Mid point, the students who were using the blocks-based version of Pencil.cc had moved from being the least interested in future computer science classes to the most interested, while both the Hybrid and Text conditions saw their level of interest decline. The differing slopes approach statistical significance ($F(2, 75) = 2.88$, $p = .06$).

While there was a slight difference in the changes across the three groups, only the Hybrid condition's Pre to Mid change showed moderate levels of significance. A Wilcoxon

signed ranks test for the Pre to Mid scores for the Hybrid condition returns $Z = 91.5$, $p = .07$, while the Blocks condition and Text condition failed to reach any level of significant ($Z = 25$, $p = .15$ and $Z = 110$, $p = .29$ respectively). This finding is interesting, as it does not entirely fit with the previous sections, which showed that students working in the Blocks condition viewed programming to be the most difficult and were not outliers with respect to enjoyment or confidence. Despite this similarity on the other measures, students in the Blocks condition were more interested in pursuing future computer science learning opportunities. In the next section, which concludes this chapter, the results from this section are brought together with the other findings from this chapter and are synthesized in order to try and answer the stated research question being addressed in this chapter: how does modality affect students' attitudes towards and perception of computer science.

Discussion

This chapter presented data towards answering how modality affects students' attitudes and perceptions of programming and computer science more broadly. The data drew from the initial student interviews as well as from the Pre, Mid, and Post attitudinal surveys (Appendix B). Two sets of analyses of student perceptions were performed in this chapter. The first set investigated students' perceptions of the use of different modalities in Pencil.cc while the second was on their emerging relationship with programming. As is often the case, the answer is not as simple "the Blocks condition improved attitudes" but instead a more complicated, nuanced answer that reveals along which dimensions attitudes shifted and when modality is the reason for that shift. This discussion provides a short summary of the findings presented above and situates them relative to the other data presented.

Students' Perceptions of Pencil.cc

The first finding from this chapter is that the use of an environment like Pencil.cc, which includes a graphical output and editor with various scaffolds differs from what high school students expect to see in an introductory programming class. Despite encountering this unexpected environment, students were able to recognize its utility, citing the environment's friendly interface, ease of writing successful programs, and ability to lay the foundation for future programming instruction as reasons for its usefulness. This shows that high school aged students are sophisticated enough about programming, computer science, their own learning to recognize the utility of the environments they are working with.

The sophistication of the learners' understanding of the learning environments and the modalities used within them emerged in analysis of student responses to an open-ended survey question asking them about what they found to be most useful about the Pencil.cc environment. In these responses we start to see stratification by modality. Blocks students were more likely to attend to blocks-related futures of the environment, citing the ease of the drag-and-drop mechanism and the visual layout of the set of possible commands as being useful, while the Text condition students attended more to features tied to the text modality, like syntax and the available in-editor scaffolds like real-time compilation warning and the Quick Reference feature.

Following this thread of investigation into students' perceptions of the utility of the Pencil.cc environment, an analysis of Likert survey questions pertaining to the authenticity and utility of the environment was presented. In looking at students' perceptions of Pencil.cc with respect to its authenticity, on the Mid survey, students on average reported that the version of Pencil.cc they used was similar to what "real programmers" did and that the environment helped them in learning to program. This paints a generally positive picture of students perceptions of

the Pencil.cc environment, however, other data suggests it is not so simple. For example, after working in Java for 10 weeks, a majority of students (79%) across all three conditions found Pencil.cc to be less useful than they had initially reported, and almost half of the students (42%) thought Pencil.cc was less like real programming after working in Java. At a condition level, only the Hybrid condition on the authenticity question showed a positive change after learning in Java. This suggests a shift occurred in students' perception of what programming is between the Mid and Post surveys. This suggests that the conceptualization of what programming is after working in Pencil.cc is different than what programming means to learners after 10 weeks of working in Java. How students' experiences with the introductory tools shaped their experience in Java will be explored in greater detail in Chapter 8.

Students' Attitudes Toward Programming

The second set of analyses presented in this chapter looked at students' attitudes towards programming grouped into four categories: confidence, enjoyment, difficulty, and interest in pursuing computer science. For each of these categories, the analysis looked both across and within conditions to try and understand the role of modality in shaping student attitudes. Across these aspects of attitudes categories, the data show student's confidence and enjoyment growing slightly, their interest dropping slightly, and a significant increase in students' perception of the difficulty of programming. That gives the highest-level overview. Breaking these aggregate trends down we see a more nuanced story emerged, finding that modality does seem to effect some of these attitudinal dimensions but not others.

When looking at perceived difficulty, plans to take future computer science courses, and students' overall confidence levels, we find differences by condition. In these three cases, it was the Blocks condition that differed from the Text and Hybrid conditions. Students in the Blocks

condition found programming to be more difficult but also experienced the greatest gains in confidence as well as the largest increase in terms of wanting to pursue future computer science courses. One possible way to explain this would be to say that students in that condition enjoyed programming more than the other two, but our analysis of enjoyment found that, while the Blocks group did report improved enjoyment, so did the Text and Hybrid conditions. Another possible explanation alluded to above is that students in the Blocks condition saw what they were doing as a simplified version of programming. Thus, the fact they were doing well in the Blocks condition (a fact that will become more clear in the next chapter) could explain their confidence and desire to take more computer science classes in the future, while also explaining how they view programming to be difficult, because what they are doing is not the same as programming, but a simplified version of it. This explanation partially holds up to the findings of the authenticity question, which shows why Blocks students saw their condition as the least authentic, but the gap between Blocks and Hybrid is small enough that, if this were the whole story, we'd expect to see a similar pattern in the Hybrid responses.

In the same way that finding differences between the conditions is an interesting result, a lack of differences also tells us something about the modalities. The lack of difference between the Blocks, Hybrid, and Text conditions with respect to the enjoyment of programming suggests that modality is not the driving characteristic behind students finding Pencil.cc-style environments engaging, at least among high school aged learners. Looking specifically at the Hybrid and Text conditions, we see little significant differences between them, but for the most part, the Hybrid condition shows more desirable numbers (more confident, slightly higher levels of enjoyment, and more likely to take another computer science course). This suggests that the

drag-and-drop ability to add commands to programs and the browsability of commands do not significantly contribute to differences on attitudinal measures.

How Did the Hybrid Condition Fare?

The Hybrid condition was designed to blend features of blocks-based and textual programming interfaces. The goal for the design is that it would be able to leverage the attitudinal strengths of the blocks-based modality while also easing the difficulty observed when students move from graphical, drag-and-drop interfaces to more professional languages. This chapter starts to shed light on whether or not this came to be with this specific hybrid approach.

First, the analysis of the perceived affordances of the three conditions found a bimodal set of responses, with relatively little overlap between the Blocks and Text responses. The responses from the Hybrid group spanned both ends of our categorical spectrum. This means that the chosen Hybrid interface was successful in drawing on features of both modalities to support learners, and that the learners themselves were aware of and attuned to those supporting features.

Among the four attitudinal dimensions analyzed, the Hybrid condition largely followed the same pattern as the Text condition, at times living between Blocks and Text (as in the case of the future computer science question), while also sometimes coming out with the highest score at the mid point (confidence) or the lowest (difficulty). Following similar trajectories as the Text condition suggests the Hybrid interface used in this study was too similar to the Text condition to gain the positive attitudinal benefits we found in the Blocks condition. Given the expanse of the design space of Hybrid environments, this suggests that more design work might be needed to bring the text editor portion of this hybrid interface closer to the Blocks interface that yielded the positive attitudinal outcomes.

Finally, in looking at how students perceived the Hybrid condition, while there was little difference across the three conditions, in both measures (authenticity and utility for learning programming), the Hybrid condition had the best outcome of the three conditions at the end of study. On the authenticity question, the Hybrid condition was the only one that saw an increase after learning Java, and on the question about if Pencil.cc helped the students learn, the Hybrid condition went from the least useful to most useful after Java was introduced. These outcomes suggest that the real value of the hybrid condition emerges only after Java is introduced, so that a Hybrid condition might not be best for learning contexts where a hybrid tool is the end goal, but instead, serves as a successful stepping-stone to text-based languages when that is in fact the goal. This question is further explored in Chapter 8.

Conclusion

This chapter presented data answering the first part of the first research question on how modality affects learners' attitudes towards and perceptions of programming and computer science more broadly. With the findings presented above, part of the larger picture of the role of modality on the learner is starting to come into view, although there is much that remains to be filled in. For example, some of these analyses only told half of the story, focusing on students shifting attitudes over the course of the first five weeks of the study when students were working in the introductory tools. Data was also collected to understand if these attitudinal shifts persisted or changed after working in Java. Likewise, little has yet been said about the nature of student programs, the practices they develop or their conceptual understanding of the ideas encountered over the first 15 weeks of the study. It is this last topic, conceptual understanding, that constitutes the second half of the first set of research questions and the focus of the next chapter.

6. Conceptual Learning Outcomes

This chapter presents findings on student's conceptual understanding of central programming concepts looking for differences and similarities across the three conditions of the study (Blocks-based, Text-based, and Hybrid Blocks/Text). The chapter is broken down into two sections. First is a presentation of students' emerging conceptualizations of core computer science ideas. This section draws on qualitative data sources (interviews and open-ended responses) to characterize how students are coming to understand the programming ideas covered in the opening five-week curriculum. The second section presents a quantitative analysis conducted with data from the three administrations of the Commutative Assessment: one given at the beginning of the study (Pre), one five weeks into the study after students had finished working in the Pencil.cc environment but before they had started working in Java (Mid), and one at the conclusion of the study after students had spent 10 weeks learning Java (Post). The quantitative analysis section begins with a brief review of the Commutative Assessment and a description of how it was administered. The first analysis investigates the relationship between modality (graphical blocks vs. textual) and specific programming concepts, revealing that modality does indeed matter, which helps to motivate the analysis conducted in the remainder of the chapter. The second half of the section looks at how conceptual understanding differs by the version of Pencil.cc used during the first five weeks of the study. Results from the Pre assessment, which serves as a baseline for the analysis that follows, is presented first. It shows a lack of differences across the three conditions. From there the analysis looks at learning outcomes by condition, answering the question: were there differences in performance on the content assessment across the three conditions of the study? The next portion of the chapter asks a similar question, but now looking at differences by modality by condition, extending an

analysis conducted on data collected during the first year of the dissertation. After analyzing the responses by modality, the next section looks at differences by the concepts included on the assessment, asking: Do learning outcomes for specific concepts differ by condition or modality? The chapter concludes with a discussion and summary of the various findings presented. Before presenting the findings, it is important to reiterate that when comparing across conditions (Blocks v. Hybrid v. Text) the only difference was the modality in the programming environment, all other class-related factors were held constant¹⁷.

Emerging Conceptual Understandings

After students concluded the five-week introductory portion of the study, they were asked short answer questions about the four central programming concepts covered in the curriculum: variables, conditional logic, iterative logic, and functions. This section presents the results of open coding these responses and grouping them by condition. The goal of this analysis is to understand if students' conceptual understanding of concepts is informed by the modality they used to first interact with and use the concepts, and if so, how. For each concept, students responded to the following open-ended prompt: What do ___ do? And how are they used in programs? Where the ___ in each question was replaced with "variables", "for loops and while loops", "if and if/else statements", and "functions". This direct questioning approach admittedly does not yield a nuanced understanding of learners' conceptualizations, but is nonetheless useful for beginning to understand how students are thinking about these concepts. Student responses to these questions were open coded using a grounded theory approach (Strauss & Corbin, 1994), in which the data themselves were used to identify emerging themes and codes. Due to this

¹⁷ One other factor that is not constant is that the classes were held at different times in the school day: 4th period, 7th period and 8th period, but given that all classes were taught by the same teacher in the same classroom at the same time in the school year, this was unavoidable.

emergent analytic approach, it is important to note the ontological inconsistencies that exist both within and across conceptual groups. For example, the analysis of variables focuses on conceptual metaphors used by the learners while the conditional logic analysis included codes for characteristics of if/else statements as well as attention to features of conditional statements. This diversity of types of codes is an artifact of the emergent analytic approach taken.

Variables

The first question students responded to from this group was: What do variables do? And how are they used in programs? In analyzing the responses given, a variety of answers were given, more specifically, students talked about variables and their uses in a number of different ways employing different metaphors in their descriptions. The first type of response identified was to use the metaphor of a variable being like a container that stores things. For example, students gave response like “*Variables are used to store a value*¹⁸” and “*Variables are values that store information, they are used to store information in.*” The idea of a variable being a container that holds things is commonly used in computer science classrooms. The second metaphor found in student responses is similar to the container idea, but instead of holding the value, variables serve as placeholders for that value. In other words, the value is stored somewhere else and the variable serves as a representation used to get access to that value. This view can be seen in responses such as “*Variables are placeholders for something such as a string, boolean, or integer.*” and “[Variables] are something that stands in or represents something else”. A third metaphor that is similar to the first two is that of a variable as a pointer. While the idea of a variable as a pointer to a value could arguably be collapsed with the prior group because it shared the feel of a placeholder, it is left separate here because the notion of a

¹⁸ Note: italicized text in this section denote direct quotes from the students’ typed-in responses.

pointer carries specific meaning in computer science and is explicit about the idea that the value itself is stored elsewhere, which is only implicit in the placeholder responses. Fewer students described variables in this way relative to the first two categories (four in total), one of these responses reads: “*Variables are values that programs use to reference pieces of information in the code.*” The last type of response given by students was to not describe the variables in terms of the values they represent, but instead, to define the variable as an object in its own right, independent of what its value is. Sample responses from this category include: “*Variables are things that change inside a function and are not things set in stone*”, “*Variables are changeable values. They are used to make stuff happen pretty much. Without them, pretty much nothing can occur in a program*”, and “*Variables are anything you want them to be. They're used to tell the computer that another thing equals something else and so on*”. All student responses were coded as one of these four categories, with a few responses being coded as more than one when students drew on more than one metaphor within their response, as in the case where a student responded “*Variables are symbols that can represent or hold information to be used, or a place holder for unknown values*”, which was coded for both container and placeholder metaphors. Only five of the 82 responses did not fit into any of these four categories, an example of one of these outliers is “*Variables are used to keep a clean and concise code in your program*”, which does not attend to what a variable is, but instead describes how they are used. Figure 6.1 shows the distribution of student responses grouped by condition.

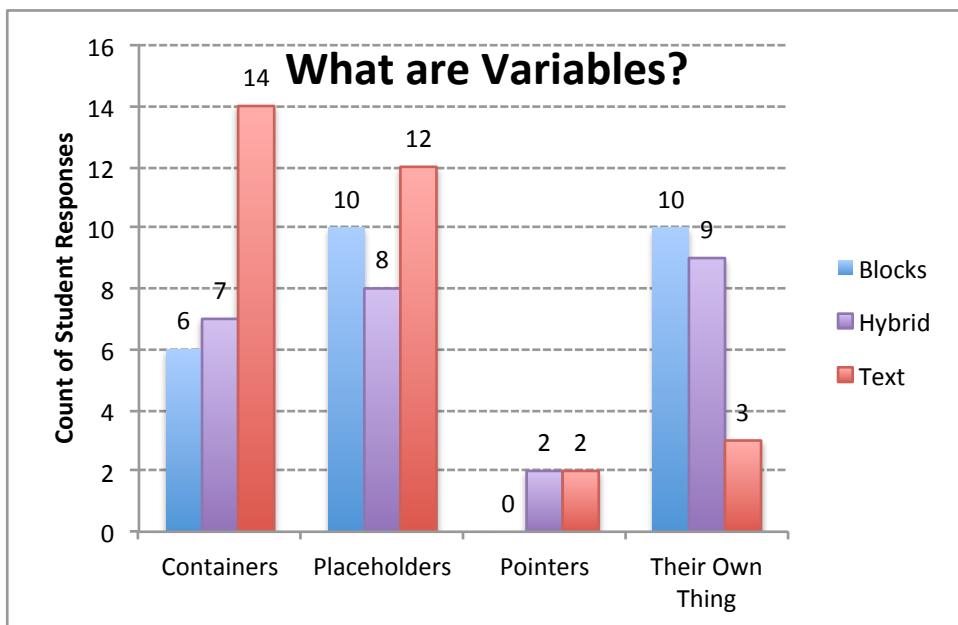


Figure 6.1. How students described variables, grouped by the form of Pencil.cc they used.

After coding all student responses, the data show that learners in the Text condition preferred the variables-as-containers metaphor, followed closely by the place holders metaphor, and were least likely to describe variables as pointers or as their own thing. Conversely, students in the block condition were most likely to use the placeholder metaphor or see variables as their own thing, and were much less likely to invoke the containers metaphor relative to their text-based peers. Throughout the various analyses of outcomes from this study, when grouping responses by condition, the Hybrid condition usually falls between the Blocks and Text conditions, sometimes falling more closely to one than the other. When looking at student responses to this conceptual question about the nature of variables, the Hybrid group aligns more closely with the Blocks condition, showing a higher frequency of treating variables as their own entity compared to text, and less likely to utilize the container metaphor favored by students in the Text condition.

One possible explanation for the higher frequency of the Blocks and Hybrid condition treating variables as their own distinct entity is in how they are presented to the user in the blocks palette. In blocks-based programming environments, variables are blocks in the same way loops are things, conditionals are things, and, in the case of Pencil.cc, visual and movement commands are things. This presentation seems to lend itself to treating variables as objects in their own right. Alternatively, the container metaphor is less intuitive from the graphical layout of the commands given the fact that variables are not visually depicted as encapsulating, holding, or in any other way containing the value. Instead, in Pencil.cc's blocks interface, the variable identified lives on one side of an = with the value on the other. A possible explanation for the Text condition's more frequent use of this explanation is that the variable-as-container metaphor is used explicitly in other courses taught by the teacher in these classrooms so, when students ask for help with variables, it seems plausible that her response would utilize this metaphor. The placeholder metaphor appears frequently across all three groups and is a perspective that aligns with how the Quick Reference page on variables describes their use. Students who sought help from the Quick Reference may have developed this intuition from the environment itself. It is important to note that none of these responses are necessarily incorrect, or more correct than the other, but instead, the differences are highlighted here to show how modality both directly (in the case of Blocks and Hybrid describing variables as their own thing) and indirectly (in the case of Text students using the container metaphor) inform emerging understandings of programming concepts.

Conditional Logic

The second question on this portion of the survey asked students about conditional logic statements, specifically asking about `if` and `if/else` statements in case students were not

familiar with the term “conditional logic”. In open coding student responses, a number of categories emerged, including students attending to how conditional statements are used to make decisions and how `if/else` blocks can introduce branching logic to a program. Codes were also added for students mentioning the need for a condition to be met, the mention of the words true or false being a component of a conditional statement, and cases where students discussed `if` and `if/else` separately or in relation to each other. There was also a code for misconceptions.

All of the codes are discussed in greater detail below. Figure 6.2 shows the result of coding student responses grouped by condition.

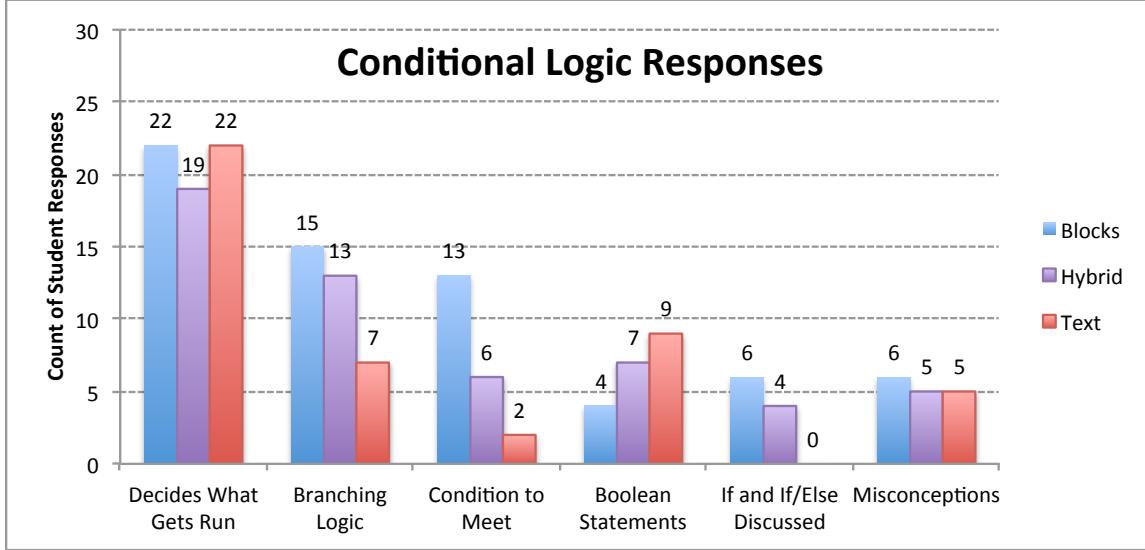


Figure 6.2. Coded student responses to the conditional logic question grouped by condition.

The first two codes in Figure 6.2 capture high level behavior of `if` and `if/else` statements. Sixty-three of the 81 responses stated that conditional logic was used to decide what code to run, and there was little difference by condition on the frequency of this response. An example of this code is the student response: “*If/else statements are used for telling a program to do something if something else is happening*”. The Branching Logic code relates to the first code, but includes responses that attend to the fact that a conditional statements can result in one thing

or another being run, i.e. that the a program's execution can branch. An example of a typical response that was coded for both of the first two codes is "*If statements are basically conditions where if a condition is met, a certain task will be done. If/else statements is almost the same thing, but if the condition is not met, then a different task will be done*". The distribution of responses across the Branching Logic question sees fewer Text condition students attend to this feature of conditional statements, a possible explanation for this will be given in the discussion of the `If` and `If/else` findings later in this section.

The next two codes, Condition to Meet and Boolean Statements, are a mutually exclusive pair that attend to how the student defines how the decision making process happens. The first code captures learners' responses that use general language suggesting that a condition needs to be met, like the response: "*If/else statements run code depending on whether a condition is met or not*". Explicitly stating that expressions are evaluated that are either `true` or `false` is the defining feature of the section code, like in the response: "*if and else statements are conditional statements. If something is true then it will do something, if something is false it will do something else*". While these two codes are conceptually similar, they are split out, as Condition to be Met is a more colloquial explanation whereas Boolean explanation is closer to textbook descriptions of conditional logic. In terms of the distribution of responses, Blocks students were much more likely to use the broader Conditions to be Met explanation, while Text students more frequently spoke about Boolean values. This distribution suggests that Text students seem to carry a more formal view of conditional logic, whereas students in the Blocks condition were more likely to use a broader, less formal description. This potentially speaks to the comfort, familiarity, and intuitiveness the modality provides to the learner. Where the Blocks conditions understand what the construct does and can thus speak about it in their own words, whereas

students from the text condition are more reliant on formal, textbook definitions to describe the behavior of the construct. For both codes, the Hybrid condition count fell between the other two.

The next code captures when learner responses discussed the role of `if` statements and `if/else` statements separately. A sample response of this variety reads “*If/else statements execute one action if a condition is true. This would be under the 'if' statement. If not, then under the 'else' statement, there would be a different set of actions*”. What is interesting about the pattern of these codes is that no students in the Text condition treated `if` and `if/else` statements as distinct things. This treatment of the two as separate may stem from the fact that, in the Blocks palette used by both the Blocks and Hybrid conditions, there are separate `if` and `if/else` blocks, whereas the Text condition never saw these two forms presented separately. While this analysis does not have the power to claim that treating `if` and `if/else` statements separately means students hold different conceptions of the construct, it does show how the environment can inform students categorization of ideas. It is important to note that across all three classes, the two forms of conditional logic were taught at the same time (i.e. they were not taught separately).

The final column captures responses that were incorrect or contained statements about `if` and `if/else` statements that were not entirely correct. These responses were equally distributed across the three conditions. A few student responses gave the impression that `if/else` statements are event-based, meaning they are always running and wait for something to happen, for example, one response reads “*If / if/else statements are used to create instances where they are to trigger once something happens. They are used to trigger when a specific time or code arrives, like if a number instance is defined, and it applies to the statement, the if/else*

statement triggers, either doing it or doing something else depending." A second example of a response of this type reads: "*If/else statements are used for telling a program to do something if something else is happening.*" Both of these responses suggest a reactive aspect of conditional logic that implies they wait for something to happen. It is worth noting some introductory programming environments such as Scratch, have an event-based blocks that demonstrates this behavior called `wait until`, which has been identified as the source of unproductive programming habits in students (Meerbaum-Salant et al., 2011). A number of students gave similar responses suggesting that `if/else` statement execute when something happens (as opposed to when a condition is true), this seemingly slight difference in language suggests a larger conceptual difference with respect to how conditional logic actually behaves.

A final aspect of the conditional logic responses that is worth highlighting is the diversity of ways students described `if/else` statements. Across the 84 responses, there are a number of different metaphors used by the students to describe conditional logic. Two students described conditional logic as a "*cause and effect*" mechanism. Other students called `if/else` statements "*Plan A and Plan B.*" A number of students described conditional logic using a navigation metaphor, "*In programs they are used to create two paths that the code can take.*" One student described conditional logic as a set of "*guidelines*" for the program, while another called them "*constraints*" for the program. This richness of metaphors highlights the diversity of resources learners have and do draw on to make sense of computational ideas as all of these metaphors can productively be leveraged to effectively reason through a conditional statement in the context of a program. A final interesting characteristic to mention from this dataset is that introduction of anthropomorphism in responses. Three students used language of this kind, saying things like "*...It's like the way a computer can be prepared*" and "*They are like a computers logical*

thinking." There is literature showing this type of language being used by younger learners in talking about computers and robots (Sharona T. Levy & Mioduser, 2007; Rücker & Pinkwart, 2015), but little prior evidence of it being used at the high school level.

Iterative Logic

Student responses to the question of what `for` loops and `while` loops do and what they are used for, were generally clear and accurate. For example, a typical response was: "*For loops makes things happen for a certain number of times. A while loop makes things happen while a condition is true. They can repeat things*". This response attends to the repeating nature of loops and identifies `for` loops as being definite (i.e. run a fixed number of times) and `while` loops being indefinite (i.e. repeat until a condition is met). In open coding the responses, a few types of codes emerged. The first two capture whether or not a given response correctly specifies how `for` loops behave and how `while` loops behave. The next code that emerged was students mentioning that loops saved the user from having to type commands over and over again. For example, one student wrote: "*Loops repeat code so you do not have to write it multiple times.*" The last three codes capture errors or misconceptions students held, which are discussed in more detail below. Figure 6.3 shows the distribution of these codes grouped by condition.

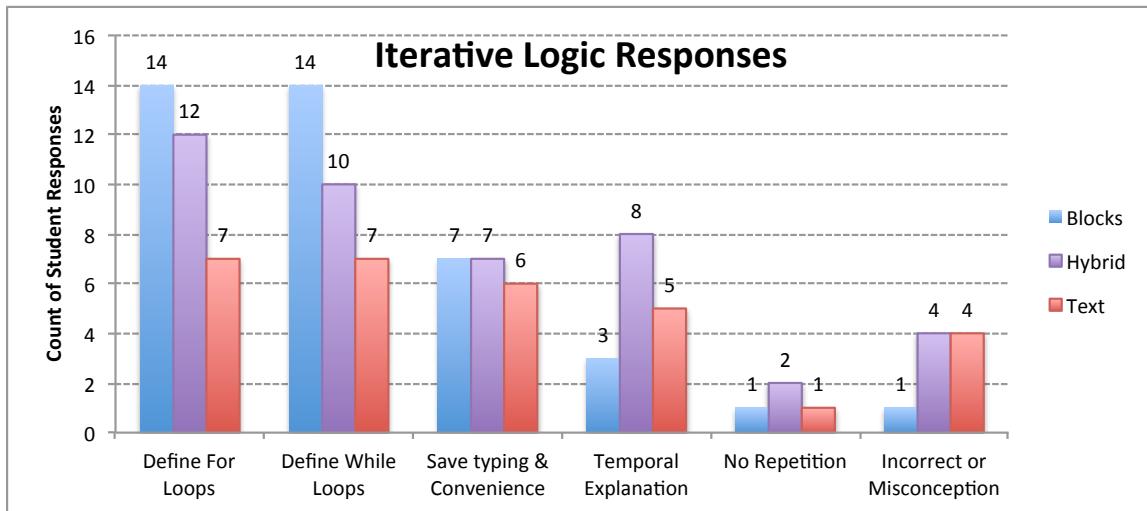


Figure 6.3. Coding of student responses to the purpose of iterative logic question.

The first two groups of columns show correct student responses describing `for` loops and `while` loops. The distribution of these columns matches the larger trend found in this study of students in the Blocks condition performing the best, while the Text condition performed the worst, with the Hybrid group being between the two. The third cluster of columns shows the number of students from each group that mentioned how looping is useful as it saves the user from having to type commands over and over again. There are two interesting things to note about this. First, is that, surprisingly, this issue came up roughly the same number of times for each condition. Given that typing is generally considered to be more cumbersome than dragging-and-dropping commands, one would expect this feature to be cited more often by the Text group. The second interesting thing to note is that only in some cases does a loop actually save typing. For example, a `for` loop that is defined to repeat five times can be replaced by copy and pasting the commands inside the loop five times, producing the same result. However, if the loop is indefinite and should repeat until a given condition is met, then it is not possible to implement

that logic without a looping construct. In this case, the loop is not saving typing but accomplishing a behavior that otherwise would not be possible.

The three remaining codes capture incorrect responses that contain some misconception. The first group, coded as “Temporal Explanations” capture responses where the students say a loop causes commands to run for a set amount of time (as opposed to a fixed number of times). For example, one student’s response starts: “*Loops make something repeat for a given amount of time...*” This misconception about repetitions being tied to time passing is interesting and not a misconception we have encountered previously in the literature. It’s possible that this is an artifact of having students learn in a turtle graphics environment where loops execute at a slow enough pace that the user can see the result of each step. In other words, when drawing a line 10 units long by asking the turtle to `for [1...10] forward 1`, the student could interpret this as run for 10 seconds, as opposed to run the `forward 1` command 10 times. It is also possible that this type of response is a result of imprecise language usage by the students. I suspect that if you were to further question the students who gave this type of response to further explain their thinking, they would not hold fast to the temporal explanation, but it is mentioned it here as it is an interesting pattern that may warrant future investigation.

There were also responses that revealed other misconceptions around looping but did not show up often enough to rise to their own category. For example, one student responded “*Loops make a program run without having to separately make the code,*” while a second student’s response was “*For and while loops are exactly what they sound like they make everything loop over and repeat. They are used in programs to restart a program for example you get a wrong answer the program will pop up once again.*” Both of these responses seem to view loops as part of the engine that drives the program, i.e. the thing that makes the program go. This is interesting

as this perception has been reported on studies of student understanding of concepts in Scratch, where it is a common practice to wrap the main logic of a program in a `forever` loop (Meerbaum-Salant et al., 2011). This is often done to make games or other programs that run continuously until manually stopped. This perspective seems less coupled to a specific modality and more to the interactive turtle graphics environment that was used, and speaks to the challenge of separating modality from the larger programming environment in which it is situated and the set of capabilities it provides. The final code is in this section captures students that gave responses that did not mention repetition, meaning they did not know what `for` or `while` loops were used for (or, more sympathetically, that they just failed to mention this defining feature of the constructs). A sample from this group reads: “*Loops are ways to simplify a programs function and can be used in several different ways*”. Only four students from the group of 81 responses fell into this category, meaning that 95% of students who went through the introductory activity were able to give correct responses to the role of looping logic in writing programs.

Functions

The final conceptual category students were asked to define on the Mid survey was functions. Two sets of codes were devised to organize the responses given by the students, but in neither case did a pattern emerge across the three conditions. The first analysis is similar to the approach presented for the responses about variables, looking at the metaphors students use to describe functions. The second analysis looks at features or characteristics that students highlighted about functions.

Across the full set of responses, a number of different metaphors were used by students to describe functions, including: functions as storage, functions as actions, functions as collections

of commands, and functions as equations. Table 6.1 includes an example response for each metaphor identified.

Table 6.1. Sample responses for different function metaphors identified.

Functions are...	Sample Response
Instruction Sets	<i>Functions are set of instructions that create things. You can change them to meet certain criteria</i>
Equations	<i>A function is an equation using two or more variables to solve another variable.</i>
Variables	<i>Functions are similar to variables where they store something, but they store a command that uses parameters, or inputs, to determine the output of the function.</i>
A Way to Do Things	<i>Functions are a way to make it easier to write large amounts of code.</i>
Storage	<i>A function is like a storage for things that need to be referenced back.</i>

Only metaphors that were used by more than 2 students are included in this analysis, so metaphors used by only a single student, such as functions are like systems and functions are like shortcuts, are not shown. Also, it is important to note that not all responses included metaphors, for example, a response like “*Functions are things with parameters that execute lines of code and reference the parameters*” are not included in this coding. Figure 6.4 shows the results for coding all student responses, grouped by condition.

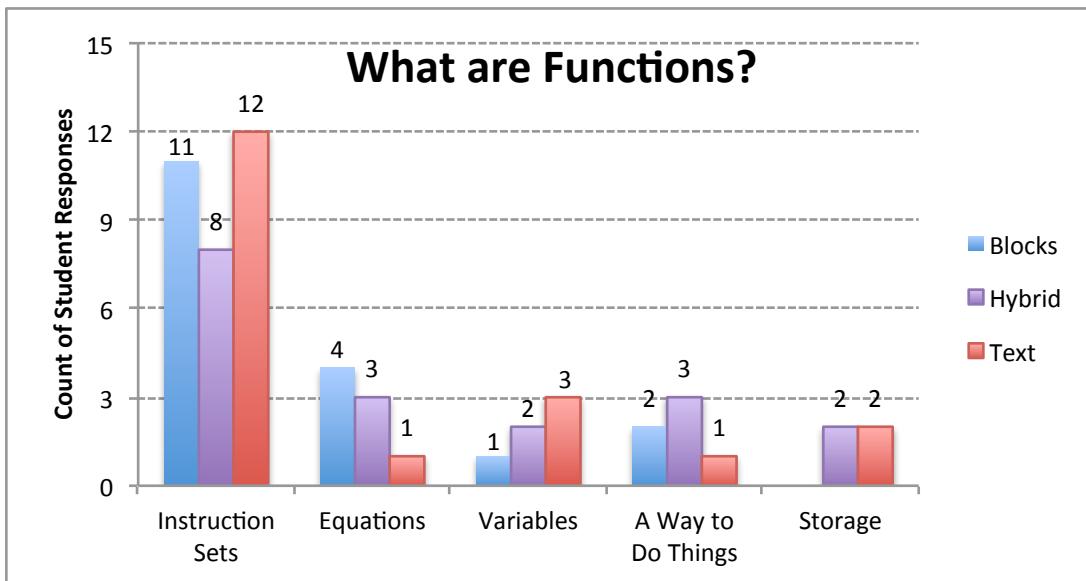


Figure 6.4. Student metaphors used to describe functions, grouped by condition.

As previously mentioned, no clear pattern emerges from this coding. There are some small patterns, but the counts are so low, that little can be gleaned from them. Instead, this analysis is presented to highlight the diversity of metaphors used and as a possible direction for future work.

The second analysis that used this data coded student responses for other aspects of functions that were attended to, such as why functions are used, characteristics of functions, and concepts related to functions. Figure 6.5 shows the results of this analysis.

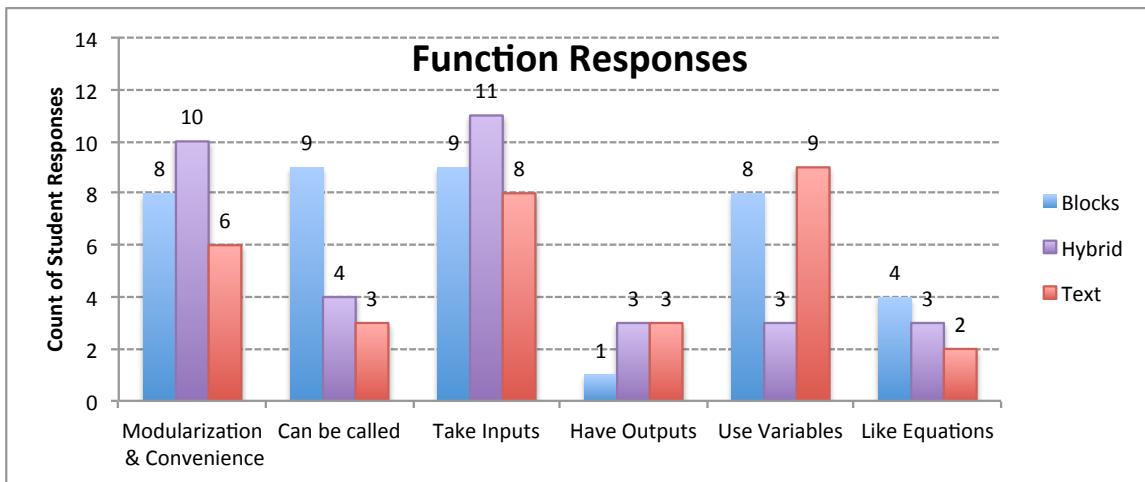


Figure 6.5. Responses to the short answer function questions coded for features of responses, grouped by condition.

Like the previous analysis on the functions responses, a clear pattern based on the version of Pencil.cc used by the student does not emerge from the data. Instead, this analysis gives insight into students' conceptions of functions more broadly. The first code, Modularization & Convenience captures responses that attend to functions being able to be called multiple times with different inputs to create different results, like the response “[Functions are] used when you want to have many different outputs so you create a function and enter in different inputs to come up with different outputs.” This code also captures responses that attended to how functions save the user from having to re-type a set of instructions every time they want to use it, as said by one student: “[Functions are] used to reuse a set of instructions without retyping it”. The second category that emerged from the open coding process was students mentioning the fact that functions are things that can be called, as one student succinctly defined a function as “*a set action that can be called upon.*” Students also attended to the fact that functions take inputs and sometimes have outputs. The previous quote used as a demonstration of functions being called multiple times was also coded for both the Inputs and Outputs categories. Many students cited

parameters as being a key feature of functions, like the student who said “*A function is a set of commands that requires parameters to perform a certain task*”. These responses were also coded in the Take Inputs category.

Interestingly, the concept of the variable was closely tied to the function concept. For example, one student wrote “[A function] is essentially a variable but in the form of a larger equation” while another responded “*Functions are a variable or variables you can make and call at a later time.*” and a third said “*A function is a special kind of variable, as it stores a list of actions to be done when the code is incorporated within the program*”. While at a certain level of abstraction the equating of variables and functions is both accurate and productive (such as when using functional languages), but functions were not used in this capacity during the five-week curriculum students followed. Instead, we suspect this relationship emerged out of the syntax for function definition used by CoffeeScript, where the function name is on the left side of an equals sign, with the parameters and definition on the right, which is quite similar to how variables are defined. Figure 6.6 shows function and variable definitions in Pencil.cc in both the blocks interface (Figure 6.6a) and text interface (Figure 6.6b).

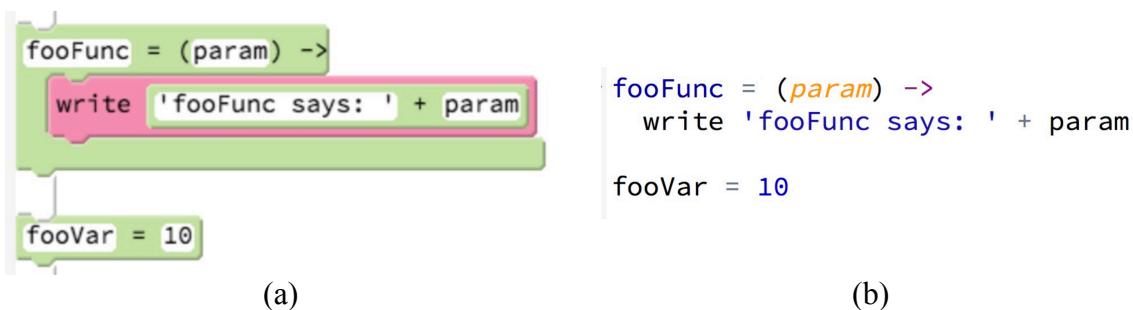


Figure 6.6. The syntax for defining functions in Pencil.cc in the blocks modality (a) and text modality (b).

While there is not a clear pattern that emerged in terms of how the three conditions informed students emerging conceptualization of functions, this association of functions with variables is a compelling piece of evidence showing how representational infrastructure and interface design can inform and shape students' emerging conceptualizations of content.

The final category that emerged from this open coding is the prevalence of students equating functions with mathematical equations. For example, the entirety of one student's response to the question was "*A function is an equation*". The linking of functions to equations seems sensible given the overlapping terminology with math classrooms. Whereas the overlap with variables emerged for representational reasons, the overlap with mathematics seems to come from terminological sources. These two differing factors, that both influence students' emerging understandings of core computer science concepts (representational and terminological), speaks to the challenge for the learner in making sense of the concepts as well as the challenge faced by educators and researchers in trying to education and interpret learning that happens in the complex world of the learner.

Having concluded our analysis of students' open-responses to prompts on the concepts covered in the five-week introductory portion of the course, the analysis now shifts to a quantitative analysis of the content assessments. This analysis begins with a brief review of the Commutative Assessment before diving into various analyses looking at differences across concept, condition, and modality. The chapter concludes with a larger discussion linking the analysis just presented with the analysis below, painting a larger picture of students' emerging conceptual understanding and the role that modality plays in this learning process.

The Commutative Assessment

The Commutative assessment is discussed in detail in the Methods chapter of this dissertation (Chapter 3), but the design of the assessment, as well as the strategy for administration during the study, are briefly reviewed here. The Commutative Assessment consists of 30 questions spread over 6 conceptual categories: conditional logic, iterative logic, variables, functions, comprehension, and algorithms. Each question on the assessment is multiple-choice and includes a short piece of code followed by a question asking students to identify the behavior of the script. The unique aspect of the Commutative Assessment is the fact that the code snippet in the question can be presented in one of three modalities: Snap! blocks (Figure 6.7a), Pencil Code blocks (Figure 6.7b), or Pencil Code text (Figure 6.7c). These three modalities are isomorphic and have the same behavior if run in their respective environments. The final important feature of the Commutative Assessment worth mentioning in this brief review is the design choice to use the existing literature on misconceptions to create the incorrect multiple-choice options. Figure 6.7 from Chapter 3 shows a sample question.

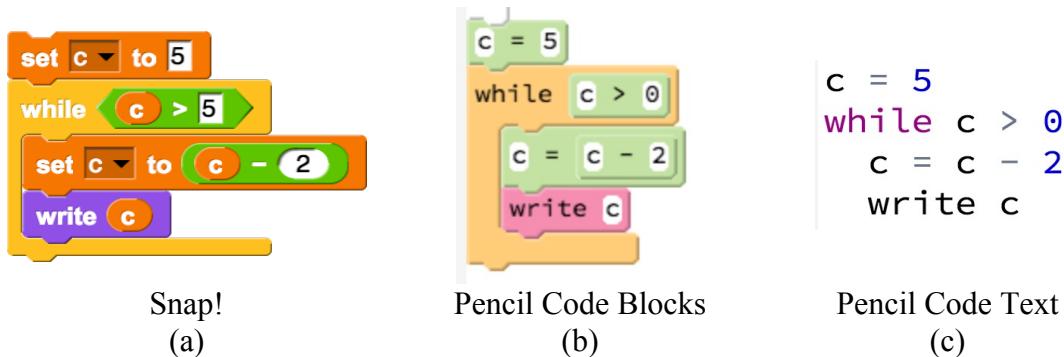


Figure 6.7. The three forms programs may take in the Commutative Assessment.

Three versions of the Commutative Assessment were created for this study. All three versions ask the same questions in the same order; the only difference is the modality of each question. Over the three administrations of the assessment (Pre, Mid and Post) students see all three modalities for every question. So if a question is asked in the Snap! modality on the first

version of the assessment, it will be in the Pencil Code blocks on the second, and the Pencil Code text on the third. Care was taken to ensure that within each conceptual category, questions in all three modality are included. So students answer questions for every concept in all three modalities on every test. Over the course of the study, students take each of the three versions of the assessment once. This means that ever student answers every question in every modality. All three versions of the survey were given at each administration, with roughly one third of the students taking each version of the assessment at each administration. Collectively, this design is meant to ensure that the results of the assessments are not skewed by having students from different conditions or at different points in time disproportionately answer a given question in a given modality.

Basic validity measures were run on the responses collected in the second year of the study and showed the assessment to have an acceptable reliability score across all items (Cronbach's $\alpha = .80$). In this section, the scores presented are calculated by averaging together every student's score for every question that fell into the grouping being presented. Grouping this way helps control for features of specific questions, and gives a more accurate within-participant score for conceptual understanding. These scores are then aggregated across the full set of participants.

Year One Concept by Modality Findings

The first analysis presented investigates if performance on conceptual questions differed by the modality the question was asked in. For this analysis, data from the first year of the study was used because the three conditions in year one were more similar, allowing the analysis to group all responses together giving more statistical power and a larger set of responses from which to investigate outcomes. Figure 6.8 shows the results of grouping student response by concept and

modality from the first year of the study. It is important to note that while the questions on the first year of the Commutative Assessment were largely the same¹⁹, the text questions were presented in JavaScript, as opposed to CoffeeScript, so there is a slight difference between the administrations across the years. Also, in this section, the blocks-based questions were rendered with the *Snap!* notation, as the learning environment students used in the first year (*Snappier!*) was based on *Snap!*

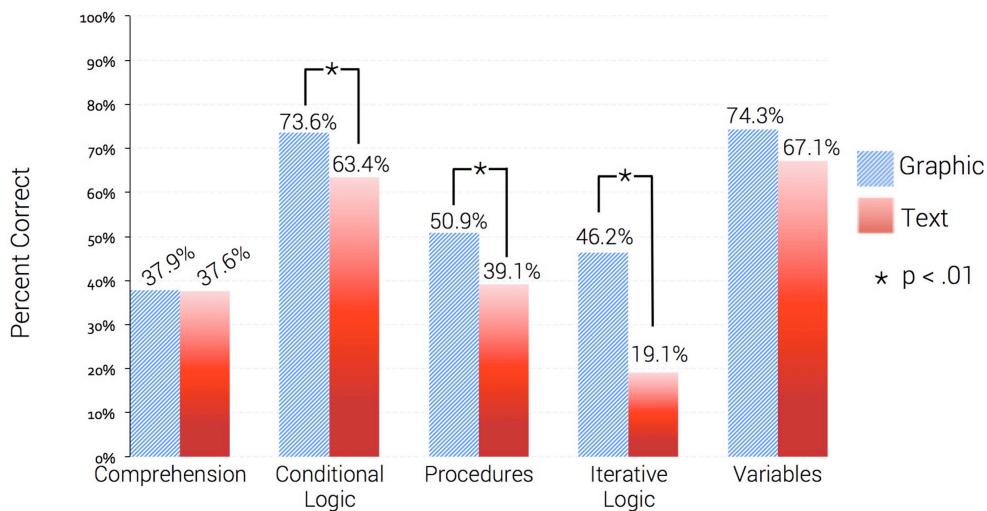


Figure 6.8. Student performance on the Commutative Assessment grouped by modality and concept.

Looking across the five conceptual categories covered in the Commutative Assessment using paired-samples t-tests, the results show that students perform significantly better with the blocks-based modality on questions related to iterative logic $t(178) = 10.40$, $p < .001$, $d = 1.57$, conditional logic $t(178) = 2.82$, $p < .01$, $d = .41$ and functions $t(178) = 2.89$, $p < .01$, $d = .41$. Students also performed better in the graphical condition on variable questions, but not significantly so, $t(178) = 1.66$, $p = .10$, $d = .25$. Interestingly, there was almost no difference in

¹⁹ A few minor revisions were made between the two years, often in the form of new incorrect responses being added to try and tease out further misconceptions, although a few code snippets were modified and a small number of questions were added.

how students performed on the comprehension questions between the two modalities $t(178) = .094$, $p = .92$, $d = .01$. These data provide evidence showing that yes, modality does affect novice programmers' understanding of basic programming concepts. Further, these data show that the effect is not uniform across concepts and does not seem to influence comprehension of programs in the same way it effects basic understanding of what a construct does within a program. Seeing that a difference does exist, the analysis continues by investigating each category more carefully, looking at how specific concepts are differentially influenced by modality and if they can be explained by misconceptions from the literature.

Iterative Logic

While iterative logic showed the largest difference in scores between blocks-based and text-based questions, a closer analysis of the questions shows that a majority of this difference can be attributed to the difficulty students have with the structure of `for` loops (du Boulay, 1986). Two of our five iterative logic questions compared a graphical `repeat` block to a text-based `for` loop (Figure 6.9).

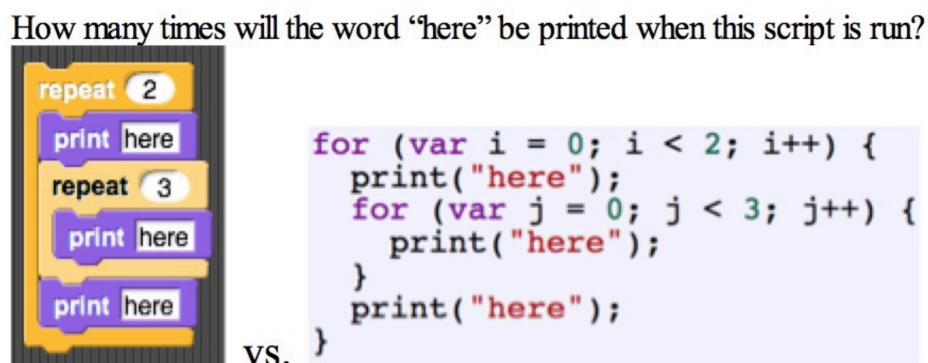


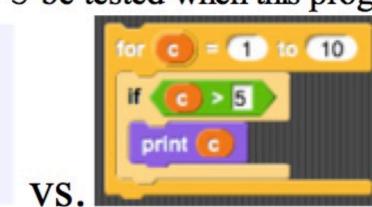
Figure 6.9. A sample iterative logic question from the Year 1 version of the assessment.

On these two questions, students performed significantly better in the graphical condition (83% correct) versus the text-based `for` loop version of the question (16.1% correct). This

provides compelling evidence for the finding that students find the `repeat` command common to blocks-based languages easier to understand than text-based `for` loops, a finding already documented in the literature (Stefik & Gellenbeck, 2011; Stefik & Siebert, 2013). By examining the incorrect responses given by students, we can glean additional information about how students understand the concepts with respect to the way they are presented. For example, on the text-based `for` loop questions, almost half of the students (49.3%) chose an answer that had each command inside the `for` loop run once and only once – suggesting it was not clear that any looping was going to occur. When answering the same questions with the graphical `repeat` blocks, only 1.5% of students chose those options. Second, in the text-based conditions, 20.7% of students chose the answer that suggested the number of times a given `for` loop would run was variable, and would be different each time it was executed. In the graphical `repeat` versions of the questions, only one student chose this option. The Commutative Assessment includes one looping question that compared a blocks-based version of a `for` loop to a text-based version (Figure 6.10).

How many times will the comparison `c > 5` be tested when this program is run?

```
for (var c = 0; c < 10; c++) {
  if (c > 5) {
    print (c);
  }
}
```



VS.

Figure 6.10. Comparing blocks-based and text-based `for` loops.

On this question, students performed comparably, answering the question correctly 19.6% percent of the time in the graphical condition and 18.0% of the time in the text-based condition. Two reasons may explain the lack of a difference on this question compared to what we saw on the two questions that use `repeat`: the confusion around the use of the term “`for`” to

capture the concept of looping and the lack of transparency in how for loops behave based on this conventional representation (du Boulay, 1986; Steifik & Gellenbeck, 2011). This outcome, along with the other for loop questions adds to the evidence that students find the word “for” unintuitive, and that “repeat” better describes the looping behavior. As there are languages that utilize the keyword “repeat” (Logo in particular comes to mind), this finding speaks more to language design than features of the modality.

The two indefinite loop questions use the `while` construct. There was little difference in performance between the blocks-based and text-based versions of these questions. For both questions, students’ performance was very similar (a difference of .6% and 2.3% for the two questions). A closer investigation of the answers given (including incorrect answers) does not show a systematic difference between the types of representations used. This suggests that, on indefinite loops, the blocks-based representation does not seem to provide any distinct advantage over a comparable text-based implementation. The lack of a difference between the two modalities when using comparable syntax/keywords, both with while loops and for loops, matches the finding from Lewis (2010), who found no significant difference in accuracy between questions asked using the repeat block in Scratch and the repeat command in Logo. This suggests that for iterative logic, the blocks-based representation does not provide additional conceptual support; meaning the nested scoping and visual syntactic information did not better support student comprehension. A closer analysis of the five iterative logic questions only reinforces what we already know about the difficulty learners have with `for` loop syntax.

Conditional Logic questions

Students performed significantly better in the blocks-based modality on three of the five conditional logic questions. On one question the students performed comparably (.34% better on

the blocks-based form), and on the last question students performed slightly better on text, scoring only 2.72% higher. On this final question, students were asked about the overall behavior of the script rather than just about the output. This brought it closer to our comprehension questions than the others, which may in part explain the better performance for the text-based representation - this issue is revisited later in the section. On the three questions where students performed better in the graphical condition, two patterns emerged in analyzing the incorrect responses, revealing a slight systematic bias. First, on the two questions where the test of an `if/else` statement evaluated to true, students in the text condition were more likely to think both the `if` and the `else` branches would execute (11.5% for text versus 7.1% in the graphical case). This misconception has been identified in the literature (D. Sleeman, Putnam, Baxter, & Kuspa, 1986) and is part of the work showing the `if/else` construct to be challenging for learners. Second, we found that students in the text condition were more likely to think the last statement is the one that is evaluated regardless of the outcome of the conditional logic surrounding it. On all three questions where this was a possible incorrect answer, students were more likely to choose it in the text-based condition (10.7% for text, versus 3.5% in blocks). This could be explained a number of ways including: students thinking that the body of a conditional statement gets executed regardless of the outcome of the conditional test, thinking the `else` outcome is always evaluated (which matches the first misconception identified and could explain two of the three questions we saw this error in), or not know how or when conditions evaluate to true so defaulting to falling through to the last statement. Overall, the finding that students performed better on blocks-based conditional logic questions matches Lewis' previous work (2010).

Variables Questions

Like with the two previous conceptual categories, students performed better (although not at a statistically significant level) on the variable questions when they were presented in the blocks-based form. A more detailed look reveals that students only performed better on the graphical case on three of the four questions in this category. On the one question where students performed better in the textual modality (Figure 6.11), one difference stands out from the others: variables are set then used, but never re-assigned, making it the simplest of the four questions.

What will be printed after running this script?

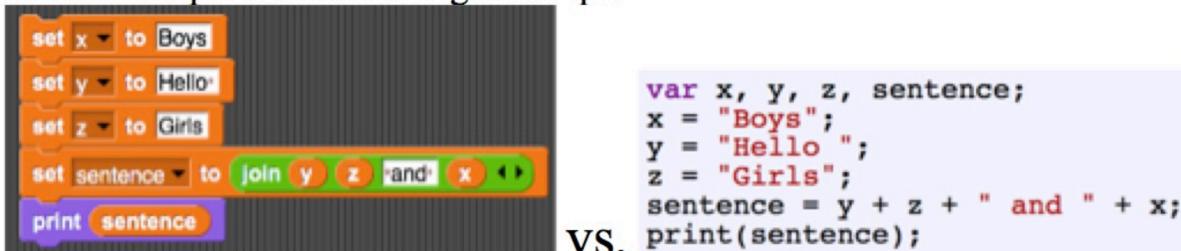


Figure 6.11. The variable question that students performed better in the text condition than the blocks-based condition.

This suggests that the text-based representation is comparable to the blocks-based version for simple variable assignment and usage, but that as statements and programs get more sophisticated (i.e. variables are assigned to other variables or variable values are set then reset), that the blocks-based modality is more intuitive for learners.

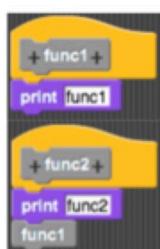
Looking at the incorrect responses given by students across the four variable questions reveals three findings that link modality to the existing misconceptions in literature on variables. First, all four questions included an option that would be chosen by students who mistakenly thought expressions do not get evaluated as part of assignment (option D in Figure 3.2 of Chapter 3) and for all four questions, this incorrect option was chosen slightly more often in text form (7.3% of text responses, 5.3% of graphical). A possible explanation is that the text form does not provide visual hints about how to parse the statement. Second, we found that on text-

based questions, students were more likely to incorrectly choose the answer that would result if variables held their initial values, meaning the values do not get overwritten (30.6% in text, 14.5% in graphical). This misconception has not been previously discussed in the literature. The hypothesis is that, in the case where students do not know what is supposed to happen when a variable that already contains a value has a new value set to it, the assumed behavior is for nothing to happen, i.e. the new value is ignored and the original value retained. Finally, students were also slightly more likely to choose answers that fit with the linked variables misconception (option A in Figure 3.2 of Chapter 3) in the text questions (23.4% of text responses, 17.4% of graphical).

Function Questions

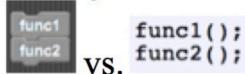
The fourth category of questions asked students about the outcome of running programs that contained function calls (Figure 6.12). On these questions, students performed better on the blocks-based version on four of the five questions we asked. Looking at the errors students made, there were a few cases where students showed signs of displaying documented misconceptions and other patterns that seem systematic, but are new to this work and can, at least partially, be explained by features of the modality.

Here are two functions:



vs.

What is printed when this script is run?



vs.

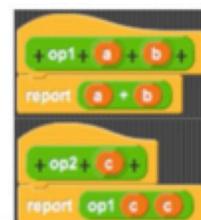
Here are three functions that each perform a mathematical operation:

```

function op1(a, b) {
    return a + b;
}

function op2(c) {
    return op1(c, c);
}
  
```

vs.



What is output of this program?

```

print(op2(10));
  
```

print op2 10

(a)

(b)

Figure 6.12. Two sample function questions.

First, one of the questions intentionally included a program that output the same word twice in a row, meaning the correct answer include the duplicated word while other choices included what students might assume was intended. Over half of the students (57%) in the text version of the question incorrectly chose the non-duplicated responses, compared to 38.6% of responses in the blocks-based version of the question. This suggests students found it easier to trace the flow in the blocks-based modality and were less likely to fall victim to what Pea (1986) calls an “intentionality bug”, where the learner assumes the computer knows the programmer’s intention. A second systematic finding from analyzing these questions reinforces a trend, observed in the variables questions, that students answering text-based questions were more likely to think that expressions do not get evaluated but instead retain the expanded form (44% for text versus 31% of graphical responses). A third trend we found is that students were twice as likely (50% compared to 22%) to think that an unbounded recursive function stopped after a fixed number of calls in the text-based form than the blocks-based modality. Finally, two of the questions included functions that return values (`report` is the keyword used in the graphical form). Figure 6.12b provides an example of this type of question. Across these two questions, students were almost twice as likely to think the return command would cause an error in the text-based form (24.5% of responses) than the blocks-based alternative (13.2% of responses). In this case, one can point to a feature of the blocks-based modality that can account for this difference. In the blocks-based language, functions that return values are depicted as ovals or hexagons that need to be nested inside another block (like `op2` in Figure 6.12b), whereas functions that do not have return statements take the shape of the interlocking blocks (like the

func1 block in Figure 6.12a). This visual difference at the place where the function is being invoked and the ability for the blocks-based representation to enforce syntactic validity, provide a pair of scaffolds for the learner that potentially explains this difference in student responses in the two modalities.

Comprehension Questions

The final type of question on the assessment is program comprehension. These questions, unlike the others, focus more on the purpose of a script rather than on specific outcomes. In each case, the question students must answer is: what does the following script do? These questions require students to mentally run the program, often for different sets of potential inputs, and then interpret that behavior into a natural language description of the behavior. Figure 6.13 shows two examples of these questions, with the correct answer being that the program swaps two values (left) and returns the largest of the three numbers (right).

a, b and tmp are variables. What does this script do?

```
set tmp to a
set a to b
set b to tmp
```

vs.

```
tmp = a;
a = b;
b = tmp;
```

(a)

The function op4 takes in 3 numbers. What does op4 function do?

```
function op4(a, b, c) {
  var tmp;
  if (a > b) {
    tmp = a;
  } else {
    tmp = b;
  }
  if (c > tmp) {
    tmp = c;
  }
  return tmp;
}
```

(b)

Figure 6.13. Two comprehension questions.

Across the full set of questions, students performed comparably on the comprehension questions by modality (a difference of less than 1%). Looking at the questions individually reveals outcomes that correlate with the trends of how students did on questions from the

conceptual category of the constructs used in the question. So, for example, question b in Figure 6.13, involves conditional logic and we found students performed better on the graphical versions of the question. Conversely, on a comprehension question that included a `while` loop, students performed better in the text condition. Because these questions involve the additional step of interpreting the behavior of scripts and the intention of the author, it becomes more difficult to map incorrect responses to specific misconceptions from the literature. Additionally, the small difference in performance between blocks-based and text-based questions is also interesting as it is the only category for which this is true, which leads to some potentially interesting conclusions. Notably, this suggests that while the graphical representation supports students in understanding what a construct does (i.e. what the output from using it is), that support does not better facilitate learners in understanding how to use that construct.

Concept By Modality Discussion

On three of our four conceptual categories there were significant differences in performance between modality, with the fourth category showing a similar, though less pronounced, trend. Three features of the blocks-based modality in particular stand out as possible explanations for this result. First, the graphical nesting of the blocks to denote scope appears to be an effective way to depict this concept, as we saw fewer errors made on blocks-based versions of questions where such misconceptions might be found. For example, it was more prevalent in the text-based condition for students to incorrectly think both branches of an `if/else` statement will be run. The difference between `{}`s and visually nested commands provides one plausible explanation for this. This finding is consistent with the discussion in the previous section on conceptual understanding and will be revisited at the end of the chapter. Second, the fact that the blocks-based modality allows for statements that can be closer to natural language

can, in part, explain some of the differences found. Notably, the command to assign values to variables takes the form of `set __ to __`, which is a closer description to what the command does than the comparable text-based language command of `var __ = __`. This difference is not a feature of the blocks-based modality, but instead an example of the language designer taking advantage of the more conversational format that the block-based modality enables. This difference can explain at least part of the differences we saw in the variable questions. Finally, the different shape of commands that return values from those that carry out actions in the blocks-based modality provides a compelling explanation for some of the differences we found in the function questions.

One of the more interesting outcomes from this work is the uniformity among student performance on the comprehension questions. There are a few possible ways to explain this. One explanation is that the gains learners get from the graphical affordances of the blocks-based modality, which support conceptual understanding of specific constructs, do not carry over to slightly more challenging comprehension tasks. A second possible explanation is that it takes longer than the time allotted in the study for the gains from the graphical layout to apply to these types of questions. If this were the case, we would expect that if given more time, we would see similar gaps in performance emerge. A third possible explanation is that the modality has little effect on student comprehension, which seems at odds with other findings presented above showing the difference to exist, but it is still possible. This section provides evidence showing that modality matters with respect to reading programs. In the next section, the analysis looks at how learning and performance differ across the three conditions of the study while trying to link the learning environment with conceptual outcomes.

Learning Outcomes by Condition

Having looking at the conceptual differences using qualitative methods and showing that modality does matter with respect to students interpreting programs, the section now turns to if and how the modality used by the learner influences their ability to read and interpret programs in different modalities and employing different concepts. The first objective of this section is to show there is no difference across the three conditions in their performance on the pre-assessment that might skew later findings. On the Pre content assessment, the mean scores by condition are: 54.3% ($SD = 12.2\%$) for Blocks, 53.4% ($SD = 16.2\%$) for Hybrid, and 51.6% ($SD = 14.5\%$) for the Text condition. Running an analysis of variation calculation on these three scores show them to not be statistically different from each other $F(2, 84) = .27, p = .76$. This lack of difference means that the three classes are not different from each other with respect to their incoming programming knowledge.

With that established, we now move forward with our analysis of learning gains by condition. Figure 6.14 shows cumulative scores for students across the three conditions on the Pre, Mid, and Post Commutative Assessment administrations. Note the y-axis on the graph does not go from 0% to 100%, but instead from 20% to 90%. This is done to make the differences in conditions more clear.

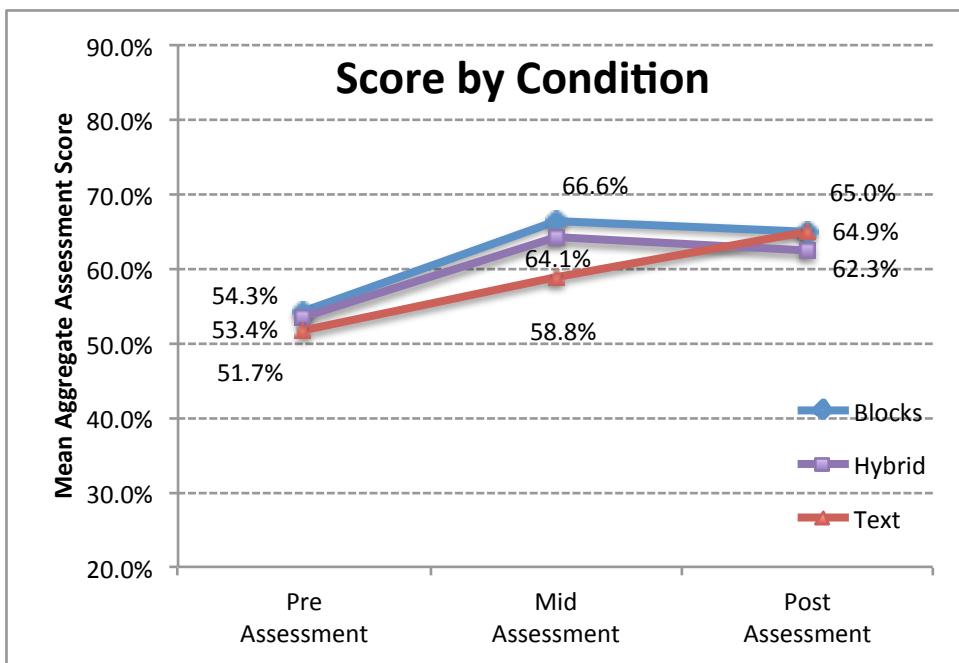


Figure 6.14. Student Commutative Assessment scores by condition over time.

The positive slope for all three conditions between the Pre and Mid assessments means that, in aggregate, students in all three classes performed better on the Mid survey than they did on the Pre. Given that this was an introductory class it is not surprising, but still noteworthy and an encouraging sign given that these three conditions cover almost the entirety of the modalities used to introduce learners to programming. For all three conditions, the improvement on test scores from the Pre to the Mid is significant (Blocks $t(24) = 6.11$, $p < .001$; Hybrid $t(26) = 6.65$, $p < .001$; Text $t(26) = 3.70$, $p = .001$). While the improvements are all significant, the Blocks condition saw the largest absolute gain, followed by the Hybrid condition, with the Text group showing the most moderate gain.

To answer the question as to whether or not students performed differently on the Mid survey, an ANCOVA calculation was run showing a significant difference in student scores by condition when controlling for Pre scores $F(2, 75) = 4.53$, $p = .01$. A Tukey HSD post hoc test shows the difference between Text and Blocks to be significant at $p = .01$, while the Hybrid-

Block difference and Hybrid-Text distinctions are not ($p = .39$ and $p = .20$ respectively). This means that the students in the Blocks condition did significantly better than students in the Text condition on the Mid content assessment controlling for Pre scores. The Hybrid condition students scored better than Text students, but not as well as Blocks, but neither difference was significant.

Turning the focus to the Post results, we see the gap between conditions that emerged at the Mid point close, with all three conditions showing very similar final scores. An ANCOVA calculation on the Post assessment controlling for Mid scores shows the three conditions to not be statistically different from each other $F(2, 74) = .85, p = .43$. This mean that even after controlling for the variance in prior scores, the three conditions' Post scores are comparable. Unlike the Pre to Mid change, only the Text condition had a positive slope on the Mid to Post scores, meaning both Blocks and Hybrid students performed worse on the Post than they did on the Mid administration of the Commutative Assessment. Comparing how the three conditions' scores changed from Mid to Post, the data show a significant difference $F(2, 75) = 5.16, p = .008$. A Tukey post hoc analysis shows there to be a significant difference between the Text and Hybrid changes ($p = .01$) and the Text and Blocks differences ($p = .03$), while there was no difference in the changes made by the Text condition relative to the Blocks condition ($p = .88$). To complete the analysis, looking at changes within each condition, none of the three conditions showed a significant change between the Mid and Post assessments: Blocks $t(26) = -.28, p = .78$; Hybrid $t(23) = -.84, p = .41$, and Text $t(25) = 1.55, p = .13$.

These data show that students in the Blocks and Hybrid conditions saw the most gains over the course of the five-week introductory period with respect to performance on the Commutative Assessment. After the transition to Java, neither the Blocks nor Hybrid conditions

improved, while the Text condition saw another incremental improvement, resulting in all three conditions performing comparably on the assessment given 15 weeks into the school year. One possible explanation of these findings is that there is a ceiling effect for learners and that the Blocks and Hybrid conditions reached that ceiling faster than the Text condition. In other words, learners in the two conditions that enabled drag-and-drop composition were able to more quickly understand the concepts at hand, while the Text condition took longer to make sense of the activity of programming before reaching the ceiling associated with the curriculum students worked through. This explanation partially fits with the attitudinal data presented in the previous chapter as students in the Text condition saw increased levels of confidence, enjoyment, and interest in computer science between the Mid and Post surveys. The finding that blocks-based learning environments allows students to learn more quickly has been shown in some small studies in informal environments (Price & Barnes, 2015), so this suggests this may be a larger, more robust phenomenon.

An interesting thing to consider is how and why student performance improved for students in the Text condition after ten weeks of working in Java given the fact that there was relatively little overlap in content between the ten weeks in Java and what was covered on the Commutative Assessment. Additionally, students did not encounter any blocks-based programs between the Mid and Post administrations. In other words, between the Mid and Post administration, in the Text condition student performance improved despite not seeing the content or the modality. This suggests that in their time working with Java on topics like basic I/O and method calling, students' general understanding of programming concepts, or at least their ability to interpret programming across different modalities, improved. This finding is unexpected and it will take effort to interpret. In the sections that follow we dig more deeply into

these data to try and put together potential explanations and gain a more nuanced understanding of these data and the role of modality and concept on learning and on learning by condition.

Condition by Modality

To better understanding the learning gains found in the previous section, we now take a closer look at the data to try and understand the source of these learning gains, specifically looking between the Pre and Mid surveys in hopes of attributing learning gains to the modalities used in the introductory learning environments. First we look at differences in outcomes by modality, before looking at conceptual outcomes. Figure 6.15 shows mean student scores on the Mid administration of the Commutative Assessment grouped by Modality and Condition. As a reminder, the three modalities used to present the questions can be seen in Figure 6.7.

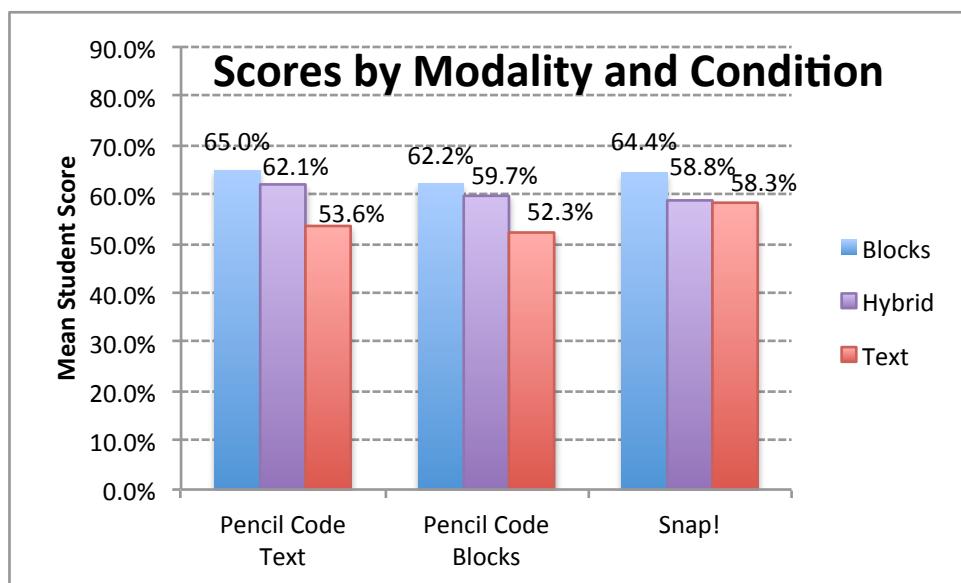


Figure 6.15. Student scores on the Commutative Assessment grouped by modality and condition.

Across all three modalities, the ranking of student performance by condition is the same: Blocks students performed the highest, Text students the lowest, with students from the Hybrid condition performing between the two, sometimes closer to the high scores from the Blocks condition (on Pencil Code Text and Pencil Code Blocks questions) and once closer to the lower Text scores (the Snap! questions). Running an analysis of covariance calculation for each group (again controlling for pre test scores) shows a difference between conditions on the Pencil Code Blocks questions $F(2, 75) = 4.77, p = .01$, but no difference for the Pencil Code Text ($F(2, 75) = 1.5, p = .23$) or Snap! ($F(2, 74) = 1.25, p = .29$) questions. A Tukey HSD post hoc test for the Pencil Code Blocks questions show there to be a significant difference between the Blocks and Text conditions ($p = .01$), but not between Hybrid and either the Blocks ($p = .60$) or Text ($p = .10$).

While not much can be definitively said about the relationship between modality and condition due to the relative lack of statistical power from this sample, there are suggestive trends that are important to note. The first, mentioned in the previous paragraph, is the consistent ordering of the three conditions in terms of performance. The fact that the Blocks condition performed highest on all three modalities suggests that the understanding that forms in one modality is not tightly coupled to that modality. An alternative interpretation of this finding is that the ability to make sense of programs developed in the blocks-based modality is not tightly coupled to that modality. This suggests a potential form of near-transfer from the blocks-to-text modality, but the data presented are not robust enough to strongly support this claim. This lack of statistical differences across modalities does, in part, fit with other work showing a lack of conceptual transfer in novices when learning a second programming language (Scholtz & Wiedenbeck, 1990; Wiedenbeck, 1993). However, it is important to note that unlike this prior

work, the question being pursued here is focused on modality as opposed to the language itself. This differing trend can be interpreted by saying that transfer across modality (i.e. from Pencil.cc's blocks interface to Pencil.cc's text interface) is 'nearer' than moving across programming language (like from Java to Python). A final thing to note is that once again the Hybrid conditions performance lives between the Blocks and Text condition, a recurring position for that condition across a number of analyses presented.

Condition by Concept

The next analysis presented looks at difference in conceptual understanding by condition and concept. This section answers the question of whether or not certain concepts are more easily learned through working in one modality versus another. Figure 6.16 shows student performance across the six concepts assessed on the Commutative Assessment. As a reminder, these scores are only from the Mid administration of the assessment, meaning students had just completed five weeks of working in their version of the Pencil.cc environment.

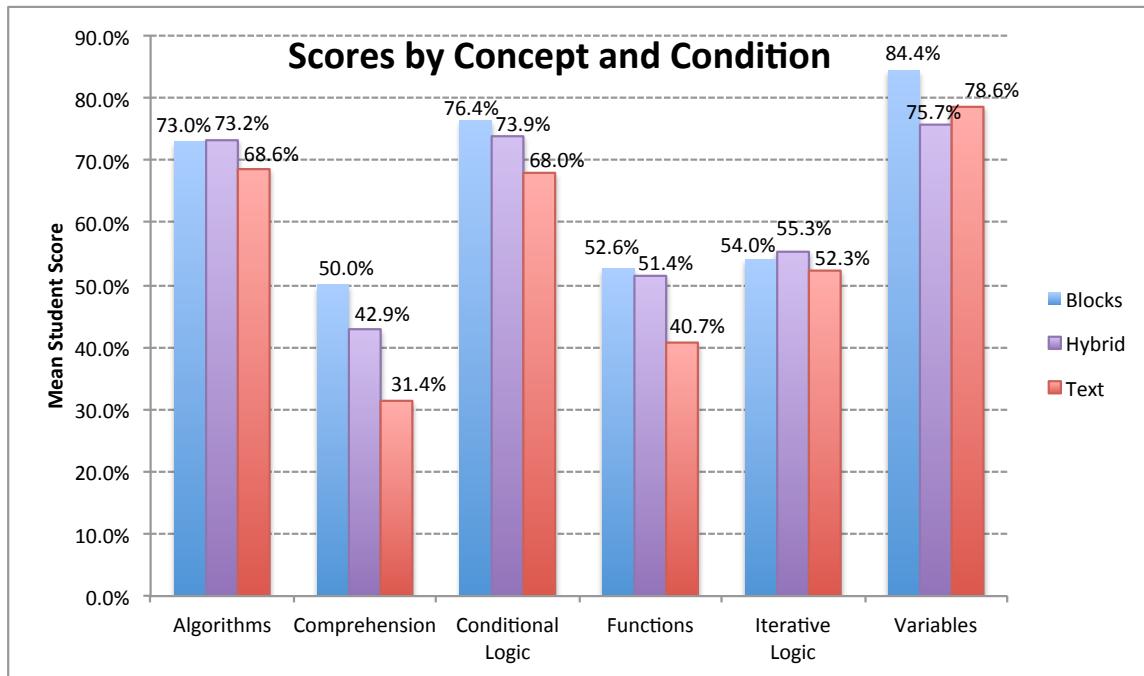


Figure 6.16. Student performance on the Mid administration of the Commutative Assessment grouped by condition and concept.

Like with the previous analyses, the Hybrid condition often scores between the Hybrid and Text, however, in this by-concept analysis, this is the case only half of the time. In these three conceptual categories (Comprehension, Conditional Logic, and Functions), we find the pattern of students in the Blocks condition scoring the highest, with Hybrid students in the middle, and the Text condition scoring the lowest. For two types of questions (Algorithms and Iterative Logic) the Hybrid condition scores the highest, while in Variables, students in the Hybrid condition scored the lowest. In none of the six categories did students from the Text condition score the highest.

Running an ANOVA calculation on each concept category shows only a significant difference across the three conditions for the comprehension questions $F(2, 80) = 4.95, p = .009$. A Tukey HSD post hoc analysis shows that significant differences exist between the Text and Blocks condition ($p = .01$) and the Text and Hybrid conditions ($p = .10$). This difference by

modality is largely driven by the extremely low score on the comprehension questions by students from the Text-based condition. The finding that students scored particularly low on the comprehension questions echoes the analysis from the year one assessment (Figure 6.8) and matches prior work on students' difficulties in drawing larger meaning and purpose when reading programs (A. Robins, Rountree, & Rountree, 2003). That the scores on these questions were the most stratified suggests that comprehension may be one place that learning with a specific modality may be helpful. When composing programs with blocks-based tools, the user has compositional units that match the larger cognitive building blocks (the command itself) allowing less cognitive effort expended on the implementation of that idea, and thus, the learner has more practice thinking at a conceptual level.

Perceived Ease-of-Use of Concepts by Condition

The last analysis in this chapter looks not at emerging conceptual understanding or performance on the assessment, but at the perceived ease of the concepts covered. On the attitudinal assessment given at the midpoint of the study, there were a series of questions asking about perceived ease-of-use of the various programming concepts covered on a 7-point Likert scale. The mean responses to the Likert questions are shown below in Figure 6.17. The higher the score, the easier a student thought it was to use the given concept.

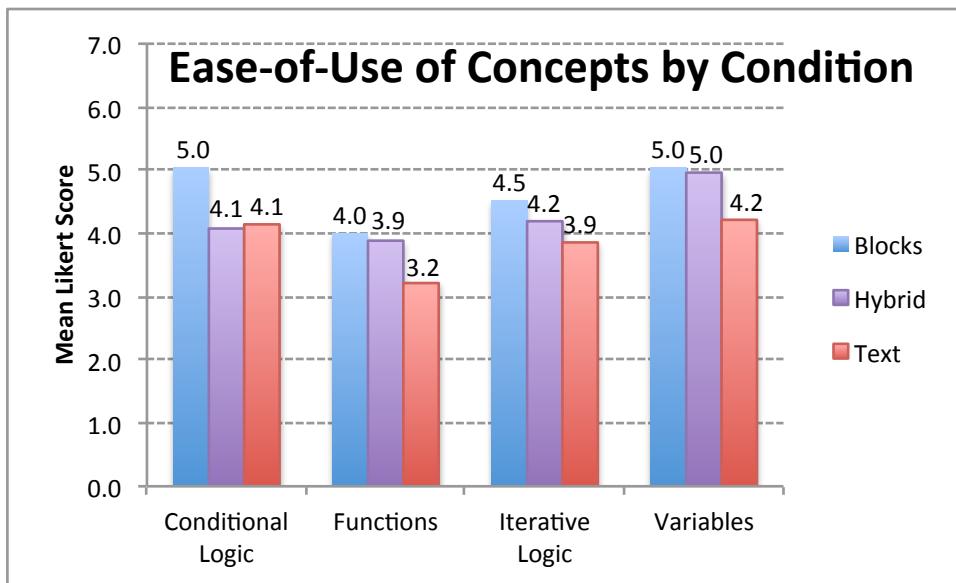


Figure 6.17. Student reported ease of using concepts in their respective version of Pencil.cc.

Running an ANOVA calculation for each conceptual category finds two of the four concepts to be statistically significant, Conditional Logic $F(2, 78) = 2.92, p = .05$ and Variables $F(2, 78) = 2.63, p = .08$. A Tukey HSD post hoc analysis for the Conditional Logic scores shows the Blocks condition to be moderately different from both the Text condition ($p = .10$) and the Hybrid condition ($p = .09$). In other words, the students found conditional logic easier to use in the Blocks condition than in either the Text or Hybrid condition. The difference between Blocks and Text conditions can be explained by the modality itself; in other words, students found using conditional logics in the drag-and-drop blocks modality to be easier than the all text condition. That the Hybrid condition was the lowest of the three, and very close to Text, suggests that learners in the hybrid condition viewed using conditional logic in their programs more like the Text students than the Blocks students. This is possibly explained by the fact that the Hybrid implementation used allowed learners to add new statements to their program via a drag-and-drop mechanism, but once added, all future edits were made with the keyboard since the editor presented code in a text format. As will be shown in the next chapter detailing how learners used

the environments, in the Hybrid condition, students often did the supplemental editing (like making minor edits to their code) with the keyboard. This may have contributed to the ease-of-use being more comparable to the Text condition than the Blocks condition. Another explanation that will be discussed in the next chapter has to do with how the students chose to use the drag-and-drop feature of the Hybrid interface, specifically, the practice of using the blocks for reference as opposed to for composition, especially for the conditional statements and iterative logic.

The second significant difference found is with the concept of Variables, where the Text condition was the outlier, having a significantly lower reported ease-of-use score than the other conditions. When comparing the Blocks condition to the Text condition, the Tukey HSD post hoc analysis score was $p = .10$, and the Blocks to Hybrid condition also did not reach a level of significant ($p = .15$). Whereas the use of a conditional statement in the Hybrid condition often included non-trivial post addition edits (i.e. defining the test and the commands to be followed after the test is evaluated), when adding variables to a program via drag-and-drop, only very straightforward edits are necessary, like entering the name or changing the value, which in both case are simple substitutions. This is in contrast to working with conditional statements or iterative blocks that also require introducing new nodes into the program's abstract syntax tree (i.e. adding new blocks to the program in the Blocks condition). In this way, the addition of variable blocks to a program in the Hybrid condition presented a template that required simple replacements, a pattern of use frequently observed in the Hybrid condition. Looking at the amount of modification after an addition also seems to explain the perceived ease-of-use for the other two categories, in that Functions usually require only the renaming or updating of elements present in the blocks. Iterative Logic in the Hybrid condition was reported between Blocks and

Text, which can be explained, in part, by the fact that definite loops (`for` loops) are more like variables and Functions requiring only direct replacement, whereas indefinite loops (`while` loops) are closer to conditional logic in that they require more substantial edits to be made. So one explanation for Hybrid iteration living roughly halfway between the Text and the Blocks responses (unlike the other three conditions) is the blending of the easily substituted `for` loop and the more complicated `while` loop. The fact that all iterative logic is grouped together means this survey does not have the specificity to tease apart these two looping constructs. Looking across the data, one explanation for ease-of-use in the hybrid condition is that the simpler a required post-addition modification is, the easier the construct to be used is perceived.

Comparing Figure 6.17, which shows the ease-of-use of concepts, and Figure 6.16, which shows scores by concept and modality, for the four concepts that overlap, it appears there is a relationship between how easy a concept is perceived to be and how well students did on that question. Running a Spearman rank-ordered correlation returns a value of $r_s = .78$, showing a high correlation between students perceived ease of use of a concept and their performance on the assessment of that concept. This suggests that students own perceptions of ease-of-use are accurate predictors of their knowledge of that concept. That said, there are some seeming outliers, like the Hybrid condition's relatively low ease-of-use score for conditional logic while performing well on those questions. Likewise, the Text condition's reported ease-of-use for variables does not align with how well they scored on the perception questions.

Discussion

This chapter investigated the conceptual learning that took place during the first five weeks of the two iterations of the dissertation study. A number of different data sources were

used and analyses conducted to tease apart differences related to students' understanding of concepts and the modality they used to learn the concepts. This work is in an effort to answer the research questions about the relationship between representation and conceptual learning. Throughout the chapter and this discussion, differences in outcomes are attributed to the modality students used. This is a reasonable causal leap to make given that all of the students in the study worked through the same curriculum with the same programming environment using the same language and had the same time-on-task, in the same classroom, with the same teacher. The only thing that differed across the three conditions of the study was the programming modality used by the programming environment and the participants themselves.

Modality Matters

One of the major contributions of this work is showing that when it comes to novices learning to program within introductory environments, modality matters. How and when this statement is true, as well as when the impacts of modality start to erode, are discussed throughout the four analysis chapters in this dissertation. This chapter shows how modality affects conceptual learning. The first section of this chapter provided a qualitative analysis of students' descriptions of various core programming concepts. This analysis showed differing metaphors used to talk about concepts by students from different conditions, as well as students attending to different aspects of a concept based on the modality used in the introductory environment. In some cases, these different patterns of responses could be linked to features of the modality. For instance, students in the Blocks condition were more likely to discuss `if` and `if/else` blocks as two distinct things due to their being portrayed as two separate blocks in the palette. In other cases, the differences did not seem tied to the modality itself, such as the case of students in the Text condition more frequently describing variables as Containers, for which the best

explanation we devised had to do with the teacher's practices and the higher likelihood of students asking for the teacher's help in explaining how variables are used in the Text condition.

This chapter shows the influence of modality on student learning after programming with a given modality as well as students' ability to comprehend programs with different modalities when learning with the same programming tool. These differences are present at the conclusion of the five-week introductory portion of the class in every analysis conducted, including: by overall performance on the assessment, in conceptual understanding, in performance on questions grouped by modality, and on perceived ease-of-use by concept. However, it is important to note that these differences are not robust across the entire data set. There are places where modality seems to have little or no effect, as was seen in the last chapter where, in numerous areas, little difference was observed with respect to various attitudinal dimensions. Also, in later chapters we will see these differences fade as students move away from the introductory environments and begin working in Java. This trend was visible on post scores of the Commutative Assessment presented in this chapter, but will explored in greater depth in Chapter 8. There were also places in this chapter where there appear to be differences by modality, but the nature of the data and the power provided by the sample size prevent us from making stronger statistical claims about differences.

Blocks versus Text

A second major contribution of this dissertation is isolating a programming environment's modality (graphical blocks, textual, and a hybrid blocks/text) providing the ability to link conceptual and learning outcomes and performance on standardized assessments with a modality, or even a specific feature of a modality. Using data from the second year, a number of differences on test performance emerged. The first finding is that students using a blocks-based

modality showed significantly higher learner gains after five weeks of class compared to their text-based peers (after controlling for the prior knowledge). A second finding shows that this difference in performance does not persist after moving onto a professional text-based language and environment. After 10 weeks of working in Java, students in the two conditions showed nearly identical scores. As discussed previously in the chapter, there are a number of possible interpretations of this data, including: that blocks-based interfaces allow students to learn faster, that learning gains from blocks-based interfaces do not transfer to text-based languages when the environment and underlying language change, or that the text-based modality better prepares learners to transition to other text-based environments.

Digging into this finding revealed a consistent pattern of students in the Blocks condition outperforming their Text-based peers. When looking at performance by the modality of the question being answered, the Blocks condition scored the highest for all three modalities (Pencil Code Blocks, Pencil Code Text, and *Snap!* Blocks). This was surprising as it meant the students in the Blocks condition did better on the Pencil Code Text questions than the students who had exclusively been using the Pencil Code text interface for the previous five weeks. Likewise, the Blocks condition did better on the *Snap!* questions, which used an interface neither condition had seen. There are a few possible ways to interpret these numbers. One interpretation is that there is some form of near transfer occurring from the Pencil.cc blocks interface to both another blocks interface (*Snap!*) and to a similar (or syntactically identical) text interface (Pencil Code Text). A slightly different interpretation is that the learning that happened by students in the Blocks condition is not so tightly coupled to the interface that it cannot be used across languages. The distinction between these two interpretations is whether the learning that occurred was about the modality or the underlying concept. Other data presented in this chapter suggests the latter

explanation is the more likely of the two; that the blocks interface helps learners develop understandings of the foundational concepts, as opposed to a type of meta-representational competence (diSessa et al., 1991) that applies across programming modalities and interfaces.

Just like with the questions-by-modality finding, students in the Blocks condition outperformed their Text counterparts in all six content categories on the Mid assessment. This means that the utility of learning to program in a blocks-based interface is not confined to one specific concept or another. This finding is interesting to interpret alongside the analysis from the first year of the study that showed that concepts are more easily parsed in the graphical modality relative to the text-based alternative, although not always statistically significantly so (Figure 6.8). Taken together, this suggests that the blocks-based modality does provide learning supports for novices while working in introductory contexts. That last clause ‘introductory contexts’ is important, as these differences did not persist once students moved on to learning Java. This is shown in Figure 6.14 and further explored in the next section. This means stronger claims about the power of blocks-based tools beyond the context and programming environment in which they are situated cannot be made. Also, as discussed in the last chapter, the blocks-based interface is better with respect to other important aspects of learning, such as confidence and authenticity.

The final analysis presented in this chapter, which directly compared the Blocks and Text conditions, looked at perceived ease-of-use of different constructs in the different modalities. This is the one place in this chapter that the data starts to incorporate dimensions of program generation alongside program comprehension. Here again the Blocks condition outperformed the Text condition across all four concepts, suggesting the benefits of blocks-based interfaces extend beyond comprehension. This dimension of the comparison is further explored in the next chapter.

The Case of the Hybrid Condition

One of the questions this dissertation is pursuing is the exploration of programming environments that blend features of a blocks-based interface with characteristics of text-based tools. In looking at conceptual learning by condition, the Hybrid design chosen for this study at times was more closely aligned with the Blocks condition and other times was more similar to the Text group, but more often than not, fell somewhere between the two. In the qualitative analysis that opens this chapter, the Hybrid condition was between the Blocks and Text conditions²⁰ on two-thirds (18 out of 27) of categories identified. This position of falling between the two can also be found in the quantitative learning outcomes analysis, where aggregated scores from students in the Hybrid condition were between the Blocks and Text scores for questions across all three modalities in four of the six concepts, and on three of the four concepts on the ease-of-use Likert questions. This suggests that the Hybrid condition is indeed a successful hybrid in that it produced results suggesting it shares characteristics of both of its ancestors. This shows that it is possible to blend the two interfaces together, and produce a new interface that shares characteristics of both ancestors and thus lives between the two of them.

Another interesting piece of insight, which can be gleaned from analyzing the Hybrid responses, is to compare them relative to the Blocks and Text conditions to see with which condition the Hybrid group more closely aligns. For example, in looking at the ease-of-use responses (Figure 6.17), the Hybrid condition is within a tenth of a point of the Blocks condition for Functions and Variables, with the Text condition being the outlier, and within a tenth of a point of the Text condition for Condition Logic. This tells us that for the Hybrid design used in

²⁰ This number also includes categories where the Hybrid condition had the number of responses as either the Blocks or Text group.

this study, using Conditional Logic was more similar to working with the text version of Pencil.cc, while the Functions and Variables were closer to the Blocks version of the environment. As discussed earlier in the chapter, this tells us something about how students used these constructs in their programs. Comparing Hybrid to the other two conditions is useful in other places as well, such as the qualitative coding, which shows us again that for Variables, the Hybrid condition was consistently closer to the Blocks condition's responses than the Text condition's (Figure 6.1).

There are still some questions about the Hybrid condition that can not yet be answered using the data presented in this chapter, such as how programming practices in the Hybrid condition compared to the other conditions or how programs authored in the Hybrid differ from the others. These questions will be pursued in the next section and then revisited in the final chapter of the dissertation where findings from the various analyses will be combined.

Conclusion

Understanding the relationship between modality and learning is a central goal of this dissertation and is consequential with respect to deciding what tools to use in classrooms and to inform the design of future introductory programming environments. The above analyses are important as they show us that there is a difference across the modalities and that the answer is not as simple as one modality is universally better than the other. Instead, these different tacks reveal different facets of the complex relationship between modality and understanding, which themselves are a part of a larger and yet more complex relationship between modality and a learner. The data in this chapter contribute to the emerging finding that modality is not a uniform monolithic thing, and that it affects different aspects of programming differently and like-wise, it

affects different learners in different ways, further complicating the challenge of trying to understand this basic relationship.

This chapter fills in another dimension of the larger questions being asked in this dissertation about the relationship between modality and learning to program. The previous chapter looked at attitudinal and perceptual outcomes from having novices use different modalities, whereas this chapter focused on learning and conceptual gains associated with working in the different modalities. What has yet to be discussed is the programming practices each modality engenders, the characteristics of programs authored across the three environments, and questions related to if and how these different modalities prepare learners for transitioning to professional text-based languages. These topics are the focus of the next two analysis chapters and continue to fill in the bigger picture of the relationship between modality and learning.

7. Practices and Artifacts

This chapter answers the third and final component of the first set of research questions being pursued in this dissertation: How do modalities influence learners' emerging programming practices and the artifacts they construct. The chapter begins with three vignettes (Erickson, 1986), one each from the three modalities used by students in the study. In each vignette, the student is trying to write the same program as part of a one-on-one interview with the lead researcher. The goal of this section is provide a sense of what it looks like for students to author programs in each modality. For each vignette, attention is paid to practices that were uniquely afforded by that specific modality. A brief discussion after each vignette summarizes the differences of the three. From there, the chapter shifts to look at programming practices across the full set of participants. To accomplish this, the computational data collected (code snapshots and programming events) are used to look at the characteristics of the programs written, patterns in how and when students chose to run their programs, and characteristics of how the blocks were used in the composition of programs. For each of these dimensions, comparisons are made across the three modalities. The chapter concludes with a discussion summarizing the findings from this chapter and setting the stage for the transition to the Java data that will happen in the next chapter.

Three Vignettes

As part of the Mid interviews, students from each of the three conditions were asked to write a short program in Pencil.cc. As a reminder, these interviews were conducted in the sixth week of the study, after students had completed the Pencil.cc portion of course and just started working with Java. Each student used the same modality during the interview that they had been

using for the previous five weeks. The analysis starts by focusing on the unique interaction patterns and distinct practices supported by the three modalities. As such, not every moment of the three programming sessions are present, but instead an effort was made to provide enough detail to give a sense of how the sessions progressed, with an emphasis placed on key moments and interesting interactions.

A total of 12 interviews were conducted at the midpoint of the study, four from each of the three conditions. The vignettes presented in this section were selected because the students in each proved to be the best demonstration of the various affordances of the modality. None of the vignettes presented were outliers from the other of the modality, but just serve as the best example (sometimes because of the bugs they encountered while writing their program or the number of different aspects of the modality they used while writing their program). We do not have a way to determine if the patterns observed are representative of the entire study population due to limitations in the data collection strategy used, thus cannot make claims about the typicality of the vignettes. Instead, they serve as specific demonstrations of the possible ways the different modalities are used by learners.

The vignettes follow students as they try to write a program in response to the following prompt:

Can you write me a program that picks a random number less than 15 and then prints out every multiple of that number that is less than 100? So, for example, if your program picked the number 11, it would print out 11, 22, 33, 44, 55, 66, 77, 88, and 99. If it picked the number 2, it would print out 2, 4, 6, 8, 10, 12, 14 and so on, up until 100.

It was not always clear to students what they were being asked to do from this concise description, so the prompt was often followed by questions from the student. Every student ultimately understood the prompt and either wrote a correct program, a close to correct program,

or was able to articulate an algorithm for producing a correct program. This specific programming challenge was selected for a number of reasons. First, it is relatively concise, so it can be quickly described and is small enough that most students were able to complete it in the time allotted. Second, the solution requires students to use variables and iterative logic (there are both definitive and indefinite looping solutions) while also potentially including conditional logic, so a number of concepts are encountered. A third useful characteristic of this problem is that there are a number of ways to solve it, including using `while` loop and adding the random number each iteration, using a `while` loop and multiplication along with a variable starting at one and increment each iteration, or calculating the number of iterations up front and using a `for` loop to control the looping. A final feature of this problem is that it has a number of natural pitfalls that students frequently encountered. Most solutions to this problem require two variables, one to keep track of the growing number and a second to store the random variable, trying to solve this problem with a single variable was a frequent approach taken and always resulted in incorrect results that students had to debug. It is also worth noting the decision to use a random number instead of asking the user for input was because we were less interested in students remembering syntax and more about their ability to incorporate programming constructs. Thus, we did not want learners to get stuck on getting student input, instead we wanted them to be able to jump right into the central logic of the program, which could be more quickly accomplished with the random number approach. While there are many ways to write this program, the most common approach (and the one taken by all three of the students profiled below) follows the logic shown in Figure 7.1.

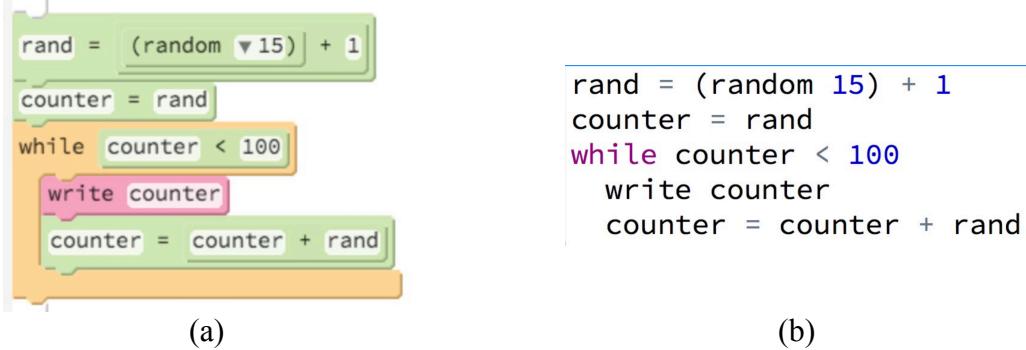


Figure 7.1. A blocks-based (a) and text-based (b) correct implementation of the prompt being worked on for these three vignettes.

There are a few things to note in the prompt and the correct solutions the students created. The first is that it asks students to use variables as well as iterative logic. All but one student used a `while` loop (one student did complete the program using a `for` loop, adding additional calculations to figure out how many times the loop would need to iterate up front). The solution also requires the use of more than one variable, a feature most students did not pick up initially, but instead had to figure it out as they worked through the program. Finally, the inclusion of the `+ 1` command in the first line of the solution is due to the fact that if the `random 15` call returns 0, the program will get stuck in an infinite loop.

Blocks Condition Vignette

The portion of the interview being used for the Blocks condition vignette lasted 10 minutes and 22 seconds. During this time, we see the student leveraging various affordances of the blocks-based modality, including browsing, drag-and-drop composition, and hover-over tips. We now carefully lead you through this episode, before concluding with a brief discussion at the end of this vignette from the Blocks condition. After asking a clarifying question, the student begins to work on his program by clicking through a number of categories in the blocks palette: Operators, Control, Sprites, Sound, Text, Art, Move, Control (where he scrolls up and down),

and finally Operators for a second time. Once returning to Operators he scrolls down and finds the `random` block, which he drags onto the canvas. He then clicks on the number input of the `random` block and changes the value from the default 6 to 15. Next, he takes a moment to think, closing his eyes, then opens them and resumes clicking through the categories: Text, Move, Control. As he scrolls up and down in the category, he says "*there's just so many possibilities, you know, different ways I could do it.*" Next he drags out the `for` block before dragging it back to the palette (so not adding it to the program). He then clicks on the Operators draw and adds the `function` block²¹ to his program. With the `function` block added, he clicks on the Control category again, scrolls down to the `while` block and hovers over it, appearing to read the hint text for the `while` block that reads "*Repeat while a condition is true.*" At this point, the interviewer intervenes, asking the student what his approach is. He responds: "*well, I want to set a variable and that variable equal to the random number. And then, put the piece of code that says 'increase by'* (long pause) *and then have that be the random number.*" He then drags out the `variable` block²², which contains the code `x = 0`, and drags the `random 15` block from his program into the right side of the assignment block. Next, he changes the left side from `x` to `rand` by typing it in, resulting in the first line of his program reading: `rand = (random 15).`

With this first statement complete, the student resumes verbalizing his algorithm. "*Ok, so like, have it increase by rand and then, at the end, say write, whatever the result of that is, repeat it, but repeat it until* (long pause) *ok, ok, ok.*" He then quickly removes the `function`

²¹ The function block does not actually say `function`, but instead is a template for defining a function that has the characters `f = (x) ->` inside a c-shaped block, with `f` and `x` being slots that can be replaced with the function name and parameter names respectively.

²² The variable block contains the text `x = 0` by default, with the `x` and `0` being slots that can be replaced by the variable name and the initialization value respectively.

block he had added and drags in the `while` block he had been looking at before, populating the left side of the `<` comparator with `x` and the right with `100`. After another long pause, the interviewer asks him what he is thinking, to which he explains a few more steps in his algorithm, culminating in his adding the statement `write rand` before his `while` loop. He then clicks on Operators and drags out the `+=` operator block, which he adds inside his `while` loop and fills in the two empty slots with `rand` and `1`, resulting in the statement `rand+=1`. After a second in thought, he changes the left side of the `+=` assignment to `x`, then quickly adds a new variable block before the `while` loop, which he uses to initialize `x`. In a flurry of quick changes, the student changes the `x += 1` command to `om += rand` and adds a `write` block inside the `while` loop to print out the variable's value and a second assignment value at the end of `while` loop. When asked what `om` meant, he said it was just a random name he came up with. By this point, the student had been working on his program for 6 minutes at 26 seconds and produced the program shown in Figure 7.2a.

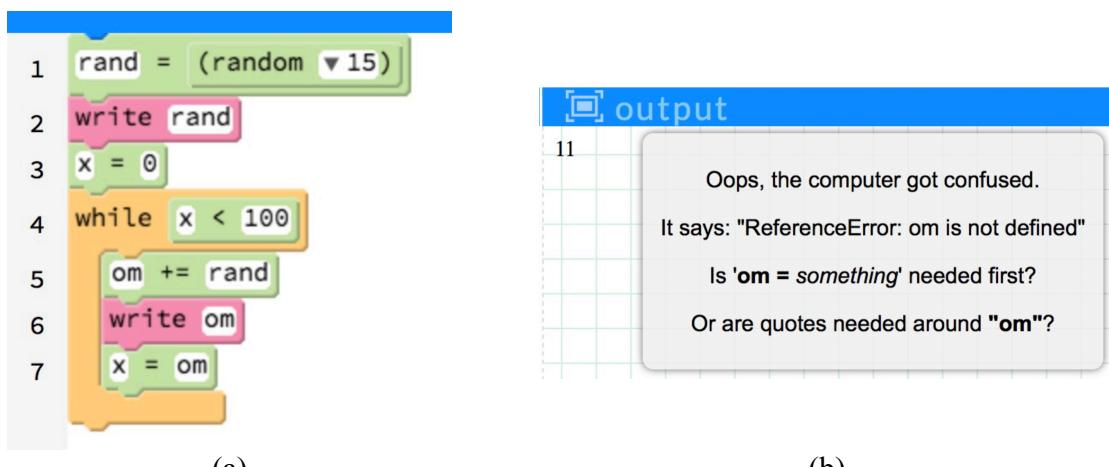


Figure 7.2. The program the student wrote in six-and-a-half minutes of work (a) and the error he saw after running it (b).

Having written this seven-line program, the student clicks the run button, and is shown the error message shown in Figure 7.2b. Upon seeing this error on the screen, the student pauses, reads the error, then says "OK" and clicks his mouse inside the slot assignment in line 5 of his program where `om` is first used. A second later, he drags a `variable` blocks into his program and puts it above the `om += rand` block, changing the two default values of the `variable` block to read `om = rand` and then clicks run button again. The updated program is shown in Figure 7.3a.

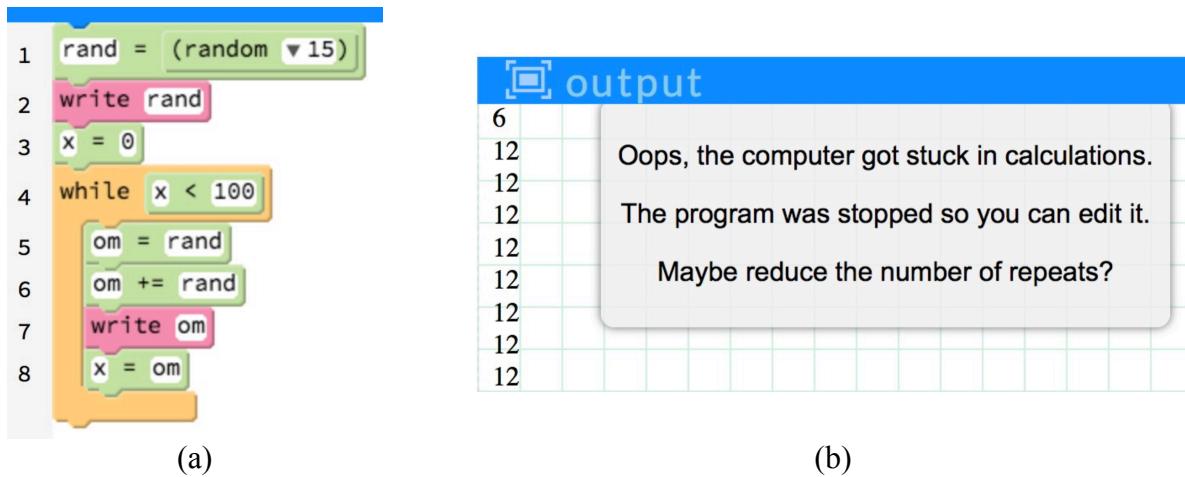


Figure 7.3. The program the student wrote in seven minutes of work (a) and the error he saw after running it (b).

This addition to his program introduces an infinite loop²³, causing the program to not print out anything for a few seconds before printing the error shown in Figure 7.3b, giving the interviewer an opportunity to interject and ask the student about the changes he just made. After a little back and forth, the student starts to explain his strategy²⁴. "*The random number* (he puts

²³ By assigning `om` to be equal to `rand` inside the `while` loop, the value of `om` never increases, thus the `while` loop never hits its stop condition.

²⁴ In the following quotes, text inside parenthesis describes what is happening on screen, words in italics are what the student says aloud. The line number reference the code showing in Figure 7.3a.

the cursor on line 2) *I want to set equal to om* (he moves the cursor first to the word `rand` in line 5, then to `om` in line 5), *and then I want it to increase om by the random number* (he moves the cursor down to line 6 and moves it from left to right as he speaks, circling `rand` with his cursor as he says *random number* aloud), *and then write it* (he moves his cursor to line 7 and again moves it from left to right, circling `om` with his cursor as he said *it*), *and then set x equal to that increased value* (he moves the cursor to the next line and moves it back and forth over the `x = om` block), *so that if this is true* (he moves the cursor up to the `x < 100` block in line 4), *I want it to repeat it again* (he moves the cursor up and down from line 4 to 8 repeatedly). *That's what I want to happen.*" During this walk through, the program stops itself and prints out a 6 followed by a string of 12s that ran off the bottom of the screen. Having explained his intentions, he then tries to figure out what happened, saying: "*well, I think it worked once* (hovering his mouse over the first 6 and twelve printed to the screen) *because it increased it, right, because the random number is 6* (moving his cursor from line 1 in his program back to the 6 on the screen) *and that increased it by six* (moving his cursor over the line 6 of his program where the addition takes place), *so it's twelve but wrote it over and over, probably because of this* (hovering his mouse over the newly added `om = rand` block). *Yeah, it's because of this* (pointing again to `om = rand`)."

The conversation continues for another minute, before the student finally says: "*I don't know, but that's my train of thought.*" After a few more minutes, the interview ends without the student resolving his infinite loop issue, so the interview helps him fix his infinite loop bug. To get the correct behavior, the interviewer only has to move the `om = rand` block from inside the `while` loop to just before it. Seeing that this small change was all that was needed to have a functioning program, the student says "*Oh, there we go*" and starts laughing about how close he was to the solution. Having completed the description of this programming session, we now

discuss the various affordances and supports provided by the blocks-based editor that were on demonstration in this vignette.

Blocks Condition Vignette Discussion

At start of the interview the student begins by clicking through different categories in the palette, sometimes scrolling through the blocks within the categories, other times quickly continuing to the next category. These actions highlight the support features of the blocks-based programming interfaces used by this student. The “browsability” of blocks-based interfaces and the logical organization of blocks were discussed earlier in this dissertation and have been documented in previous work (Weintrop & Wilensky, 2013a, 2015b, 2013b). The fact that the student cycled through nine categories before dragging out his first block speaks to how easy and fluid this aspect of blocks-based tools can become. Alternatively, this pattern of use can be interpreted as the learner is forming a dependence on the browsable category in order to write a program. If the goal of the introductory environment is to prepare learners for more professional text-based languages, this pattern is potentially problematic. Whether or not the browsability of the Blocks and Hybrid modalities produces a negative outcome when shifting to a modality that does not have the feature will be explored in Chapter 8. After adding his first block to the program, the student changes the default value in `random` from 6 to 15. This may seem trivial, but the presence of a default value serving as a template for how the block is used, is a powerful and often transparent feature of blocks-based tools. In other non-blocks-based interviews, we see students typing commands like `random [1..15]`²⁵ and other incorrect statements, which show the potential errors the templates can alleviate. We next see the student drag out a `for` block,

²⁵ It is worth mentioning that this command is valid in Pencil.cc, but produces a slightly different behavior.

pause, hover the block over the canvas, then put it back in the palette, thus not adding it to his program. This is comparable to typing in a command and then deleting it in a text editor. The speed and casualness with which this can be done in the blocks-based interface speaks to how quickly the user can compose programs and how the modality provides the ability to focus on what statements to include rather than on the act of typing, which is often non-trivial for novices.

The next affordance of programming in blocks, which is seen in this vignette, is the learner hovering his mouse over the `while` block to see a natural language description of how it can be used. A number of students mentioned this as being useful in their interviews, saying things like “[In Pencil.cc] *you don't have to type or remember code. you can just put your mouse over it and it will tell you what it does.*” As the student said this, she moved her mouse over a block to show the hint text as a demonstration. It is worth noting that there is no natural analog in a textual modality, at least not before a command has been added to the program. Instead, text modalities need supports outside of the editor itself (akin to the Quick Reference menu) to achieve a similar behavior.

After using the hint, the next noteworthy event comes when the student drags out the `variable` block and then drags his `random 15` command into the right hand slot, resulting in the statement `x = random 15`. What is interesting about this is that the chronology of creating this statement does not match the left-to-right order of its final form. In other words, the student defines the right side of the command (`random 15`) before the left (`x =`), which is a useful feature of the blocks-based modality that is less natural (although certainly possible) in text-based interfaces. This ease-of-editing was on display as the student's algorithm took shape. This can be seen in his removing the entire `function` block in a single drag-and-drop action then

adding the `while` block, again with a single drag-and-drop command. These edits are made quickly and with fewer keystrokes/clicks than would be necessary in a text-based editor.

Continuing the discussion of blocks-based features supporting the novice programmer from this vignette, we want to highlight the ease with which the student explained his program *with* the blocks. In explaining what he wanted to have happen after programming the infinite loop, he used the cursor to point to specific commands or parts of commands as he built his explanation. This is not unique to the blocks-based modality, as it is similarly possible to do this with textual programs, but the added spacing provided by blocks, the different colors, and visually depicted slots for text entry all contributed to the ease of communication. The final aspect of this vignette that should be noted is what was missing from this 10 minutes of programming: the student never struggled with syntax, keywords, or the mechanics of implementing his ideas. He paused only when trying to figure out what he wanted to do. This speaks to how the blocks-based modality can push the mechanistic and syntactic aspects of the act of programming to the background and allow the user to focus on the structure and algorithmic challenges inherent in composing programs. Having presented the blocks-based vignette and a discussion of what features of the modality were employed by the student, we now turn to the Hybrid condition.

Hybrid Condition Vignette

The vignette selected to serve as an example of what it looked like for a learner to program in the Hybrid modality lasted a total of 9 minutes and 8 seconds, so roughly the same duration as the vignette chosen to represent the Blocks condition. In this vignette, we see the student leverage some of the affordances highlighted in the previous section, but also see an increased use of the keyboard, blending conventional blocks-based and text-based authoring

patterns. The student starts by clicking through the various categories in the blocks palette, even as the interviewer is still describing the details of the program. With the goal of the program outlined, the student first drags the `random` block into the palette, then puts the cursor next to the default value of 6 and replaces it with 15. After asking a clarifying question, he then hits the return button, giving himself a blank line to work on, and types: `increase=0`. He then returns to the blocks palette, clicks through a few categories (Sound and Move) before opening the Control group and dragging a `for` block onto the canvas. The `for` block provides a template that defines the looping structure and provides a space to define what will happen inside the loop. With the `for` loop in his program, he deletes the empty command nested under the `for` statement and then moves his cursor up to change the portion of the code that determines how many times the loop will execute. After deleting the placeholder inside the `for` loop the editor displays a red x next to the loop's definition. This denotes there is an error in the code, in this case, because there is nothing inside the `for` loop. The student hovers his mouse over the error, sees the message 'UNEXPECTED TERMINATOR,' pauses for a second, then clicks on the Text category of the palette and drags in a `write` block, which causes the red x to disappear.

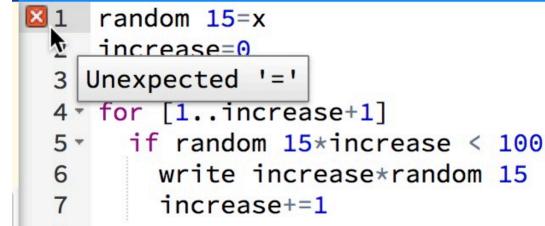
The `write` block has the default argument of '`hello.`'. After dragging the block in, the student deletes these characters and replaces them with `increase*` and then highlights the first line of the program (`random 15`), copies it and then pastes it after the `*`, giving him the line: `write increase*random 15`. The student next adds a conditional statement to his program by typing the line `if random 15*increase < 100`. These characters come from memory, as he does not use the block palette or Quick Reference for help. He then makes a few more modifications to his program using the keyboard, including changing the code that controls the loop and adding a line to increment his program then stops typing and looks at his code. At this

point, he has been programming for two minutes and 48 seconds, and has written the program shown in Figure 7.4a.

```

1 random 15
2 increase=0
3
4 for [1..increase+1]
5 if random 15*increase < 100
6 write increase*random 15
7 increase+=1
  
```

(a)



A screenshot of a code editor window. The code in the editor is:

```

1 random 15=x
2 increase=0
3 Unexpected '='
4 for [1..increase+1]
5 if random 15*increase < 100
6 write increase*random 15
7 increase+=1
  
```

The third line, "3 Unexpected '='", is highlighted with a red box. A red 'x' icon is positioned next to the first line, "1 random 15=x".

(b)

Figure 7.4. The Hybrid student's first run program (a) and the error message displayed in the editor (b).

As the student reflects on his program, the interviewer asks him to explain it. He pauses a second, then starts his explanation by saying "*Alright, so, so this part of the code selects a random number* (he clicks and drags his cursor over the first line of his program, highlighting line one). *Increase represents when the number is multiplied, so it could be 0, 1, 2, 3, etcetera.* *This is going to* (points the cursor at the word `for`, then pauses), *increase + 1 is going to be one* (highlights the `increase+1` portion of line 4), *so it repeats one time. Then we're going to do the random number again* (moving his cursor over `random 15` in line 5, but not highlighting it) *and if that times whatever I set increase to is 0, then it should write the product of what I'm multiplying it by and the random number* (again pointing to the various parts of line 6 as he mentions them, but not highlighting them)". With his program explained, the interviewer asks him if he wants to run the program or if there is anything else he would like to add. He pauses and says "*I'm going to set this to a variable because I'm not sure if this random 15* (clicks and drags to highlight the `random 15` in line 5) *is the same as this random 15* (hovers his mouse of line 1). *I'm just going to set it to x.*" He then adds the characters `=x` to the end of line one, so the line reads: `random 15=x`, which causes the editor to display a red x next to the first line.

The student sees this and says: "*wait, what's this?*" and hovers over the x, which reveals the message Unexpected '='. Figure 7.4b shows the red x and the error message the learner sees at this point. The student reads this message, pauses for a second, then adds a second = to the line so the first line of the program reads random 15==x²⁶. This addition to the program resolves the compile-time syntax error and the red x disappears.

Upon running the program, he gets an error saying 'x is not defined'. He pauses, thinks for a minute, then deletes the ==x at the end of line and then adds x= in front of the random 15 command. This fixes the error, and he hits run again. His program then prints out the character 0 and then stops. This leads to the student quietly reading through his program. The interviewer then starts to ask questions about the program getting the student to verbalize his intentions and how he thinks the program is running. The details of the discussion focus more on the specific assignment than the modality, so are not included here. It is worth noting that, throughout this discussion, the student frequently highlights words and lines as he thinks through the program, as was shown above. As part of his trying to understand what is happening in his program, he changes the +1 in line 4 to +2 and runs the program. It now prints out 0 on the first line and then 2 on the second line, which gives him some insight into the behavior. He next says "*I'd also like to see what x is*" and then puts the cursor at the end of line one and types in write x. He then asks "*that is it right?*" and drags a write block into the program, compares the syntax of the newly added command with the command he just typed and says "*yeah*" before deleting the write command he just drag-and-dropped into his program. This strategy of using blocks to check syntax ended up being an unexpected but common practice that will be discussed below.

²⁶ In CoffeeScript, like many other programming languages, the double equals sign (==) is used to compare the equality of two objects whereas as a single equals sign (=) is used for assignment. The double equals sign is not what the student wants in this scenario.

When the student next runs the program with his debug statement in place the output is: 2 0 10, which causes the student to say "*that's strange. Oh.*" and changes the `random 15` command that is still in line 6 to use the variable `x`. He now re-runs the program which produces: 6 0 6, each printed on a separate line, which, based on the student's facial expression, was what he was expecting to see.

A moment after running this program the student has a revelation. "*Oh, I know, I know what to do*" and then smiles. The interviewer inquires and he continues: "*I think a while loop would be much better.*" He then highlights the `for` loop line, deletes it all at once, types in `while`, and then copies and pastes the condition out of the `if` statement, producing the program showing in Figure 7.5. "*Yeah, this ought to work better.*" Upon running the program, the program demonstrates the correct behavior. After a short concluding discussion, the student is thanked and the interview ends.

```

1 x=random 15
2 increase=0
3
4 while x*increase < 100
5     write increase*x
6     increase+=1
7

```

Figure 7.5. The students final, correct program

Hybrid Vignette Discussion

This student was chosen for the demonstration of what it looks like to program using a hybrid blocks/text interface because he took advantage of various features of the dual modality during the interview. There are also episodes in this interview where his approach and pattern of

interaction are different than those taken by learners in the fully blocks condition. The first interesting thing to note from this vignette is how the student moved back and forth between dragging in commands from the palette and typing them directly into his program. This could be seen right from the beginning in that the first command (`random 6`) in the program was added via dragging the block onto the editor and the second command (`increase=0`) was typed. For the third command (a `for` block) he goes back to the palette to again drag-and-drop a command into his program, which he then modifies with the keyboard (deleting the placeholder for the nested statement). This shows the hybrid interface is successful in supporting two modes of composition, allowing students to type commands if they know the syntax and are more comfortable on the keyboard, while also having the browsability and ease of composition that come with the drag-and-drop-supported blocks-based modality. Another thing to note about dual modality composition is the support it provides via scaffolds that can be developmentally appropriate for learners at different levels and that allow the learner to be in control of their own learning. For example, this student was able to type out an `if` statement from memory, but used the drag-and-drop feature to add a `for` loop, suggesting he had better knowledge of the syntax of one of these concepts compared to the other. In the vignette, we also see the learner clicking through categories in search of commands, showing that the browsability of the Blocks modality is still present.

Another noteworthy feature in this vignette is the presence of in-editor feedback for compile-time errors²⁷ and how the student responds. Twice during this portion of the interview

²⁷ Compile-time errors (as opposed to run-time errors) are syntactic errors in a program that the environment is able to detect before the program is run. Common examples of compile-time errors include incorrect keywords (like typing `whiel` instead of `while` or forgetting necessary punctuation like semicolons, brackets, or curly braces).

the student introduced a compile time error; in both cases it was while he was manually typing in commands. One of these errors happened when the student typed in the line: `random 15=x`, which caused an error. What was interesting about this is that the student's next move was to add a second `=`, producing the line `random 15==x`, which is still incorrect, but now is a semantic bug as opposed to a syntactic error. This is noteworthy as this type of error would be very difficult to make in the blocks interface where more constraints are placed on how commands can be assembled. By this we mean blocks modalities can prevent the user from adding a statement like `random 15==x` to the program based on the shape of the block. For example, in Scratch, the equality comparison block has an oval shape, so it cannot be added to a script, while in the Pencil.cc blocks interface, there is not standalone `==` block to be dragged into a program. While it is possible to have compile time errors in the blocks interface, this never happened in any of the interviews conducted with students working in that modality.

The introduction of compile-time errors was frequently observed in both the Text and Hybrid modalities, but in the Hybrid case, there are additional supports provided by the interface that can help address this. One clear example occurred later in the interview. When the student wanted to add a debugging statement to his program, he typed in the line `write x` and then dragged out a `write` block and placed it below the line he just typed to check the syntax. Upon seeing that what he had typed matched what appeared when he dropped the write block, he deleted the second command and continued working. This pattern of using the blocks as a way to check syntax ended up being frequently employed throughout the course. In this capacity, the blocks were not serving as a means for remembering what is possible, or a way to author new portions of a program, but instead serving as a supplemental verification for students to double-check to make sure they were doing things correctly. In the Text condition, to verify syntax,

students have to go to the Quick Reference page, which is a slower, more cumbersome process than dragging a block into the canvas in the Hybrid interface. This pattern was unexpected and tells us something about types of supports novice programmers need and want: in-editor scaffolds to quickly verify syntax.

Another thing to note from this interview is that the student utilized a number of common text-editing techniques: notably copy-and-pasting lines of code to move them around and highlighting blocks of text either to denote something to the interviewer or to delete portions of his program. When highlighting portions of the code during his explanations, he sometimes highlighted whole lines and other times just portions of a larger command. Seeing the student make these types of moves is not particularly surprising as high school students are usually comfortable with text manipulation. This is noteworthy in that the Blocks condition does give the student the ability to do this type of character-by-character highlighting, showing another small difference between the Hybrid (and Text) modalities and the Blocks interface. It shows that the text editing practices were present in the Hybrid condition, which also included capabilities not present in conventional text editors.

Text Condition Vignette

The text vignette lasts twelve minutes and six second and, like the others, begins with the interviewer explaining the programming challenge. In the previous vignette we saw the learner start to use the keyboard as a form of input, in this vignette we see how programming differs when the learner only has the keyboard input and lacks other features of the blocks-based modality (like the browsable categories and visual syntactic information). When the interviewer mentions the program picking a random number less than 15, the student immediately types `random [1–15]` in the first line of the editor, saying: "*I'm pretty sure that's how that works.*"

The interviewer then continues with the rest of the program details. After hearing the description, the student then tries to store the random number into a variable, typing: , defer a after the] on line one of his program²⁸. The syntax he has typed in is not correct (a fact made clear by a red x appearing on the left side of the editor like shown in Figure 7.4a). Seeing this error, the student moves his mouse to the top right of the screen, opens the Quick Reference menu and says he is going to look up defer. As he is doing this, he comments how he cannot remember exactly how to use this command. In looking through the Quick Reference, he sees the menu item for the random command and opens that up. He reads through its contents, then closes it and says "*Would I have to do await? I have to do await.*"²⁹ He types in the command await at the beginning of the line, which resolves the error, although it will not produce the behavior he is expecting.

With his variable command in place and no compile-time errors, he asks the interviewer for clarification on the programming task, then types a*2 on the second line of the program, saying "*That's multiplication right?*" He pauses for a second, then says: "*I'm going to check that*" then adds write in front of a*2 and hits the run button. The program gives him a runtime error that includes the message: "You might not need a comma here." Seeing this message, he removes the comma and hits run again, which confusingly gives him another runtime error with the message "Is there a missing comma?" This causes him to make some more minor modifications to the syntax in the first line of his program and then open up the Quick Reference again to look at the entry on random. As he is doing this he says: "*another thing I could say I*

²⁸ The defer command is a custom commands to Pencil.cc and is used to pass control to a paused process. It is often used with the await command to read in input from the user.

²⁹ The await command pauses execution until an asynchronous process completes. It is often used with the defer command to read in input from the user.

don't like about [pencil.cc] is that sometimes it's so vague on what the problem is." After some more tinkering with syntax and getting more errors, the interviewer steers the student towards the variables entry in the Quick Reference menu. This shows him the correct syntax for setting a variable, which he types in. With his variable in place, he is ready to move on with the logic in his program. He quickly says "*I can use an if statement*" and then says aloud "*if a is less than 100 a plus a then write a*" while typing out the statement: `if a is < 100 a = a + a` `write a`. He then runs his program, getting another runtime error, this one saying "unexpected <." To fix this error, he deletes the `is` from his program, not using any help from the editor or interviewer. Figure 7.6a shows the student's program up to this point in the interview.

```

1 a = random 1, 15
2 if a<100
3   a = a+a
4   write a

```

(a)

```

1 b = 0
2 a = random 1, 15
3 while b<100
4   b = b+a
5   write b

```

(b)

Figure 7.6. The student's text-based program at the middle of the interview (a) and the end (b).

With all the syntax errors resolved, he runs the program and gets the output 8. He runs it again and sees a 14. Upon seeing these two numbers output individually, he says, "*I have to do while loop*" and deletes the `if` and types in `while`, leaving the rest of the program intact. Running the program again, he now sees: 14 28 56 112 each printed on their own line. The interviewer then asks about the reason for the shift from `if` to `while`, which prompts an explanation of the `if` and `while` statements, that ended with him saying: "*while is like, while this is true, I will keep on doing this continuously and that's what it did here.*" He runs the program a few more times, and then sees the first line of output for one of his runs is 22, so he

quickly adds `write a` on a new line before the `while` loop saying “*now we'll see the multiple of the numbers we're doing.*”

He continues by running the program again for which the computer picks the number 2 and prints out 2 4 8 16 32 64 128. The student looks at it and says: “*wait, that's not right, why is it like that?*” Looking at his program he continues, “*a plus a, no I can't do that, I have to make this a different variable.*” He then replaces `a = a + a` with `b = a + a` and changes `write a` to `write b` and runs the program again. This causes an infinite loop.³⁰ After a minute, the program prints out a 9 followed by a string of 18s that run off the bottom of the screen. The student then reads through his program saying “*if a is less than 100 then do b equals a plus a*” at this point, he puts his cursor after that line, hits the return and types `a = b` while saying “*a equals b*” aloud. After a pause, he says “*I don't want to do that*” and deletes the commands he just typed in one character at a time. He then quietly says: “*after it writes b, a equals b*” typing `a = b` on a new line at the end of his program. He runs the program again. This time the program completes, but gives the same behavior he had before adding the variable `b`. The interviewer asks the student what is happening, which prompts him to go on a lengthy explanation of how the number is doubling every time instead of incrementing by the random number. After a minute of talking through his program he says “*Oh, I see what I can do here b equals zero*” and types `b=0` on a new line at the top of his program “*and then b equals b + a*” changing the first line inside his `while` loop to be `b = b + a`, explaining “*what this will do here is b is zero and then I keep adding a, so I keep adding that number and I'll get, there we go!*” finishing this statement as he watches his program run and produce the desired output. The final version of his program can be seen in

³⁰ The infinite loop occurs because he has changed his program so that `b` is the variable that increases each iteration but the `while` condition is checking to see if `a < 100`, thus, `a` never changes and the loop never ends.

Figure 7.6b. After a short discussion on what he expects in the coming weeks learning Java, the interview ends.

Text Vignette Discussion

Having presented short vignettes from a Blocks interview and Hybrid interview, this vignette shows a typical interaction of a student working in the Text condition. There are a few things of interest in this vignette, especially when compared with the previous two. The first thing that stands out is the number of errors the student encountered. This includes both compile-time errors due to incorrect syntax as well as runtime errors stemming from improper use of commands. Early in the interview, the student spends almost a minute trying various combinations of commas and keywords in an attempt to set a variable using the `await` and `defer` primitives. Things like this rarely happened in either the Blocks or Hybrid condition. In the case of the Blocks interface, the lack of typing in commands alleviates the syntax burden, while the ability to hover over a block to see what it does helps navigate students towards the correct commands. The Hybrid interface also has the hover-over blocks feature that can help with syntax by allowing students to drag blocks onto the canvas to check syntax, as was demonstrated by the student in the Hybrid vignette. In the Text condition, the student has to rely on the Quick Reference, which resides outside of the editor space itself. While using the Quick Reference is often helpful for students, in this vignette, we see an issue with this approach. Early in the vignette, when the student is trying to set a variable, he goes to the Quick Reference in hopes of finding information about the `defer` keyword, which is not what he actually wants. Even with the logical organization of topics in the Quick Reference, the student still needs to know what to look for.

Another interesting difference in this vignette from the previous two that relates to the aspects mentioned in the previous paragraph was the student's reliance on the compiler as a form of support. At various points during the interview, the student got an error from the compiler (either runtime or compile time). In response to this, the student made a series of small changes to the statement where the error resided, thus using the compiler error as a way to tell if he had figured it out or not. This can sometimes work, but in the case of this student, resulted in a syntactically valid statement that did not do what he had expected (like the statements `await random[1-15], defer a`, which is valid but does not do what the student had intended).

Relying on the compiler for syntactic guidance is often employed by veteran programmers who have not used a language or a keyword in a long time and are struggling to remember details of the command, however, for a novice to engage in the practice is very different. An extreme version of this practice was observed in another interview with a student in the Text condition.

After adding the `random 15` command at the start of her program, this student then typed `mult 100`. When asked what that meant, she said: "*I was trying to print the multiple all the way up to one hundred, because it worked for random fifteen.*" In other words, she was completely guessing that there may exist a command for multiple and only realized it was not a command when the interface told her. Most often this type of guess-and-check for novices is not particularly fruitful and even when the student succeeds in accomplishing what he or she set out to do, will have little understanding as to why the program worked.

A last comment to make about this vignette in comparison to the other two is to point out how little the student used the mouse. The interface requires that everything be typed in character-by-character, but it is still possible to highlight words or copy-and-paste commands that have already been typed in. Throughout the interview, the student rarely used the mouse

cursor, choosing instead to delete words (and lines of words) one character at a time, and unlike the Hybrid vignette, did not use the mouse cursor to reference code during the interview. Instead, he relied on reading aloud without a visual cue to communicate where his attention was within the program. While this is not a big deal in the context of the interview, it is possible to imagine scenarios where this lack of the use of the mouse is detrimental. Having provided rich descriptions of students working in each of the three modalities, this chapter now transitions to looking across the full set of students, using these vignettes to guide the investigative approach taken in the remained of this chapter.

Programming Practices Across Conditions

Having provided a rich description of what it looks like for novices to program in the three modalities, the remainder of this chapter will look at the full set of participants to reveal larger, more systematic trends across the three conditions. The data for this section were collecting by the logging system built into Pencil.cc. Information about the type of data collected can be found in Chapter 3 and is summarized in Table 3.6. This section begins with high-level descriptive data on programming practices and program characteristics grouped by condition and by assignment. A total of 145,207 Pencil.cc events were collected from the students across the three conditions. Table 7.1 summarized the average number of times each type of event occurred for each student, grouped by condition³¹. As a reminder, the three events that start with the prefix ‘block’ capture blocks-based composition events, while the other five denote larger, program-wide events (like running, loading, and saving a program).

³¹ Throughout this section, numbers are reported per student to control for the fact that the three conditions did not have the same number of students. Only students who ran their program at least 10 times were included in these calculations to not skew the average by including students who did not complete the assignment due to absences.

Table 7.1. The average number of each event types per student that was collected during the five-week introductory portion of the study, grouped by condition.

Event Type	Blocks	Hybrid	Text	Total	F or t Statistic
run	733.34	1073.62	742.97	846.41	$F(2, 89) = 8.71; p < .001$
load	76.76	110.17	69.13	84.81	$F(2, 89) = 7.18; p = .001$
save	85.93	91.45	68.13	81.38	$F(2, 89) = 1.59; p < .21$
new	24.41	27.28	23.88	25.14	$F(2, 89) = 1.44; p < .24$
logout	9.24	16.69	9.69	11.80	$F(2, 80) = 1.25; p = .29$
block-drop-addition	864.86	118.21	NA	491.53	$t(33) = 11.67, p < .001$
block-drop-floating	114.38	NA	NA	114.38	NA
block-drop-deletion	288.21	NA	NA	288.21	NA

This table shows that there was a significant difference in how often students ran and loaded their programs. A Tukey HSD post hoc analysis shows that students in the Hybrid condition ran their programs much more often than the other conditions (compared to Blocks $p < .001$, compared to Text $p = .003$), while there was no difference in number of runs between Blocks and Text students ($p = .86$). Similarly, the students in the Hybrid condition also loaded their programs more often than students in the Text condition ($p = .002$) and Blocks condition ($p = .005$) with no significant difference existing between the Blocks and Text students ($p = .97$). Students in the hybrid condition also saved, logged out, and created new programs more often than their Blocks and Text peers, although not at a statistically significant level. Looking at block-drop-addition, the one block-level event type for which we have data from multiple conditions, we see the Blocks condition adding commands to their programs using drag-and-drop at a much higher rate relative to students in the Hybrid condition. These high-level trends will be explored in much greater detail throughout the remainder of the chapter.

Running Programs

The first detailed analysis looks at the rate at which students ran their programs, which gives some insight into what it looked like to program in each of the three modalities. The five-week curriculum used for the first phase of the study included 13 individual assignments. Table 7.2 shows the breakdown of the average number of run events recorded by assignment per student. Given that each condition had the same amount of time to complete each assignment, looking at the total number runs per assignment serves as a proxy for understanding how quickly students were able to write and edit their programs³². An ANOVA calculation was run for each row to see if the number of runs differed at a statistically significant level for each assignment. The assignments are organized chronologically going from the top to the bottom.

Table 7.2. Run events collected for each assignment broken down by condition.

#	Assignment	Blocks	Hybrid	Text	Total	F-Statistic
1	Quilt	90.50	85.86	94.97	90.64	$F(2, 86) = .29; p = .75$
2	Madlibs	55.43	104.89	52.19	70.51	$F(2, 86) = 7.13; p = .001$
3	Tip Calculator	25.46	28.20	31.75	28.87	$F(2, 76) = 2.67; p = .08$
4	Paint by Quadrant	52.41	100.45	69.55	74.14	$F(2, 87) = 6.53; p = .002$
5	Movie Recommendation Engine	45.77	72.81	52.07	57.27	$F(2, 84) = 3.92; p = .02$
6	Grade Ranger	25.53	41.88	52.76	40.28	$F(2, 72) = 3.27; p = .04$
7	Guessing Game	67.00	107.70	73.97	82.21	$F(2, 85) = 2.33; p = .10$
8	Radial Art	37.50	70.67	52.92	53.91	$F(2, 81) = 4.69; p = .01$
9	Spiral	44.50	74.93	48.35	56.63	$F(2, 85) = 8.79; p < .001$
10	Polygoner	46.44	49.04	42.05	45.97	$F(2, 78) = 1.62; p = .20$
11	Connect 4	93.03	104.29	83.10	93.23	$F(2, 87) = 1.63; p = .20$
12	Brick Wall	56.04	86.19	56.50	66.63	$F(2, 82) = 7.85; p < .001$
13	Final Project	158.17	212.41	145.78	171.24	$F(2, 87) = 1.66; p = .20$

Below, Figure 7.7 shows these same data as a line chart, giving a sense of how the numbers fluctuated over the course of the five-week curriculum. The figure also shows the

³² There are other, more nuanced ways to interpret the average number of runs per student numbers that will be explored later in this section.

concept that was the focus of each lesson and denotes the assignments where statistical significance was found.

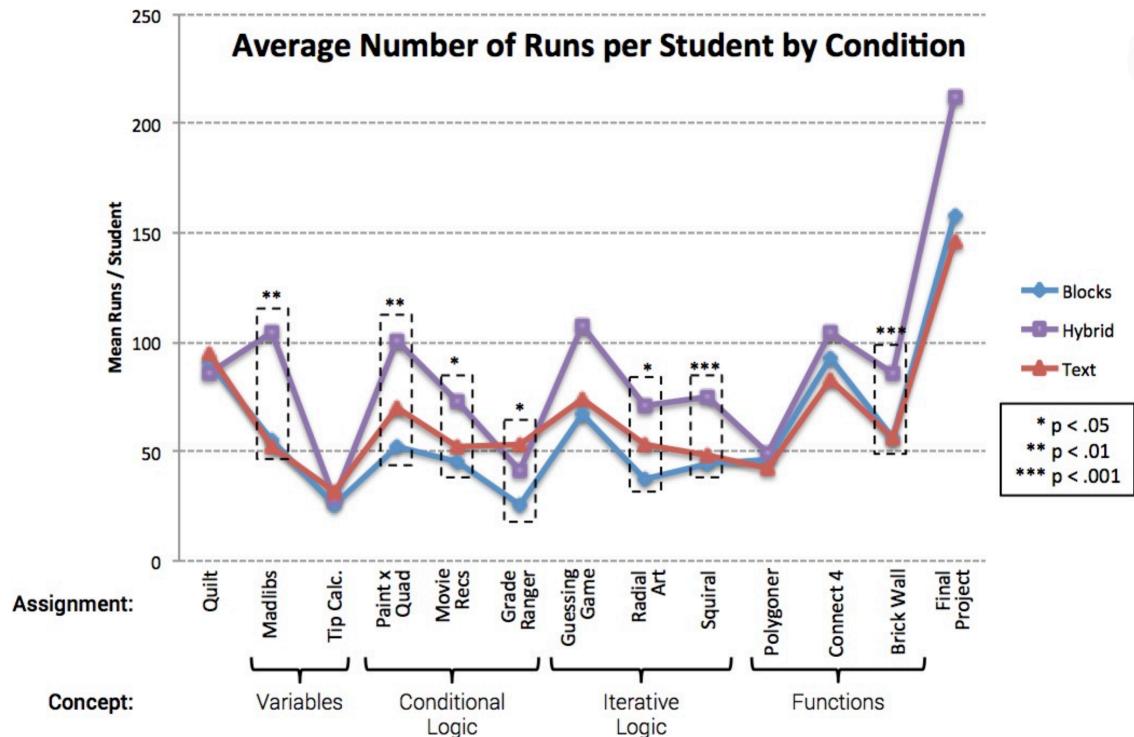


Figure 7.7. The average number of runs by students for each project broken down by condition.

That first thing that stands out in Figure 7.7 is that throughout the five-week curriculum students in the Hybrid condition consistently ran their programs more often than either of the other conditions. This was true for ten of the thirteen assignments, including the last seven assignments. This pattern is in contrast to much of what was found in the previous two chapters, which looked at attitudes and learning outcomes where the Hybrid condition's results usually fell between the Blocks and Text conditions. A possible explanation for this is that the blocks interface has the ease-of-composition of the drag-and-drop modality, which makes it easy to

quickly add commands to the program to see if they work. At the same time, it also allows for syntax errors, due to the lack of constraints on how and where commands can be added. Together, these characteristics (ease of adding commands and error-prone manual entry) can lead to users running their programs. Runs motivated by either of these two practices are in addition to running programs to see what happens (which is the main reason why students in the Blocks condition would run their program). In this case, the blended Hybrid interface results in a summative behavior (i.e. students do both) as opposed to reductive outcome (i.e. the Hybrid interface relieves the user from having to do certain things). This is just one possible explanation. Unfortunately, the data collection strategy for this dissertation did not include compiler error messages or keystroke level changes so it is difficult to validate this hypothesis in the current study.

Looking at the ANOVA values from Table 7.2 and the stratification of the three conditions on different assignments, we start to see how concepts influence the frequency of students running their programs. For example, all three of the assignments that focused on conditional logic were found to have statistically significant differences in average number of runs by condition. Likewise, two of the three iterative logic assignments were significant, with the third approaching statistical significance. Only one of the three functions assignments and one of the two variables assignments show significant differences in how often students ran their programs. Put concretely, these data show that, based on modality, students in the different conditions ran their programs with significantly different frequencies for assignments focusing on conditional logic and most assignments looking at iterative logic. For all six of these assignments, the Blocks condition ran their programs least often. Our explanation for this finding is that the Blocks modality's prevention of syntax errors removes the need for students to run

their program to see if the syntax is correct. Similarly, students are less likely to find themselves making small changes to their program and re-running it in quick succession to see if the change fixes an issue. Both of these outcomes contribute to fewer overall runs. To support this hypothesis, we now turn to the timestamps to look at patterns in the time between consecutive runs to see if it tells the same story.

Elapsed Time Between Consecutive Runs of Programs

A second way to compare programming practices by condition is to look at the amount of time that elapses between consecutive runs by condition. The goal of this analysis is to understand how quickly students re-run their programs, which gives insight into another dimension of their forming programming practice. Namely, do students develop incrementally and systematically or are there big bursts of runs followed by extended stretches of no runs. Figure 7.8 shows the average amount of time that passes between consecutive runs for each assignment grouped by condition. This data only considers runs that happened on the same assignment and within 15 minutes of each other³³.

³³ Fifteen minutes was selected as an arbitrary cutoff for the longest amount of time that might elapse between consecutive runs. The cutoff was added to control for instances where the data show hours passing between consecutive runs.

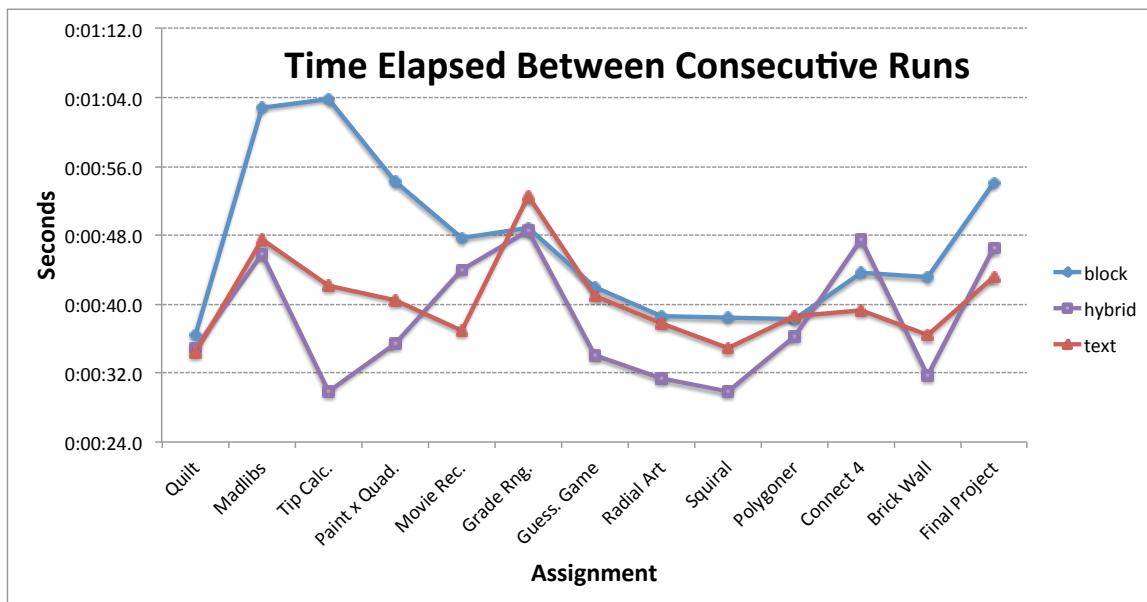


Figure 7.8. The average amount of time that elapsed between consecutive runs.

This graph shows that at the start of the year, the Blocks condition took the longest amount of time between runs. As the five-week curriculum progressed, the Blocks time grew closer to the other two conditions, while the overall time between runs mostly declined. This decreasing pattern continued until the last few assignments, when the time between runs grew, which also correlates with the complexity and difficulty of the assignments³⁴. The fact that the Blocks condition was slower on average at the outset is interesting and unexpected given a feature of blocks is their ease of composition, especially for novices early in their programming careers. A possible explanation for this is that, since there are no syntax errors, students do not develop the practice of using the compiler as a mechanism to check if a given command will work. Thus, there are fewer episodes of quick, consecutive runs where the learner is trying to find the right syntax. This explanation fits with the analysis presented in the previous section and

³⁴ The Connect 4, Brick Wall, and Final Project were more difficult than the assignments that preceded them as they included more of the concepts taught over the course of the five weeks relative to the previous assignments.

is backed up by the data, which shows there to be fewer runs in quick succession in the Blocks case than either the Text or Hybrid. Table 7.3 shows the average number of runs that happened less than five seconds after the previous run for each condition by assignment per student.

Table 7.3. The average number of runs per student that happened within five seconds of the previous run, grouped by condition and assignment.

	Quilt	Madlibs	Tip Calc.	Paint by Quadrant	Movie Recs	Grade Ranger	Guessing Game	Radial Art	Spiral	Polygoner	Connect 4	Brick Wall	Final Project
Blocks	6.4	5.5	3.5	5.4	8.1	5.8	18.5	2.1	3.3	4.0	5.4	5.0	15.3
Hybrid	7.9	29.2	6.7	15.2	15.7	6.7	31.6	8.4	7.7	12.1	11.1	11.5	19.7
Text	10.8	12.7	5.9	19.7	11.2	9.9	19.7	6.6	8.3	5.7	11.5	7.8	28.8

What stands out in Table 7.3 is that, while the condition that had highest number of quick succession runs rotates between Hybrid and Text, for every assignment, the Blocks condition had fewer runs that happened within 5 seconds of the previous run. This data confirms the trend shown above, that the Blocks condition took longer between runs, in part due to having fewer quick-succession runs. The primary explanation is that the Blocks modality, and it's prevention of syntax errors saves students from having to make quick, minor tweaks to fix syntax errors in their programs. Another explanation for the trend of Blocks being slower is due to a critique of the blocks-based approach to programming brought up by learners early in the study, namely that programming with blocks is slower than authoring in a text-based modality. The slowness of dragging-and-dropping commands and often having to assemble a number of blocks to define a single instruction could also explain the on-average longer delays between runs (this idea will be revisited with new data later in the chapter). This aspect of authorship, paired with the lower frequency of quick-successions runs, can in part explain the slower authoring patterns shown in

Figure 7.8. Having looked at the overall characteristics of the programs and patterns linked to students running their programs, the analysis now shifts to look at composition patterns that happened between runs.

Characteristics of Programs

Another dimension to investigate differences across conditions is looking at characteristics of the students' final versions for each of their assignments. Figure 7.9 shows the average size of each program completed by students in the first five weeks of the course³⁵. The length measurement used in chart is the number of characters in the final project. While there are more sophisticated ways to calculate this measure, given that all students were given the same assignment and had relatively constrained instructions, total number of characters serves as a useful proxy for more complex measures of length³⁶.

³⁵ The Final project is left off because the size dwarfed the other assignments and also due to the fact that the size of the program was greatly influenced by the type of final project students chose to do. For example, students who authored text-driven story programs had projects that were much larger than other more syntactically complex projects.

³⁶ For the blocks condition, length is calculated based on the characters within the blocks. So a blocks-based an text-based program made comprised of the same commands will have an identical length.

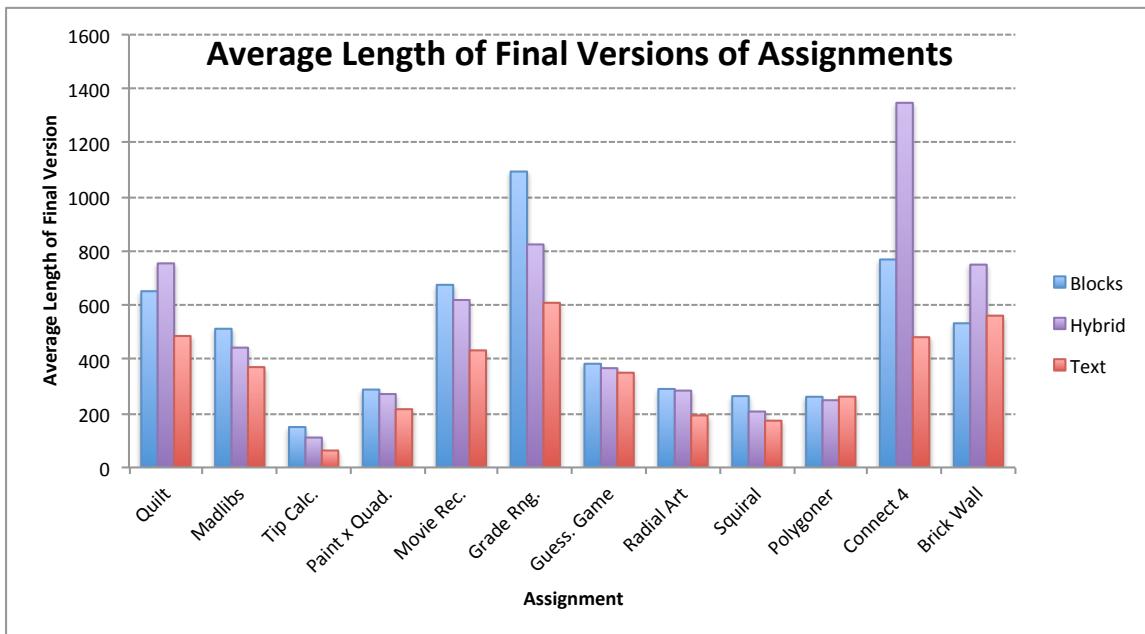


Figure 7.9. The average size of student authored programs by condition.

Although students were completing the same assignment regardless of the condition, differences in the length of the assignments do emerge. On ten of the twelve assignments, the Text condition produced the shortest solutions (on average), with Blocks students writing the longest average programs on eight of the assignments and Hybrid being the longest for four assignments. Running an ANOVA calculation for each of the assignments, four were found to have statistically significant differences at the $p < .05$ level: Tip Calculator ($F(2, 82) = 4.78, p = .01$) , Grade Ranger ($F(2, 71) = 5.26, p = .01$), Radial Art ($F = (2, 83) = 3.51, p = .03$) and Connect 4 ($F(2, 87) = 2.90, p = .05$). The assignments with the greatest stratification focused on conditional logic (Paint by Quad, Movie Recommendation Engine, and Grand Ranger) and the last two assignments from the functions portion of the course (Connect 4 and Brick Wall). The variance in the conditional logic assignments is similar to what was seen in the runs-by-assignment analysis (Figure 7.7), but that pattern does not continue with the iterative logic assignments or the functions assignments. This variation in the Connect 4 and Brick Wall

assignments may come from the fact that those two assignments were by far the most difficult in that they asked students to incorporate logic from previous parts of the course and required the most amount of code to accomplish relative to the other assignments³⁷. The fact that we see a difference in conditional logic is another piece of evidence towards the larger trend of modality affecting students' learning and using those constructs, which was identified in the previous chapter as well as in work by others outside of this study (C. M. Lewis, 2010).

Blocks-based Usage in the Hybrid Condition

Since, in both the Hybrid and the Blocks conditions, students had the ability to add new commands to their programs through the use of dragging-and-dropping blocks from the palette, comparing patterns of adding blocks provides insight into how modality affected programming practices. It also provides some insights into how the two modalities differed. Figure 7.10 shows the average number of blocks added to a program per run for students in the Blocks and Hybrid conditions, broken down by assignment.

³⁷ The Grade Ranger and Movie Recommendation Engine assignments' numbers are inflated due to the amount of text included in the assignment.

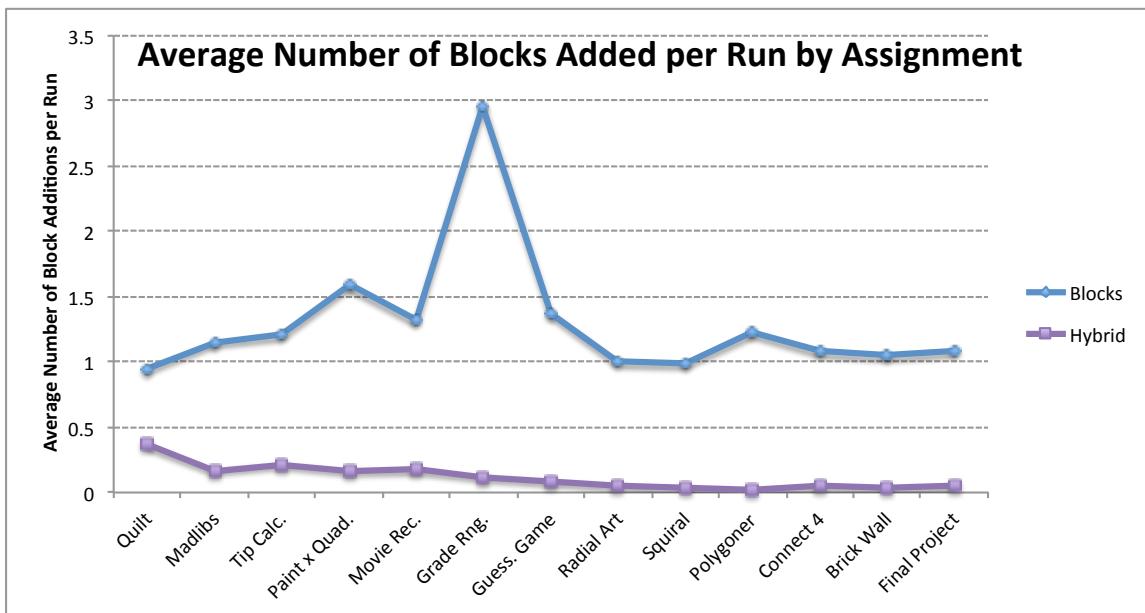


Figure 7.10. The Average number of blocks added to a program for each assignment.

There are a few things that stand out from this chart. The first is the gap between the Blocks condition and the Hybrid condition. For every assignment, students in the Blocks interface added more than twice as many blocks for each run of their program, meaning they added blocks to the program at more than twice the rate of students who worked in the Hybrid interface. There are a number of possible explanations for this. First, as was shown above, the students in the Hybrid condition ran their programs more often than those in the Blocks condition, which would result in a smaller number of block additions per run. This explanation only tells part of the story as the absolute number of blocks additions was also much higher in the Blocks condition. A second, more compelling explanation comes from the fact that the Hybrid condition allows both the drag-and-drop addition of commands as well as direct keyboard input of the textual modality, as was seen in the Hybrid vignette presented earlier in this chapter. This explanation says that students in the Hybrid condition added fewer blocks in writing their programs because they supplemented adding blocks by directly typing in

commands. This explanation also explains the decreasing slope of the Hybrid condition's plot over the course of the five weeks. As student's familiarity and experience grew, students in the Hybrid condition were more likely to directly edit the program than to use the drag-and-drop mechanism. This explanation is supported by data collected during the interview, as a number of students in the Hybrid condition commented on their preference towards typing over the drag-and-drop composition strategy. For example, one student said: "*for the most part, I just type the code myself. I don't think the blocks are useful other than showing what you can do.*" This finding replicates other similar work which looked at how, when given a choice to use either a text-based or blocks-based modality, students shift from blocks to text as their experience grows (Matsuzawa et al., 2015).

In contrast to the declining use of blocks for composition in the Hybrid condition, the students in the Blocks condition show a relatively consistent block-addition-to-run ratio. The outlier for this trend in the Blocks condition was the Grand Ranger assignment, which saw a spike in the number of blocks added between runs. This happened because this assignment asked students to work with multiple conditional statements and, since each conditional statement required a number of blocks to be added (the conditional itself, the comparator, and blocks for one or both of the argument slots), in total, more blocks were added to make basic changes to the program. Whether or not the requirement for using the blocks-based form of authorship affects students' approach to programming in Java is the topic of the next chapter.

Quick Reference Usage

The last dimension of programming practices this chapter investigates is if and how students used the Quick Reference feature of the Pencil.cc environment. As a reminder, the Quick Reference menu is an in-editor resource that provides instructions on various aspects of

the Pencil.cc language and environment. Like the blocks palette, the Quick Reference guide is conceptually organized and provides definitions and examples of all of the central topics covered in the five-week introductory curriculum. Logging was put in place to track when and how the Quick Reference feature was used. This information gives us insight into when students need assistance beyond what is provided by the modality itself. Thus, looking at Quick Reference is not about understanding that feature itself, but to understand shortcomings of the modalities in providing sought after support.

Overall, students working in the different modalities visited the Quick Reference manual at very different rates. The Quick Reference was used a total of 2,591. By condition, that number breaks down as follows: Blocks loaded 229 pages, Hybrid loaded 553, and the students in the Text condition loaded 1,809 Quick Reference pages. Running an ANOVA calculation on the total number of look-ups by condition shows the usage of this resource differed significantly by condition $F(2, 82) = 10.7, p < .001$. A Tukey HSD post hoc analysis shows the Text condition to be significantly different from both the Hybrid and the Blocks conditions at the $p < .001$ level, while the Blocks and Hybrid conditions are not statistically different from each other ($p = .34$). Looking at the pages for the specific concepts that were the focus of the curriculum, we see a similar pattern in use of the Quick Reference feature. Figure 7.11 shows these numbers.

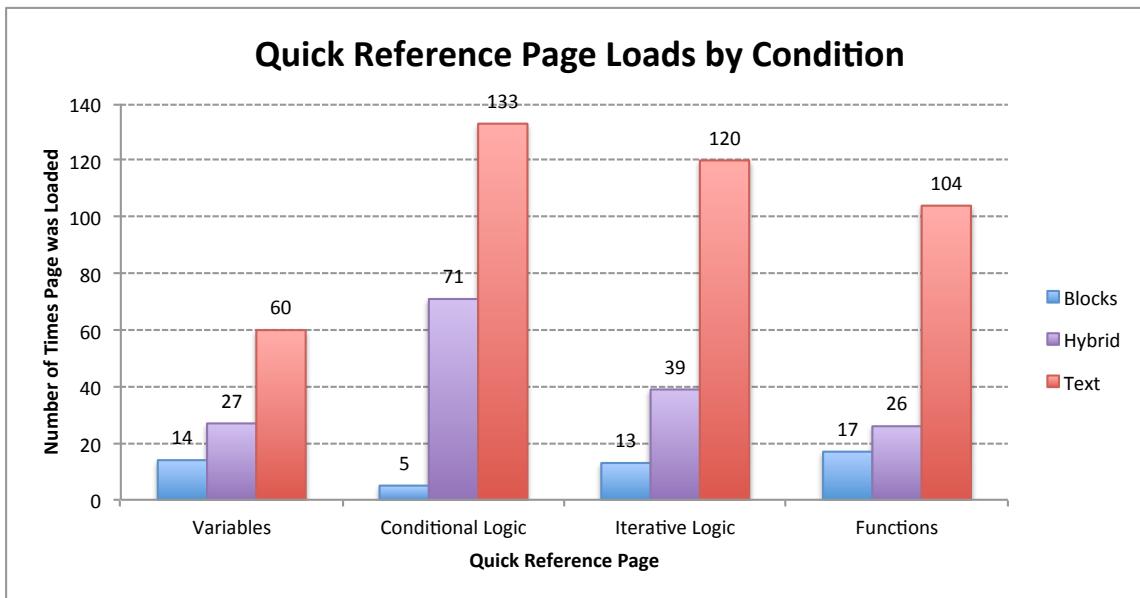


Figure 7.11. The number of times the Quick Reference pages were loaded for the four concepts covered in the introductory curriculum, grouped by Condition.

This figure shows a consistent pattern in Quick Reference usage. For every category, the Text condition used this reference the most, followed by the Hybrid condition, with the Blocks condition using it least often. Running an ANOVA on each of these categories shows the difference between usage to be significantly different for all four concepts (Variables: $F(2, 82) = 3.29, p = .04$; Conditional Logic: $F(2, 82) = 5.89, p = .004$; Iterative Logic: $F(2, 82) = 10.39, p < .001$; Functions: $F(2, 82) = 10.7, p < .001$). What this chart shows is a systematic difference in reliance on programming assistance that resides outside of the editor. Our explanation for this is that because the text modality does not provide the same scaffolds that the Blocks and Hybrid modalities do, namely a list of available commands via the blocks palette and hover-over tips for each command, that students in the Text condition had to look elsewhere for guidance. This explanation is corroborated by the vignettes presented earlier in the chapter, where we saw the student in the Text condition refer to the Quick Reference menu, but not the Hybrid or Blocks students. In the final section of this chapter, we summarize the findings presented.

Discussion

This chapter sought to understand the programming practices engendered by the blocks-based, text-based, and Hybrid modalities of the Pencil.cc environment. Using a variety of methods and data sources, the chapter depicts a number of facets of the learning to program process. Collectively, they paint a vivid picture of the practices and patterns that develop as novices learn to program across three different modalities. This final section provides a discussion that looks across these various data sources to summarize what was learned in this chapter.

Using the Vignettes

The first half of this chapter presented vignettes of students working in the three modalities used in this study. The goal of this section was to provide a thick description of what it looked like for beginners to author programs in different modalities after having worked in them for five weeks. In using the Mid interviews, we see students demonstrating how they had come to rely on various scaffolds and affordances of the different modalities and environments. This discussion section is broken down into four subsection, each of which covers a different step in writing a program: deciding what to do, doing it, making sure it is right, explaining it. For each phase, the three modalities are addressed.

Deciding What to Do

All three students were able to come up with a strategy for accomplishing the task set before them, however, the role of the environment in supporting this process differed. The student in the Blocks modality relied heavily on the blocks palette, browsing through the various categories a number of times both at the beginning of the programming process, as well as

throughout the activity. We also saw this student use the hover-over tip feature of the editor to gain additional information about the set of commands available to him. The student from the Hybrid condition was also observed browsing through the blocks palette early in the interview, thus taking a similar approach to the idea generation portion of the programming activity. The vignette of the student from the Text condition on the other hand, did not have the blocks palette available to him, so could not use it as a resource. Instead, the student drew ideas from memory. This is especially clear at the outset of the activity when he quickly begins typing and uses incorrect commands that he says he remembered needing for previous programs. The Blocks and Hybrid vignettes also used prior knowledge to construct their programs, for example, knowing to store their random numbers in variables, but these three vignettes do show how modality facilitates this initial phase of writing a program. In other interviews, we saw students in the Text condition browse the Quick Reference menu as a source of ideas although this use occurred less frequently and was often less useful than how the Blocks and Hybrid students used the blocks palette. Looking across the full set of students, the use of the Quick Reference by student in the Text condition proved to be a common practice and one that was distinct (or at least distinct in frequency) to the text modality.

Writing the Program

The next phase of writing a program on display in these vignettes was the students actually composing the program. It is in this phase that the three modalities differed the most. In the Blocks condition, we see the student using the cursor to drag-and-drop commands into the palette, assembling the program block-by-block, command-by-command. We also see the student incrementally building up commands, like when he dragged the `random` block into his program, changed its argument from 6 to 15, then dragged out the `variable` block, and finally

moved the modified `random` block inside the `variable` block. In this way, the command was not constructed in a sequence that matched the left-to-right presentation of the final statement, which is the natural way to compose in a textual modality³⁸. Throughout the Blocks vignette, the student always added and moved commands using the cursor, only using the keyboard to change arguments inside the blocks. In contrast, the student from the Text condition was unable to add or edit the program using a drag-and-drop approach, instead typing in every command in the program one character at a time. Using this approach, the student in the Text condition encountered a number of compile-time syntax errors, something that did not happen in the Blocks vignette. In both the Blocks and the Text use-cases, the form of authorship was imposed by the modality, the Text condition was not able to drag-and-drop commands and the Blocks condition was not able to author his program entirely with the keyboard.

The Hybrid condition proved to be the most interesting of the three vignettes in terms of how the program was authored given the modality supported both drag-and-drop blocks-based additions and keyboard-driven textual authorship. In the vignette presented, we see the student fluidly moving between these two forms of composing a program. He dragged the commands `random` and `for` into his program, while typing in variable declarations and a `while` loop. We also saw the student take advantage of conventional text-editing strategies, including copying and pasting chunks of text to duplicate logic in his program, a technique supported in the Blocks

³⁸ It is worth noting that it is possible to author programs in a non-left-to-right sequence in a text-based modality, although this is more cumbersome as it requires the author to move from the character keys to the arrow keys (or mouse/touchpad) and back. This authoring pattern was not observed during the four text-based interviews.

modality, but rarely (if ever) used³⁹. This vignette also showed the student encountering syntax errors like the student in the Text vignette. Collectively, this vignette shows how a student from the Hybrid condition, after working in the modality for five weeks, had become comfortable using both a blocks-driven and text-based approach to authoring programs.

Correcting Errors and Making Sure the Program Works

After composing their programs, students must then check to see if what they just wrote behaves as expected and if not, make the necessary modifications. All three of the vignettes followed students writing the same program and then saw them encounter various errors on their journey towards a working program. While all three students had bugs in their initial attempts to write the program, the types of struggles the students encountered were not all the same. Most notably, the Hybrid and Text vignettes showed the student encountering syntax errors, whereas the Blocks vignette never encountered the red X that, in the editor, denotes a syntax error. Between the Hybrid and Text condition we saw different strategies employed to verify syntax or look for help. In the Text condition, the Quick Reference menu became the main source of syntactic help. While it proved useful, it was not without its difficulty, as the learner must know what he or she is looking for as the Quick Reference menu is conceptually organized, as opposed to having an entry for every single command⁴⁰. While the Quick Reference menu is a feature specific to Pencil.cc, it is a common way to organize help resources. The Hybrid student on the other hand, was able to use the blocks palette and the drag-and-drop feature to check syntax by

³⁹ During the five weeks of data collection and observation, no student was seen using the copy/paste technique in the Blocks modality even though it is possible using the conventional `ctrl+c` and `ctrl+v` key bindings.

⁴⁰ For example, the Quick Reference menu has a single entry for Arithmetic that includes the various mathematical operators available to the programmer.

dragging out commands to see their syntax. This strategy was observed in every Hybrid interview conducted and was widely used during the five-week curriculum.

In all three vignettes, students made sure the program was working by running the program and then evaluating the output relative to what was expected. This run-evaluate-edit cycle was the catalyst for many of the edits across the vignettes. For example, in the Hybrid vignette, the student, on seeing that only two values were being displayed, replaced his `if` statement with a `while` loop. Similarly, when the students saw the output of their program double, as opposed to increase by a fixed amount, they responded by revising their programs. It is worth noting this pattern of run-evaluate-edit did not differ by modality and it did not appear as if modality played a significant role in how students went about this component of the programming process.

Explaining the Program

The last step in writing a program that we saw in all three vignettes was the students explaining the programs they had written. In all three modalities, students used the cursor as a pointer to direct the interviewer's attention but with subtle differences. In the case of the Text and Hybrid interviews, students would point to specific characters and words, move their cursor back and forth over lines, or click-and-drag over the text, to highlight portions of the statement. In the Blocks modality, the student similarly guided gaze using the mouse and had additional visual indicators like color and shape to explain what was happening, but lacked the ability to highlight portions of the code to make it clear what specifically he was referring to. While this is a relatively small difference, it is these small things that collectively lead to different experiences and lead students to develop different programming practices. In this way, modality facilitates this portion of the authoring process in a slightly different way.

Wrapping up the Vignettes

Across the three modalities we saw different practices employed during all phases of the programming activity. This analysis intentionally did not project any normative evaluation of the different moves made by the learners because there is no one “right” or “best” way to write a program. Instead, this section documents how the different modalities support novices in forming different programming practices and affords different compositional strategies. The big question that remains from this analysis is the one that is tackled in the next chapter: If and how these programming practices transfer to more conventional, professional programming languages and environments?

Differences in Programming Practices and Artifacts

The second half of this chapter used the computational data collected to look across the full set of participants to understand how the differences identified in the vignette analysis manifested themselves in aggregate outcomes across all of the students that participated in this study. This analysis included looking at programming patterns in the form of number of programming runs, types and frequencies of errors, and finer-grained patterns of composition observed across the three modalities. In this section we summarize this work, drawing across the various measures used to identify aggregate trends in programming practices and constructed artifacts by modality.

Programming Practices by Condition

Looking at the data logs collected during the first five weeks of the study for differences by condition, characteristics of the modality start to emerge. Students from the Blocks condition ran their programs the least frequently, spent the most time between runs, and also produced the

longest programs on average. Characteristics of the modality provide potential explanations for these findings. First, the lack of syntax errors due to the shape and construction constraints provided by the blocks kept students from engaging in a cycle of making small syntactic changes and re-running the program to see if the change works. By backgrounding syntax, this modality allows students to focus on the semantic and algorithmic aspects of writing the program, which require more concentration and could in part explain the fewer number of runs and the more time taken between consecutive runs. Second, the ease of dragging-and-dropping commands relative to typing them in character-by-character could explain the longer programs produced by students in this condition. Since it is easier to add more commands, students are more likely to do so. There are also practices we do not see the students from the Blocks condition engage in, notably, the Quick Reference menu is relatively rarely used compared to the Hybrid and Text conditions. As discussed above, our explanation looks to the other various supports the modality provides for partially alleviating the need for learners turning to this resource.

Whereas students in the Blocks condition ran their programs the fewest number of times and at the slowest pace, the students in the Hybrid condition ended up at the other end of the spectrum. Students working in the Hybrid modality ran their program the most often, having the shortest average delay between consecutive runs, and also were found to have re-run their programs in under five seconds the most number times of the three modalities. These outcomes were a little surprising given that the Hybrid condition has largely resided between the Blocks and Text conditions for many of the dimensions of programming explored in this dissertation. A possible explanation for the frequency of runs was briefly proffered earlier in this chapter. Since students in the Hybrid condition can add commands to their programs by dragging-and-dropping them, an action that is quicker and easier than typing out the command character by character,

students could quickly build out their programs. At the same time, the text-based canvas does not provide the syntactic scaffolds of the blocks modality, so it was possible for students to introduce syntax errors into their programs. This allows students to quickly write programs that contain errors, which then need to be debugged, which is often done through tinkering and making small changes. This iterative development produces quick turn-arounds and a rapid succession runs of their program. This is one potential explanation, but the data suggests this is only part of the story, since Figure 7.10 shows the use of the blocks feature declining over the course of the curriculum, while the speed and number of runs relative to the other conditions did not. Unfortunately, this dissertation did not gather keystroke level data in the logs, so we do not have a complete view into the programming practices in aggregate for students in the Hybrid or Text conditions, so the explanation for these patterns remains incomplete and is left as an avenue of future research.

Students working in the Text condition ended up writing the shortest programs and often landed between the Hybrid and Blocks students on the measures used to evaluate the programs and practices in aggregate. Students in the Text condition also used the Quick Reference feature of the environment far more often than either of the other two. We think these two features are related. Students working in the Text modality had to contend with syntax errors while having the fewest number of in-editor scaffolds available to help. Authoring shorter programs is a logical outcome when encountering more syntax errors that impede progress and when having to add content to a program faster than character-by-character. Further, relying on the Quick Reference menu would also slow down the authorship process as it resides outside of the components of the editor directly involved in the act of authoring the program. The Text vignette provided a glimpse into some of the challenges associated with the Text modality when he

encountered syntax errors and was unable to fix it on his own using only the Quick Reference menu. The Text condition of the introductory portion of the study was the closest to what students will be doing in the next phase of the class, so the discussion of programming practices and artifacts will continue in the next chapter of this dissertation.

Collectively, looking at different aspects of programming practices and features of programs authored in the three modalities, we can see differences emerge. In some ways these differences mirror trends that were documented in the previous two analysis chapters, while in other instances, some of the trends and outcomes were new and unexpected. For example, throughout the last two chapters, the Hybrid condition has largely lived between the Blocks and Text conditions, a natural home for a modality that is a mix of the other two. In this chapter, however, we see places where the Hybrid condition lives as an outlier, showing how blending blocks-based and textual programming modality can produce a new modality that engenders practices and uses distinct from its two parents.

Programming Practices by Concept

One of the features of the study design of this dissertation is the ability to look at how different factors affect learning to program and if and when those factors interact with each other. The guiding question for this dissertation looks at the relationship between modality and learning to program. In this chapter, we can see how and when modality interacts with the various concepts that are foundational to programming. The last chapter showed how modality and concepts interact with respect to learners' comprehension of programs (Figure 6.8). This chapter compliments those findings by providing insight into the interaction of modality and concept for students during the composition of programs.

Figure 7.7 shows how often students ran their programs by assignment while also overlaying the concept being taught for each assignment. That figure and the analysis that it summarizes shows how students ran their programs at different rates for all three assignments focused on conditional logic and two of the three iterative logic assignments. Conditional logic emerged again as an outlier in Figure 7.10, which showed the average number of blocks being added to a program between consecutive runs. This was especially true for the Grade Ranger assignment, which asked students to take a number between 0 and 100 in as input, then report what the letter grade for that score would be (91-100 returns an A, 81-90 returns a B, and so on). In this assignment, students added more than twice as many blocks per run than any of the other assignments, with the other two conditional logic assignments also falling among the four assignments that had the most number of blocks added per run. Similarly, there is also a spike showing that the time between runs increases for this assignment. These data provide evidence for the claim student have often made (and was reported in Chapter 4) of blocks-based programming being perceived as slower than the text-based alternative. Triangulating this data provides a way to show that while it may not be universally true that Blocks authorship is slower than text-based programming, in the case of conditional logic, more blocks are required to construct a statement and thus blocks-based construction is slower than comparable statements in the text modality.

Conclusion

This chapter presented a third and final analysis of the data collected in the first five weeks of the school year. Collectively Chapters 5, 6, and 7 paint a detailed picture of high school students learning to program in three different modalities. With this chapter, we showed what it looked like to author programs in the different modality through three detailed vignettes and

looked across the full set of students to tease out systematic practices and trends within the programs and log data collected over the course of the five weeks. Taken together, this chapter, along with the preceding two, provides a complete picture of how modality influences novice programmers. The major outstanding question this dissertation has yet to answer is if and how students' experiences in these three introductory programming environments impact their early Java learning. This question will be answered in the next and final analysis chapter in this dissertation.

8. Transitioning to Java

One of the over-arching goals of this dissertation is to understand the role of modality in introductory environments in terms of if and how it prepares learners for later computer science learning opportunities. Put more concretely, do blocks-based programming environments effectively prepare learners for later text-based programming? And how does the blocks-based modality compare to isomorphic text and hybrid blocks/text interfaces with respect to preparation for future computer science learning? This chapter presents data and analysis towards answering these questions. Understanding this is consequential as many uses of blocks-based tools in formal educational contexts presuppose that such tools will help prepare students for later instruction in text based languages. However, little empirical work supports this position, and as one of the students in this study said during an interview: “*I can guarantee that the transition between languages will be hard to do.*”

This chapter begins with an analysis of student responses to questions from the Mid and Post attitudinal surveys which investigate learners’ experiences working in either a blocks-based, text-based, or hybrid blocks/text interface, specifically focusing on if students found that experience to be useful preparation for Java. As part of this analysis details about what was learned in the introductory portion of the class that transferred to Java are investigated by condition, trying to identify the strengths of each. Throughout this section, the results are supplemented with data from interviews conducted with students from all three conditions. The next portion of this chapter looks at how attitudes towards and perceptions of programming shifted between the end of the introductory five-week curriculum and the end of the study ten weeks later. This work sheds light on how perceptions shifted after working in Java. The final

section of this chapter looks at programming practices developed and the successes of students from the three conditions in their early Java programming assignments. This final section uses the programs written by students in the first ten weeks of the Java portion of the class to understand the lasting impact the various modalities had on students programming ability. The chapter concludes with a discussion of these various analyses, tying them together to paint a larger picture of students' transitions from the three modalities used in the introductory portion of the study to Java.

Perceptions of Introductory Programming Environments as a Preparation for Java

In this section, interviews and survey responses are analyzed to try and understand how students viewed their experiences using the different modalities with respect to the transition to Java. The section starts with data from the Mid and Post interviews, supplementing these data with excerpts from interviews conducted with students at the midpoint and conclusion of the study. The first analysis of this section investigates whether or not students themselves found the experience of working with Pencil.cc useful for preparing them from later Java learning. This analysis begins with students' responses to the 10-point Likert question asking them on the Mid survey "What I learned with Pencil.cc will help me learn Java" and then on the Post survey: "What I learned in Pencil.cc has helped me in Java." In both cases, a higher value means stronger agreement with the statement. It is important to remember, when asked about Pencil.cc, students from each condition envision a programming environment with a different modality, so when students in the Blocks condition think about Pencil.cc, they are thinking about a blocks-based programming environment⁴¹. Student responses to this question by condition are shown in Figure

⁴¹ This question asks about Pencil.cc as opposed to a specific modality because students do not necessarily know that there are multiple versions of Pencil.cc. There is possible conflation

8.1. Please note the y-axis in the chart does not start at zero, this was done to make the trends clearer.

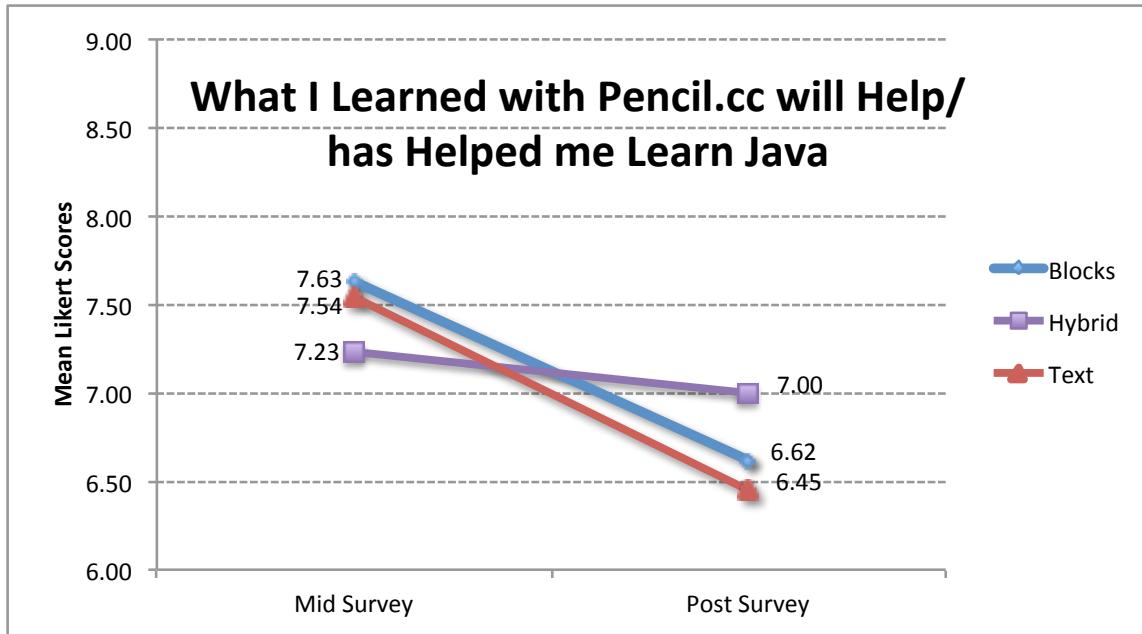


Figure 8.1. Student responses to whether or not they thought their time spent working in Pencil.cc was helpful for learning Java.

On this question, the average Mid survey response was 7.5($SD = 1.7$) and the average post survey response was 6.8 ($SD = 2.2$), showing that students collectively agreed with the prompt that the introductory tools were helpful, but did not do so in a particularly strong manner. Between the Mid and Post administration, all three conditions show a negative slope, meaning that after working in Java for 10 weeks, students viewed what they had done during the first five weeks of school as less useful than they had before the Java portion of the study. Overall, the average score on the Mid survey was higher than the post survey ($Z = 1217.5$, $p = .00$ running a Wilcoxon signed ranks test on normalized z-scores). The changes between the Mid and Post

between modality and non-modality-related features of Pencil.cc, but as all non-modality features of Pencil.cc are shared by all three conditions, these differences would not account for the emerging differences observed.

surveys were also significant for the Text condition ($Z = 147, p = .00$) and the Block condition ($Z = 149, p = .01$). Only the Hybrid condition failed to reach statistical significance ($Z = 100, p = .86$). For both the Mid and Post surveys, the responses are relatively tightly clumped together, thus we see no differences by condition on the Mid survey ($F(2, 78) = .40, p = .67$) or the Post survey ($F(2, 80) = .43, p = .65$). This means that at neither the Mid nor Post administration did students in the Blocks, Text, and Hybrid conditions report a significantly different opinion on the utility of their introductory experience relative to the other conditions. Due to the Hybrid condition deviating slightly from the other two condition on the Mid survey (slightly less helpful) and the Post survey (slightly more useful), the data show a statistically significant difference when looking at the change in perceptions by condition $F(2, 74) = 3.38, p = .04$. A post hoc Tukey HSD test shows a significant difference in the change in perceived helpfulness between the Hybrid and Text conditions ($p = .04$) but not between Hybrid and Blocks ($p = .12$) or Blocks and Text (.86).

This analysis shows that students' perceptions of the helpfulness of the introductory modalities decreased over time. When comparing across the three interfaces used in the introductory portion, the learners in the Hybrid condition initially saw the tools as the least helpful, but by the Post survey, this position shifted to the point where the Hybrid modality was viewed as the most helpful. This is possibly explained by some of the findings given in Chapter 4, where the Hybrid condition, being both similar enough to and sufficiently different from Java, was initially seen as having little in common with Java but for the links to become more clear after working in Java. An alternative explanation given earlier cited the potential value for exposing multiple modalities to the learner up front. The similarity of response values and trends between this question and the question of whether or not students thought Pencil.cc improved

their programming abilities (Figure 5.4 in Chapter 5) shows these two questions are linked in terms of student perceptions. This can be seen by running a validity test showing the two questions are getting at the same underlying belief (Cronbach's $\alpha = .83$). This is evidence showing that the more students' felt they learned using a specific modality, the better prepared for Java they felt. This is not surprising but is another piece of data showing that students see similarities in what it means to program across the modalities and environments.

Understanding if and how Pencil.cc does or does not prepare students for transitioning to Java is one of the central research questions being pursued in this dissertation. As such, additional questions were asked on the Mid and Post surveys trying to understand exactly how and where students saw the utility of the introductory environments they used. These questions were asked using an open-ended format, giving students more freedom to express their own perceptions. Having laid out a high level trend on student reactions to the introductory portion of the course broken down by modality, the analysis next digs into specific concepts and aspects of the introductory portion of the course that they identified as being helpful once they transitioned to Java.

Helpful Aspects of Introductory Programming Environment for Transitioning to Java

The idea of "helpful" can mean many different things, so to further understand exactly how students found the introductory modalities to be helpful and to investigate if the type of perceived help it offered differed by condition, students were asked to respond to the following free-response prompt: "The thing I learned in Pencil.cc that will be most useful in Java is."⁴².

⁴² These charts are similar to those presented in Chapter 5 but are not the same. In chapter 5, the question was about student perceived differences between the introductory environments and Java. Here, the question looks at if/how the introductory tool was helpful for the transition to java.

Again, it is important to remember that when asking about Pencil.cc in general, the question will call to mind either a blocks-based, text-based, or hybrid blocks/text interface depending on what condition the student was in. Student responses to this question were conceptually grouped and are summarized in Figure 8.2. Each code is discussed in the paragraphs following the summative figure. Cohen's κ was run to determine agreement and consistency of the application of these codes, and found there to be moderate agreement between the coders, $\kappa = .73$, all differences were resolved through discussion. The coding manual used to code these responses can be found in Appendix E.

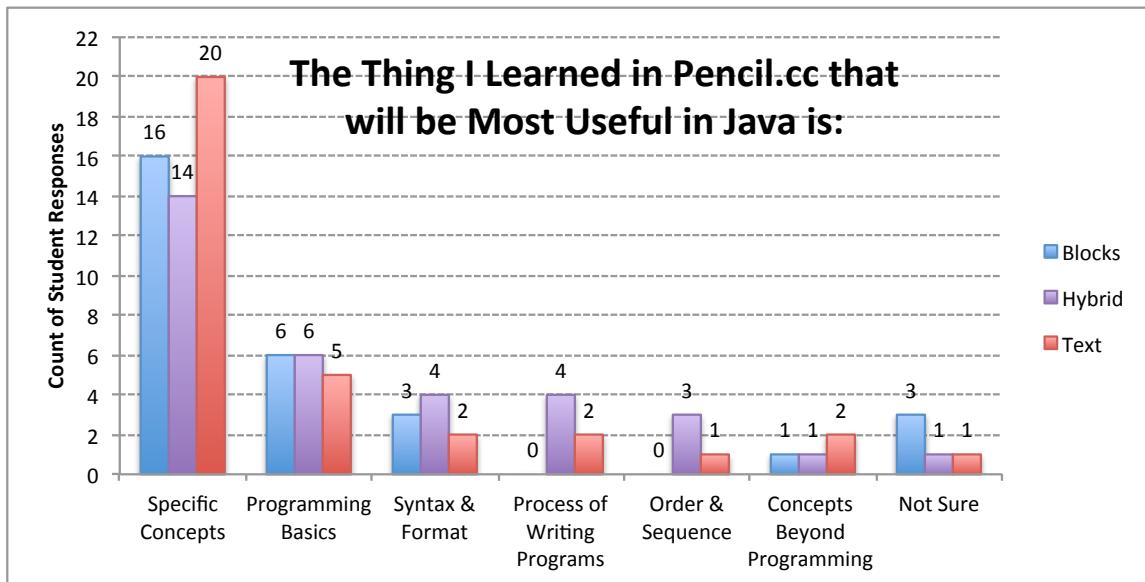


Figure 8.2. Student responses to perceived useful aspects of Pencil.cc with respect to the upcoming transition to the Java language.

The first and most frequent feature of introductory environments cited for being useful in Java was the Specific Concepts code. A response was given this code anytime a student mentioned a specific concept from the set of topics that had been covered in the first five weeks of class (variables, conditional logic, iterative logic, and functions). For example, one student responded: “*I learned how to use for loops, variable referencing, functions and parameters,*

while loops, and if statements." Not all students mentioned all concepts, a number of students gave responses like "*The concepts of variables and if statements*" or more succinctly "*functions*."

A more detailed breakdown of the Specific Concepts category is presented later in this section.

After Specific Concepts, the next category identified was Programming Basics, which includes responses that speak to general programming knowledge, as can be seen in the response "*I learned the basic components of programming, which was a mystery to me before this class.*" Now, *I will be able to apply all of this information to Java.*" The Syntax and Format category was the next most common code and captures responses that speak directly to those two aspects of programming, like in the response: "*That spacing, commas, and syntax matters*". The next two codes are related, the Process code captures students attending to the steps taken to write a program including decomposing the problem, designing an algorithm, and then the steps required to execute the stated plan, which can be seen in responses like: "*The process of planning the right steps in order to make your program run the way you want it to.*" The Order and Sequence code was applied to responses that talked about how programs run or attended to the relationship between consecutive commands, like the response: "*Coding runs from the top to the bottom*". The Concepts Beyond Programming code captures students attending to important knowledge that is relevant for programming but that is not specifically about programming, including things like problem solving and being organized. The final code is for students who responded that they did not know how the introductory tool would be useful for their upcoming Java work.

The first thing to note about these responses is the overwhelming frequency of students attending to specific concepts as being helpful. This suggests that the most direct relationship students expected between the introductory tools and Java was conceptual; that concepts encountered, like variables and functions, would be useful for their work in Java. This sentiment

was also captured during the interviews; for example, one student from the Blocks condition described the introductory environment as being “*like a stepping stone. It reminds you about assigning variables and...different methods and assigning each class and functions. Because every code, well, every program has functions, has code, has variables...so it kind of helps us wrap us around that idea of being organized and the concepts*”’. In other words, the introductory environment covers the same concepts, but in a way that is easier to understand. The fact that this category was the most frequently cited for all three conditions suggests that modality was not a deterrent for seeing the conceptual similarities between introductory and professional programming environments.

A second interesting trend is the general similarity in responses of the Text and Hybrid conditions compared to the Blocks condition. This can be seen in the Programming Basics, Process, Order & sequence, Meta Programming Concepts, and Not Sure categories. This pattern matches the findings in the perceived differences analysis in Chapter 5 and Chapter 6’s findings on conceptual learning showing that in certain contexts the Hybrid condition was found to be more similar to the Text condition than the Blocks. The two categories where the Hybrid and Text conditions outnumber the Blocks responses, Process and Order and Sequence, reveal something about the difference in utility across the modalities. The fact that no Blocks students attended to Process while six students across the other two conditions did, suggests that the text manipulation aspects of the Text and Hybrid forms of Pencil.cc helped students see procedural similarities between writing programs in different environments. For example, one student in the Hybrid condition responded: “*The understanding of what you want your program to do. By knowing the step by step process of what you want, you will be actually able to know how the program works.*” Nothing in this response, at the surface level, seems coupled with modality, but

nonetheless, no Blocks students seemed to attend to these procedural dimensions of programming on the mid survey. One potential explanation draws on the authenticity finding identified in Chapter 4 as contributing to the potential transfer of procedural strategies as the Hybrid and Text environments were viewed as more similar to “real programming.”

After working in Java for 10 weeks, students were again asked this same question on the Post survey, with the tense changed from future to past. Figure 8.3 shows student responses at this point in time. Cohen’s κ was run to determine agreement and consistency of the application of these codes, and found there to be moderate agreement between the coders, $\kappa = .63$, all differences were resolved through discussion.

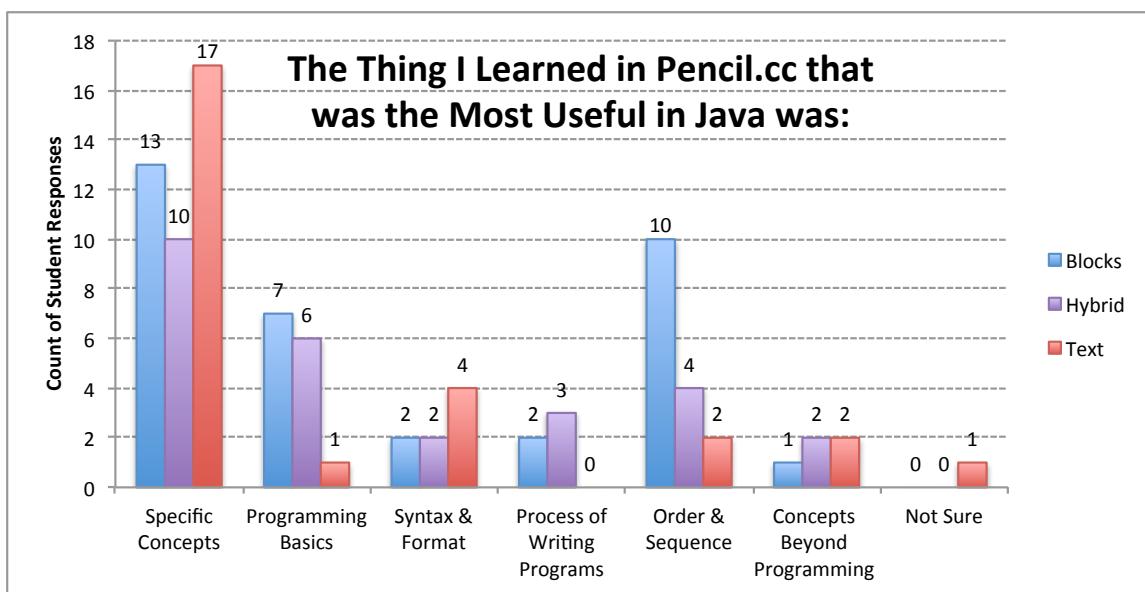


Figure 8.3. Student responses to a question on the useful aspects of Pencil.cc and the modality it used after working in the Java language for 10 weeks.

The first that stands out in this Figure compared to the analysis of the Mid responses is the frequency of Blocks students citing Order & Sequence on the post survey relative to the Text & Hybrid conditions. This is especially interesting given the fact that no students in the Blocks condition predicted this would be the case on the Mid survey. This suggests that a strength of the

Blocks modality is that it makes clear to the learner the order in which commands execute and that this feature only becomes salient after the transition to the textual modality. A similar pattern can be seen in the change in the Process category, where Text and Hybrid decreased and Blocks increased, suggesting that features less tightly coupled to the modality grew in perceived helpfulness in the Blocks condition.

This Mid to Post comparison is interesting in that it sheds light on the differences between what students thought would be useful and what they claimed to be useful across conditions. The growth in the frequency of Order & Sequence being cited by Blocks students combined with a decrease in Syntax & Formatting is evidence towards what features of the blocks-based modality are useful in the transition to Java. Namely, that making explicit order is clear and helpful, while syntactic features of the blocks-based interface are not so useful. Likewise, the high counts of responses in the Specific Concepts category in both the Mid and Post show that students can recognize the conceptual similarities across the transition of modalities and environments.

Given the high frequency of responses in the Specific Concepts category and the fact that it is comprised of many constituent concepts, we next tease apart that category to better understand what concepts were actually cited by name in the responses. Table 8.1 shows the frequency that students cited the four concepts covered in the opening five-week curriculum.

Table 8.1. Student responses to the concepts perceived to be most useful for Java that were learned in Pencil.cc.

	Functions & Parameters		Variables		Iterative Logic		Conditional Logic	
Mod. \ Time	Mid	Post	Mid	Post	Mid	Post	Mid	Post
Blocks	10	2	4	8	3	0	6	1
Hybrid	8	0	5	9	7	0	3	1
Text	11	1	6	16	8	1	6	2

Total	29	3	15	33	18	1	15	4
-------	----	---	----	----	----	---	----	---

There are a few things to note in this table. First, looking only at the Mid responses, students mentioned Functions & Parameters in their concept responses significantly more frequently than other concepts (29 times, compared to 15, 18, and 15 for the other three concept areas). Doing this same comparison for Post responses, we see an overwhelming focus on Variables (33) compared to the other three concepts (3, 1, and 4). This is most likely explained by the fact that the first concept covered in the Java portion of the class was variables and Input/Output, so this increased attention on variables is not especially surprising. Functions was the last topic covered in the introductory curriculum, which suggests there may be a recency bias in the responses. The last thing to note in this table is the relative lack of variance across the three conditions. For each concept, looking down the columns, the numbers are relatively consistent, suggesting that the modality did not significantly influence student responses with respect to what concepts were cited. The one exception is Iterative Logic, where only 3 Blocks students cited it compared to 7 and 8 in Hybrid and Text students respectively. This is slightly surprising as the ease of loops is often cited as a strength of blocks-based programming languages, although other research shows that it does not support deeper conceptual understanding, a finding discussed in the Chapter 6. A possible explanation for the greater frequency of iterative logic for the textual conditions could be that the need to remember particularities of syntax made the concept more salient to the learner.

Perceptual Outcomes Discussion

Two interesting findings stand out from the analysis of students' perceptions of the three introductory modalities with respect to their utility for learning Java. First was how the Hybrid condition fared relative to the other two. Students in the Hybrid condition initially viewed the

Hybrid condition to be the least helpful relative to how students in the Blocks and Text conditions responded. However, after working in Java for 10 weeks, the students in the Hybrid condition saw that experience as the most helpful relative to the other two modalities. This trend stands out from most of the other trends in this dissertation, where the Hybrid condition is aligned with either the Blocks or the Text students. Here however, the Blocks and Text conditions are similar with the Hybrid environment serving as the outlier. This could be one place that Hybrid is not acting as a best-of-both-worlds tool, but instead is a case where the whole is greater than the sum of the parts. In other words, the Hybrid interface is contributing more than either modality it is built on can individually. One potential explanation for this is the fact that the Hybrid condition is the only one where students see code represented in more than one way, which can help lead students to the perspective that not all programming interfaces or languages are the same. This view, coupled with the scaffolds of the blocks-based features and the authenticity of the textual canvas collectively could explain this positive outcome for the Hybrid condition.

The second finding that stands out from this analysis is the emergence of students in the Blocks condition citing order and sequence as being something learned in the introductory tool that was helpful in Java. Before working in Java, no students in the Blocks condition cited this reason. Afterward, a third of students cited this as being something they learned in the Blocks modality that was helpful in Java. The novices' ability to focus their attention on order and how programs fit together instead of syntax or other mechanistic distractions is cited as one of the conceptual strengths and learning benefits of the blocks-based approach to programming (Maloney et al., 2010). These claims are often made without data from learners to support it.

This finding suggests that indeed, the blocks-based modality is effective for helping learners understand the role that order and sequencing play in the practice of programming.

Changes in Attitudes and Perception in Java

Part of understanding and evaluating the lasting impact of working in different modalities during the introductory portion of the course is investigating how the attitudes and perceptions of programming that formed during their use persisted or changed as students moved on to Java. This section is a continuation of the analysis presented in Chapter 5 that looked at students' attitudes and perceptions of programming. Whereas that previous analysis looked at student responses from the Pre and Mid surveys and the changes between them, this section looks at shifts from the Mid to the Post survey. While each of the diagrams in this section show the trajectory over the full 15 weeks of the study, the analysis starts by looking specifically at the Java portion of the study, before incorporating findings from the earlier analysis to paint a larger picture of students' trajectories over the 15 weeks. The charts in this section are the same as those shown in Chapter 5. Note, the charts do not start at zero on the y-axis and do not all cover the same portion of the 10-point Likert scale, but are on the same scale. This means it is safe to compare changes (slopes) across the charts, but not absolute values or vertical position.

Confidence in Programming Ability

The first attitudinal dimension discussed is students' perceived confidence in their own programming ability. As a reminder from the earlier discussion of this topic, the aggregate confidence scores is the average of the two Likert statements: I will be good at programming (or I am good at programming on the Post test) and I will do well in this course. The aggregated confidence measure at the Pre, Mid and Post points in time are shown in Figure 8.4.

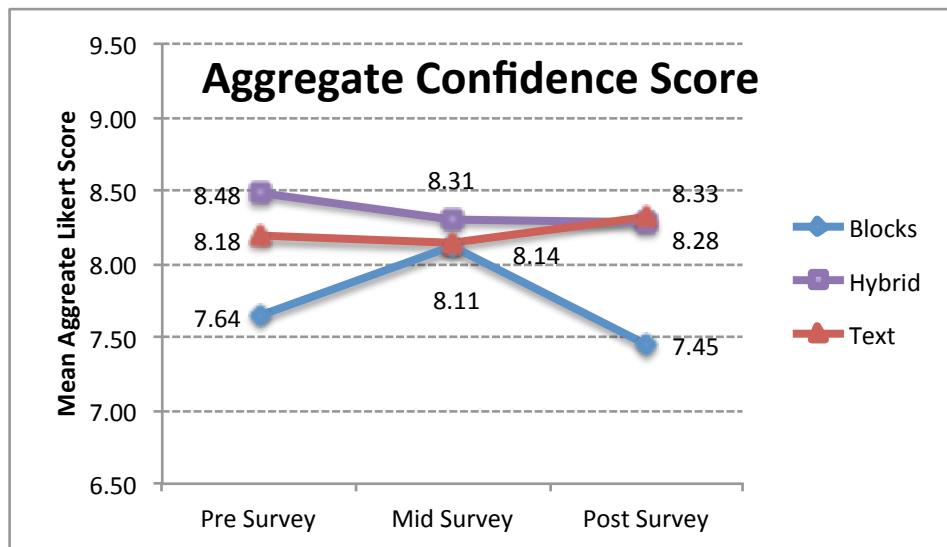


Figure 8.4. Calculated levels of students' confidence in programming at three points in the study.

The mean confidence scores on the Pre (8.11, SD = 1.47), Mid (8.19, SD = 1.67), and Post (8.00, SD = 1.86) surveys show a relatively minor downward trajectory between the Mid and Post and Pre and Post surveys, meaning students' confidence in their programming ability decreased over the 15 weeks. The decrease from Mid to Post survey was not significant when grouping the three conditions together ($Z = 945$, $p = .14$). As previously reported, there was no difference between the three conditions at the Mid point ($F(2, 74) = .10$, $p = .90$), nor was there a significant difference on the Post survey ($F(2, 80) = 2.07$, $p = .13$), although the numbers were trending in that direction. An analysis of the changes between Mid and Post for each of the three conditions also fails to return a significant result ($F(2, 74) = 1.11$, $p = .33$). Collectively, this means there was not a significant difference across the three conditions, although the trends suggest that with more statistical power, a difference may emerge.

Looking at changes within groups, we see some moderate significance emerge in the Blocks condition ($Z = 55.5$, $p = .10$), but not in the Hybrid ($Z = 58.5$, $p = .64$) or Text ($Z = 94$, $p = .69$) numbers. In the case of the Blocks condition, the results show students' confidence in their

programming ability decrease, showing that after working in Java, their overall confidence in their programming ability decreased relative to where it was after working in the blocks-based interface of Pencil.cc. From the beginning to the end of the study, none of the three conditions show a meaningful change in their programming confidence (Blocks: $Z = 158.5$, $p = .82$, Hybrid: $Z = 61$, $p = .48$, Text: $Z = 130$, $p = .82$). Taken at this level, the data leads one to conclude that the modality did not have an impact on students' confidence, however this conclusion misses the interesting trajectory followed by the Blocks condition.

The Blocks condition saw a significant improvement in their confidence after five weeks in Pencil.cc, followed by a decrease after ten weeks working in Java. At a surface level, this suggests that students thought Blocks was improving their programming, then, revising this impression after working in Java. There are a number of possible explanations for this. One is that students thought they had become better programmers after working in Blocks, which could explain the increase, however, data presented in Figure 5.4 in Chapter 5 on whether or not the different modalities had made the students better programmers, does not support this explanation, as Blocks students didn't show a different outcome than the other two conditions. A second possible explanation that is supported by the data draws from findings from the conceptual outcomes chapter showing that students in the Blocks condition performed the best on the Mid content assessments. This performance potentially explains this improvement, as students answering questions correctly would lead to them feeling more confident in their ability. However, this explanation does not fully hold up, as students in the Blocks condition also scored highest on the Post survey, at the same time point as they are reporting a decreased confidence. The increase in confidence for students working in a blocks-based interface could explain other findings showing an increased retention for students using these types of graphical tools in their

first computer science course (Cliburn, 2008; Johnsgard & McDonald, 2008), but does potentially call into question the effectiveness of such an approach for preparing students for future learning of computer science as the gains with respect to confidence do not persist.

Enjoyment of Programming

The second attitudinal dimension included on the survey was to understand if students enjoyed programming and if so, how it differed by condition both during their time using the introductory tools and their time in Java. The aggregate enjoyment score is a composite of responses to the following three questions: I like programming, Programming is Fun, and I am excited about this course. Figure 8.5 shows the average aggregate enjoyment score by student across the three surveys.

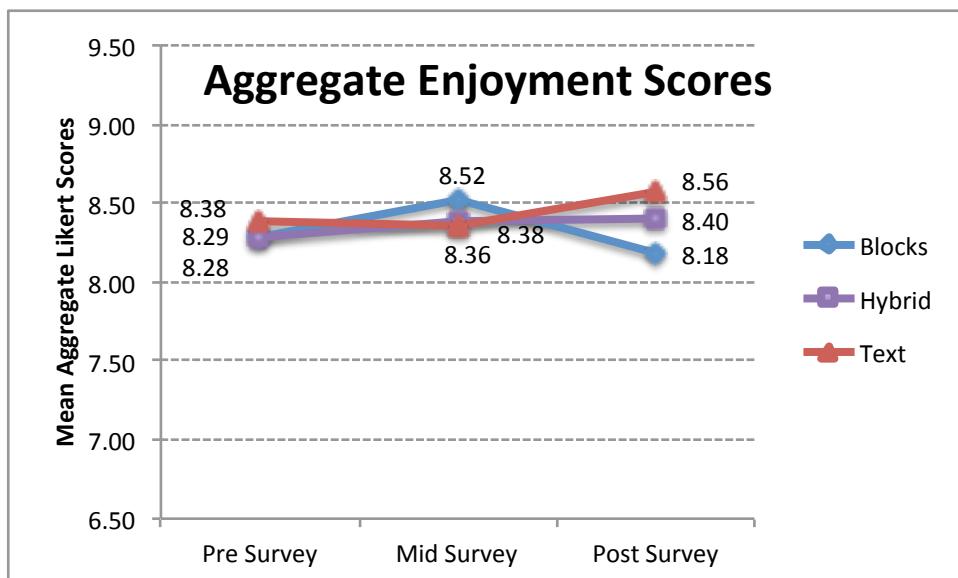


Figure 8.5. Calculated levels of students' enjoyment of programming by condition at three points in the study.

Between the Mid and Post surveys, there are no statistically significant changes, this includes looking across the Mid scores ($F(2, 78) = .08, p = .93$), across the Post scores ($F(2, 80) = .39, p = .66$), and the changes between Mid and Post by condition (i.e. the slopes) ($F(2, 74) =$

.76, $p = .47$). Likewise, a within-group Wilcoxon signed rank test does not reveal any significant changes between Mid and Post within condition (Blocks: $Z = 38$, $p = .22$; Hybrid: $Z = 96.5$, $p = .97$; Text: $Z = 83$, $p = .93$). Qualitatively, the graphs show trends that would match expectations, namely that the Blocks condition sees a decrease in their enjoyment of programming while the Text condition sees their enjoyment increase. However, the relatively minor changes do not allow for stronger claims to be made. The main take away from these numbers, like with the Pre to Mid analysis, is that modality seems to have little effect on student enjoyment of programming, which is both true when using the modality as well as after leaving the modality behind and transitioning to a more conventional text-based programming language.

The choice to use aggregated enjoyment scores provides a more reliable measure of the underlying attitudinal aspect being measured, but also potentially masks some more nuanced perspectives the students may hold and mute some trends in the data. This can be seen by looking at some of the underlying enjoyment measures. For example, looking at the responses to the “Programming is Fun” Likert question (Figure 8.6a) and “I am Excited About this Course” question (Figure 8.6b) side-by-side reveals additional insight into how the Hybrid condition is viewed relative to the two others.

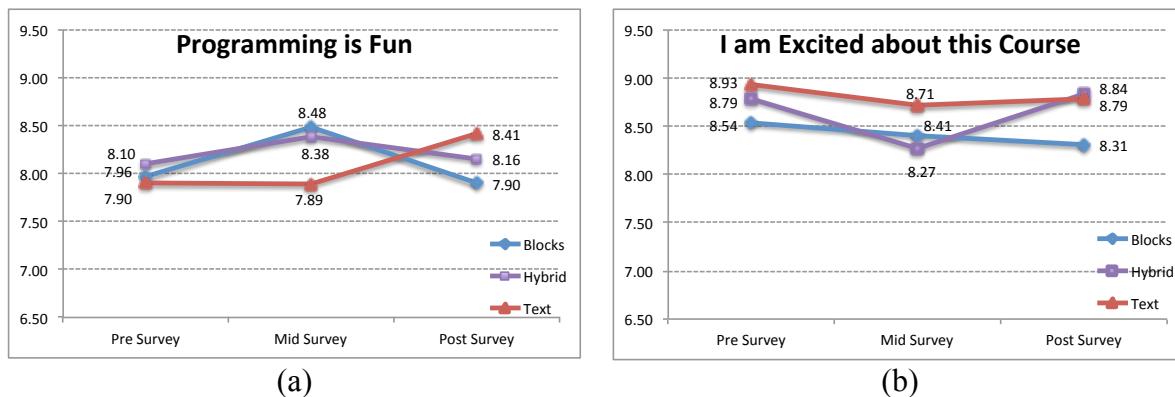


Figure 8.6. Average student responses to the Likert prompt Programming is Fun (a) and I am Excited about this Course (b) grouped by condition.

What is interesting about these two sets of responses is that, in the Programming is Fun chart, the Hybrid condition's path is similar to that of the Blocks condition, where it increases from Pre to Mid, then decreases from Mid to Post, while the Text condition has the inverse trends. However, on the second chart, I am Excited about this Course, the opposite is true; the Hybrid condition is more similar to the Text condition, showing a negative slope from Pre to Mid that is closer to the Text condition, then seeing that slope shift positive in the Mid to Post time period. There are a few things that can be gleaned from this. First, this provides evidence that the design of the Hybrid condition was successful in finding a space between the textual and graphical interfaces. Second, this shows that modality affects different aspects of students' impressions in different ways.

Programming is Hard

The question asking if students found programming to be difficult is the third attitudinal dimension being investigated. Figure 8.7 below shows the average student responses to this question by condition.

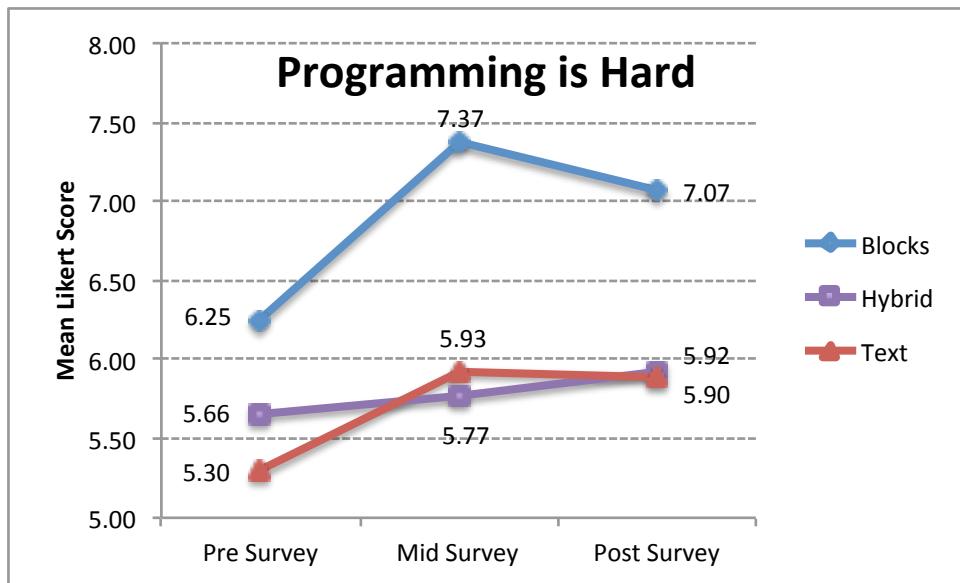


Figure 8.7. Average responses to the Likert statement: Programming is Hard.

As discussed in Chapter 5, there was a significant difference in students' perceptions of the difficulty of programming on the Mid survey ($F(2, 78) = 4.36, p = 0.02$), with a Tukey HSD post hoc analysis showing the Blocks to Hybrid and Blocks to Text differences being significant ($p = .04$ and $p = .03$ respectively). Even after responses move closer together on the Post survey, a moderate effect can be seen showing differences across the three groups ($F(2, 80) = 2.52, p = .08$). A Tukey HSD post hoc test does not reveal any significant in the pairs, but does show that the source of the difference stems from the Blocks condition being the outlier (Blocks/Hybrid $p = .15$; Blocks/Text $p = .12$; Text/Hybrid $p = 1.00$). Despite the different signs of the slope between the Blocks and Hybrid conditions, the difference is not large enough to reach a level of statistical significance ($F(2, 74) = .35, p = .71$). Looking within condition changes, again we do not find any significant changes within a condition for any of the three groups (Blocks: $Z = 84, p = .27$; Hybrid: $Z = 71.5, p = .55$; Text: $Z = 89.5, p = .55$). Overall, students' responses to the Programming is Hard question reveals that working in the modality itself has a significant impact on perceived difficulty, that transitioning to Java tempered this perceived gap slightly, but the

difference remains. In other words, the shift to Java did not affect those difficulty perceptions that formed in using the introductory environments.

Interest in Future CS

The final attitudinal survey question presented in this analysis inquires after students' interest in pursuing future computer science learning opportunities. It asked students to give a response on a 10-point Likert scale to the prompt: I Plan on Taking More Computer Science Courses after this One. Student responses at all three points in time, grouped by condition are shown in Figure 8.8.

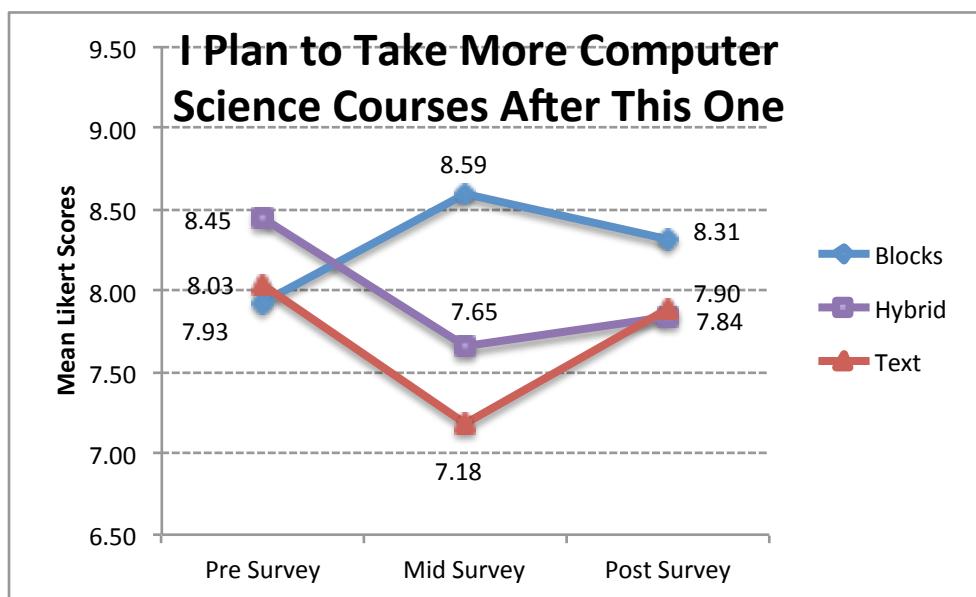


Figure 8.8. Average responses to the Likert statement: I plan to take more computer science courses after this one, grouped by condition.

Students at the Mid point of the study showed diverging interest in taking future computer science courses. Students in the Blocks condition showed an increased interest while students in the Hybrid and Text conditions showed a decreased interest. After students transitioned to Java and the question was asked again, the trajectory of student responses

changed; Blocks students were less interested in future computer science courses, while Text and Hybrid students' interest increased. The difference in the three conditions at the Post survey did not reach statistical significance ($F(2, 80) = .32, p = .72$), nor did the difference in the deltas across the three conditions ($F(2, 74) = 1.03, p = .36$), despite the opposite signs of the slopes. Looking within each condition by running a Wilcoxon signed rank test does not find significant changes for any of the three conditions (Blocks: $Z = 26, p = .32$; Hybrid: $Z = 99.5, p = .87$; Text: $Z = 82.5, p = .21$). These data suggest that the introductory language does influence students' likelihood of wanting to take another computer science class after they have transitioned, but not at a statistically significant level (with the power available in this study).

Attitudinal Changes Discussion

The analysis of students attitudes relied largely on the Pre, Mid, and Post surveys administered, using statistical methodologies to identify how and where attitudes differed over time and across conditions. Taken collectively, looking across all four of the attitudinal dimensions pursued, we see little significant effect of the transition to Java in changing student attitudes or perceptions. This lack of significance is in part due to the relatively weak statistical power of the study design due to the number of students in each condition and the three-way comparisons. With that being said, there are some noteworthy trends. Foremost among them is the fact that students in the Blocks condition showed a negative trend in all four of the categories they were asked about on the Post survey relative to their responses on the Mid survey. At the same time, for both Text and Hybrid, on three of the four dimensions, attitudes improved over the time spent working in Java, suggesting an opposite trend, that the move to Java improved students overall attitudes and perceptions of programming. In other words, students in the Text and Hybrid conditions saw their attitudes improve over the ten weeks of working in Java, while

students in the Blocks condition saw the opposite effect. These trends suggest that there are potential consequences to the decision of what language you choose to start with. For example, if a computer science sequence is setup so that in the first class students only use an introductory language and in the second class students use a professional text-based language, then these data would recommend using a blocks-based language, as students likelihood of taking future courses after using the introductory language was highest in that condition. However, if the course is setup such that students start with an introductory language and then transition to a professional language as part of the same course, then the decision of introductory modality is less important as student opinions are not different after the transition has occurred.

A second interesting trend to notice across the four categories is the frequency of slopes inverting between the two time periods. When the changes were positive between the Pre and Mid surveys, they often became negative from Mid to Post, and vice versa. This potentially reads as a sort of dampening effect, where the introductory environment pushes learners out towards some (relative) extreme and then the shift to Java brings that dimension of the learners' attitude or perception back towards their initial position. The trend was followed on nine of the 12 individual trend lines mapped in the four aggregated figures. While this characteristic is shared across the three conditions of the study, the order of the slopes (increase then decreases or vice-versa) differs by modality. On all four charts, the Blocks condition peaks on the Mid survey, then declines afterwards. The Text condition has the opposite trajectory (decrease then increase) on three of the four categories. The Hybrid condition only changes slopes once across the three categories, twice having a positive slope for both time segments and once having a negative slope. The larger interpretation of this finding will be revisited as part of the overall discussion at the conclusion of this chapter.

Differences in Java Programs

Along with interviews and written surveys, all of the programs that students authored throughout the fifteen-week study were collected. For the Java portion of the study, each student computer was instrumented so that a call to compile a program would send a copy of that program, along with the compiler output to a remote server controlled by the researcher. In this way, every student program authored and every run of the program was logged. This section digs into this data to try and understand if there were different patterns in programming practices or varying frequencies of errors and successes across the three conditions. This section begins by looking at frequency of compilations and levels of successes and continues with an analysis looking at the types of error encountered and if they differed by introductory modality. Collectively, this analysis adds another dimension to the picture that is being filled in around how introductory modality informs later programming experiences.

Frequency of Compilations Over Time

The first programming practice investigated was to see if there were differences across the three conditions with respect to how often students attempted to run their programs. As a reminder, in this class, students run their programs by calling the `javac` command from the terminal. This form of compiling and running of programs is different than many introductory programming classes, which use development environments that provide built-in compilation support (like clicking a button to compile). As discussed in Chapter 3, this approach was an intentional pedagogically-driven decision made by the teacher. As was shown in the previous chapter, there were differences across the three conditions using the introductory tools with respect to how often students ran their programs (Figure 7.7), so here we look to see if those differences persisted. Students in the Blocks condition ran the `javac` command an average of

142.3 times ($SD = 67.1$), the same value for the Text conditions was 130.9 ($SD = 61.1$) and 150.9 ($SD = 79.2$) for the Hybrid Condition. These numbers are not statistically significant ($F(2, 80) = .594, p = .55$), meaning in aggregate, there was no difference in the number of calls to javac based on the modality students used in the first five weeks of the school year. These numbers are visually depicted in Figure 8.9, which shows the average number of compilations for each student across the three conditions by week^{43,44}. This chart includes both successful compilations as well as calls the resulted in an error.

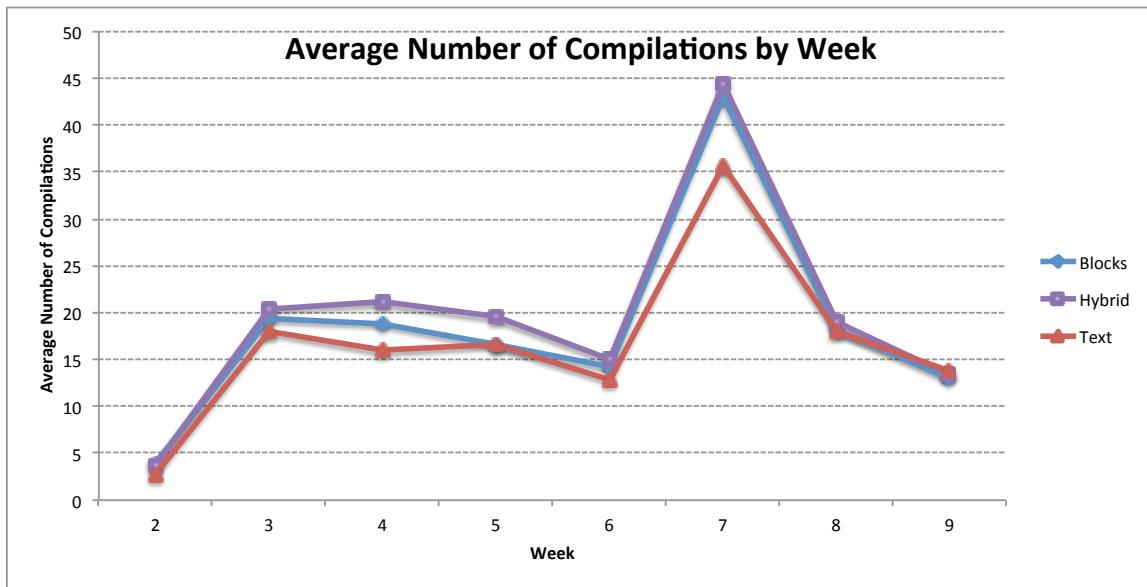


Figure 8.9. The average number of compilations of Java programs by student by week.

Although none of the differences between the three conditions reach statistical significance, this chart does start to show some patterns. First, in six of the eight weeks, the Hybrid condition had the most calls to compile, while the Text condition had the fewest number

⁴³ Unless otherwise specified, all charts in this section show per-student averages to control for the fact that not all classes had the same number of students.

⁴⁴ The chart starts at week two of the Java unit as no calls to compile happened during the first week of the Java unit, largely due to other non-Java related activities, like administering the attitudinal and content assessments, getting Java development environments setup and presenting final projects from the first portion of the course.

of calls in the same number of weeks (six out of the eight covered). Second, for the most part, there was roughly the same number of compilations that happened per week. Running an ANOVA calculation on each week of Figure 8.9 finds no week to have a significant difference in the number of runs by previous modality (week 4 comes the closest with $F(2, 77) = 1.79$, $p = 0.17$). This chart suggests that the introductory modality had relatively little effect on how often students attempted to run their programs.

The chart shows some unexpected trends comparing week to week, like the dip in week 6 followed by the spike in week 7. This is in part due to one of the challenges of doing research in schools: the unpredictability of the school calendar and the number of days that students are not in the classroom or are not working on what one might expect. Week 6 of the study was Thanksgiving week, so those numbers are lower than they otherwise might be because students were only in class 3 days that week. Students also missed two days of classes in week 4 and one day in week 5. This explains some of the fluctuations. The same data from the figure above is presented again in Figure 8.10, this time showing the average number of runs per student per day in class. The relationship between conditions is the same across these two figures, but this updated figure below gives a better sense of the activity by week.

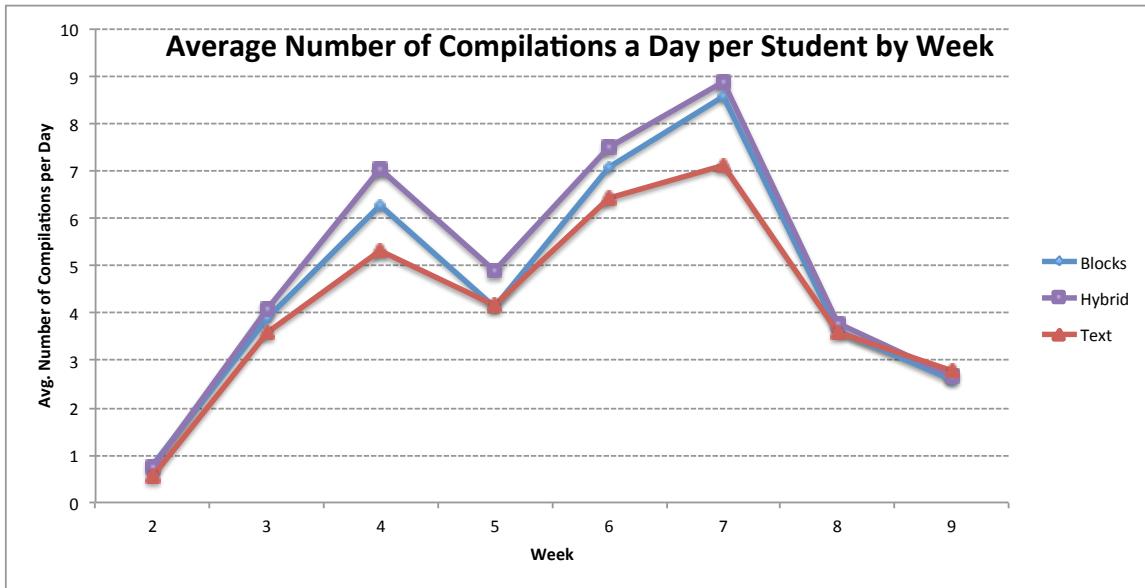


Figure 8.10. The average number of javac calls per student per day grouped by week.

The above Figure still shows spikes (like weeks 4 and 7) and dips (like in weeks 5, 8, and 9), although now, these are explained by how the teacher chose to spend class time as opposed to external factors (like school holidays). For example, in week 7, students were introduced to the `char` variable type through an assignment where they were asked to write a short program, then try and run it with different values to see what would happen. As a result, there was a spike in week 7 as these types of assignments (that would have student call `javac` over and over again) were not the norm. Other assignment related trends are discussed later in this section.

So far, the charts shown in this section have included all calls to `javac`, grouping together both successful compilations as well as calls to the compiler that produced errors. We now tease apart these two outcomes to see if students were differentially successful or error prone based on the modality used in the introductory portion of the course. Figure 8.11 uses the same data as the two previous charts, but now only includes successful compilations.

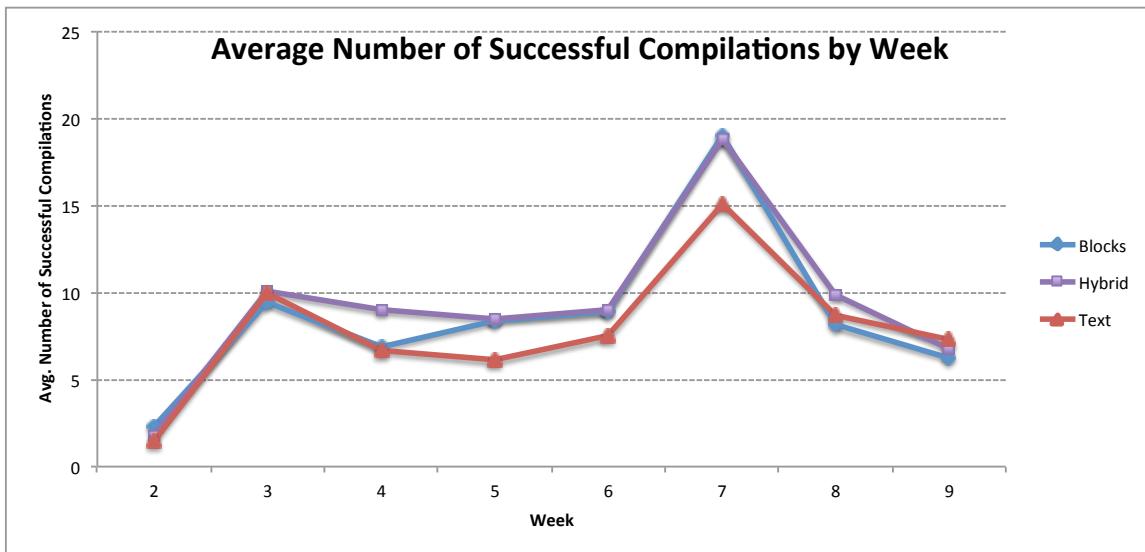


Figure 8.11. Average number of successful compilations by condition.

The pattern in this chart largely matches the data from the previous charts. The Text condition frequently had the lowest average number of successful compilations per student, while students who had worked in the Hybrid version of Pencil.cc showed the highest number of successful compilations. Since the students were all working on the same assignments in the same programming environment, the slight differences in successful runs does suggest something about programming practice. The higher frequency of the Block and Hybrid students shows these students having higher levels of success in writing syntactically valid programs, although the magnitude of this difference is relatively small, and seems to start to fade by the end of the study. There are a number of possible explanations for this. One explanation could be that students in these conditions were more likely to compile their program at intermediate steps along the way. In other words, as they were writing their program, they would check to see if the portion they had written was correct before continuing with the next portion. A second explanation could be that students in the Blocks and Hybrid condition ran their programs more frequently once they were completed (i.e. they finished their programs then ran it over-and-over

again), thus inflating their successful compilation counts. A final explanation is as simple as the fact that students in the Text condition did not call `javac` as often as students in the other two classes. In the following paragraphs, we explore these different potential explanations.

To identify the cause of these trends, we can look at consecutive successful `javac` calls for the students participating in the study and focus on the size of the changes students made between them. This analysis only looks at successful runs and does not consider how many failed `javac` calls might have taken place between them. The intention with this investigation is to understand the incremental nature of the programming approach taken by students from the three conditions and to see if different development practices can be found. To measure the distance between two programs, we use the Levenshtein distance between the texts of the two programs. Levenshtein distance captures the minimum number of single-character edits (i.e. insertions, deletions or substitutions) required to change one string into the other. Table 8.2 shows the results of this analysis. The columns capture the size of the Levenshtein distance between two consecutive successful programs, while the cells show the average number of compiles of that distance per student. The lower the number, the less often a program with that distance from the previous successful compilation was run by that student. For example, the left-most column of numbers shows that, on average, students in the Blocks condition complied a program that was identical to the last program they compiled 7.00 times over the course of the 10 weeks, while the Hybrid condition recompiled programs an average of 7.40 times and the Text condition only did this 6.16 times.

Table 8.2. The frequency of successful compilations with a given Levenshtein distance from the last successful compilation of the same program.

Levenshtein Distance								
0	1	2	3	4	5 - 10	11 - 25	26 - 100	> 100

Blocks	7.00	3.37	5.70	1.33	2.37	4.30	3.52	6.56	3.33
Hybrid	7.40	3.68	5.88	1.84	2.60	4.64	4.72	6.48	3.76
Text	6.16	3.00	5.58	1.23	2.13	3.77	3.48	5.87	2.55

Table 8.2 shows that students in the Text condition made fewer small changes to their programs, fewer large changes to their program, and also re-ran their programs without making any changes less often than the two conditions. In other words, the Text condition had fewer successful runs than the other two conditions. The data does not show that these fewer compilations are a result of them making larger sets of changes between runs, thus ruling out that explanation for the fewer number of successful runs. There is some evidence for the difference in number of runs coming from the students in the Text condition rerunning their programs less often and making fewer calls after modifying only a few characters in their program, but these differences are not so large as to convincingly explain the larger trend. These numbers tell the same story as Figures (Figure 8.9) and (Figure 8.10), both of which show the Text group to have called `javac` least often. The most likely explanation for this is that students who used the text-only modality in the first five weeks of the course are slower to author programs in Java, but this study did not collect keystroke data, which is the data source needed to provide strong evidence for this outcome.

Figure 8.9, which shows all compilations, includes both successful compilations, as well as compilations that resulted in errors. Figure 8.12 below shows the average number of `javac` calls the produced an error broken down by week. All of the errors captured at this point were compile time errors, meaning the program violated some syntactical requirement of the Java language (e.g. a missing semicolon or misspelled keyword). This is in contrast to runtime errors, which only emerge once the program is run. An analysis was done looking for runtime errors in the programs collected, but no runtime errors were detected. This is largely due to the types of

programs written, which did not include constructs that are the most frequent culprits of runtime errors (e.g. array indexing and divide-by-zero calculations).

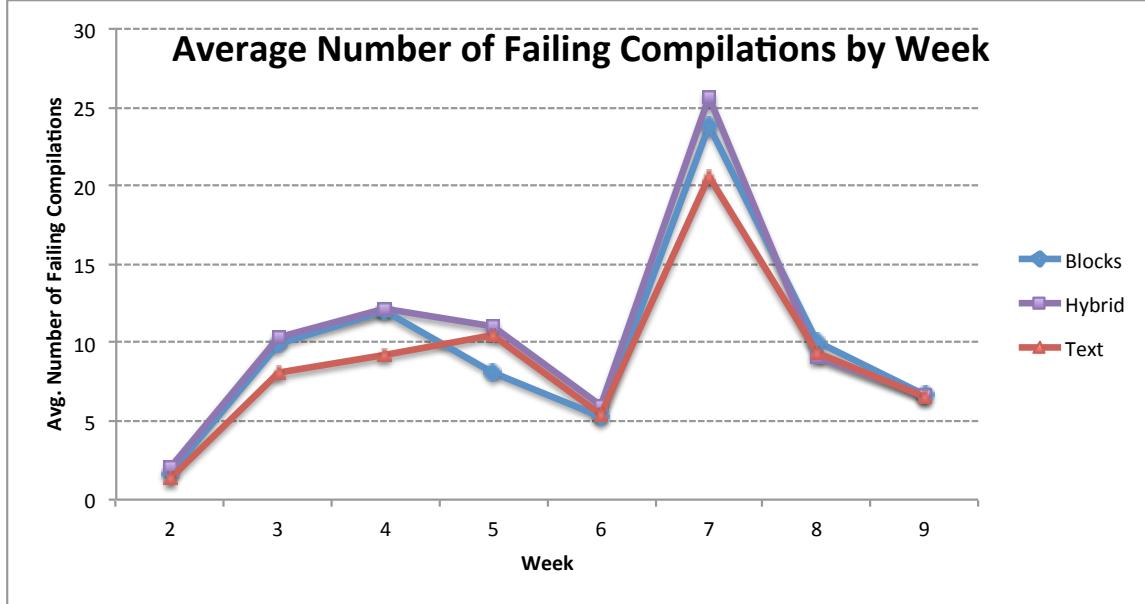


Figure 8.12. The average number of compilation calls that resulted in at least one error, grouped by condition and week.

The pattern in this figure roughly matches that of the previous two, with the Text condition again having the lowest numbers, but we see three weeks where the conditions split, (weeks 4, 5 and 7). Whereas with the successful compilations we can gain insight into the programming practices by looking at the nature of the changes made between runs, with the errors we have additional information in the form of the type of error that was detected. This information can be used to further understand and explain this graph. The next section starts to explore the patterns of errors observed in this data.

Before digging into the nature of the errors, the last chart we present in this section, Figure 8.13, shows the ratio of successful to unsuccessful javac calls over the ten weeks of the Java portion of this study. This chart controls for the overall number of compilations so rules out that explanation for the less successful number of compilations in the Text condition.

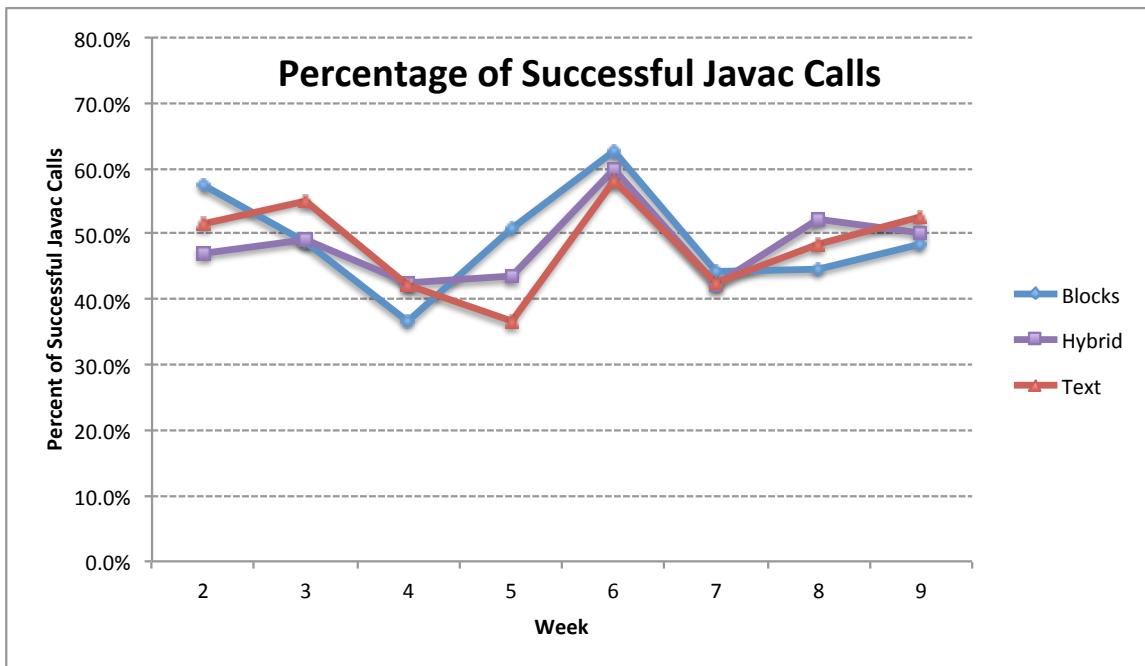


Figure 8.13. The percentage of syntactically correct calls to `javac` by condition.

One might expect the lines in this figure to have a positive slope, denoting that students are progressively improving (or at least writing more correct code) as their experience grows. However, this positive trend is not always present due to the fact that the students are constantly learning new material. While many weeks see the three conditions follow similar trajectories, there are some weeks where the three conditions deviate from each other, notably, weeks 5 and 8, while other weeks see outliers (like weeks 3 and 4).

To explain the stratification that happened in weeks 5, and 8, we look to the weekly curriculum to see if there is a plausible explanation based on the activities students are working on. In week 5, students work on two projects that ask students to write short Java programs to display images. In both cases, students find a URL online, then display the image residing at that URL in a visual Java container (called a `JOptionPane`). This is the first time students are doing anything with images since starting their work in Java. In fact, this assignment is similar to

a commonly used feature of Pencil.cc, namely, that students can have their turtles ‘wear’ an image by passing a URL into the `wear` command. This assignment is the closest to any of the turtle geometry or visual assignments given in Pencil.cc. Given this fact, it seems reasonable to expect that students who fared better in Pencil.cc may perform better on this assignment, which is indeed what happened.

Week 8 again sees students from the three conditions experiencing different levels of success with respect to successful compilation rates. This time, however, the Blocks condition is the least successful of the three conditions, with the Hybrid condition having the highest success and the Text condition being in the middle. Week 8 saw the introduction of input into Java programs using the `Scanner` class. Interestingly, whereas week 5 had the Blocks students excel when working on a program related to displaying images, in week 8 they appear to struggle to write syntactically correct programs when the assignment is based on taking input from the user. All students had read in data from a user as it was part of the Pencil.cc curriculum but the syntax they used was very different than what was required in this part of the course. A possible explanation for the poorer performance of students in the Blocks condition is that students from the Text and Hybrid had a more comfort writing text-based programs to handle user input, and thus more quickly were able to author syntactically correct programs. An investigation into the outliers previously mentioned in weeks 3 and 4 revealed no clear link between the assignment and the frequency of errors. Having looked at aggregated and temporal error patterns for this portion of the study, in the next section, attention turns away from whether an error occurred and towards what type of error it was, in hopes of linking features of the modality to early Java error and programming patterns.

Types and Frequencies of Java Errors

The last figure shown in the previous section charted the frequency of students' errors as they advanced in Java. This gives us some sense of how students were faring in Java, but does not provide insight into the types of errors they were making. In this section, we dig further into this information, trying to understand the nature of errors that were being made to see if any patterns could be attributed back to the introductory modality students used. As a reminder, every time a student makes a call to `javac` (the Java compilation command) the logging system put in place makes a record for the call that includes who the student was, what he or she typed in (usually the name of the program, but also any arguments provided), the contents of the program, as well as all errors reported by the Java compiler. The first step in this process is to try and normalize and categorize each error captured by the logging system. The plurality of compilation errors produced by the Java compiler has been documented as both a source of difficulty for novices (Nienaltowski et al., 2008; Traver, 2010) as well as an opportunity for improving introductory programming environments (T. Flowers, Carver, & Jackson, 2004; Hristova, Misra, Rutter, & Mercuri, 2003).

Before looking at frequency of specific types of errors, we first look at some overall frequency numbers, trying to understand if there are differences in total number of failing calls to `javac` by student, the average number of errors per student (since there can be more than one error per program), and the number of errors per compilation by student. Table 8.3 shows these high-level descriptive patterns.

Table 8.3. High-level descriptive patterns of failing compilations and errors.

	Failing <code>javac</code> calls per student	Compilation errors per student	Compilation errors per failing <code>javac</code> call
Blocks	75.11	165.78	2.23
Hybrid	80.04	212.04	2.5
Text	69.55	164.26	2.21

Looking at this table, we see a pattern similar to that shown in the previous section. The Hybrid condition had the highest average number of javac calls that returned a compilation error per student. Similarly, the Hybrid condition had the most number of errors per student over the course of the ten weeks. The far right column shows the average number of errors per javac call. Again, we see the students who spent the first five weeks of the course working in the Hybrid version of Pencil.cc had the most number of compilation errors per program. This chart shows relatively little difference between Blocks and Text, but shows students in the Hybrid condition to be relative outliers. These numbers match the figures shown in the previous section where the Hybrid condition was often plotted above the lines representing the Text and Blocks conditions.

We now shift from total number of errors to the frequency of different types of errors. The collection and analysis of Java error messages is not without its challenges. Due to the process by which `javac` compiles programs, the compiler often does not (and at times cannot) provide meaningful error messages to the programmer. For instance, a missing ‘;’ could be described by the error message “expected ‘;’ on line 11” or by the rather generic error message “not a statement”. In addition, many error messages stated by the compiler are class specific (e.g. “Class names, ‘VarRefConcat’, are only accepted if annotation processing is explicitly requested”). In order to make the analysis more meaningful, errors were grouped into broadly specified error types. For example, the class name error above was classified as an “Incorrect javac Call” as that is the most common cause of that particular error. The logic used to conduct this categorization can be found in Appendix F. Figure 8.14 shows the 10 most frequently found errors collected by

compilation, grouped by condition⁴⁵. The values in this chart are reported on a per-compilation basis to control for how often students chose to compile as well as the fact that the three conditions did not have the same number of students.

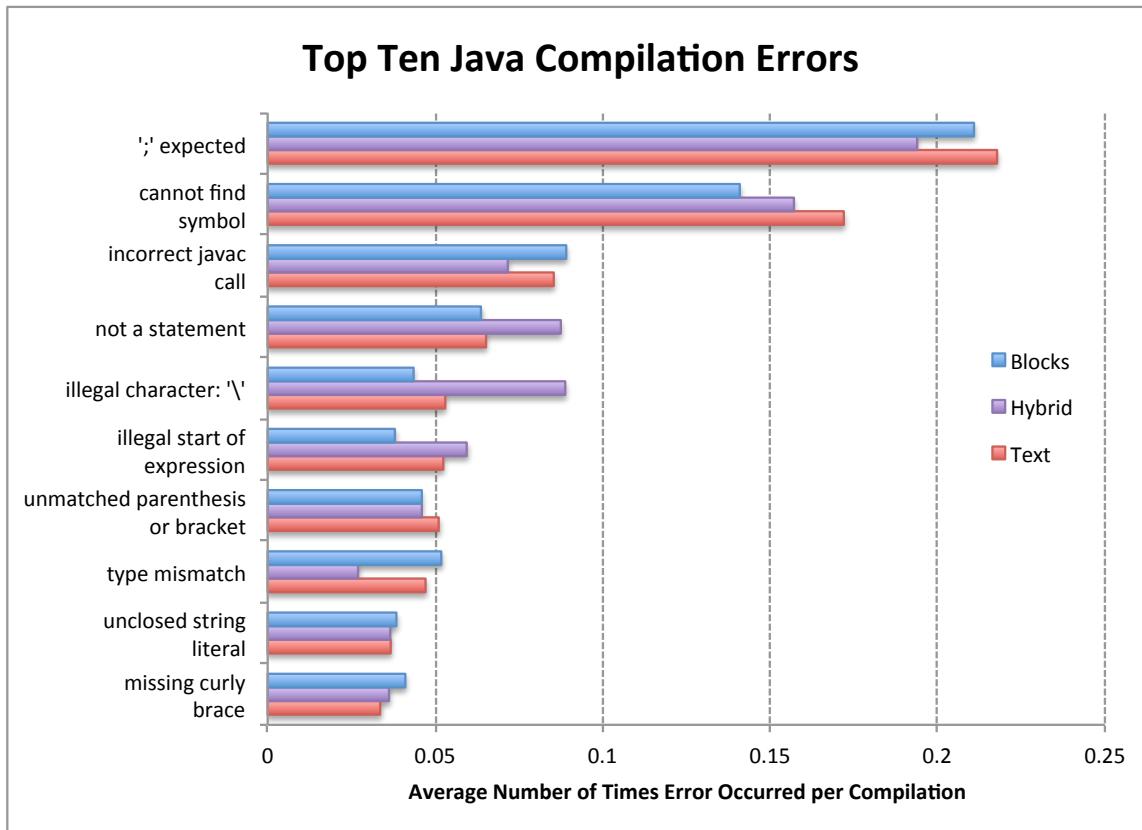


Figure 8.14. The ten most frequently encountered Java errors, grouped by condition.

There are a few things to notice about this chart. Running an ANOVA calculation on each error, looking for statistically significant differences between the groups finds that none of the errors rise to the $p < .05$ level of significance. The two errors that come the closest are the two categories where the Hybrid condition is the outlier: “not a statement” ($F(2, 82) =$

⁴⁵ Note: this figure includes the top 10 errors, each of which occurred over 500 times. This cutoff was chosen because there was natural break in the data between this error and the next most common error, which occurred over 100 times less often.

2.26, $p = .11$) and “illegal character: ‘\’” ($F(2, 82) = 2.54, p = .08$). These two categories will be discussed later in this section.

The first thing that stands out about this figure is how often the first two errors were encountered relative to every other error. The most common error was: “‘;’ expected”, which is seen when students forget to end a statement with a semi-colon, a syntactic requirement of Java. The second most common error: “cannot find symbol”, occurs when students try and use a variable before it has been defined. Neither of these errors are possible in Pencil.cc, as semi-colon terminators are not required and variables do not need to be defined before they are used (at least in most cases). While this Pencil.cc explanation seems reasonable, it is important to note that regardless of prior programming experience, novices frequently encounter these two errors. The literature shows these two mistakes to be very common, in fact, both Jadud (2005) and Jackson et al. (2004) identified these two mistakes as the most frequently encountered in their data. In this way, the findings of this analysis replicate findings documented elsewhere in the literature.

Looking across the ten errors, we see that half of the ten most frequently occurring errors were seen least often by students in the Blocks condition. A possible explanation for this outcome is that, because Java code is so unlike the blocks-based modality used by the students in the first five weeks, they were more attentive to the specific syntax they were being forced to learn. Text and Hybrid students on the other hand, were already accustomed to manipulating a text-based language, but had used an entirely different syntax, so they may have assumed a higher level of similarity across the text-based languages. Another explanation for the fewer number of errors made by students from the Blocks condition relies on the data analyzed in Chapter 6, which found the Blocks-based students performed the best on the Mid assessments.

The explanation thus becomes, students in the Blocks condition learned the most about programming in the first five weeks of the course, so had the least amount of conceptual difficulty in the early weeks, and thus, had the fewest errors. This explanation does not completely hold up, as the differences in the performance on the content assessment eroded over the course of the ten weeks of working in Java, but that is not reflected in this data.

As previously mentioned, the two categories that showed the largest difference between the introductory modalities both had the Hybrid condition as an outlier. These two errors are: “not a statement” and “illegal character \”. The “illegal character \” error was encountered frequently because early assignments asked students to include escape characters in their output text (which includes tabs, quotes, and backslashes). In Java, the “\” character is used to denote an escaped sequence. This error often occurred in cases where students wanted to output text, but were missing the enclosing quotes or tried to escape a character not inside a string to be output. In blocks-based modalities, this type of error rarely occurs as the keywords themselves are prefabricated (i.e. a novice cannot escape a keyword) and because strings are visually denoted inside slots and quotes are not needed. A possible explanation for the students coming from a Hybrid modality encountering this error is that they were used to seeing and working with commands that have the opening and closing quotes provided. Also they did not have the higher level of attention to detail when working in Java that seems to accompany the students from the Blocks condition. The other error frequently seen, “not a statement”, is usually the result of typing only part of a statement, leaving it incomplete and meaningless. One explanation for this is that, at the start of the year, Hybrid students did not have the predefined commands and sequences that could be dragged into their programs but instead gained experience in the text-based modality, so are more intrepid and less

cautious. In this way students had the familiarity with the modality but previously had more scaffolds in place to help them compose their programs. The combination of fewer scaffolds with less caution may explain these outcomes. Other errors in this list could be encountered as a result of students having this orientation of confidence without scaffolds, including “unclosed string literal” and “illegal start of expression”, two other errors most often encountered by students from the Hybrid condition. If these types of mistakes are indeed an outcome from working in the hybrid interface for the reasons discussed above, this would be a place where this specific interface is producing the worst-of-both-worlds, rather than the opposite, which was the intended outcome.

A final important thing to point out about this figure is that all of the errors in the chart are tied to the contents of the programs students’ wrote, with the exception of one. The “Incorrect javac call” error is a category we created to capture the various errors associated with issues related to calling javac. These errors include mistyping the program file name, trying to compile a file that does not exist, or giving (or forgetting) arguments that do not match what the program expected. There is no difference in the prevalence of this error by introductory modality, which is not surprising given that no students were asked to do this type of compilation and program calling in Pencil.cc. Collectively, this analysis by error reveals some minor trends towards types of error by previous modality experience, but a clear link between types of errors and introductory modality did not emerge. These findings will be further discussed in the context of everything else presented in the next section of this chapter.

Java Programs Discussion

In previous chapter, programs written by students were used as a data source to try and understand if and how the modality students had used during the first five weeks of the school year influenced their early Java programming practices. The first investigation looked at frequency of compiling programs and the success rate of those `javac` calls. Overall, no obvious differences were observed. Students across the three groups showed similar programming patterns with respect to frequency of failures and successes in their calls to compile their programs, comparable patterns in the size of changes between successful calls to `javac`, and encountered the same types of compilation errors with roughly the same frequency.

Even though clear differences did not emerge, there were some trends that suggest some difference did exist. Of the three conditions, students who had spent their first five weeks working in a text-based modality had the fewest number of calls to compile their programs, despite the same amount of time on task. This suggests that students in the Text condition took more time between compiles, resulting in fewer overall calls to `javac` relative to their peers. Other alternatives were explored (like Text students made larger edits between runs, or that students in the Text condition needed to compile their programs less frequently because they were more efficient and finished their programs sooner), but the data did not support these alternative explanations.

In looking at the ratio of successful to unsuccessful calls to `javac`, there were a few points where the three conditions started to vary. When assignments involved images, students with prior experience working in a fully blocks-based modality excelled, while those same students seemed to struggle on assignments related to reading in user input. The plausible explanations given for these two findings drew on different characteristics of the programming activity. For the image assignment, the visual nature of the assignment seemed the most plausible

explanation, while the variance for the input assignment was part of an explanation that was also used to explain some of the error patterns observed later. When looking at patterns in the types of errors observed, there were a few places where the Hybrid condition was an outlier in terms of seeing specific errors more frequently and, in the Blocks condition, a larger pattern of fewer errors. The explanation given for these has to do with students in the Blocks condition attending more closely to the syntactic details of Java due to how different the modality was. The students in the Hybrid condition were the least well suited for Java due to their prior experience working in a text-based modality with an introductory version of Java that had scaffolds. While these explanations seem reasonable, there is only weak support for them in this data, so for now they remain conjectures, with the hope of returning to explore them in more detail in future work.

The final discussion point from this section is less about the research questions and the modalities being investigated and more about the challenges associated with doing this work in classrooms. Trying to make sense of the data at the highest level gathered over the course of weeks in the classroom cannot be interpreted without considering the complex milieu of the classroom and the school infrastructure in which it resides. These issues can take the form of school holidays, differences in the types of assignments being given, and the pedagogical choices the teacher makes and the fact they can change from day-to-day and week-to-week. This, in conjunction with other challenges related to studying modality previously discussed (like the difficulty of separating modality from language and assignments) make it difficult to find clean and clear findings, but at the same time, engaging in such work is essential for answering the types of question being pursued in this dissertation in ecologically valid ways.

Discussion

This chapter began by looking at students' perceptions of whether or not the introductory environment was useful for the eventual transition to Java and if so, what features were useful for that transition. The chapter then continued by looking at attitudinal shifts of students that occurred during their first 10 weeks of learning to program in Java. Finally, the chapter presented data looking at student successes in their early Java programming, using logs of Java programs to gain insight into if and how the modality used in the introductory environment helped students at the outset of their Java experience. Collectively, these analyses illuminate different facets of the larger question of how the modality used in an introductory programming environment does or does not prepare learners for the transition to a professional text-based programming language. A brief discussion for each of the three analyses was provided within each section, here, the three larger trends from these three analyses are presented and connections are made across them trying to pull together these three avenues of inquiry to tell the larger story on learners transitioning to Java. As the goal of this dissertation is to understand differences by modality, that is where we choose to focus in this section.

Across the three analyses there were as many facets of learning to program in Java that seemed to be affected by the introductory modality students had used as were places where modality seemed to make little difference. Asking students to reflect on what they found to be useful from the introductory environment produced largely uniform results across the three conditions. The exception being the emergence of students who used the blocks-only modality highlighting how their time using the visual programming representation helped them see the importance of Sequence and Order in learning to program. The investigation of programming practices based on the computational logs collected yielded a similar lack of difference across the three conditions. While some trends emerged, like the fact that students from the Text condition

had fewer calls to `javac` and that there were a number of errors more frequently encountered by students from the Hybrid condition, both of these findings are relatively minor compared to the larger potential trends that could have emerged. This suggests that the introductory modality used plays either no role or only a relatively minor role in shaping programming practices after transitioning to a professional programming language, or at least, modality does not differentially affect emerging practices.

Where differences based on modality do appear, the largest differential impact is in how attitudes change during the first ten weeks of working in Java. Interestingly, it appears that much of the influence of modality on attitudes when working in Java was shaped by the attitudes students held after the five weeks working in the introductory tool. As was shown in Chapter 5, the Blocks condition largely had the most positive effect on various dimensions of students' attitudes. What this chapter found, is that student attitudes showed a negative slope for every dimension that was captured. At the same time, the students from the Blocks condition saw a relative improvement in three of the four attitudinal dimensions (with the fourth showing a decrease of only .03 points on a ten-point scale). One way to read these trends is that the different modalities used in the introductory portion had the effect of fanning out the attitudes (Blocks improved while Text decreased with Hybrid living in the middle) and then the shift to Java moves students back to being closer to the attitudes held at the outset of the study. This makes some sense as students coming into the course held various preconceived notions about what it meant to program in Java and had incoming dispositions about the activity.

While the framing of the discussion and analysis has focused on how the modality from the introductory portion of the study informed students' experience in Java, the shift between the two phases of the study included more than just changing the programming language. In the

curriculum designed for the first five weeks of the study, care was taken to give assignments that would result in all of the students producing similar programs as well as opportunities for students to be creative and expressive, creating unique and personally meaningful programs. In the Java portion of the class, for a variety of reasons, the assignment become much more formulaic and standardized across the class, leaving less room for creativity and expression. Similarly, the pedagogy in the class shifted to a model that relied more heavily on direct instruction (either through demonstration or following examples in the textbook). This was in contrast to the exploratory, self-directed approach used during the introductory portion. These shifts in the culture of the classroom were related to the shift in modality (more heavily scaffolded environment can support different types of activities), but also had to do with the teacher taking back the reins of curriculum design from the researcher. At the same time the cultural shift happened, so to was there a drastic shift in the nature of the programming environment being used. Whereas the Pencil.cc environment provided a number of built-in scaffolds independent of modality, the text editor used for the Java portion of the class was intentionally spartan, providing no coding support to the learner. While this shift was shared for all students it, along with the cultural shift, provide two dimensions that confound the study design, which was trying to isolate language as the major difference between the two phases of the study. While this does not undermine any of the data, analysis, or findings presented in this chapter, it does suggest that more studies of a similar design need to be conducted before the findings in this chapter become robust enough to be responsibly applied to diverse classrooms and learning contexts.

Conclusion

This chapter is the fourth and final analysis chapter of this dissertation and fills in the last remain big piece of the analytic approach taken in understanding the role of modality on learning. The three analyses presented in this chapter took different approaches towards understanding how the modality a learner uses in an introductory course impacts their experience and approach to programming in a professional language. In taking these different approaches to understand this question, this chapter shows how and where modality informs students' experiences in learning to program with Java and places where little residue from the time spent working with different introductory modalities was found. These findings have potentially large implications with respect to the suitability of various introductory tools based on the larger goal of the learner and the educator. These implications, along with a longer summative discussion to capture the full breadth of the findings will be presented in the next and concluding chapter of this dissertation.

9. Discussion and Conclusion

The final chapter of this dissertation summarizes the work undertaken and recapitulates the findings presented throughout the document, providing a larger framing for the contributions made and the implications of what was learned. Over the previous four analysis chapters, different aspects of the relationship between modality and learning to program were investigated. In this summative chapter, I link these analyses to provide clear answers to the stated research questions pursued in this work. The chapter begins by restating the research questions and briefly describing the course pursued to answer them. The summary of the findings follows. It starts with a comparison of the findings from the textual and blocks-based conditions of the study and then brings the hybrid blocks/text condition into the story. The implications of this work follow and reflect on what these findings mean for pedagogy, classroom curricula, and learning contexts. Next is a section that serves as a discussion for the major focus of this work: the relationship between modality and learning to program. Finally, the limitations of the current study are discussed with care taken to discuss potential future work to address each of the limitations identified.

Review of the Program of Research

This study sought to understand the relationship between modality and learning to program. The concept of modality is intended to capture both the design of the representation used as well as the types of interactions made possible by that design. In this way, modality is not a characteristic of a representational system in isolation, but instead captures the larger sphere of representation-with-actor. The domain of interest is computer science, specifically, novices learning to program. This area is particularly well suited for pursuing questions of

modality because of the diverse set of modalities used in introductory programming contexts and the interactive nature of working with programming languages and environments. Stated concretely, this dissertation pursued three sets of interrelated research questions seeking to understand the impact of students' learning in blocks-based, text-based, or hybrid blocks/text modalities. The three sets of research questions pursued in this work are as follows:

1. (a) For text-based, blocks-based, and hybrid blocks/text programming tools, what is the relationship between the programming modality used and learners' perceptions of programming with respect to confidence, authenticity, enjoyment, and to their broader attitudes towards the field of computer science? (b) How does the representational infrastructure used affect learners' emerging understandings of programming concepts?
c) What programming practices do learners develop when working in each of these three modalities? And, for each of these questions, how do the answers differ across blocks-based, text-based, and hybrid blocks/text environments?
2. (a) How do understandings and practices developed while working in different introductory programming modalities support or hinder the transition to conventional text-based programming languages? (b) How does a learner's understanding of and attitudes towards programming change as learners shift from introductory environments to more widely used, professional programming languages? How is this different among text-based, blocks-based and hybrid blocks/text introductory modalities?
3. Can we design hybrid introductory programming environments that blend features of blocks-based and text-based programming that effectively introduce novices to programming and computer science more broadly? How does such an environment

perform relative to blocks-based and text-based programming tools with respect to conceptual understanding, development of productive programming practices, and attitudinal, motivational, and engagement outcomes for learners?

To answer these questions, a quasi-experimental, mixed-methods study design was developed and executed. Three isomorphic programming environments were developed. The environments used the same programming language and had the same set of capabilities, but differed in the modality used: one environment was fully text-based, one was fully blocks-based, and a third presented users with a text canvas but also provided a blocks-based palette, thus supporting the addition of commands through a drag-and-drop mechanism as well as character-by-character editing.

The study was conducted in three high school programming classes, run in the same room by the same teacher during three different class periods. Each class worked through the same curriculum using just one of the three modalities. The study began on the first day of school and lasted 15 weeks, the first five weeks were spent using the introductory programming environments, followed by ten weeks of following the students as they transitioned to Java, a professional text-based programming language. The strength of the study design is that it controls for many (but not all) of the confounding factors that make comparative classroom studies and studies of programming languages and modality so difficult. The study controls for teacher effects and curricular effects because they were held constant across the three conditions. Students were drawn from the same student body, which helps to control for larger school culture effects. To try and isolate modality from other aspects of introductory programming environments, the three introductory environments were built on the same platform, used the

same underlying programming language (CoffeeScript), provided the same capabilities in the same runtime, and differed only in how the commands were presented and edited. In other words, the environments were isomorphic with modality being the only difference. The three modalities are shown below in Figure 9.1. With these tools and this study design, we are able to answer the stated research questions in a rigorous and compelling way.

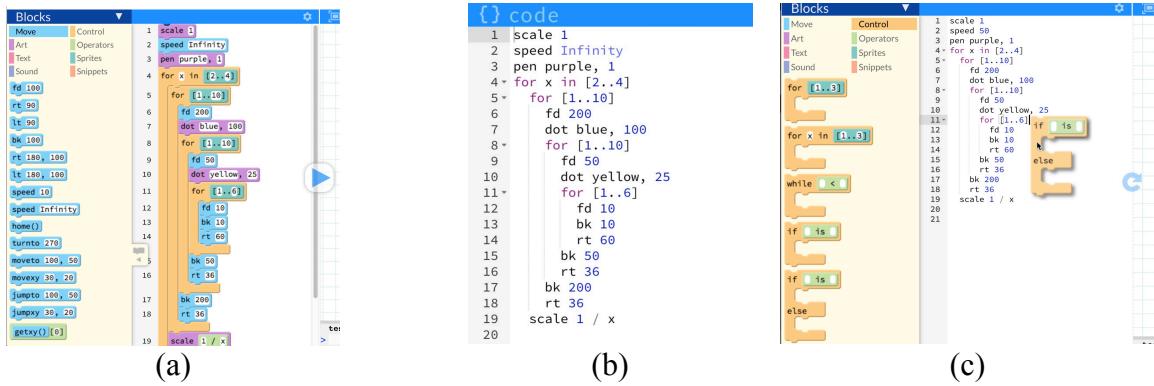


Figure 9.1. The Blocks (a), Text (b), and Hybrid (c) environments used in the study.

Summary of Findings

This section serves as a high-level review of the finding from the four analysis chapters and, for the first time, draws conclusions across the full set of evidence presented. The first portion of this section compares the Blocks and Text conditions to make claims about modality's influence on attitudes, conceptual outcomes, and programming practice, as well as if and how those differences inform and affect the transition to Java. The second half of this section gives a similar treatment to the Hybrid condition, situating it relative to the two modalities from which its design was drawn.

Comparing Blocks and Text Modalities

Blocks versus Text: The First Five Weeks

In Chapter 4, students' attitudes towards and perceptions of programming were investigated. The data found that, while working in the introductory modality, students using the blocks-based modality reported higher levels of enjoyment and a greater interest in taking future computer science classes than students who worked in the Text condition. Additionally, these students saw their confidence increase more than students in the Text modality, but this is partially explained by students starting with a lower level of confidence in the Blocks condition, resulting in students in the two classes ended with roughly the same levels of confidence after five weeks. No difference in enjoyment of programming was found between the Blocks and Text students. All of the numbers reported were in the positive half of the response range, meaning even with the differences, all students had self-reported positive programming experiences. Where the Text condition was found to be more successful than the Blocks condition at the end of five weeks was in students' perceptions of the authenticity of the introductory programming experience. Students' responses from the Text condition showed that those students found what they were doing to be more similar to what real programmers do and viewed it as more useful in preparing them for their upcoming transition to Java.

Looking at the differences in student performance on the Commutative Assessment at the completion of the introductory curriculum revealed a similar pattern. After controlling for prior knowledge, students in the Blocks condition scored significantly higher on the content assessment than students in the Text condition. The Blocks condition scored higher on questions across all three modalities (Pencil.cc Text modality, Pencil.cc Blocks modality, and Snap! Blocks). They also scored higher on all six of the content areas covered by the assessment (variables, conditional logic, iterative logic, functions, comprehension, and algorithms). Students were also asked to report how easy they found it to write programs that included the four

concepts that were the focus of the curriculum (variables, conditional logic, iterative logic, and functions). The Blocks condition reported that all four of the concepts were easier to use than students in the Text condition. Along with these aggregate outcomes, a second analysis looked at students' responses to short answer questions asking for written explanations of the meaning and use of the main concepts of the curriculum. This analysis revealed some differences between the conditions, but, overall, responses were found to be more similar than different. A difference that did emerge was that students in the Blocks condition were more likely to see instances of similar concepts as distinct. For example, rather than speaking about concepts in general, they viewed `if` and `if/else` statements as distinct and `for` and `while` loops as independent. This perspective seems linked to the blocks palette and the choice of what to display as a distinct block and how and where things are grouped.

Despite these differences, the students in each condition seemed similar in their conceptualization of computer science ideas, with no systematic difference emerging when they talked about what conditional logic does or how and when functions are used. Similarly, we did not find a difference in frequency or types of misconception identified in students' written descriptions of the concepts between the Blocks and Text conditions. This suggests that the Blocks and Text modalities play a relatively small role in shaping conceptual understanding of programming concepts.

This dissertation also included an analysis of the practices students developed in writing their programs and the types of programs they wrote. Vignettes were used to understand programming practices, with clear differences emerging between the two modalities. The Blocks modality requires students to drag-and-drop commands onto the canvas in order to assemble their programs, while the Text interface makes students use the keyboard and type in commands

character-by-character. This resulted in students using a different composition mechanism and produced a number of side effects. For example, the Text condition encountered syntax errors more frequently than the blocks condition due to the blocks modality enforcing syntax correctness. Similarly, the Blocks condition used the Blocks palette as an external memory aid to determine what was possible and used the hover-over tooltip for assistance in figuring out what commands were used for. In contrast, since the Text modality did not provide this information, students had to rely more on the Quick Reference menu to figure out what was possible in the language and the correct syntax to use. A second difference observed was the ease with which students in the Blocks condition could incrementally build up commands by dragging and dropping components sequentially while ignoring the components' position in the final command. In other words, students could build up complex commands left-to-right, right-to-left, or from the inside out, in a way that was possible, but unintuitive and not observed in the Text modality. In this way, the Blocks modality provides a type of authoring pluralism that is less well-supported in the Text modality (at least when used in a conventional text editor). Looking at the full set of programs collected by the automated logging system used in the study showed the students in the Text condition produced shorter programs, ran their programs more often, and had more quick succession runs relative to the students in the Blocks condition. These patterns can be explained by what was observed in the vignette, which showed how students in the Text condition encountered syntax errors and had to run their programs more often, and in quick succession, as part of their debugging process. This shows that modality does change how students author programs and that these different authorship mechanisms have consequences beyond the specific mechanics of writing the program.

Taken together, these data reveal important differences between the Blocks and Text modalities. In most dimensions, attitudinal and perceptual outcomes were better for the Blocks condition during the time spent using the introductory tools. Likewise, in terms of conceptual learning, the data show the Blocks-based condition to be more effective for teaching high school students programming basics within the constraints of the study. These constraints, such as the relative short duration of the curriculum, its fast pace, and the use of the visual programming execution environment, mean that we cannot make larger claims about how robust this finding is for other modalities, curricula, age groups, etc. This point will be addressed later in the limitations and future work section of this chapter. This dissertation also showed how the different mechanism for constructing a program (typing versus dragging-and-dropping) affected other aspects of the programming experience, including help seeking, frequency of compilations, and the length of the programs. It is also likely that there is an interaction between the practices that formed and students' attitudes and conceptual outcomes. For example, one explanation that fits the data is, since the Blocks students could ignore the details of syntax, they were better able to focus on the conceptual aspects of programming, i.e. what is the concept and how can I use it? Additionally, spending less time on errors and debugging allowed them to dig deeper into the use and behaviors of constructs and also affected their attitudes with respect to confidence and interest in the field, even if it came at the expense of perceived authenticity.

Blocks versus Text: Transitioning to Java

The second set of research questions ask if and how attitudes, practices and concepts learned in a given modality in an introductory environment carry over to learning a text-based professional language. Looking at students' attitudes during the second phase of the study, the opposite trend from the changes in made in the first five weeks can be seen. Students coming

from the Text condition see their confidence, enjoyment, and interest in computer science rise, while the aggregate scores for the Blocks condition decrease. Even with the decreasing trajectory, students in the Blocks condition still attained high aggregate scores for interest and perceived difficulty of programming. Additionally, compared to perceptions at the midpoint of the study, students in both the Blocks and Text conditions saw their time in the introductory modalities as less useful for learning Java and less authentic relative to professional programming practices after the ten weeks of working in Java.

Looking at conceptual outcomes at the end of the 15-week study, we see the gap that emerged in performance between the Blocks and Text conditions close. At the conclusion of the study, the two conditions showed no difference on the content assessment. The Text condition's score between the Mid and Post administrations improved, while the scores for students in the Blocks condition remained at roughly the same level. There were a few subtle differences observed in the programming practices developed by learners from the two conditions, but overall the programming practices were relatively indistinguishable from each other. Students from the two conditions showed roughly similar success and failure rates for the compilation calls, produced comparable programs, and encountered the same types of errors with roughly the same frequency. This data suggests there is little lasting impacting on programming practices between the introductory modality and working in a professional, text-based programming language.

Blocks versus Text: Summary

Taken together, these data show students having rather different experiences while using the different modalities in the introductory portion of the course, but that those differences eroded as the students transitioned to Java. Attitudinally, even though students ended up in

roughly the same place, the data show students taking a rather different path to get there. The Blocks students saw gains in the introductory portion, while the Text condition saw gains after transitioning to Java, suggesting that part of the benefit of the text-based introductory environment was not experienced until after it was left behind. A similar pattern was observed for conceptual learning. Students in the Blocks and Text modalities showed similar scores on the final test, but again, the path there was quite different with learners in the Blocks condition seeing all of their learning gains coming during the first five weeks while the Text condition students saw consistent incremental growth over the two phases. A number of possible explanations for this were given in Chapter 5 (like ceiling effects or students not having enough time on task). These and other aspects of the outcomes will be explored in greater detail in the implications section later in this Chapter.

Overall, the results of this dissertation show that modality makes a significant difference in learners' early programming experiences in a variety of ways. This dissertation also reveals that these differences begin to fade as students leave the introductory modality and move on to more conventional programming languages. With this work we were able to tease apart aspects of this story to show how modalities fostered productive attitudes, supported effective programming practices, and facilitated students in learning foundational programming and computer science concepts. In the next section, a review of the Hybrid condition is given, before a longer discussion on the implications of these findings is presented.

The Case of the Hybrid Modality

The third set of research questions asked in this dissertation pertained to the design of hybrid blocks/text programming environments, asking: Is it possible to design a “best-of-both worlds” introductory programming modality? Overall, the Hybrid environment used in this study

was found to produce outcomes similar to the Blocks condition in some dimensions while being more closely linked to the Text condition in others. At the same time, there were also instances where the students in the Hybrid conditions were outliers relative to their Blocks and Text peers. Below, these findings are summarized.

The Hybrid Condition: The First Five Weeks

Over the first five weeks of the study, students in the Hybrid condition showed attitudinal changes that were similar to those observed in the Text condition: little change with respect to confidence or enjoyment of programming and a decrease in interest in taking future computer science courses. When asked about how the introductory environment compared to what real programmers do and if the introductory environment made them a better programmer, the Hybrid students gave responses similar to the Blocks students, which were lower than their Text-based peers. Together, these findings show that the Hybrid environment was not particularly successful with respect to cultivating positive attitudes and interest in computer relative to Blocks or Text alternatives.

On the mid-point administration of the commutative assessment, the Hybrid condition scored between the Blocks and Text students overall. Grouping questions by modality, the Hybrid condition's aggregate scores were close to the high-mark set by the Blocks condition on the Pencil.cc Text and Pencil.cc Blocks questions and close to the lower scores set by Text condition on the Snap! Blocks questions. The Hybrid condition also netted out between Blocks and Text students on three of the six conceptual categories (conditional logic, functions, and comprehension). The Hybrid condition scored the highest on algorithms and iterative logic and the worst on variables (only narrowly). When asked about the perceived ease-of-use of various programming constructions, the Hybrid students gave responses closer to the “they were easy to

use” responses given by the Blocks condition for variables and functions, but closer to the lower scores of the Text condition for conditional logic. Again, this shows that the Hybrid condition shares the features of the other two modalities. The analysis of students’ responses to the open-ended conceptual questions showed that the Hybrid students once again have characteristics similar to the Blocks condition in some respects and the Text condition in others. Like the Blocks condition, students in the Hybrid condition showed a higher likelihood to treat related concepts as their own entities (like treating `if` and `if/else` concepts separately and viewing variables as their own distinct entities). This finding fits in with the explanation of the presence and nature of the blocks palette in shaping this view. We also saw patterns akin to the Text condition, such as students favoring technical definitions of more colloquial explanations of ideas (as seen in the students discussion of conditional logic). There were also conceptual outcomes unique to the Blocks condition, like the increased rate of defining looping constructs temporally. Taken together, these results highlight how the Hybrid condition has successfully blended the Blocks and Text modalities, with the Hybrid modality often resulting in students showing attitudes and results that live in the space between the two other modalities. There were also a few places where the Hybrid condition is distinct from the other two modalities, suggesting that, in some ways, the Hybrid modality is not just is simply the sum of the other two modalities.

The dimension where the Hybrid condition was least like the other two modalities was in the programming practices students developed. Over the course of the five-week introductory curriculum, Hybrid students wrote the longest programs and also ran their programs more frequently than either of the other two modalities. The vignettes reveal one potential explanation for this. In the vignette, we saw the student fluidly move back and forth between using the drag-

and-drop mechanism of the blocks modality and editing statements and adding new commands with the keyboard. This means students could quickly add fully formed statements, making it easy to author longer programs, but also quickly make minor edits or introduce syntax errors through keyboard input, both of which help explain the increased run frequency. The analysis of programming practices also found unique affordances of the Hybrid modality, such as students using the blocks as a way to check the syntax of typed-in commands. We also saw that over time, the students in the Hybrid condition used the drag-and-drop mechanism for adding commands less and less. Together, these two trends suggest that high school aged students prefer the keyboard-based form of input and that the drag-and-drop mechanism is a helpful way to bootstrap authorship early and an intuitive way to verify statement structure and syntax.

The Hybrid Condition: Transitioning to Java

Whereas the Hybrid condition did not seem to produce the desired, positive outcomes with respect to attitudes of and perceptions towards programming during the first five weeks, things start to change after the transition to Java. When asked to reflect on their time in the introductory modality, students in the Hybrid condition reported their time in the introductory tool as being the most helpful and the most similar to real programming when compared with the other two modalities. In the four other attitudinal categories evaluated, the Hybrid condition saw relatively little change, having three categories showing slight increases (enjoyment, perceived difficulty, and interest) and a minor decrease in one (confidence). Students' scores on the Commutative Assessment decreased a small amount after working in Java for ten weeks, suggesting the modality was not an outlier with respect to preparation for future text-based learning in a different language. Looking at various characteristics of programming practice for students in the Hybrid condition showed them to adopt an approach more similar to the Blocks

condition in terms of frequency of running their programs and in the size and nature of their incremental programming edits. Where things differed for the Hybrid condition is an increased frequency of certain types of Java errors made during the first ten weeks of learning the language. Students in the Hybrid condition showed a higher propensity for having compilation errors generated by incomplete quoted strings in their programs, which can result in a number of different types of errors. This error is interesting in how it relates to specific features of the Hybrid modality, which provided all of the necessary open/closing quotes when adding statements to a program in the introductory environment. So here, we have a nice, albeit relatively nuanced, example of a practice, fostered in the Hybrid modality, carrying over to the professional text-based language with detrimental effects.

The Hybrid Condition: Summary

The major take away from this analysis is a definitive answer to part of the third research question, showing that it is possible to design Hybrid modalities. The specific Hybrid modality used in this study shows that the design choices made in the creation of new modalities and learning environments can produce outcomes similar to either of the source modalities used, as well as unique outcomes distinct from the designs that served as its inspiration. In this study, we found places where the Hybrid condition succeeded in drawing on the strengths of both modalities. For example, in providing the scaffolding and ease of composition of the Blocks modalities while also conveying the perceived authenticity students associated with the text-based interface. At the same time, there were instances where the Hybrid condition produced something closer to a “worst-of-both-worlds” outcome, as could be seen in places where the programming practices that relied on the blocks palette resulted in students encountering certain types of errors more often. In this case, they had developed comfort and familiarity with the

textual representation, but did not have the supports that accompanied the practices they had developed. Taken together, the Hybrid condition in this study shows the potential for this line of work in developing effective programming environments. It also shows one of the many possible ways to blend blocks-based and text-based programming environments and sheds light on the potential set of outcomes from doing so. The next section discusses the implications of these results before taking a few steps back from this specific study to situate the findings in the larger context of modality, learning, and design.

Implications

Having reviewed the findings from the two-year study, this section discusses some of the concrete implications of these discoveries. First, the implications of modality choice as it relates first to the learner are discussed. Similar discussions looking first at teachers and then at schools follow. These focus on how modality choice potentially impacts the larger educational infrastructure that surrounds the formal computing education learning opportunities provided to learners today.

Implications of Modality on the Learner

This dissertation is the first careful study into how modality impacts learners. It shows how modality affected students' attitudes, perceptions and conceptual learning. Thus, it supports the claim that modality has a direct impact on learners' experiences with programming and their early computer science classroom learning experiences. Further, given that modality was conceptualized in this work as characterizing the relationship between representation and user, the impact of modality will necessarily be unique for each student based on their predispositions, prior experiences, and incoming knowledge.

The choice of modality can facilitate engagement, help foster a positive classroom culture, and shape how a learner feels about the domain. The choice of modality is especially important early in learners' interactions with the field, as negative early experiences may turn them away. At the same time, the choice of modality will influence how learners experience future computer science learning opportunities. As was shown in this dissertation, productive dispositions fostered by blocks-based modalities early in the study did not carry over to more professional programming languages, resulting in less positive experiences down the road. Here, we refer to the limited extent of modality on impacting learners – it matters while students are working with it, but modality choice for introductory environments seems to have relatively small long-term attitudinal, perceptual, or conceptual impact. This is not to say it is not an important decision. Negative early experiences may result in students' choosing to withdraw from the course or lose interest and not put forth the same effort they may have if the early experiences were more positive.

A complicating aspect of modality choice in formal education spaces is the fact that students are entering their first computer science learning opportunities with an increasingly diverse set of prior programming experience. In this study, some students had never programmed before, while others had just spent the summer trying to learn trendy, professional development frameworks. Given that all students in the same class usually learn with the same environment and are asked to complete the same set of assignments, keeping advanced learners engaged while also not leaving true novices behind is a challenge. Modality choices made to support one type of learner may negatively affect the other. This came up a few times in this study, when advanced students lamented having to use a blocks-based modality, instead wanting to go straight into learning Java. In cases such as these, hybrid modalities, like the one used in this study, show

promise for achieving both the low-threshold needed for true novices as well as the high ceiling for students with more prior experience. As will be discussed in greater detail in the next section, modality does not inherently make a language more or less powerful, instead it just shifts how one interacts with it. Further, much of computer science is less concerned with syntax and details of a programming language and instead focuses on issues related to problem solving and critical thinking. Modality choice directly impacts learners, but through framing and carefully selected activities, the drawback of beginner modalities on more advanced learners may be mitigated without sacrificing the benefits they hold for the novices they were designed for.

Finally, this dissertation shows that high school students are able to think critically about the tools they are using and can articulate strengths and drawbacks of such tools. This finding shows the sophistication that high school students have with respect to their own preferences and perceptions as it relates to programming and learning. Students' ability to discuss the various ways that a modality is useful for learning shows they see how and why these tools can be instructive and useful for their own learning. This implies that when choosing a modality for specific pedagogical or affective reasons, students should be encouraged to use the tools in the ways they find meaningful. If a student finds the scaffolds or features of a modality to be a distraction because they do not feel they need them, they should be allowed to use the tool as they see fit. Likewise, this dissertation shows that high school-aged learners have pre-conceived notions about what "real" programming is, and what it looks like. Given this preconception, it is not necessarily beneficial to try and convince high school aged students that blocks-based programming is the same thing as text-based programming. Alternatively, by taking advantage of the sophistication of the learner and respecting their knowledge, framing introductory, highly scaffolded modalities as being productive for learning can potentially alleviate issues of

inauthenticity or a lack of perceived uselessness. Shifting utility towards usability and learnability may help learners identify the value of different modalities, irrespective of their prior experience.

Implications of Modality on the Teacher

The choice of modality will have a large impact on the experience of the teacher and their experiences in the classroom. Modality can influence classroom culture, pedagogical approaches, and in part shapes the curriculum that is followed. In choosing a given modality, the teacher is setting in motion various aspects of the course and their own position in it. Modalities designed to support novices in programming independently will impose different challenges on the teacher compared to a modality with fewer beginner-oriented features. A teacher's preference for direct instruction versus letting learners discover and explore on their own should be taken into account when choosing a modality. When working in a modality designed for beginners, the learners' reliance on the teacher for guidance is decreased, thus the teacher can spend more time in one-on-one support. At the same time, if students are better able to make progress on their own, there is less potential for teachable moments – instances when students ask questions that lead to productive class discussion. One of the first year teachers brought up this point as he explained his experience teaching in the blocks modality: "*the point of the environment is that it shouldn't generate a whole lot of questions, like 'how do I do this?' - it's more intuitive.*" The teacher went on to explain that while this is empowering for the learner, it gives him fewer opportunities to engage in productive discussions on different aspects of programming.

Just as modality choice shapes the role of the teacher in the classroom, it can also shape the curriculum. Modalities designed to facilitate exploration and creativity allow for different types of assignments compared to modalities designed for efficiency or clarity. If a teacher

prefers every student to author a program that looks the same, choosing a modality that makes discovery easy may prove counterproductive to the teachers desired form of assignment. There are also class management and grading considerations in choosing a modality. If assignments are open-ended or assigned in a modality that make it easy for students to go beyond what has been covered in class, the teacher is more likely to encounter more diverse solutions or solutions that include extra features beyond what was asked. This was a frequent occurrence over the two years of this study, especially among more advanced students who sought to challenge themselves on assignments they were able to complete quickly.

Along with impacting students and the role of the teacher, modality can also shape classroom culture. As one teacher who participated in this study pointed out: “[Blocks-based programming] *creates a different feel to the room...blocks take away the foreign feel, it looks friendly, and it's something you can do right away, and because of that, the culture in the room is different, kids are more prone to talk to their neighbors, more prone to feel OK about joking around.*” While modality is not the only contributor to a classroom culture, more inviting and playful tools can help shape a certain set of classroom norms.

A final, potential afterthought for a teacher in choosing a modality is considering the larger technological infrastructure of the class. Are assignments going to be submitted in a specific online format? Is the teacher planning on running all of the students programs to make sure they work and meet the requirements of the assignment? The environments used in the introductory portion of this class were all browser-based, which made it tricky for students to submit their work as they did not have a local file to submit. The teacher in the second year of the study would do her grading of student projects by walking around the room asking students to show her their work. While this worked for the purposes of this teacher, it had its limitations

as the teacher could only spend a few seconds on each and did not have a way to give detailed or written feedback to students.

Implications of Modality Choice on Schools and Administrators

A major implication of the findings in this dissertation affects, not the students or teachers, but should inform the larger infrastructure in which these learning opportunities are situated. The decision of if and when to transition from an introductory modality to a professional programming environment is consequential and has many potential repercussions. For example, if the transition from a blocks-based introductory tool to a text-based language takes place the week before the drop/add deadline, what will be the implications of that transition relative to if it happened a week after students could no longer drop the course? The findings in this dissertation say you would likely see more students choose to leave the course if the transition happened before that deadline. Likewise, what if students were asked to sign up for next year's classes in week four of the study? Would we expect to see the same number of students from the Blocks class enroll in a future computer science class as from the Text condition? The data presented in this study suggest that that timing of transitioning between modalities should be carefully considered and external deadlines, like enrollment dates and drop/add deadlines should be considered.

This also opens up the larger conversation about whether or not the transition from introductory tools to professional languages is necessary. In the United States, until recently, the AP Computer Science exam, which is the closest thing the country has to a national computer science curriculum, was essentially a Java programming exam. In order for students to receive college credit in computer science, they had to learn to program in Java. This is now changing with the introduction of the AP Computer Science Principles course, which focuses less on text-

based programming languages and instead emphasizes broader computer science concepts, such as algorithms, problem solving, and data and information. Deciding the importance of programming and whether or not to prioritize professional text-based languages is a consequential decision that has large effects on how computer science will be taught and the experiences that learners will have. This dissertation sheds light on some of the implications of this decision, showing both the promise of introductory programming tools as well as some of the challenges associated with transitioning modality and language early in a learners computer science career. As one teacher said when asked about how concepts carried over from the blocks-based introduction to Java, the transition was “*rough, I think [the students] lost what they were doing [in the blocks-based tool] with what they were doing in Java.*” Whether or not this transition is necessary is an important question for administrators and department chairs to decide given the shifting nature of computer science education where the decision is no longer being made for them.

On Modality, Learning, and Design

While this dissertation was focused on three specific modalities and the domain of computer science, the implications for this work extended beyond the particularities of this study. At the highest level, this dissertation is concerned with understanding the relationship between modality and learning and then relating those findings to the design of new modalities. In this work, the conceptualization of modality included not just the static representation, but also its affordances and the various ways it was appropriated by learners. In this way, describing modality is not purely an exercise in describing visual characteristics, but also includes how various features are taken up by users and how they do and do not enable the learner. Over the course of the fifteen weeks of the study and the three modalities used, new insights into the

relationship between modality and learning emerged as well as implications for their study and design. This section characterizes what we learned about modality through conducting this work.

Modality Matters

One contribution of this work shows how and when modality matters with respect to learning to program and the design of introductory programming tools. By following novices learning to program with three isomorphic modalities, we showed the various ways that modality does and does not impact learners. Over the two years of the study, modality was found to influence learners in many different ways, including program comprehension ability, program composition strategies, emerging conceptual understanding, and various dimensions of affect and attitudes towards the discipline. In the first year of the study, the data showed that students performed better on questions asked using a blocks-based modality than a textual form. In the second year of the study, students using blocks-based, text-based, and hybrid blocks-text tools ended up performing differently on content assessments, reporting different levels of interest in the field, and viewed the utility and authenticity of their learning experiences differently. Along other dimensions, little difference was found across the modalities, including confidence, enjoyment, especially in conceptual understanding after students stopped using the introductory modalities. These findings show the importance of recognizing and considering the various ways that modality affects learners and the context in which learning is taking place.

The definition of modality used in this dissertation uses the term to characterize how one interacts with a given representation and the role the design of the representation plays in supporting various uses and interaction patterns. This dissertation thus provides insight into the various ways a representation design can shape and support different types of uses. Across the

three modalities used in the introductory portion of the study, various types of supports were provided for different aspects of the practice of programming. For example, the drag-and-drop capability of the Blocks modality provided a form of interaction not possible with the Text modality, thus these two modalities produce distinctly different authorship patterns with different challenges and supports provided. In this way, the modality shapes how things are done, but not what can be done. A second example could be seen in the first year of the study, where the Snap! Blocks modality was shown to be easier to comprehend for novices than a text-based modality. A careful analysis of student responses was able to identify features of the modality that supported novice comprehension, such as the shape of function calling blocks helping learners know what their behavior will be. Other dimensions such as color, the choice of words used in the language, and the presentation and arrangement of the representation also supported learners in various ways including the construction of programs as well as helping them develop ideas and communicate information about their programs.

Throughout this dissertation, much care was taken to highlight the various ways that learners used the modalities. Both within and across conditions, this dissertation reported on different features of modalities being appropriated by learners in different ways. This provides a concrete example of what Noss and Hoyles (1996) call *webbing*. The construct of webbing is intended to capture the rich, diverse, and interrelated features of learning environments that provide support to the learner. Webbing describes “a structure that learners can draw upon *and reconstruct* for support – in ways that they choose as appropriate for their struggle to construct meaning” (Noss & Hoyles, 1996, p. 108). The term webbing was chosen to capture the full network of supports provided to the learner, not just a single scaffold within the environment. To understand the role of modality in supporting the learning process through this lens, one must

view the various features of the environment being used in concert, as opposed to elements used in isolation. In this work, this means recognizing how the shape of a block, the drag-and-drop mechanism, and the arrangement of block in the palette all collectively contribute to helping the learner make meaning. Further, through this lens, we can remain faithful to the recognition that learning is not uniform, but is unique to the individual. This is to say, modality matters not just in terms of what it makes possible, but also for the various ways it allows diverse sets of learners to make meaning in their own, personally meaningful ways. Modality cannot and should not be evaluated by looking at a single student, nor should it be designed to promote a single specific practice, but instead, the design of modality should be seen as an opportunity to support diverse practices and forms of expression.

Modality is Malleable

In addition to showing that modality choice is consequential, this dissertation shows that modality is not fixed. Instead, modality is malleable; it can be designed, changed, blended, and extended. This perspective opens the door to the larger enterprise of creating new modalities through the revision of existing forms as well as the creation of entirely new ways of expressing ideas and interaction with representational systems. We see this dissertation contributing to a new and growing discipline on the study and design of representations that has been argued as a possible future direction for the Learning Sciences (Papert, 2006; Wilensky et al., 2005; Wilensky & Papert, 2010). Further, this dissertation argues that in considering the design of modality, it is important to look beyond the static represented form and consider the practices and interaction patterns made possible by the modality. This shift is important as new representations increasingly are coupled with, and rely on, the capabilities of computational media. The interactivity and feedback enabled through computer-based representations adds a

new dimension to the design of notations. By including practices and usage patterns as part of a broader conceptualization of modality, new considerations and design opportunities emerge.

Programming languages provide an especially rich context for the design of new modalities due to what Papert (1980) called the *Protean* nature of computers. Computers provide the ability to introduce layers of abstraction between the way a representational infrastructure is presented to the user and the form those instructions must take so they can be executed by the machine. From this relatively blank canvas a vast design space emerges for the creation of new modalities. Looking at the three modalities used for this work we can start to see the various dimensions along which computationally situated modalities can be defined. Visual rendering (color, shape, location on the screen, etc.) is a first dimension that a modality design can explore. Likewise, how and when other representational systems are incorporated can differ. By this we mean, if natural language descriptions will be used, compared to symbolic representations or programming language primitives can differ by modality. This can be seen in comparing Scratch Jr. (Flannery et al., 2013) and its use of glyphs to Scratch (Resnick et al., 2009) where natural language expressions are used, to Pencil Code (Bau et al., 2015) which provides visual supports on top of programming keywords, and finally to Logo (Papert, 1980), which is a fully text language. Figure 9.2 shows the turn right command as it is represented across these four tools.



(a)



(b)



(c)

right 90

(d)

Figure 9.2. The same concept (turning right) conveyed in four modalities: Scratch Jr. (a), Scratch (b), Pencil Code (c), and Logo (d).

The four representations shown in Figure 9.2 are all static for the user, once defined they do not change. However, modality need not be static. Looking at the design potential of modality across a temporal dimension, we can start to imagine new representations that change over time. For example, we can have visual cues like moving arrows or blocks that change shape or size based on uses or the point in the composition process. Likewise, the modalities presented above and used in this dissertation are all virtual. A growing number of programming tools take advantage of physical devices, like Tern (Horn & Jacob, 2007), Robo-blocks (A. Sipitakiat & Nusen, 2012), and Cubelets (Schweikardt & Gross, 2006), which each allow the user to express computational statements without using a screen.

As modality is intended to describe interactions, designing modalities extends beyond just the visual depiction of the representation. The design activity also includes various interaction capabilities that influence the mechanics of interaction with and use of the modality. The clearest distinction of this aspect of modality in this dissertation is the difference between dragging-and-dropping blocks on the canvas versus typing commands in character-by-character with the keyboard. While the textual modality supports in Pencil.cc were rather minimal, other integrated development environments (IDEs) include a richer suite of functionality to support different interaction patterns and authoring mechanism. For example, autocomplete allows users to type only a few characters before a curated list of potential commands appears. Another example comes in the form of templates, where the user can browse a set of pre-defined sets of commands to choose larger blocks of code to accomplish set tasks. For more advanced programmers, IDEs provide other programming authoring/editing mechanisms such as code refactoring tools that allow the user to edit multiple lines of code at once or rearrange large portions of code based on the needs and wants of the author.

A final important dimension to discuss in terms of the malleability of modalities is to point out that a user need not be pinned to a specific modality. A growing number of environments are supporting users in moving between multiple modalities. The learners in this dissertation study were held to a specific modality for the purpose of the research study, not because of design constraints. Pencil Code, the environment that was the basis for Pencil.cc, allows learners to freely move back-and-forth between a text-based and blocks-based modality, with changes in one modality being reflected in the other. A number of projects are looking at supporting this same dual-modality representation for various programming languages include Java (Matsuzawa et al., 2015), Python (Bart et al., 2015), and Grace (Homer & Noble, 2014). The ability to provide multiple modalities within the same environment further opens up the set of possibilities to modality designers. A strength of this approach is that it gives agency to the learner to decide not just how to use the various features of a single modality, but also to choose the modality they want to use.

Modality and Structurations

The findings presented in this dissertation with respect to the role of modality on learners complement existing work on representational infrastructure and its impacts. Prior work, both theoretical and empirical, has shown that representational infrastructure has implications along a number of dimensions including expressive power, learnability, communicability, and shaping the nature of emerging understanding (diSessa, 2000; Kaput et al., 2002; Sherin, 2001; Wilensky & Papert, 2006, 2010). Much of this work has looked at the nature of notational systems, the prototypical example of which is the comparison between roman numerals and Hindu-Arabic numerals, and the widespread impact of the shift between these two ways of representing the same concepts (Swetz, 1989). Wilensky and Papert (2006, 2010) use the term structuration to

capture the relationship between a representational infrastructure and its ability to encode knowledge of a given domain. This dissertation is primarily concerned with modality, which differs from structurations in that modality is concerned with a person's interactions with a given structuration and how the design of that representational system supports various uses, whereas structuration is more foundational, characterizing a representation's ability to encode and express ideas of a given domain. While modality differs from structuration, Structuration theory provides a productive lens for thinking about modality and the findings from this study. Wilensky & Papert (2006, 2010) describe five properties for evaluating a given structuration: power, cognitive, affective, social and diversity, many of which are on full display in this dissertation. Below we discuss each of these structuration properties as they pertain to modality broadly and then link them to the specific modalities explored in this dissertation.

The Power Property

The power property of a structuration describes the set of things that it is possible to express with a given representational system. We view the power of a structuration and its modality as orthogonal; modality does not change what can be expressed with a given representation but instead, characterizes how one goes about saying it. That is not to say the two are unrelated. We see two ways that modality influences the power properties of a structuration. The first has to do with perception. The first year of the study showed some students to hold the perception that blocks-based modality as less powerful than the text-based alternative, saying things like ““blocks are limiting, like you can't do everything you can with Java”. While technically not true, it is revealing that the students held this view. This means, that while modality does not change the power of a structuration, it can change the perceived power of it, which potentially results in the same outcome. In this case, it might mean students taking the

structuration less seriously or seeing it as less valuable. The second interaction between power and modality we see is in how a modality can make things easier than they might otherwise be. If the same idea can be expressed in two modalities, but one requires less cognitive load, less time, or fewer steps, that may lead to other benefits related to the power principle. For example, a new modality might make discovery more likely as exploration of ideas with that structuration may be better supported. This was observed in this study in how students relied on the blocks palette as a way to support their program construction; this helped novices do more in the blocks-based modality than the isomorphic text modality.

The Cognitive Property

The cognitive property of a structuration captures its learnability; does a specific form of representation make ideas more intuitive or accessible? This dissertation shows that modality plays a potentially strong role in shaping the learnability of a given structuration. While working in the blocks-based modality of Pencil.cc, students showed greater learning gains relative to students working in the text-based modality. These two modalities were isomorphic, shared language semantics, and most other environmental factors were held constant. In other words, the structuration was the same, the modality differed, and after five weeks, learning outcomes also differed. This finding has potentially large implications for thinking about the role of modality with respect to the evaluation and design of new structurations.

The Affective Property

The third property for evaluating structurations is the Affective Property which describes how one feels about a given representational infrastructure with respect to things like enjoyment, engagement, and other emotions. Like with the Cognitive Property, this dissertation showed that,

along some dimensions, modality played a role in shaping a learner's affect for a given structuration. This dissertation found subtle differences in terms of affective dimensions like enjoyment and confidence, and stronger differences in terms of students' relationship to the discipline of computer science and whether or not they would continue in the field. There is also the notion of perceived authenticity and utility, which falls under the Affective Property and was also found to be affected by modality. This provides evidence that there is an interaction between structuration and modality with respect to affect, although in this dissertation, that interaction was less pronounced than the cognitive property.

The Social Property

The next property for evaluating structurations captures how well an idea can be communicated in a given structuration. Does the form make it easy to describe ideas, share insights, and convey information? While the construct of modality is meant to capture the relation between an individual and the structuration, features that support that interaction may also facilitate person-to-person communication. We saw some examples of this in the vignettes presented in Chapter 7, where students had slightly different ways of using the modality to assist in their explaining their programs to the interviewer. This data does not fully capture the breadth of everything included in social aspects of a structuration, but suggests there is some interaction between the two. This dissertation provided glimpses of the social properties of modalities but was not the focus of the work, thus exploring this dimension remains an open question and a possible avenue for future work.

A second relevant aspect of social dimensions of modalities considers the larger, online communities that can form around tools that leverage specific modalities. The most notable example of this is the online Scratch community that includes over 15 million projects. On the

Scratch website, visitors can run programs written by others, inspect the code behind the project, and “remix” the program, starting with a project and making their own additions and modifications (Fields et al., 2014; Roque, Kafai, & Fields, 2012). While such online communities are not tightly coupled to modality, the most vibrant online learner communities for sharing programs are associated with blocks-based languages. Further, they are some aspects of the blocks-based modality that might facilitate the remixing culture that tools like Scratch seek to promote, specifically, the “readability” of the programming representation makes it easier for novices to parse and understand a program written by another. Again, the existence and ongoing success of an online community built around a programming environment is not specific to modality, but modality plays a role and it impacts the social properties of a structuration.

The Diversity Property

The final structuration property is the Diversity Property. This property describes how well a structuration is at supporting different ways of thinking or solving problems, what Turkle and Papert (1990) call *epistemological pluralism*. Along with the Cognitive and Affective Properties, this seems like the place where modality and structuration intersect heavily, as all three are characteristics of the relationship between the structuration and the individual. In Chapter 4, when reviewing the findings from the first year of the study, we saw a nice example of the Blocks modality supporting a form of construction that, while possible in a textual modality, is less well supported. The episode we are referring to saw a student author a condition statement by first dragging out an = block, then defining the two signs of the comparison, then dragging out the `if` block and finally defining the behavior to be carried out if it evaluated to true. These steps show the student authoring this conditional statement from the inside out, as opposed to from the left to right, the sequence suggested by the text modality. Throughout the

two years of the study, we saw examples of students building complex statements diverse ways, rather than the relatively uniform approach students in the text condition followed. This shows that modality has the ability to influence and support a diversity of ways of constructing with blocks-based tools.

Modality and Milieu

In this final section of our extended discussion on modality, we look at how modality interacts with the larger context in which it is encountered. Modality is inherently situated into a larger network of designed entities and established practices that collectively shape the experience of the learner. For example, the last section explored the relationship between modality and structuration, showing the two to be interconnected but distinct. Likewise, there are a number of other situational aspects in which a modality lives and interacts. One major influence on modality is the environment in which it is situated, which can inform and shape how a modality is used. In this study the three modalities were presented inside the larger Pencil.cc context. Pencil.cc included the visual execution space, defined and constrained the types of things that are possible with the modality (like Turtle Geometry and text input/output), and included additional scaffolds outside of the modality that influenced learners' actions. To that last point, Pencil.cc's Quick Reference menu provided guidance to learners on the usage and capabilities of the language in a way distinct from the supports available through the modality. The data collected in Pencil.cc showed how usage of the Quick Reference menu was different based on the modality of the student.

The idea of environment influencing modality extends beyond the specific technological setting or medium being used to include the larger socio-cultural context in which it is being used. Bronfenbrenner (1979) argues for a concentric nesting of contexts in which the individual

resides, starting with the microsystem (peers, family, classroom) and growing outward through a mesosystem (school, friends-of-friends), exosystem (media, neighborhood, extended family) and finally macrosystem (cultural factors). Each of these sequential contexts informs and influences how an individual interacts with a given modality. For this study, the microsystem played a large role in shaping usage patterns. The presence of teachers, peers, and the one-laptop-per-student arrangement of the classroom all influenced how learners interacted with the modality. The ability to ask a neighbor or listen to the teacher carefully introduced different concepts changed how and when the learner engaged with the modality and the role the modality played. Influences of the larger dimensions of the learning context could also be seen, including the school system and how technology was integrated into the K-12 learning experience and also the larger cultural values placed on technology and learning to program.

In this study and the specifics of the modalities used we can also see aspects of the exo- and macrosystems and play. Specifically, the blocks-based design takes advantage of a puzzle-piece metaphor, where visual cues denote how and when commands can fit together. This design approach assumes that the user has experience with puzzles or toys that share this interlocking assembling characteristic, be it Legos, train tracks, or jigsaw puzzles. Drawing on the form-function shift framework (Saxe, 1999), Horn (2013) calls these cultural forms and argues that such cultural practices and knowledge can be leveraged in the design of intuitive computing interfaces. The blocks-based modality used in these classrooms serves as one example of a cultural form being productively leveraged with positive outcomes.

The fact that modality influences the classroom culture affects both the learner as well as the teacher, whose role changed between the three conditions of the study. In interviews with the teacher from the second year of the study, she described how the level of detail she needed to

provide differed by modality and how students independence differed by modality, with the Blocks students being the most self-sufficient (Weintrop & Wilensky, 2016a). Tightly coupled with the role of the teacher is the set of pedagogical strategies supported by different modalities. As discussed in the previous section, with modalities that provide high levels of scaffolding to support learner independence, teachers' pedagogical strategies can shift away from direct instruction to one-on-one personal attention. The opposite is also true, when using modalities with little in the way of novice support, the strategies the teacher uses differ and the opportunities for meaningful conversation also shift.

Just as the modality interacts with classroom culture and teaching practices, so too are the curriculum and the set of activities the students work through similarly affected. Asking students to have onscreen sprites move around and draw a specific shape will allow learners to leverage different affordances of a modality than a different activity, like sorting numbers. More concretely, in an activity involving motion, having a modality that supports natural language expressions like `turn right 15 degrees` on a block, will provide a different form of support than if a student had to type in the command `rt(15)`; to accomplish the same behavior. Likewise, the browsability of blocks-based modalities can facilitate creativity and exploration in open-ended assignments differently than other modalities that do not have that ease of discovery. This feature of the blocks-based modality is not surprising given the constructionist roots of the modality.

Limitations and Future Work

While this study provided insight into the relationship between modality and learning in the domain of computer science, it does have limitations with respect to the claims that can be made. This section reviews the limitations and discusses potential future directions that may be

taken to try and address them. The first limitation of this study is related to the students who were recruited to participate. The school where the study took place was a selective enrollment institution. This means that all of the students that participated in the study have historically been successful in formal educational contexts. Thus the findings of this dissertation do not necessarily apply to underperforming students who have not had success in conventional classroom settings. A second limitation of this study relates to the student population as this study took place in an elective class. This means the students who participated in the study had chosen to take part in a computer science learning opportunity, suggesting they showed a pre-disposition for being more interested or placed a higher value on the concepts being taught. A final participant-related limitation of the study has to do with the gender breakdown of the study. In both years, female students made up less than one third of the student in the class. The gender breakdown was beyond the control of the researchers as student recruitment for the classes was outside of the scope of the study, but is none-the-less not representative of the greater student population. All three of these limitations can be addressed by conducting future iterations of the study at different schools where these limitations are not necessarily true. In some school districts around the country, computer science is becoming a graduation requirement for high school. Conducting a similar version of this study at a non-selective enrollment school where all students must take the class would directly address all of these limitations and is one intended future direction for this work.

A second limitation of this study has to do with the teachers who volunteered to participate. Finding a teacher who is willing to teach the same curriculum using three different modalities is difficult. Any teacher willing to take on such a challenge will have a level of confidence and experience that is rare among in-service computer science teachers.

Understanding how modality affects less experienced and less confident teachers is an open and important question to answer. Just as the way to address the student-related limitations of this study was to replicate the study at a different site, the solution to the generalizability of the findings due to the teacher can similarly be addressed this way. Working with a less accomplished and experienced teacher (or set of teachers) is a direction of future work that goes hand-in-hand with working with a different population of students, and is an intended avenue of future work.

A third limitation of the study has to do with duration of the second phase of the study. The study followed students for the first ten weeks of the Java portion of the course. For a number of reasons, the amount of content covered in the first 10 weeks of learning Java was much smaller than the breadth of concepts students encountered in the 5-week introductory portion of the course. There are a number of reasons for this, including the level of detail topics are given, the ease of compiling running programs in Pencil.cc compared to Java, and the various supports provided by different modalities that have been the focus of this dissertation. As a result, some of the concepts covered in phase 1 were not encountered during phase 2, which limits our ability to make claims about whether or not the conceptual learning of that concept transferred. Similarly, when the students do re-encounter the concepts, a great deal of time will have passed. For example, more than 15 weeks elapsed between when students used iterative logic in Pencil.cc and when they learned it in Java. An alternative study design would have students use an introductory modality to learn a concept, then immediately transition to Java to learn the same concept, and then repeat this introductory modality then Java pattern for each concept. The sequencing was initially proposed by one of the teachers in the study who said she might use this approach in future courses. The means she sees pedagogical value in the

introductory modalities, but would like to use them in a way that is different than how they were used for this study. We hope to investigate this pedagogical approach in future work.

A fourth limitation related to the structure of the study stems from the fact that the courses took place at different points in the school day. The Hybrid condition was fourth period, in the middle of the day, whereas the Blocks and Text conditions were seventh and eighth period respectively. Being at the end of the day has a number of potential impacts on students in the classroom. First, at the end of the school day, students may not be as attentive as they would be during earlier periods, this may be due to fatigue from a long school day, or excitement about post-school activities. Likewise, students in the classes at the end of the day were more likely to be absent than classes in the middle of the day. For example, a few times during the study, football players were excused from their eighth period class to prepare for their upcoming game. As there were football players in the eighth period class, those three students received less instruction relative to their non-football playing peers. While this limitation is unavoidable, it is nonetheless important to note. The remedy for this limitation is similar to the solution for the other study-design related limitations, more iterations of the study that vary the time of day for each condition, thus giving us the ability to counter balance this dimension of the study.

There are also limitations to this study related to the specific languages and tools that were used. For this work, Pencil Code's blocks editor was used to represent the blocks-based modality, while its text editor served as the canonical text editor. In the case of the blocks editor, other blocks-based modalities, like Scratch and Snap!, include additional features not supported by Pencil Code]. Likewise, Pencil Code's text editor included some built-in scaffolds like syntax highlighting and automatic indentation, but not others that are common in text-based coding tools like auto-complete. Also, the choice of CoffeeScript to serve as the underlying language in

the introductory condition and Java as the transitioned-to professional language are both only one of many possible ways such instruction could take place. The course just as easily could have used JavaScript as the underlying language for the introductory portion and Python as the professional language. In all of these cases, the design decisions made do not invalidate the findings, but instead impose a set of qualifiers. We do not yet know if students working in a blocks-based JavaScript environment then transitioning to Java (or even text-based JavaScript) would produce different results. The exploration of different languages and different underlying syntaxes is work that is being actively pursued. Just as the choice of language and specific modality influence the findings, so too does the programming paradigm used and the design of the curriculum. The fact that roughly half of the introductory assignments relied on drawing or Turtle Geometry in some capacity does change the way students interact with the modality. Again, this does not invalidate any of the claims made, but instead slightly constrains the generalizability, an issue that can (and hopefully will) be addressed through future iterations of similar studies with new languages, curricula, and paradigms.

Along with this list of limitations, there are also other avenues of future work that we hope to pursue and that this dissertation starts to address but by no means is definitive about. The most obvious is the fact that the Hybrid environment used in this study was just one of many new modalities. Looking at how other emerging modalities and new ways of blending existing modalities influences and potentially improves the learning of the powerful ideas of computing offers many opportunities for exciting work and new findings. A second avenue of future work is to look at the impact of modality on different types of students. Do students struggling students see more, less, or different benefits from working in a specific modality compared to students who have excelled in academic settings? A third direction of future work is taking a similar

modality-centric approach to concepts beyond programming and computer science. As argued by Wilensky & Papert (2006, 2010), there is great promise and much work to be done looking at restructurations across a diverse array of disciplines. As new fields becoming increasingly computational, new tools, interfaces, and modalities are being invented to support new computational endeavors. Research projects similar to this dissertation could fit well amongst the various activities that accompany the formation new technologically enhanced tools and practices and the larger emergence of new computational fields.

Conclusion

This dissertation sought to answer foundational questions looking at the relationship between modality and learning in the domain of computer science. This work serves as the first systematic investigation of modality in formal high school computer science classrooms. Through the design of the study, this dissertation makes claims about the role modality places in shaping novice programmers' attitudes, perceptions, and conceptual understanding. It also revealed ways that modality can shape emerging programming practices. Further, in following students as they moved from introductory programming environments to more conventional professional languages, contributions were made with respect to our understanding of how and when the impacts of working in introductory tools carry over into later programming environments.

The title of the dissertation states its central contribution: modality matters. The fuller picture revealed by the two iterations of this classroom-based study found that, while modality mattered along a number of dimensions while students were working in it, the introductory modality had relatively little lasting impact with respect to attitudes and conceptual learning after transitioning to other tools. This dissertation also provides a theoretical contribution in terms of

how to think about and evaluate modality as a distinct aspect of a learning environment.

Additionally, it showed that modality should be thought of as a feature of a learning experience that can be designed for, and that features of an interface and the capabilities of a system can shape the way the user engages with it. While modality is a characteristic of all representational systems, computer science, and computing education research in particular, is especially well suited to serve as the context for conducting this work due to the malleability of the medium and the blossoming of new languages, modalities, and learning opportunities.

Collectively, this dissertation contributes to our understanding of the design of introductory programming environments. Given the increasingly digital world that we live in, determining how best to prepare today's learners for the computational futures that await them is of critical importance. This work is intended to help push this larger endeavor forward, providing methods, evidence, and ideas to contribute to educating the next generation of computationally literate citizens.

10. References

- Abelson, H., & DiSessa, A. A. (1986). *Turtle geometry: The computer as a medium for exploring mathematics*. The MIT Press.
- ACM/IEEE-CS Joint Task Force on Computing Curricula. (2013). *Computer Science Curricula 2013*. ACM Press and IEEE Computer Society Press.
- Adams, J. (2007). *Alice in Action with Java* (1 edition). Boston, Mass: Cengage Learning.
- Allen, E., Cartwright, R., & Stoler, B. (2002). DrJava: A lightweight pedagogic environment for Java. *ACM SIGCSE Bulletin*, 34(1), 137–141.
- American Association of University Women. (1994). *Shortchanging Girls, Shortchanging America*. Washington, DC: AAUW Educational Foundation.
- Anderson, J. R., Corbett, A. T., Koedinger, K. R., & Pelletier, R. (1995). Cognitive Tutors: Lessons Learned. *The Journal of the Learning Sciences*, 4(2), 167–207.
- App Inventory Java Bridge*. (2014). Retrieved from <https://code.google.com/p/apptomarket/>
- Armoni, M., & Ben-Ari, M. (2010). *Computer Science Concepts in Scratch*. Retrieved from <http://onlinelibrary.wiley.com/doi/10.1002/app.1975.070190908/abstract>
- Armoni, M., Meerbaum-Salant, O., & Ben-Ari, M. (2015). From Scratch to “Real” Programming. *ACM Transactions on Computing Education (TOCE)*, 14(4), 25:1-15.
- Astrachan, O., & Briggs, A. (2012). The CS principles project. *ACM Inroads*, 3(2), 38–42.
- Baker, R. S., & Yacef, K. (2009). The state of educational data mining in 2009: A review and future visions. *JEDM-Journal of Educational Data Mining*, 1(1), 3–17.
- Bakhtin, M. M. (1981). *The dialogic imagination: four essays*. Austin: University of Texas Press.

- Baroth, E. C., & Hartsough, C. (1995). Experience Report: Visual Programming in the Real World. *Visual Object Oriented Programming, Edited by MM Burnett, A. Goldberg & TG Lewis, Manning Publications, Prentice Hall*, 21–42.
- Bart, A. C., Tilevich, E., Shaffer, C. A., & Kafura, D. (2015). From interest to usefulness with BlockPy, a block-based, educational environment. In *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE* (pp. 87–89). IEEE.
- Bau, D., Bau, D. A., Dawson, M., & Pickens, C. S. (2015). Pencil Code: Block Code for a Text World. In *Proceedings of the 14th International Conference on Interaction Design and Children* (pp. 445–448). New York, NY, USA: ACM.
- Bayman, P., & Mayer, R. E. (1983). A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Communications of the ACM*, 26(9), 677–679.
- Begel, A. (1996). *LogoBlocks: A graphical programming language for interacting with the world*. Electrical Engineering and Computer Science Department. MIT, Cambridge, MA.
- Begel, A., & Klopfer, E. (2007). Starlogo TNG: An introduction to game development. *Journal of E-Learning*.
- Behnke, K. A. (2013). SLASH: Scratch-based visual programming in Second Life for introductory computer science education Poster Session. In *Proceeding of the 44th ACM technical symposium on Computer science education*. Denver, CO.
- Berland, M., Baker, R. S., & Blikstein, P. (2014). Educational Data Mining and Learning Analytics: Applications to Constructionist Research. *Technology, Knowledge and Learning*. <http://doi.org/10.1007/s10758-014-9223-7>
- Berland, M., & Lee, V. R. (2011). Collaborative Strategic Board Games as a Site for Distributed Computational Thinking. *International Journal of Game-Based Learning*, 1(2), 65–81.

- Berland, M., Martin, T., Benton, T., Petrick Smith, C., & Davis, D. (2013). Using Learning Analytics to Understand the Learning Pathways of Novice Programmers. *Journal of the Learning Sciences*, 22(4), 564–599. <http://doi.org/10.1080/10508406.2013.836655>
- Blackwell, A. F., Whitley, K. N., Good, J., & Petre, M. (2001). Cognitive factors in programming with diagrams. *Artificial Intelligence Review*, 15(1–2), 95–114.
- Blikstein, P. (2013). Digital fabrication and “making” in education: The democratization of invention. In J. Walter-Herrmann & C. Büching (Eds.), *FabLabs: Of Machines, Makers and Inventors* (pp. 1–21). Bielefeld: Transcript Publishers. R
- Blikstein, P. (2013). Gears of our childhood: constructionist toolkits, robotics, and physical computing, past and future. In *Proceedings of the 12th International Conference on Interaction Design and Children* (pp. 173–182).
- Blikstein, P., & Wilensky, U. (2009). An Atom is Known by the Company it Keeps: A Constructionist Learning Environment for Materials Science Using Agent-Based Modeling. *International Journal of Computers for Mathematical Learning*, 14(2), 81–119.
- Blikstein, P., Worsley, M., Piech, C., Sahami, M., Cooper, S., & Koller, D. (2014). Programming Pluralism: Using Learning Analytics to Detect Patterns in the Learning of Computer Programming. *Journal of the Learning Sciences*, 23(4), 561–599.
- Bonar, J., & Liffick, B. W. (1987). A visual programming language for novices. In S. K. Chang (Ed.), *Principles of Visual Programming Systems*. Prentice-Hall, Inc.
- Bontá, P., Papert, A., & Silverman, B. (2010). Turtle, Art, TurtleArt. In *Proceedings of Constructionism 2010 Conference*. Paris, France.

- Boroditsky, L. (2001). Does Language Shape Thought?: Mandarin and English Speakers' Conceptions of Time. *Cognitive Psychology*, 43(1), 1–22.
- Brady, C., Weintrop, D., Gracey, K., Anton, G., & Wilensky, U. (2015). The CCL-Parallax Programmable Badge: Learning with Low-Cost, Communicative Wearable Computers. In *Proceedings of the 16th Annual Conference on Information Technology Education* (pp. 139–144). New York, NY, USA: ACM. <http://doi.org/10.1145/2808006.2808039>
- Bronfenbrenner, U. (1979). *The Ecology of Human Development: experiments by nature and design*. Harvard University Press.
- Bruckman, A. (1997). *MOOSE Crossing: Construction, community, and learning in a networked virtual world for kids*. MIT.
- Bruckman, A., Biggers, M., Ericson, B., McKlin, T., Dimond, J., DiSalvo, B., ... Yardi, S. (2009). Georgia computes!: Improving the computing education pipeline. In *ACM SIGCSE Bulletin* (Vol. 41, pp. 86–90). ACM.
- Bruckman, A., & Edwards, E. (1999). Should we leverage natural-language knowledge? An analysis of user errors in a natural-language-style programming language. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems* (pp. 207–214). ACM.
- Bruckman, A., Jensen, C., & DeBonte, A. (2002). Gender and Programming Achievement in a CSCL Environment. In *Proceedings of the Conference on Computer Support for Collaborative Learning: Foundations for a CSCL Community* (pp. 119–127). Boulder, Colorado: International Society of the Learning Sciences.
- Bruner, J. (1973). *Beyond the Information Given: Studies in the Psychology of Knowing* (1st edition). New York: Norton & Co.

- Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A., & Miller, P. (1997). Mini-languages: a way to learn programming principles. *Education and Information Technologies*, 2(1), 65–83.
- Buechley, L., & Eisenberg, M. (2008). The LilyPad Arduino: Toward wearable engineering for everyone. *Pervasive Computing, IEEE*, 7(2), 12–15.
- Buffum, P. S., Lobene, E. V., Frankosky, M. H., Boyer, K. E., Wiebe, E. N., & Lester, J. C. (2015). A Practical Guide to Developing and Validating Computer Science Knowledge Assessments with Application to Middle School. Retrieved from <http://research.csc.ncsu.edu/learndialogue/pdf/LearnDialogue-Buffum-SIGCSE-2015.pdf>
- Burke, Q., & Kafai, Y. B. (2010). Programming & storytelling: opportunities for learning about coding & composition. In *Proceedings of the 9th International Conference on Interaction Design and Children* (pp. 348–351). ACM.
- Chadha, K. (2014). *Improving App Inventor through Conversion between Blocks and Text* (Honors Thesis). Wellesley College.
- Chandhok, R. P., & Miller, P. L. (1989). The design and implementation of the Pascal Genie. In *Proceedings of the 17th conference on ACM Annual Computer Science Conference* (pp. 374–379). ACM.
- Cliburn, D. C. (2008). Student opinions of Alice in CS1. In *Frontiers in Education Conference, 2008. FIE 2008. 38th Annual* (p. T3B–1). IEEE.
- Code.org Curricula. (2013). Code.org. Retrieved from <http://code.org/educate>
- Collective, T. D.-B. R. (2003). Design-based research: An emerging paradigm for educational inquiry. *Educational Researcher*, 5–8.

- Collins, A., Joseph, D., & Bielaczyc, K. (2004). Design research: Theoretical and methodological issues. *The Journal of the Learning Sciences*, 13(1), 15–42.
- Colmerauer, A. (1985). Prolog in 10 Figures. *Commun. ACM*, 28(12), 1296–1310.
<http://doi.org/10.1145/214956.214958>
- Confrey, J., & Smith, E. (1994). Exponential functions, rates of change, and the multiplicative unit. *Educational Studies in Mathematics*, 26(2–3), 135–164.
- Cooper, S., & Cunningham, S. (2010). Teaching computer science in context. *ACM Inroads*, 1(1), 5–8.
- Cooper, S., Dann, W., & Pausch, R. (2000). Alice: a 3-D tool for introductory programming concepts. *Journal of Computing Sciences in Colleges*, 15(5), 107–116.
- Cooper, S., Dann, W., & Pausch, R. (2003). Teaching objects-first in introductory computer science. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education* (p. 195).
- Danielak, B. A. (2014). *How electrical engineering students design computer programs*. University of Maryland, College Park, MD.
- Dann, W., Cooper, S., & Ericson, B. (2009). *Exploring Wonderland: Java Programming Using Alice and Media Computation*. Prentice Hall Press.
- Dann, W., Cooper, S., & Pausch, R. (2011). *Learning to Program with Alice*. Prentice Hall Press.
- Dann, W., Cosgrove, D., Slater, D., Culyba, D., & Cooper, S. (2012). Mediated transfer: Alice 3 to Java. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education* (pp. 141–146). ACM.

- Denny, P., Luxton-Reilly, A., Tempero, E., & Hendrickx, J. (2011). Understanding the syntax barrier for novices. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education* (pp. 208–212). ACM.
- Dijkstra, E. W. (1982). How do we tell truths that might hurt? In *Selected Writings on Computing: A Personal Perspective* (pp. 129–131). Springer.
- Dijkstra, E. W. (1989). On the cruelty of really teaching computing science. *Communications of the ACM*, 32(12), 1398–1404.
- DiSalvo, B., Guzdial, M., Bruckman, A., & 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), 272–315.
- diSessa, A. A. (2000). *Changing minds: Computers, learning, and literacy*. Cambridge, MA: MIT Press.
- diSessa, A. A., & Abelson, H. (1986). Boxer: a reconstructible computational medium. *Communications of the ACM*, 29(9), 859–868.
- diSessa, A. A., Hammer, D., Sherin, B. L., & Kolpakowski, T. (1991). Inventing graphing: Meta-representational expertise in children. *Journal of Mathematical Behavior*, 10, 117–160.
- Donzeau-Gouge, V., Huet, G., Lang, B., & Kahn, G. (1984). Programming environments based on structured editors: The MENTOR experience. In D. Barstow, H. E. Shrobe, & E. Sandewall (Eds.), *Interactive Programming Environments*. McGraw Hill.
- Doukakis, D., Grigoriadou, M., & Tsaganou, G. (2007). Understanding the programming variable concept with animated interactive analogies. In *Proceedings of the The 8th Hellenic European Research on Computer Mathematics & Its Applications Conference (HERCMA '07)*.

- du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1), 57–73.
- Duncan, C., Bell, T., & Tanimoto, S. (2014). Should Your 8-year-old Learn Coding? In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education* (pp. 60–69). New York, NY, USA: ACM. <http://doi.org/10.1145/2670757.2670774>
- Education Policy Committee, A. (2014). *Rebooting the Pathway to Success Preparing Students for Computing Workforce Needs in the United States*. Association for Computing Machinery.
- Eisenberg, M. (2003). Mindstuff Educational Technology Beyond the Computer. *Convergence: The International Journal of Research into New Media Technologies*, 9(2), 29–53.
- Erickson, F. (1986). Qualitative methods in research on teaching. In M. C. Wittrock (Ed.), *Handbook of research on teaching* (pp. 119–161). New York: Macmillan.
- Fagin, B., & Merkle, L. (2003). Measuring the effectiveness of robots in teaching computer science. In *ACM SIGCSE Bulletin* (Vol. 35, pp. 307–311). ACM.
- Feinstein, A. R., & Cicchetti, D. V. (1990). High agreement but low Kappa: I. the problems of two paradoxes. *Journal of Clinical Epidemiology*, 43(6), 543–549.
- Felleisen, M., Findler, R. B., Flatt, M., & Krishnamurthi, S. (2004). The TeachScheme! project: Computing and programming for every student. *Computer Science Education*, 14(1), 55–77.
- Feurzeig, W., Papert, S., Bloom, M., Grant, R., & Solomon, C. (1969). *Programming-languages as a conceptual framework for teaching mathematics* (BBN Report No. 1889). Cambridge, MA: Bolt, Beranek, and Newman.

- Feurzeig, W., Papert, S., & Lawler, B. (2011). Programming-languages as a conceptual framework for teaching mathematics. *Interactive Learning Environments*, 19(5), 487–501.
- Fields, D., Giang, M., & Kafai, Y. (2014). Programming in the wild: trends in youth computational participation in the online scratch community. In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education* (pp. 2–11). ACM Press.
- Findler, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., & Felleisen, M. (2002). DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2), 159–182.
- Fisher, A., Margolis, J., & Miller, F. (1997). Undergraduate women in computer science: experience, motivation and culture. In *ACM SIGCSE Bulletin* (Vol. 29, pp. 106–110). ACM.
- Flannery, L. P., Silverman, B., Kazakoff, E. R., Bers, M. U., Bontá, P., & Resnick, M. (2013). Designing ScratchJr: Support for early childhood learning through computer programming. In *Proceedings of the 12th International Conference on Interaction Design and Children* (pp. 1–10). ACM.
- Flowers, T., Carver, C. A., & Jackson, J. (2004). Empowering students and building confidence in novice programmers through Gauntlet. In *Frontiers in Education, 2004. FIE 2004. 34th Annual* (p. T3H/10-T3H/13 Vol. 1). <http://doi.org/10.1109/FIE.2004.1408551>
- Flowers, T. R., & Gossett, K. A. (2002). Teaching problem solving, computing, and information technology with robots. *Journal of Computing Sciences in Colleges*, 17(6), 45–55.
- Fraser, N. (2013). *Blockly*. <https://developers.google.com/blockly/>: Google.

- Garcia, D., Harvey, B., & Barnes, T. (2015). The Beauty and Joy of Computing. *ACM Inroads*, 6(4), 71–79. <http://doi.org/10.1145/2835184>
- Garlan, D. B., & Miller, P. L. (1984). GNOME: An introductory programming environment based on a family of structure editors. *ACM Sigplan Notices*, 19(5), 65–72.
- Garlick, R., & Cankaya, E. C. (2010). Using Alice in CS1: A quantitative experiment. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education* (pp. 165–168). ACM.
- Gibson, J. J. (1986). *The ecological approach to visual perception*. Psychology Press.
- Goldenberg, E. P., & Feurzeig, W. (1987). *Exploring language with Logo*. Cambridge, MA: MIT Press.
- Goldman, R., Eguchi, A., & Sklar, E. (2004). Using educational robotics to engage inner-city students with technology. In *Proceedings of the Sixth International Conference of the Learning Sciences (ICLS)* (pp. 214–221).
- Goode, J., Chapman, G., & Margolis, J. (2012). Beyond curriculum: the exploring computer science program. *ACM Inroads*, 3(2), 47–53.
- Goodman, D. (1988). *The Complete HyperCard Handbook* (2nd ed.). New York: Bantam Books.
- Green, T. R. G. (1977). Conditional program statements and their comprehensibility to professional programmers. *Journal of Occupational Psychology*, 50(2), 93–109.
- Green, T. R. G., & Petre, M. (1992). When visual programs are harder to read than textual programs. In *Human-Computer Interaction: Tasks and Organisation, Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics)*. GC van der Veer, MJ Tauber, S. Bagnarola and M. Antavolits. Rome, CUD. Citeseer.

- Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming environments: A “cognitive dimensions” framework. *Journal of Visual Languages and Computing*, 7(2), 131–174.
- Green, T. R. G., Petre, M., & Bellamy, R. K. E. (1991). Comprehensibility of visual and textual programs: A test of superlativism against the ‘match-mismatch’ conjecture. *ESP*, 91(743), 121–146.
- Grover, S., Cooper, S., & Pea, R. (2014). Assessing computational learning in K-12 (pp. 57–62). ACM Press. <http://doi.org/10.1145/2591708.2591713>
- Grover, S., & Pea, R. (2013). Computational Thinking in K-12: A Review of the State of the Field. *Educational Researcher*, 42(1), 38–43.
- Guzdial, M. (2004). Programming environments for novices. *Computer Science Education Research*, 2004, 127–154.
- Guzdial, M. (2010). Does contextualized computing education help? *ACM Inroads*, 1(4), 4–6.
- Guzdial, M. (2015). *Learner-Centered Design of Computing Education: Research on Computing for Everyone* (Vol. 8). Retrieved from <http://www.morganclaypool.com/doi/10.2200/S00684ED1V01Y201511HCI033>
- Guzdial, M., & Ericson, B. (2009). *Introduction to computing and programming in Python, a multimedia approach*. Prentice Hall Press. Retrieved from <http://dl.acm.org/citation.cfm?id=1695818>
- Hancock, C. M. (2003). *Real-time programming and the big ideas of computational literacy*. Citeseer.
- Harel, I., & Papert, S. (Eds.). (1991). *Constructionism*. Norwood N.J.: Ablex Publishing.

- Hartmann, B., MacDougall, D., Brandt, J., & Klemmer, S. R. (2010). What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 1019–1028). ACM.
- Harvey, B. (1997). *Computer science logo style: Beyond programming* (Vols. 1–3). The MIT Press.
- Harvey, B., & Mönig, J. (2010). Bringing “no ceiling” to Scratch: Can one language serve kids and computer scientists? In J. Clayson & I. Kalas (Eds.), *Proceedings of Constructionism 2010 Conference* (pp. 1–10). Paris, France.
- Henriksen, P., & Kölling, M. (2004). Greenfoot: Combining object visualisation with interaction. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (pp. 73–82).
- Hils, D. D. (1992). Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, 3(1), 69–101.
- Holbert, N., & Wilensky, U. (2011). FormulaT Racing: Designing a Game for Kinematic Exploration and Computational Thinking. In *Proceedings of the 7th Games, Learning, & Society Conference*. Madison, WI.
- Holt, R. C., & Cordy, J. R. (1988). The Turing Programming Language. *Commun. ACM*, 31(12), 1410–1423. <http://doi.org/10.1145/53580.53581>
- Homer, M., & Noble, J. (2014). Combining Tiled and Textual Views of Code. In *IEEE Working Conference on Software Visualisation (VISSOFT)* (pp. 1–10). Victoria, BC: IEEE.
- Hopscotch. (2014). New York, NY: Hopscotch. Retrieved from <http://www.gethopscotch.com/>

- Horn, M. S. (2013). The role of cultural forms in tangible interaction design. In *Proceedings of the 7th International Conference on Tangible, Embedded and Embodied Interaction* (pp. 117–124). New York, NY, USA: ACM. <http://doi.org/10.1145/2460625.2460643>
- Horn, M. S., AlSulaiman, S., & Koh, J. (2013). Translating Roberto to Omar: computational literacy, stickerbooks, and cultural forms. In *Proceedings of the 12th International Conference on Interaction Design and Children* (pp. 120–127). New York, NY.
- Horn, M. S., & Jacob, R. J. K. (2007). Tangible programming in the classroom with tern. In *CHI '07 extended abstracts on Human factors in computing systems* (pp. 1965–1970). New York, NY, USA: ACM.
- Horn, M. S., Solovey, E. T., Crouser, R. J., & Jacob, R. J. . (2009). Comparing the use of tangible and graphical programming languages for informal science education. In *Proceedings of the 27th international conference on Human factors in computing systems* (pp. 975–984). ACM Press.
- Horn, M. S., Weintrop, D., Beheshti, E., & Olson, I. C. (2012). Spinners, dice, and pawns: Using board games to prepare for agent-based modeling activities. Presented at the AERA, Vancouver, Canada.
- Horn, M. S., Weintrop, D., & Routman, E. (2014). Programming in the Pond: A Tabletop Computer Programming Exhibit. In *Proceedings of the Extended Abstracts of the 32nd Annual ACM Conference on Human Factors in Computing Systems* (pp. 1417–1422). New York, NY, USA: ACM. <http://doi.org/10.1145/2559206.2581237>
- Horstmann, C. S. (2012). *Java Concepts: Early Objects* (7 edition). Hoboken, NJ: Wiley.
- Hour of Code. (2013). Code.org. Retrieved from <http://code.org/learn>

- Howland, K., & Good, J. (2014). Learning to communicate computationally with flip: A bimodal programming language for game creation. *Computers & Education*.
- Hristova, M., Misra, A., Rutter, M., & Mercuri, R. (2003). Identifying and correcting Java programming errors for introductory computer science students. In *ACM SIGCSE Bulletin* (Vol. 35, pp. 153–156). ACM.
- Hundhausen, C. D., Farley, S. F., & Brown, J. L. (2009). Can direct manipulation lower the barriers to computer programming and promote transfer of training? *ACM Transactions on Computer-Human Interaction*, 16(3), 1–40. <http://doi.org/10.1145/1592440.1592442>
- Hutchins, E. L., Hollan, J. D., & Norman, D. A. (1985). Direct manipulation interfaces. *Human-Computer Interaction*, 1(4), 311–338.
- Ioannidou, A., Repenning, A., & Webb, D. C. (2009). AgentCubes: Incremental 3D end-user development. *Journal of Visual Languages & Computing*, 20(4), 236–251.
- Jadud, M. C. (2005). A first look at novice compilation behaviour using BlueJ. *Computer Science Education*, 15(1), 25–40.
- Jadud, M. C., & Henriksen, P. (2009). Flexible, reusable tools for studying novice programmers. In *Proceedings of the fifth international workshop on Computing education research workshop* (pp. 37–42). ACM.
- Jamieson, P. (2010). Arduino for teaching embedded systems. Are computer scientists and engineering educators missing the boat? *Proc. FECS*, 289–294.
- Johnsgard, K., & McDonald, J. (2008). Using Alice in Overview Courses to Improve Success Rates in Programming I. In *IEEE 21st Conference on Software Engineering Education and Training, 2008. CSEET '08* (pp. 129–136). <http://doi.org/10.1109/CSEET.2008.35>

- Johnson, G. W. (1997). *LabVIEW graphical programming: practical applications in instrumentation and control*. McGraw-Hill School Education Group.
- Jona, K., Penney, L., & Stevens, R. (2015). "Re-mediating" Learning. In *Proceedings of the 11th Annual Conference on Computer Supported Collaborative Learning (CSCL)*. Gothenburg, Sweden.
- Kafai, Y. B., Peppler, K. A., & Chapman, R. N. (2009). *The Computer Clubhouse: Constructionism and Creativity in Youth Communities. Technology, Education--Connections*. Teachers College Press.
- Kahn, K. (1999). From prolog to Zelda to ToonTalk. In *Proceedings of the International Conference on Logic Programming* (pp. 67–78).
- Kaput, J., Noss, R., & Hoyles, C. (2002). Developing new notations for a learnable mathematics in the computational era. *Handbook of International Research in Mathematics Education*, 51–75.
- Kay, A. (2005). Squeak etoys, children & learning. *Online Article, 2006*. Retrieved from ftp://ofset2.unix-ag.uni-kl.de/speeches/jrfdez/malaga08/doc/etoys_n_learning.pdf
- Kay, A., & Goldberg, A. (1977). Personal dynamic media. *Computer*, 10(3), 31–41.
- Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*, 37(2), 83–137.
- Kelleher, C., Pausch, R., & Kiesler, S. (2007). Storytelling alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 1455–1464).
- Kemeny, J. G., & Kurtz, T. E. (1980). *Basic programming*. John Wiley & Sons, Inc.

- Koh, K. H., Basawapatna, A., Nickerson, H., & Repenning, A. (2014). Real Time Assessment of Computational Thinking. In *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on* (pp. 49–52). IEEE. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6883021
- Kölling, M., Brown, N. C. C., & Altadmri, A. (2015). Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education* (pp. 29–38). New York, NY, USA: ACM. <http://doi.org/10.1145/2818314.2818331>
- Kölling, M., & McKay, F. (2016). Heuristic Evaluation for Novice Programming Systems. *Trans. Comput. Educ.*, 16(3), 12:1–12:30. <http://doi.org/10.1145/2872521>
- Kölling, M., Quig, B., Patterson, A., & Rosenberg, J. (2003). The BlueJ System and its Pedagogy. *Computer Science Education*, 13(4), 249–268.
- Kölling, M., & Rosenberg, J. (1996). Blue—a language for teaching object-oriented programming. In *ACM SIGCSE Bulletin* (Vol. 28, pp. 190–194). ACM.
- Lave, J. (1988). *Cognition in practice: Mind, mathematics, and culture in everyday life*. Cambridge Univ Pr.
- Lee, I., Martin, F., & Apone, K. (2014). Integrating computational thinking across the K–8 curriculum. *ACM Inroads*, 5(4), 64–71.
- Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., ... Werner, L. (2011). Computational thinking for youth in practice. *ACM Inroads*, 2(1), 32–37.
- Lee, T. Y., Mauriello, M. L., Ahn, J., & Bederson, B. B. (2014). CTArcade: Computational thinking with games in school age children. *International Journal of Child-Computer Interaction*. <http://doi.org/10.1016/j.ijCCI.2014.06.003>

- Lego Systems Inc. (2008). *Lego Mindstorms NXT-G Invention System*. Retrieved from
<http://mindstorms.lego.com>
- Levy, S. T., & Mioduser, D. (2007). Does it “want” or “was it programmed to...”? Kindergarten children’s explanations of an autonomous robot’s adaptive functioning. *International Journal of Technology and Design Education*, 18(4), 337–359.
- Levy, S. T., & Wilensky, U. (2011). Mining students’ inquiry actions for understanding of complex systems. *Computers & Education*, 56(3), 556–573.
- Lewis, C. M. (2010). How programming environment shapes perception, learning and goals: Logo vs. Scratch. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (pp. 346–350). New York, NY.
- Lewis, J., & DePasquale, P. (2008). *Programming with Alice and Java*. Boston: Addison-Wesley.
- Luria, A. R. (1982). *Language and cognition*. (J. V. Wertsch, Ed.). Winston ; Wiley, Washington, D.C. : New York ; Chichester :
- Made with Code. (2014). Google. Retrieved from <https://www.madewithcode.com/>
- Malan, D. J., & Leitner, H. H. (2007). Scratch for budding computer scientists. In *ACM SIGCSE Bulletin* (Vol. 39, pp. 223–227). ACM.
- Maloney, J. H., Peppler, K., Kafai, Y., Resnick, M., & Rusk, N. (2008). Programming by choice: Urban youth learning programming with Scratch. *ACM SIGCSE Bulletin*, 40(1), 367–371.
- Maloney, J. H., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4), 16.

- Margolis, J. (2008). *Stuck in the shallow end: Education, race, and computing*. The MIT Press.
- Margolis, J., & Fisher, A. (2003). *Unlocking the clubhouse: Women in computing*. The MIT Press.
- Martin, F., Mikhak, B., Resnick, M., Silverman, B., & Berg, R. (2000). To Mindstorms and Beyond. *Robots for Kids: Exploring New Technologies for Learning*, 9.
- Matsuzawa, Y., Ohata, T., Sugiura, M., & Sakai, S. (2015). Language Migration in non-CS Introductory Programming through Mutual Language Translation Environment. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 185–190). ACM Press. <http://doi.org/10.1145/2676723.2677230>
- McNerney, T. (2004). From turtles to Tangible Programming Bricks: explorations in physical language design. *Personal and Ubiquitous Computing*, 8(5).
<http://doi.org/10.1007/s00779-004-0295-6>
- Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2011). Habits of programming in Scratch. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education* (pp. 168–172). Darmstadt, Germany: ACM.
- Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. M. (2010). Learning computer science concepts with scratch. In *Proceedings of the Sixth international workshop on Computing education research* (pp. 69–76).
- Mendelsohn, P., Green, T. R. G., & Brna, P. (1990). *Programming languages in education: The search for an easy start*. Academic Press London.

- Miller, P., Pane, J., Meter, G., & Vorthmann, S. (1994). Evolution of novice programming environments: the structure editors of Carnegie Mellon University. *Interactive Learning Environments*, 4(2), 140–158.
- Modrow, E., Mönig, J., & Strecker, K. (2011). Wozu JAVA? *LOG IN*, 31(168), 35–41.
- Moher, T. G., Mak, D. C., Blumenthal, B., & Levanthal, L. M. (1993). Comparing the comprehensibility of textual and graphical programs. In *Empirical Studies of Programmers: Fifth Workshop* (pp. 137–161). Ablex, Norwood, NJ.
- Mönig, J., Ohshima, Y., & Maloney, J. (2015). Blocks at your fingertips: Blurring the line between blocks and text in GP. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)* (pp. 51–53). <http://doi.org/10.1109/BLOCKS.2015.7369001>
- Moskal, B., Lurie, D., & Cooper, S. (2004). Evaluating the effectiveness of a new instructional approach. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education* (pp. 75–79).
- Motil, J., & Epstein, D. (1998). JJ: a Language Designed for Beginners. Retrieved from <http://www.csun.edu/~jmotil/BeginLanguageJr.pdf>
- Mullins, P., Whitfield, D., & Conlon, M. (2009). Using Alice 2.0 as a first language. *Journal of Computing Sciences in Colleges*, 24(3), 136–143.
- Myers, B. A. (1990). Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, 1(1), 97–123.
- Nemirovsky, R. (1994). On ways of symbolizing: The case of Laura and the velocity sign. *The Journal of Mathematical Behavior*, 13(4), 389–422.
- Nienaltowski, M.-H., Pedroni, M., & Meyer, B. (2008). Compiler error messages: What can help novices? In *ACM SIGCSE Bulletin* (Vol. 40, pp. 168–172). ACM.

- Nishida, T., Kanemune, S., Idosaka, Y., Namiki, M., Bell, T., & Kuno, Y. (2009). A CS unplugged design pattern. In *ACM SIGCSE Bulletin* (Vol. 41, pp. 231–235). ACM.
- Norman, D. A. (1990). *The design of everyday things*. New York: Doubleday.
- Norman, D. A. (1991). Cognitive artifacts. In J. M. Carroll (Ed.), *Designing interaction: Psychology at the human-computer interface*. New York, NY: Cambridge University Press.
- Norman, D. A. (1993). *Things that make us smart: Defending human attributes in the age of the machine*. Basic Books.
- Noss, R., & Hoyles, C. (1996). *Windows on mathematical meanings: Learning cultures and computers*. Dordrecht: Kluwer.
- Ong, W. (1982). *Orality and Literacy: The technologizing of the world*. London: Routledge.
- Palmer, S. E. (1978). Fundamental aspects of cognitive representation. In E. Rosch & B. B. Lloyd (Eds.), *Cognition and categorization* (Vol. 259, pp. 259–303). Hillsdale, N.J.: Lawrence Erlbaum Associates.
- Pane, J., & Miller, P. (1993). The ACSE multimedia science learning environment. In *Proceedings of the 1993 International Conference on Computers in Education* (pp. 168–173).
- Papadimitriou, C. H. (2003). MythematiCS: in praise of storytelling in the teaching of computer science and math. *SIGCSE BULLETIN*, 35(4), 7–9.
- Papastergiou, M. (2009). Digital game-based learning in high school computer science education: Impact on educational effectiveness and student motivation. *Computers & Education*, 52(1), 1–12.

- Papert, S. (1972). Teaching Children to be Mathematicians Versus Teaching About Mathematics. *International Journal of Mathematical Education in Science and Technology*, 3(3), 249–262.
- Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. New York: Basic books.
- Papert, S. (1993). *The children's machine: Rethinking school in the age of the computer*. New York: Basic Books.
- Papert, S. (2006). Afterword: After how comes What. In R. K. Sawyer (Ed.), *The Cambridge Handbook of the Learning Sciences* (pp. 581 – 586). Cambridge University Press.
- Parsons, D., & Haden, P. (2007). Programming osmosis: Knowledge transfer from imperative to visual programming environments. In S. Mann & N. Bridgeman (Eds.), *Proceedings of The Twentieth Annual NACCQ Conference* (pp. 209–215). Hamilton, New Zealand.
- Pattis, R. E. (1981). *Karel the robot: a gentle introduction to the art of programming*. John Wiley & Sons, Inc.
- Pea, R. D. (1986). Language-independent conceptual“ bugs” in novice programming. *Journal of Educational Computing Research*, 2(1), 25–36.
- Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., ... Paterson, J. (2007). A survey of literature on the teaching of introductory programming. In *ACM SIGCSE Bulletin* (Vol. 39, pp. 204–223). ACM.
- Perlis, A. J. (1962). *The computer in the university*. MIT Press.
- Petre, M. (1995). Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, 38(6), 33–44.

- PicoBlocks. (2008). Playful Invention Company. Retrieved from
<http://www.picocricket.com/download.html>
- Piech, C., Sahami, M., Koller, D., Cooper, S., & Blikstein, P. (2012). Modeling how students learn to program. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education* (pp. 153–160). ACM.
- Powers, K., Ecott, S., & Hirshfield, L. M. (2007). Through the looking glass: teaching CS0 with Alice. *ACM SIGCSE Bulletin*, 39(1), 213–217.
- Price, T. W., & Barnes, T. (2015). Comparing Textual and Block Interfaces in a Novice Programming Environment (pp. 91–99). Presented at the ICER '15, ACM Press.
- Reed, D., Wilkerson, B., Yanek, D., Dettori, L., & Solin, J. (2015). How exploring computer science (ECS) came to Chicago. *ACM Inroads*, 6(3), 75–77.
- Repenning, A., Ioannidou, A., & Zola, J. (2000). AgentSheets: End-user programmable simulations. *Journal of Artificial Societies and Social Simulation*, 3(3).
- Repenning, A., & Sumner, T. (1995). Agentsheets: A medium for creating domain-oriented visual languages. *IEEE Computer*, 28(3), 17–25.
- Resnick, M. (1997). *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*. A Bradford Book.
- Resnick, M., & Rusk, N. (1996). The Computer Clubhouse: Preparing for life in a digital world. *IBM Systems Journal*, 35(3.4), 431–439. <http://doi.org/10.1147/sj.353.0431>
- Resnick, M., Silverman, B., Kafai, Y., Maloney, J., Monroy-Hernández, A., Rusk, N., ... Silver, J. (2009). Scratch: Programming for all. *Communications of the ACM*, 52(11), 60.

- Resnick, M., & Wilensky, U. (1993). Beyond the deterministic, centralized mindsets: New thinking for new sciences. In *annual meeting of the American Educational Research Association, Atlanta, GA*.
- Robins, A., Rountree, J., & Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, 13(2), 137.
- Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2), 137–172.
- Roque, R., Kafai, Y., & Fields, D. (2012). From tools to communities: Designs to support online creative collaboration in Scratch. In *Proceedings of the 11th International Conference on Interaction Design and Children* (pp. 220–223). ACM.
- Roque, R. V. (2007). *OpenBlocks: An extendable framework for graphical block programming systems* (Master's Thesis). Massachusetts Institute of Technology.
- Rücker, M. T., & Pinkwart, N. (2015). Review and Discussion of Children's Conceptions of Computers. *Journal of Science Education and Technology*.
- Ryoo, J. J., Margolis, J., Lee, C. H., Sandoval, C. D., & Goode, J. (2013). Democratizing computer science knowledge: transforming the face of computer science through public high school education. *Learning, Media and Technology*, 38(2), 161–181.
- Sammet, J. E. (1981). The Early History of COBOL. In R. L. Wexelblat (Ed.), *History of Programming Languages I* (pp. 199–243). New York, NY, USA: ACM.
- Santori, M. (1990). An instrument that isn't really (Laboratory Virtual Instrument Engineering Workbench). *IEEE Spectrum*, 27(8), 36–39. <http://doi.org/10.1109/6.58432>
- Saxe, G. B. (1999). Cognition, development, and cultural practices. In T. E. (Ed.), *Culture and Development. New Directions in Child Psychology*. SF: Jossey-Bass.

- Scholtz, J., & Wiedenbeck, S. (1990). Learning second and subsequent programming languages: A problem of transfer. *International Journal of Human-Computer Interaction*, 2(1), 51–72.
- Schweikardt, E., & Gross, M. D. (2006). roBlocks: a robotic construction kit for mathematics and science education. In *Proceedings of the 8th international conference on Multimodal interfaces* (pp. 72–75). ACM.
- ScratchBlocks*. (2014). Retrieved from <https://github.com/blob8108/scratchblocks2>
- Scribner, S., & Cole, M. (1981). *The psychology of literacy* (Vol. 198). Harvard University Press Cambridge, MA.
- Sengupta, P., & Wilensky, U. (2009). Learning Electricity with NIELS: Thinking with Electrons and Thinking in Levels. *International Journal of Computers for Mathematical Learning*, 14(1), 21–50.
- Settle, A., Goldberg, D. S., & Barr, V. (2013). Beyond computer science: computational thinking across disciplines. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education* (pp. 311–312). New York, NY, USA: ACM.
- Shapiro, R. B., & Ahrens, M. (2016). Beyond Blocks: Syntax and Semantics. *Commun. ACM*, 59(5), 39–41. <http://doi.org/10.1145/2903751>
- Sherin, B. L. (2000). How students invent representations of motion: A genetic account. *The Journal of Mathematical Behavior*, 19(4), 399–441.
- Sherin, B. L. (2001). A comparison of programming languages and algebraic notation as expressive languages for physics. *International Journal of Computers for Mathematical Learning*, 6(1), 1–61.

- Sherin, B. L., diSessa, A. A., & Hammer, D. (1993). Dynaturtle revisited: Learning physics through collaborative design of a computer model. *Interactive Learning Environments*, 3(2), 91–118.
- Sipitakiat, A., Blikstein, P., & Cavallo, D. P. (2004). GoGo board: augmenting programmable bricks for economically challenged audiences. In *Proceedings of the 6th international conference on Learning sciences* (pp. 481–488).
- Sipitakiat, A., & Nusen, N. (2012). Robo-Blocks: designing debugging abilities in a tangible programming system for early primary school children. In *Proceedings of the 11th International Conference on Interaction Design and Children* (pp. 98–105).
- Slany, W. (2014). Tinkering with Pocket Code, a Scratch-like programming app for your smartphone. In *Proceedings of Constructionism 2014*. Vienna, Austria.
- Sleeman, D., & Brown, J. S. (1982). Intelligent tutoring systems. Retrieved from <https://hal.archives-ouvertes.fr/hal-00702997/>
- Sleeman, D., Putnam, R. T., Baxter, J., & Kuspa, L. (1986). Pascal and high school students: A study of errors. *Journal of Educational Computing Research*, 2(1), 5–23.
- Smith, D. C. (1977). *Pygmalion: A Computer Program to Model and Stimulate Creative Thought*. Birkhäuser.
- Smith, D. C., Cypher, A., & Spohrer, J. (1994). KidSim: Programming agents without a programming language. *Communications of the ACM*, 37(7), 54–67.
- Smith, D. C., Cypher, A., & Tesler, L. (2000). Programming by example: novice programming comes of age. *Communications of the ACM*, 43(3), 75–81.

- Smith, D. C., Cypher, A., & Tesler, L. (2001). Novice Programming Comes of Age. In H. Lieberman (Ed.), *Your wish is my command: Programming by example* (pp. 7–20). Morgan Kaufmann.
- Smith, N., Sutcliffe, C., & Sandvik, L. (2014). Code Club: Bringing Programming to UK Primary Schools Through Scratch. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (pp. 517–522). New York, NY, USA: ACM.
- Soloway, E., Guzdial, M., & Hay, K. E. (1994). Learner-centered design: The challenge for HCI in the 21st century. *Interactions*, 1(2), 36–48.
- Sorva, J. (2008). The same but different students' understandings of primitive and object variables. In *Proceedings of the 8th International Conference on Computing Education Research* (pp. 5–15). ACM. Retrieved from <http://dl.acm.org/citation.cfm?id=1595360>
- Sorva, J. (2012). *Visual Program Simulation in Introductory Programming Education*. Aalto University, Espoo, Finland.
- Squire, K. (2005). Changing the game: What happens when video games enter the classroom. *Innovate: Journal of Online Education*, 1(6).
- Stead, A., & Blackwell, A. F. (2014). Learning Syntax as Notational Expertise when using DrawBridge. Presented at the Psychology of Programming Interest Group, University of Sussex.
- Stefik, A., & Gellenbeck, E. (2011). Empirical studies on programming language stimuli. *Software Quality Journal*, 19(1), 65–99.
- Stefik, A., & Hanenberg, S. (2014). The Programming Language Wars: Questions and Responsibilities for the Programming Language Community. In *Proceedings of the 2014*

- ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (pp. 283–299). New York, NY, USA: ACM.
- Stefik, A., & Siebert, S. (2013). An Empirical Investigation into Programming Language Syntax. *ACM Transactions on Computing Education*, 13(4), 1–40.
- Stieff, M., & Wilensky, U. (2003). Connected chemistry—incorporating interactive simulations into the chemistry classroom. *Journal of Science Education and Technology*, 12(3), 285–302.
- Stonedahl, F., Wilkerson-Jerde, M., & Wilensky, U. (2010). MAgICS: Toward a Multi-Agent Introduction to Computer Science. *Multi-Agent Systems for Education and Interactive Entertainment: Design, Use and Experience*, 1–25.
- Strauss, A., & Corbin, J. (1994). Grounded Theory Methodology: An Overview. In *Strategies of Qualitative Inquiry* (pp. 158–183). Thousand Oaks, CA: Sage Publications, Inc.
- Sudol, L. A., & Studer, C. (2010). Analyzing Test Items: Using Item Response Theory to Validate Assessments. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (pp. 436–440). New York, NY, USA: ACM.
- Suppes, P. (1966). The Uses of Computers in Education. *Scientific American*, 215, 206–220.
- Suzuki, H., & Kato, H. (1995). Interaction-level Support for Collaborative Learning: AlgoBlock—an Open Programming Language. In *The First International Conference on Computer Support for Collaborative Learning* (pp. 349–355). Hillsdale, NJ, USA: L. Erlbaum Associates Inc. <http://doi.org/10.3115/222020.222828>
- Swetz, F. (1989). *Capitalism and arithmetic: The new math of the 15th century*. La Salle, Illinois: Open Court.

- Tangney, B., Oldham, E., Conneely, C., Barrett, S., & Lawlor, J. (2010). Pedagogy and processes for a computer programming outreach workshop—The bridge to college model. *Education, IEEE Transactions on*, 53(1), 53–60.
- Taylor, C., Zingaro, D., Porter, L., Webb, K. C., Lee, C. B., & Clancy, M. (2014). Computer science concept inventories: past and future. *Computer Science Education*, 24(4), 253–276.
- Teitelbaum, T., & Reps, T. (1981). The Cornell program synthesizer: a syntax-directed programming environment. *Communications of the ACM*, 24(9), 563–573.
- Tempel, M. (2013). Blocks Programming. *CSTA Voice*, 9(1).
- Tew, A. E., & Dorn, B. (2013). The Case for Validated Tools in Computer Science Education Research. *Computer*, 46(9), 60–66. <http://doi.org/10.1109/MC.2013.259>
- Tew, A. E., & Guzdial, M. (2010). Developing a validated assessment of fundamental CS1 concepts. In *Proceedings of the 41st ACM technical symposium on Computer science education* (pp. 97–101).
- Tew, A. E., & Guzdial, M. (2011). The FCS1: a language independent assessment of CS1 knowledge. In *Proceedings of the 42nd ACM technical symposium on Computer science education* (pp. 111–116). ACM.
- Traver, V. J. (2010). On compiler error messages: what they say and what they mean. *Advances in Human-Computer Interaction*, 2010. Retrieved from <http://www.hindawi.com/journals/ahci/2010/602570/abs/>
- Turkle, S., & Papert, S. (1990). Epistemological pluralism: Styles and voices within the computer culture. *SIGNS: : Journal of Women in Culture and Society*, 16(1), 128–157.
- Tynker. (2014). Mountain View, CA: Tynker. Retrieved from <http://www.tynker.com/>

- Van Deursen, A., Klint, P., & Visser, J. (2000). Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6), 26–36.
- Vossoughi, S., & Bevan, B. (2014). Making and Tinkering: A Review of the Literature. *National Research Council Committee on Out of School Time STEM*. Washington, DC: National Research Council, 1–55.
- Vygotsky, L. (1986). *Thought and language*. Cambridge, MA: MIT Press.
- Wagh, A., & Wilensky, U. (2012). Evolution in blocks: Building models of evolution using blocks. In C. Kynigos, J. Clayson, & N. Yiannoutsou (Eds.), *Proceedings of the Constructionism 2012 Conference*. Athens, Greece.
- Weintrop, D., Beheshti, E., Horn, M., Orton, K., Jona, K., Trouille, L., & Wilensky, U. (2016). Defining Computational Thinking for Mathematics and Science Classrooms. *Journal of Science Education and Technology*, 25(1), 127–147. <http://doi.org/10.1007/s10956-015-9581-5>
- Weintrop, D., & Wilensky, U. (2012). RoboBuilder: A program-to-play constructionist video game. In C. Kynigos, J. Clayson, & N. Yiannoutsou (Eds.), *Proceedings of the Constructionism 2012 Conference*. Athens, Greece.
- Weintrop, D., & Wilensky, U. (2013a). Supporting computational expression: How novices use programming primitives in achieving a computational goal. Presented at the American Education Researchers Association, San Francisco, CA, USA.
- Weintrop, D., & Wilensky, U. (2014a). Program-to-play videogames: Developing computational literacy through gameplay. In *Proceedings of the 10th Games, Learning, & Society Conference* (pp. 264–271). Madison, WI.

- Weintrop, D., & Wilensky, U. (2014b). Situating programming abstractions in a constructionist video game. In G. Futschek & C. Kynigos (Eds.), *Proceedings of Constructionism 2014*. Vienna, Austria.
- Weintrop, D., & Wilensky, U. (2015a). The challenges of studying blocks-based programming environments. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)* (pp. 5–7). <http://doi.org/10.1109/BLOCKS.2015.7368989>
- Weintrop, D., & Wilensky, U. (2015b). To Block or Not to Block, That is the Question: Students' Perceptions of Blocks-based Programming. In *Proceedings of the 14th International Conference on Interaction Design and Children* (pp. 199–208). New York, NY, USA: ACM. <http://doi.org/10.1145/2771839.2771860>
- Weintrop, D., & Wilensky, U. (2015c). To Block or not to Block, That is the Question: Students' Perceptions of Blocks-based Programming. In *Proceedings of the 14th International Conference on Interaction Design and Children*. Boston, MA.
- Weintrop, D., & Wilensky, U. (2015d). Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (pp. 101–110). New York, NY, USA: ACM.
- Weintrop, D., & Wilensky, U. (2016a). Bringing Blocks-based Programming into High School Computer Science Classrooms. In *Paper presented at the Annual Meeting of the American Educational Research Association (AERA)*. Washington DC, USA.
- Weintrop, D., & Wilensky, U. (2016b). Cognitive affordances of blocks-based programming in a two dimensional construction space. Presented at the The 46th Annual Meeting of the Jean Piaget Society Annual Meeting, Chicago, IL.

- Weintrop, D., & Wilensky, U. J. (2013b). Designing for computational expression: Four principles for the design of learning environments towards computational literacy. In D. J. Loveless, B. Griffith, M. E. Bérci, E. Ortleib, & P. M. Sullivan (Eds.), *Academic knowledge construction and multimodal curriculum development* (pp. 86–110). Hershey, PA: IGI Global.
- Werner, L., Denner, J., Campe, S., & Kawamoto, D. C. (2012). The fairy performance assessment: measuring computational thinking in middle school. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education* (pp. 215–220).
- Werner, L., McDowell, C., & Denner, J. (2013). A first step in learning analytics: pre-processing low-level Alice logging data of middle school students. *Journal of Educational Data Mining*, 5(2), 11–37.
- Whitley, K. N. (1997). Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages & Computing*, 8(1), 109–142.
- Whorf, B. L., Carroll, J. B., & Chase, S. (1956). *Language, thought, and reality: Selected writings of Benjamin Lee Whorf*. MIT press Cambridge, MA.
- Wiedenbeck, S. (1993). An analysis of novice programmers learning a second language. In *Empirical Studies of Programmers: Fifth Workshop: Papers Presented at the Fifth Workshop on Empirical Studies of Programmers, December 3-5, 1993, Palo Alto, CA* (p. 187). Intellect Books.
- Wilensky, U. (1995). Paradox, programming, and learning probability: A case study in a connected mathematics framework. *The Journal of Mathematical Behavior*, 14(2), 253–280.

- Wilensky, U. (1997). *StarLogoT*. Center for Connected Learning and Computer-Based Modeling, Northwestern University. <https://ccl.northwestern.edu/cm/StarLogoT/>.
- Wilensky, U. (1999). *NetLogo*. Evanston, IL: Center for Connected Learning and Computer-Based Modeling, Northwestern University. <http://ccl.northwestern.edu/netlogo>.
- Wilensky, U. (2001). Modeling nature's emergent patterns with multi-agent languages. In *Proceedings of EuroLogo* (pp. 1–6). Linz, Austria.
- Wilensky, U., Brady, C. E., & Horn, M. S. (2014). Fostering Computational Literacy in Science Classrooms. *Commun. ACM*, 57(8), 24–28. <http://doi.org/10.1145/2633031>
- Wilensky, U., & Novak, M. (2010). Teaching and Learning Evolution as an Emergent Process: The BEAGLE project. In R. Taylor & M. Ferrari (Eds.), *Epistemology and Science Education: Understanding the Evolution vs. Intelligent Design Controversy*. New York: Routledge.
- Wilensky, U., Papert, A., Sherin, B., DiSessa, A. A., Kay, A., & Turkle, S. (2005). *Center for Learning and Computation-Based Knowledge (CLiCK)*. Proposal to the National Science Foundation - Science of Learning Center.
- Wilensky, U., & Papert, S. (2006). *Restructurations: Reformulations of knowledge disciplines through new representational forms*. (Manuscript in preparation).
- Wilensky, U., & Papert, S. (2010). Restructurations: Reformulating knowledge disciplines through new representational forms. In J. Clayson & I. Kallas (Eds.), *Proceedings of the Constructionism 2010 conference*. Paris, France.
- Wilensky, U., & Rand, W. (2014). *Introduction to Agent-based Modeling*. Cambridge, MA: MIT Press.

- Wilensky, U., & Reisman, K. (2006). Thinking like a wolf, a sheep, or a firefly: Learning biology through constructing and testing computational theories—an embodied modeling approach. *Cognition and Instruction*, 24(2), 171–209.
- Wilensky, U., & Resnick, M. (1999). Thinking in levels: A dynamic systems approach to making sense of the world. *Journal of Science Education and Technology*, 8(1), 3–19.
- Wilkerson-Jerde, M. H., & Wilensky, U. (2010). Restructuring Change, Interpreting Changes: The DeltaTick Modeling and Analysis Toolkit. In J. Clayson & I. Kalas (Eds.), *Proceedings of the Constructionism 2010 Conference*. Paris, France.
- Wilson, A., & Moffat, D. C. (2010). Evaluating Scratch to introduce younger schoolchildren to programming. *Proceedings of the 22nd Annual Psychology of Programming Interest Group (Universidad Carlos III de Madrid, Leganés, Spain)*.
- Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, 49(3), 33–35.
- Wolber, D., Abelson, H., Spertus, E., & Looney, L. (2011). *App Inventor: Create Your Own Android Apps*. Sebastopol, Calif: O'Reilly Media.
- Wolfram, S. (2002). *A New Kind of Science* (1st ed.). Wolfram Media.
- Yaroslavski, D. (2014). *Lightbot*. Retrieved from <http://lightbot.com>
- Zweben, S., & Bizot, B. (2014). 2013 Taulbee Survey. *COMPUTING*, 26(5).

11. Appendix A – Introductory Curriculum

This section presents the thirteen assignments that the students worked through during the five-week introductory portion of the study. This appendix serves as a supplement to Table 3.1, which provides a high-level overview. The assignments below are presented in the order that were given to students. For each assignment, a text description is given (sometimes with an accompanying image) and a brief statement of the goal of the assignment. Next, we include the exact text that was given to the students during the study. The assignments were designed by the author with the help of the teacher who was participating in the study. The exact format and wording of the assignments is that of the teacher, who reformatted the assignments to fit into the assignment template she uses in her classes.

Quilt

Description

Students will write a program to draw an image of something that represents them. The images they create will then be “stitched” together to form a quilt for the class. A sample quilt patch was shown to the class.

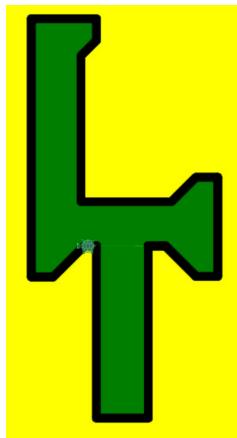


Figure A.1. The sample quilt shown created by the teacher and shown to the students.

Goal

The goal of this assignment is to familiarize students with the Pencil.cc environment, specifically how to assemble and run programs and some of the basic movement and drawing commands.

Text shown to students

You will be using <http://pencil.cc> to create a "quilt," this should be a drawing that represents you and something you care about, you can include multiple things.

To visit an example of a completed quilt go to: <http://share.pencil.cc/home/ccsf0-nudavid-quilt>

- a. Go to <http://pencil.cc>, fill in the drop downs and choose quilt for the assignment
- b. Remember, while you are creating your quilt you are still investigating the workspace. How do you get your turtle to move forward? If you are not sure where can you "quickly" (hint) look? How do you get the turtle to use a pen (to draw)?

When you physically draw on paper do you move forward and then pick your pen or do you first pick your pen and then move forward to draw? Do you think it is the same for coding? Test out both ways and see which works.

MadLibs

Description

This assignment has students author and implement a short “MadLib”. A MadLib is a text game where one person (usually the author of the MadLib) asks the player for a part of speech (“give me a proper noun” or “give me an adjective”). These suggestions are then plugged into the story.

Goal

This assignment is intended to introduce students to the concept of variables and the basic text functionality of Pencil.cc

Text shown to students

Part 1: On a scrape piece of paper with a partner, create an algorithm for how to play MadLibs

You need to gather two adjectives and three nouns.

Does order matter? How can you store users response? How can you use the users response?

Part 2: As individuals, in pencil.cc, choose the MadLib’s drop down and start to program using your algorithm

How can you store users response? How can you use the users response?

Tip Calculator

Description

This assignment asks students to create a basic tip calculating program that prompts the user for a bill amount and a tip percentage, then reports back to the user how much the tip should be.

Goal

This assignment gives students further experience with variables, this time using them to store numbers and doing basic mathematical calculations with variables.

Text shown to students

Write an algorithm to calculate the tip for a meal based on 20%., you must use variables to represent numbers The result should be the amount of tip (example-4 on a \$20 meal)

Paint by Quadrant

Description

In this assignment, students write a program that has the on-screen turtle follow the mouse cursor around the screen. As the turtle moves, it leaves a trail behind it. The color of the trail is determined by the color. The teacher demonstrated this assignment in class so students could see the behavior.

Example shared in class:

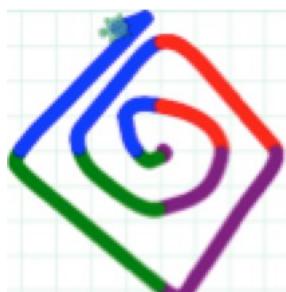


Figure A.2. A static picture of the completed Paint by Quadrant program.

Goal

This assignment is intended to introduce learners to conditional logic, using a visual depiction of how conditional logic can change the outcome of a program.

Text shown to students

[Students are first shown a demonstration of a working program.]

Plan of attack? What's the first thing we should do?

What does that look like? Write down on scrap piece of paper.

When directed, open pencil. What blocks/code could you use to complete action?

Movie Recommendation Engine

Description

This assignment asks students to develop a simple movie recommendation engine. The program will prompt the user with a pair of questions about their movie preferences and then, based on those preferences, recommend a movie to the user.

Goal

The goal of this assignment is for the students to gain further experience using conditional logic, in this case, comparing user input to predefined values to change the output of their program.

Text shown to students

Create an algorithm to suggest a movie to watch on Netflix based on a few (multiple) conditions:

- 1) At least two categories to choose from (ie comedy or action or drama or etc...)
- 2) Once the category is chosen, the user must choose another sub category (ie comedy would be romantic comedy or comedic parodies...)
- 3) End with suggesting a movie (must be school appropriate)

Grade Ranger

Description

This assignment asks student to write a program that will convert a letter grade (like B+) to a percentage (87% - 89%).

Goal

This program is intended to serve as a more complex application of conditional logic by having multiple nested conditions.

Text shown to students

In our program A-, A and A- (etc.) each have a different value so we need conditionals to help sort it out.

1. Ask the user for a letter (A-D)
2. Ask the user to enter either a +, -, or = (= for ‘regular A, + would be B+ and – would look like A-).
3. After taking in these two inputs, the program will tell the user what that range of grades is using if/else or conditionals. Here are the grade ranges: So, if the user types in an ‘A’, then a ‘+’, the program will say: A+ is from 97-100.

The grade ranges are:

A+97-100
 A 93-96
 A-90-92
 Repeat for B, C, D
 F, F- and F+ are all equal to 0-59

Guessing Game

Description

This assignment is a canonical introductory programming activity. Students write a program that randomly picks a number between 1 and 100 and then the user has a fixed number of tries to guess the number. After each guess, the program tells the user if their guess was too high, too low, or correct.

Goal

This assignment introduces learners to iterative logic while also asking them to use conditional logic to determine the appropriate feedback.

Text shown to students

Part 1: You will be programming a game that stores a number in a variable and allows the user to guess the number. Use conditionals to check the two numbers. When the guess is too low, the program says “too low”, when the guess is too high, the program says “too high”, when the user guesses it, the program says – you got it! Write an algorithm for the program.

Part 2: Change the game so the user only gets 7 guesses, so the program stops when the user guesses correctly or if the users uses up all 7 guesses. After the users guesses the number of fails in 7 attempts, ask the user if they want to play again, if yes, then let them play again, if not, end the game.

Radial Art

Description

This assignment has students create a visual pattern that demonstrates radial symmetry. Students were shown the example displayed in Figure A.3.

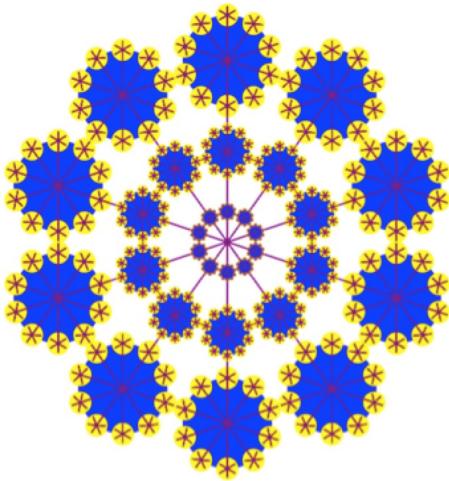


Figure A.3. An example program for the Radial Art assignment.

Goal

The goal for this assignment was to built student intuition about iterative logic.

Text shown to students

Part1: Draw a geometric picture in pencil.cc, use at least 3 colors.

Part 2: Use a loop to repeat the picture

Squiral

Description

Students were asked to draw a pair of “squirls” (shown in Figure A.4).

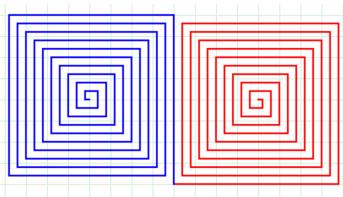


Figure A.4. A pair of squirls.

Goal

This assignment was intended to give students the experience of using loops and variables in conjunction. To draw a squiral, the students need to increment a value over time to produce the visual pattern shown.

Text shown to students

Recreate the following pictures, you may choose either for or while loops and you may pick the color. One squiral starts in and moves out, one squiral starts out and gets smaller

Polygoner

Description

In this assignment, students are given a basic function that draws a square and are asked to modify the function so that it takes in the number of sides and the length of sides as arguments and draws the specified polygon.

Goal

This assignment introduces learners to the concept of functions and parameters.

Text shown to students

Recreate the code in pencil.cc in the polygoner project (shown in), test it out.

What did you notice the program was doing? What is poly? How are sides and len used? What does poly (6, 100) do?

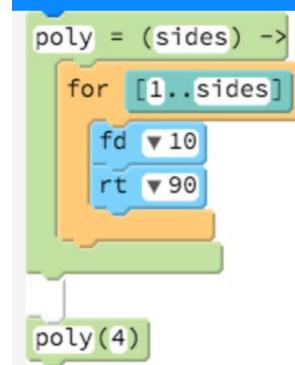
```

poly = (sides) ->
  for [1..sides]
    fd 10
    rt 90

poly(4)

```

(Shown to the Text and Hybrid conditions)



(Shown to the Blocks condition)

Figure A.5. Templates for the Polygoner project.

Working in the polygoner project, first correct the logic so the function draws a polygon of `sides` number of sides. Next, add the option to change the lengths of the sides of the shape and the color of the polygon by adding a new parameters. Don't forget to add this parameter into the existing functions being used!!

Connect 4

Description

This project asks students to draw a Connect 4 board (a grid) and then add the ability for users to play connect four by rendering a black or red dot where the user clicks. No logic with regard to winning or making sure the players followed the rules was required.

Goal

The goal of this project was to ask students to define multiple functions and call them inside of each other.

Text shown to students

Part 1: Create a function that draws 8 lines that are equally spaced apart, use a parameter that will allow the user to make the lines vertical or horizontal.

Part 2: We need to create the code for when the user (player 1) clicks a spot on the grid a black dot is created at that spot, when the user clicks again (player two) a red dot appears in the new spot.

You can starts with this function that has already been created.

```
Click (e) ->
```

```
Moveto e
```

Brick Wall

Description

This penultimate assignment asks students to combine conditional logic, repeating logic, variables, and functions. The assignment prompts the user for a number and then draws a brick wall with that many rows. Figure A.6 shows an example of what an 11 -row brick wall should look like.

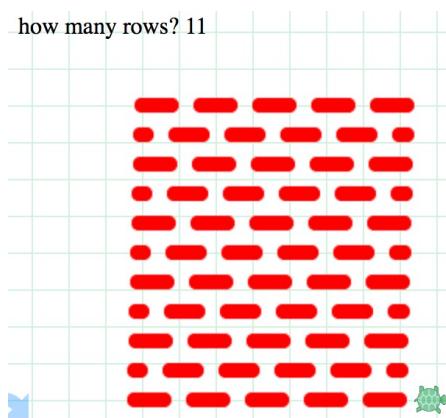


Figure A.6. An 11-row brick wall.

Goal

The goal of this assignment is to ask students to combine the four concepts they have learned over the five week curriculum into a single program.

Text shown to students

- a. Create a function to draw a brick.
- b. Next create a second function that creates a row of bricks across your screen.
- c. Create another function that draws an “odd” row of bricks
- d. Create a function that will let the user choose the number of bricks to be drawn

Final Project

Description

For the final project, students are given a week to create a project that incorporates all the concepts covered since the beginning of the year. On the last day of the curriculum, students share their projects with their peers. Students are given a number of project ideas, including making games, interactive stories, or music videos.

Goal

The goal is for the students to demonstrate their understanding of the ideas covered in the course. Further, like the introductory assignment, this assignment gives students an opportunity to bring their own interests into the classroom.

Text shown to students

Create a game or program that has the following:

- 3 functions with parameters
- `for` loop
- `while` loop

- variables
- variable referencing
- `if/else` conditionals

12. Appendix B – Attitudinal Survey

This appendix entry includes the pre attitudinal survey in its entirety as well as the new questions added for the Mid and Post survey. The survey was administered online at the beginning, middle, and end of the study. The demographic questions (birthday, grade, gender, etc.) were only asked on the Pre assessment.

Pre Attitudinal Survey

- 1) Name
- 2) Student ID
- 3) Programming Class
- 4) Birthday
- 5) Grade
- 6) Gender
- 7) Race/Ethnicity (multi-select box with other free response option)
- 8) What language (or languages) do you speak at home?

The following questions are asked on a 10-point Likert scale and were asked on all three administrations of the survey.

- 9) Programming is fun
- 10) I will be good at programming
- 11) Programming is hard
- 12) I know more than my friends about programming
- 13) Most women can learn to program
- 14) In the future, I would like a job that involves programming
- 15) I like programming
- 16) Programming is a talent - you either have it or you do not
- 17) My family encourages me to learn to program
- 18) Knowing how to program is important
- 19) My friends like using computers
- 20) I can become good at programming
- 21) I like the challenge of programming
- 22) I think programming will be useful in the future
- 23) I cannot learn to program well if the teacher does not explain things well
- 24) I plan to take more computer science courses after this one.
- 25) Computer Science is all about programming

The following set of questions give the student a text field to type in their responses and were only asked on the Pre survey.

- 26) Why are you taking this course?
- 27) What do you hope to learn in this course?
- 28) The thing I am most excited about for this class is:
- 29) Do you have any friends taking this course? If so, how many?
- 30) How did you hear about this course?
- 31) To be successful in programming courses, students need to:

The following three questions were asked on all three administrations of the survey

- 32) I define programming as:
- 33) The most important thing about programming is:
- 34) The hardest thing about programming is:

The following questions are asked on a 10-point Likert scale on all three surveys.

- 35) I will do well in this course
- 36) I am excited about this course
- 37) I think learning to program can help me with other classes
- 38) I think learning to program will help me with things outside of school
- 39) I think about the programs that control the devices I use in my everyday life.

The final set of questions were multiple choice and only asked on the first administration of the survey.

- 40) How much time do you spend on a computer at home each day?
 - a) I don't use a computer
 - b) Less than 1 hour
 - c) Between 1 and 2 hours
 - d) Between 2 and 3 hours
 - e) More than 3 hours
- 41) What do you do on the computer outside of school?
- 42) What types of computational devices do you own/use regularly? *Check all that apply.*
 - a) Laptop computer
 - b) Desktop computer
 - c) Tablet (iPad, Surface, etc.)
 - d) Smartphone (iPhone, Samsun Galaxy, etc)
 - e) Portable Media Player (iPod, portable movie player, etc.)

- f) Game console (Xbox, Play Station, Wii, etc.)
- 43) Have you taken any programming courses previously? If so, what course(s) and when?
- 44) Have you ever used these languages/programming tools? *Check all that apply.*
- a) Scratch or Snap!
 - b) App Inventor
 - c) Alice
 - d) HTML, CSS or Javascript
 - e) Java, C++ or C#
 - f) Python, Lisp or Scheme
 - g) Pencil Code
 - h) Other: _____
- 45) Do you know any professional programmers? If so, who?

Mid Attitudinal Survey

This section only includes new questions to the survey.

- 1) The thing I learned in Pencil.cc that will be most useful in Java is:
- 2) The thing will be the most different about programming in Java compared to programming in Pencil.cc is:
- 3) The thing I like most about Pencil.cc is:
- 4) The thing I like least about Pencil.cc is:

The following questions were asked on a ten-point Likert scale.

- 5) What I learned with Pencil.cc will help me learn Java
- 6) Pencil.cc has made me a better programmer
- 7) I think Pencil.cc was a good use of class time
- 8) Pencil.cc is similar to what real programmers do
- 9) I will do well in this course
- 10) I am excited about this course
- 11) What are variables? How are they used in programs?

The last set of questions on the mid survey asked about the concepts covered in the introductory portion of the course. Each question starts with a 7-point Likert question followed by a free response question.

- 12) How easy was it to use variables in Pencil.cc?
- 13) What do for loops and while loops do? How are they used in programs?
- 14) How easy was it to use loops (for and while) in Pencil.cc?
- 15) What do if and if/else statements do? How are they used in programs?
- 16) How easy was it to use if and if/else statements in Pencil.cc?
- 17) What is a function? How are functions used in programs?
- 18) How easy was it to use functions in Pencil.cc?

Post Attitudinal Survey

This section includes the new questions added to the survey for the final administration.

- 1) The thing I learned in Pencil.cc that was the most useful in Java is:
- 2) The thing that is the most different between Pencil.cc and Java is:
- 3) Now that I am programming in Java, the thing I miss the most about Pencil.cc is:
- 4) Now that I am programming in Java, the thing I miss the least about Pencil.cc is:

The following questions were asked on a 10-point Likert scale.

- 5) What I learned in Pencil.cc has helped me in Java
- 6) Pencil.cc made me a better programmer
- 7) I think Pencil.cc was a good use of class time
- 8) Pencil.cc is similar to what real programmers
- 9) I will do well in this course
- 10) I am excited about what we will be doing the rest of the year in this course
- 11) I am more excited about programming now than I was at the start of the year

13. Appendix C – The Commutative Assessment

This appendix includes a full version of the Commutative Assessment. Details about the design of the assessment and how it was administered can be found in Chapter 3. Every program included in the Commutative Assessment can be displayed in either Pencil Code Blocks, Pencil Code Text, or Snap! Blocks. For the appendix, only one version of each script is provided. The assessment in its entirety, including the instructions and information about the assessment, is presented below exactly as it was shown to students.

Programming Concepts

Take your time on these questions. If you reach a question you do not know the answer to, please make your best educated guess. Your score on this activity will not count towards your grade in the class.

For all of the questions below, please assume the all functions exist and behave as the name suggests. For example, if there is a program the includes the command: word.getLastLetter(), you can assume this function returns the last letter of that word.

Calls to the function "write" will print the value passed in and anything wrapped in ‘’ (single quotes) denotes that those words are printed on the screen, so the line: write "Hi!" will print 'Hi!' on screen.

A note on the text-based questions

For the text-based questions we are using the **CoffeeScript** programming language. CoffeeScript is different from many other programming languages (like Java and JavaScript) in a few, minor ways:

- 1 – CoffeeScript does not use semicolons at the end of statements
- 2 – CoffeeScript uses white space to denote commands being nested inside each other, instead of the more common {}s
- 3 – CoffeeScript uses few parenthesis

Below is a short program implemented in both Java and CoffeeScript that highlights these differences. The two programs will have the same behavior when run.

```
if (true) {  
    write("In the if statement");  
}  
write("After the if statement");
```

Java

```
if true  
  write "In the if statement"  
  write "After the if statement"
```

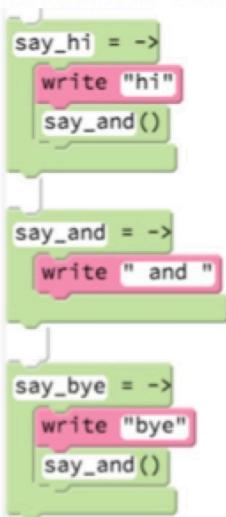
CoffeeScript

What are the values of x and y after this script runs?

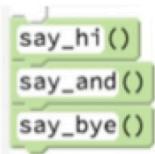
```
x = 10  
y = 5  
y = x  
x = x + 5
```

- a. x is equal to 15; y is equal to 15
- b. x is equal to 10; y is equal to 5
- c. x is equal to 5; y is equal to 10
- d. x is equal to 10, 15; y is equal to 5, 10
- e. x is equal to 'x + 5'; y is equal to 'x'
- f. x is equal to 15; y is equal to 10

Here are three functions.

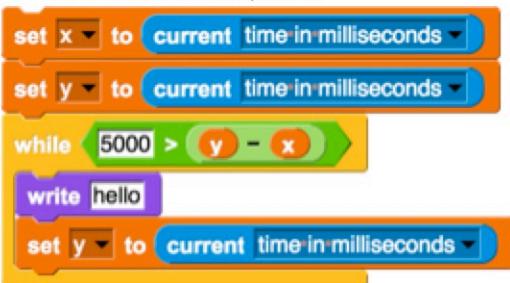


What will be printed when this program is run?



- a. 'hi and bye and'
- b. '' [nothing will be printed]
- c. 'hi bye and'
- d. 'hi and bye'
- e. 'hi and and bye and'

What does this script do?

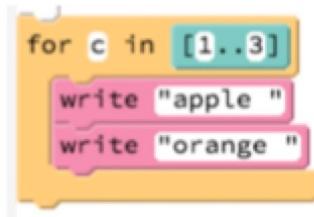


- a. This program will cause an error
- b. Prints 'Hello' 5000 times
- c. Prints 'Hello' once after 5 seconds has elapsed
- d. Prints 'Hello' continuously for 5 seconds
- e. Prints 'Hello' once

If you want to write a program that asks a user to type in a sentence, then reports back to the user the number of times the letter 'e' appears in that sentence, which of these things would your programming language not need to be able to do:

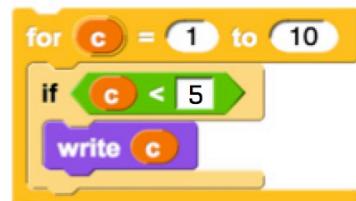
- Store user entered information
- Display text on the screen
- Compare two letters to each other to determine if they are the same
- Convert letters into numbers and numbers into letters
- Create and modify data as a program runs

What will be printed when this program is run?



- '' [nothing will be printed]
- 'apple orange apple orange apple orange'
- It will be different each time you run it
- 'apple apple apple orange orange orange'
- 'apple orange'

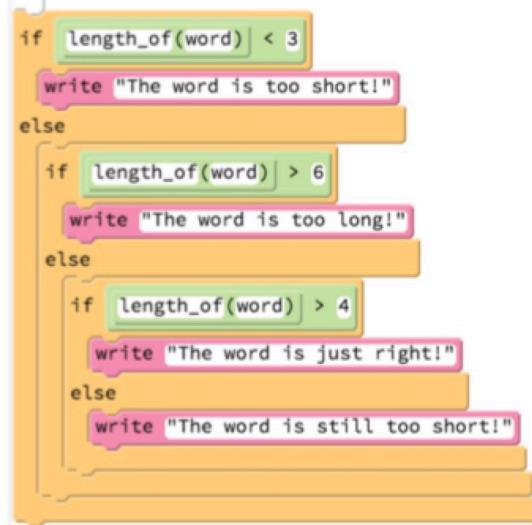
How many times will the comparison $c < 5$ be tested when this program is run?



- 0
- 1
- 5
- 10
- 50
- It will be different each time you run it

The `length_of` function returns the number of letters in a word. So `length_of("cat")` will return 3, because cat is 3 letters long.

Here is a program that uses the `length_of` function.



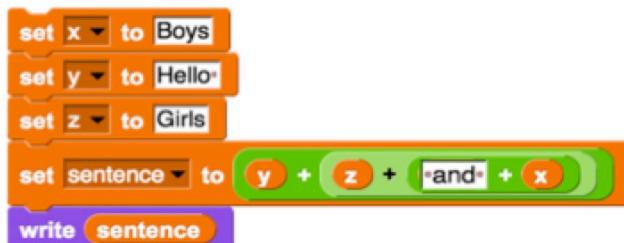
If `word` was set to "Hello", what would be printed?

- 'The word is still too short!'
- 'The word is just rightthe word is still too short!'
- 'The word is just right'
- 'The word it too long!'
- 'The word is too short!'

If `word` was set to "Hi", what would be printed?

- 'The word is too short!'
- 'The word is just right'
- 'The word is still too short!'
- 'The word it too long!'
- 'The word is just rightthe word is still too short!'
- 'The word is too short!The word is still too short!'

What will be printed after running this script?



- 'sentence'
- 'BoysHello Girls'

- c. 'Hello Girls and Boys'
- d. 'y + z + " and " + x'
- e. 'Hello Boys and Girls'

What does this script do?

```
if x > 10
    x = 10
else
    if x < 5
        x = 5
```

- a. It always sets x equal to 5
- b. This program will cause an error
- c. Makes sure the value of x is less than 5
- d. Makes sure the value of x is not equal to 10
- e. Makes sure the value of x is between 10 and 5

You are writing a guessing game program. Your program randomly chooses a number between 1 and 100, then asks the player to guess the number. If the player's guess is too high the program will tell them to guess lower, if their guess is too low, the program will tell them to guess higher. The player gets 10 guesses.

Below are the six necessary parts of the program, but in a jumbled order.

- A. Compare the guess to the mystery number and report back 'Too High', 'Too Low!', or 'You Guessed it!'
- B. Increase the **guess_counter** by one to record the guess attempt
- C. Check to see if the player has used up all 10 guesses, if so, print: 'You are out of Guesses!"
- D. Ask the user for their guess and then read it in
- E. Ask the user if they would like to play again
- F. Randomly choose the mystery number and set the user's **guess_counter** to 0

Please answer the following true/false questions:

True or False: Part F must be the first step in the program

True or False: It is necessary that part A comes before part B

True or False: Part C must come before part E

True or False: Part B is optional; the game will work without it

True or False: There is only one way to write the code for these six parts of the program so that the game works

Because the guessing game gives the player 10 tries, some of the steps in the program will need to be run once for each guess the player makes, while other steps will only be run once per game. Which steps will only be run once for each time the game is played?

- a. E and F
- b. A, B, C, and D
- c. B, C, and E
- d. Only E

What are **x**, **y**, and **z** equal to after this program is run?

```

x = "yes"
y = x
z = x
x = "no"
y = "maybe"
z = x

```

- a. x is equal to 'yes, no'; y is equal to 'yes, maybe'; z is equal to 'yes, no'
- b. x is equal to 'no'; y is equal to 'maybe'; z is equal to 'yes'
- c. x is equal to 'yes'; y is equal to 'maybe'; z is equal to 'no'
- d. x is equal to 'no'; y is equal to 'maybe'; z is equal to 'no'
- e. x is equal to 'no'; y is equal to 'maybe'; z is equal to 'x'

The **contains** function return true if the first string contains the second string. So “**Yes**”.**contains**(“**Y**”) will return true.

What will be printed after running this script?

```

set str to Hello-World!
if str contains He
  write I include He
if str contains Three-Ls
  write and I include three Ls
if str contains !
  write and I include !

```

- a. ‘I include He and I include !’
- b. ‘and I include !’
- c. ‘’ [nothing will be printed]
- d. ‘I include He and I include three Ls and I include !’
- e. ‘I include He’

What will be printed when this program is run?

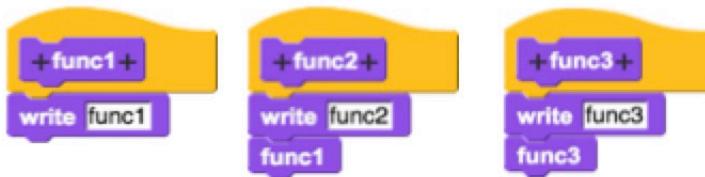
```

c = 5
while c > 0
  c = c - 2
  write c

```

- a. it will be different each time you run it
- b. '3'
- c. '531'
- d. '31'
- e. '543210'
- f. '31-1'

Here are three functions.



What is printed when this script is run?



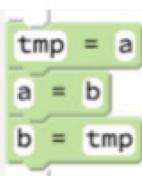
- a. 'func1 func2 func1'
- b. 'func1 func2'
- c. This program would cause an error
- d. " [nothing is printed]
- e. 'write func1 write func2'

What is printed when this script is run?



- a. 'func3 func3 func1'
- b. 'func3 func1 func3'
- c. 'func3 func1'
- d. " [nothing is printed]
- e. This program would cause an error
- f. 'func3' will be printed over and over until the script is stopped

a, **b**, and **tmp** are variables. What does this program do?



- a. Makes a and b equal to each other
- b. Rearranges the variables a, b, and tmp
- c. This program do not do anything
- d. Swaps the values of a and b
- e. This script doesn't do anything

Imagine you are writing a computer program that will ask you for a class period, then print what class you have that period on the screen. When you try and use the program, you realize it always prints out your first period class. Which of the following potential "bugs" could not cause this error:

- a. The steps you follow to match the input period to your stored schedule is incorrect
- b. You are using the wrong command to print words onto the screen
- c. You are reading in the class period incorrectly
- d. You accidentally stored your first period class for every period of the day

Below is a program that will read in the names of all students in a class, and then print out a list of all the students whose first name ends with the letter 'a':

```
Read in a list of students in the class and store their names in a container called all_students
Create a container called a_names to hold students whose name ends in 'a'
Go through all of the elements in all_students and for each name:
    <do-something-here>
Print out all the students in a_names
```

Here are five possible steps that could be used to finish the program

- A. Check to see if the student's first name ends in 'a'
- B. Print out the students names
- C. Add the students name to **a_names**
- D. Remove the student's name from **all_students**
- E. End the program

Which steps should replace <do-something-here> so the program will work correctly:

- b. A then B
- c. A then B then C
- d. A then C
- e. Just C
- f. C then D then E

When the program is working correctly, which statement will always be true?

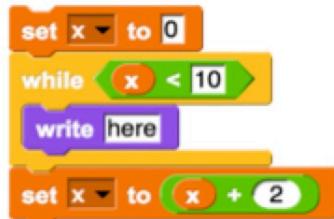
- a. **a_names** has fewer or the same number of names in it as **all_students**
- b. No two students will have the same name
- c. There will be no student names in **a_names**
- d. If you run the program twice for the same class, you will get a different list of names printed out each time

The function **op4** takes in 3 numbers, what does this function do?

```
op4 = (a, b, c) ->
  if a > b
    tmp = a
  else
    tmp = b
  if c > tmp
    tmp = c
  return tmp
```

- Set all three inputs to a temporary value
- Returns the largest of the three numbers
- Randomly returns one of the three numbers
- Return the smallest of the three numbers
- This program will cause an error

How many times will the word "here" be printed when this script is run?



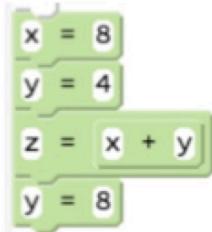
- 0
- 1
- 5
- 10
- 'here' will be continuously printed until the script is stopped
- It will be different each time you run it

What will be printed after running this script?

```
x = 10
if x > 7
  write "inside the if "
else
  write "inside the else "
  write "all done"
```

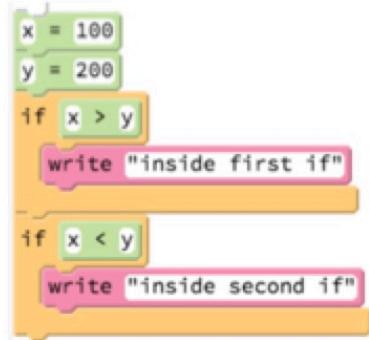
- 'inside the if inside the else all done'
- 'inside the if'
- 'inside the if all done'
- 'all done'
- 'inside the else all done'

What are **x**, **y** and **z** equal to after this program is run?



- a. x is equal to 8; y is equal to 4; z is equal to 84
- b. x is equal to 8; y is equal to 8; z is equal to 16
- c. x is equal to 8; y is equal to 8; z is equal to 12
- d. x is equal to 8; y is equal to 4, 8; z is equal to 12
- e. x is equal to 8; y is equal to 8; z is equal to 'x + y'

What will be printed after running this script?



- a. 'Inside first if'
- b. 'Inside first ifInside second if'
- c. 'It will be different each time'
- d. 'Inside second if'

How many times will the word "here" be printed when this script is run?

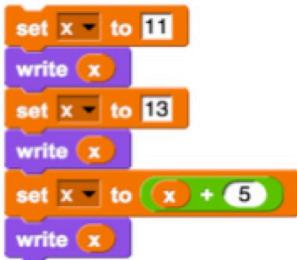
```

repeat (2)
  write "here"
  repeat (3)
    write "here"
  write "here"
end

```

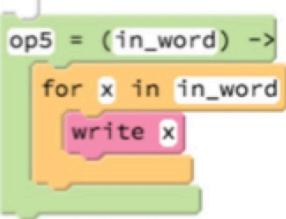
- a. 3
- b. 6
- c. 10
- d. 18
- e. It will be different each time you run it

What is printed when this program is run?



- a. '11' then '11' then '11'
- b. '11' then '13' then '18'
- c. '' [Nothing will be printed]
- d. This program will cause an error
- e. 'x' then 'x' then 'x + 5'
- f. '18' then '18' then '18'

The function **op5** takes a word in as an input.
What does **op5** function do with that word?



- a. Prints the word one letter at a time in the original order
- b. Prints the word once for each letter of the word (so for a 3-letter word, the whole word is printed 3 times)
- c. Prints the last letter of the word that is passed in
- d. Prints the word one letter at a time in reverse order
- e. This program will cause an error

Below are three functions that each perform a mathematical operation:

```

op1 = (a, b) ->
  return a + b

op2 = (c) ->
  return op1(c, c)

op3 = (d) ->
  return d * d

```

What is output of this program?

```
write op2(5)
```

- a. '55'
- b. '5'
- c. '' [nothing is printed]
- d. '10'
- e. This program would cause an error

What is output of this program?

```
write op1(op3(4), 5)
```

- a. '21'
- b. '9'
- c. '81'
- d. '' [nothing is printed]
- e. This program would cause an error

14. Appendix D – Interview Protocols

This appendix includes the interview protocols used for the interviews conducted at the beginning, middle, and end of the 15-week study.

Pre Interview Protocol

Programming and Computer Science Background

Tell me about your programming experience?

ECS students:

- Tell me about ECS
- What did you like about the class?
- What didn't you like?
- What do you think the goal was?
- What did you learn?
- How did it prepare you for this course?
- Tell me about the projects in the class:

Tell me about your Scratch experience?

Initial Perceptions of Pencil.cc

About Pencil.cc

- Do you have any PencilCode experience?
- Is Pencil.cc programming?
 - What about it makes it programming?
 - What about it makes it different from 'real programming'?
- What is easy about pencil.cc, what is difficult?
- Why do you think we are starting the year with pencil.cc?

Program Comprehension

For this portion of the interview, students are shown a program in the modality they will see in their class and asked questions about it. The two programs and subsequent questions are presented below.

```

await read 'Type in a word please?', defer word
if word.length < 8
    x = 0
    while x < word.length
        write word[x]
        x = x + 1
else
    write word + ' is too long!'

```

Figure D.1. The first program shown to students in the pre interview.

- What does it do?
- Are all the lines necessary?
- If we wanted to make it print out all the letters twice in a row what would you change? (h,h,e,e,l,l,o,o)
- If we wanted to make it print the word twice in a row, what would we change? (h,e,l,l,o,h,e,l,l,o)?

```

myNumber = (random 100)
write 'The number is: ' + myNumber
countDowner = 0
if myNumber > 60
    countDowner = 20
else
    if myNumber > 30
        countDowner = 10
    else
        countDowner = 5

while myNumber > 0
    write myNumber
    myNumber = myNumber - countDowner

```

Figure D. 2. The second program shown to students in the pre interviews.

- What does it do?
- Are all the lines necessary?
- If we wanted to add another option where if the number was between 40 and 60 it would count down by 15, how we would do that?

Program Generation

The last portion of the interview asks students to write the following program:

Write a program that prompts the student for their grade and then reports back if the student is a underclassman if they enter 9 or 10, and an upperclassmen if they enter 11 or 12.

Mid Interview Protocol

Reflection on the Course

- How has the class been going so far?
- Did you like working in Pencil.cc
 - What was your favorite thing about it?
 - What was your least favorite thing about it?
- If a friend of yours asked you to describe what you have done in class so far this year, what would you tell them?

Transition to java

- You just started working on Java, how has that been going?
- How is it different than what you did in Pencil.cc in the first 5 weeks of school?
- Do you think what you learned in pencil.cc is helpful for Java? If so what and how?
- What has been the biggest difference between Pencil.cc and Java?

Final Project Discussion

Ask student to explain their final project and how it works

Program Generation

The last activity of the mid interview asks students to write the following program:

Have the computer pick a random number less than 15 and then print out every multiple of that number that is less than 100. So if you pick 8, it would print 8, 16, 24....96.

Post Interview Protocol

Reflection on the Course and Java

- How is the class been going so far?
- Do you like Java?
- What have you liked? And what have you not liked?
- What have you learned so far about programming in Java?
- What types of things can you do with Java?

- What types of things could you do with Pencil Code?
- What is it important to know to be good at Java?
- What do you think the easiest thing?
- Do you like programming in Java?
 - Did you prefer programming with Pencil Code?
- If a friend of yours asked you to describe what you have done in class so far this year, what would you tell them?

Java Compared to Pencil.cc

- So how does what you're doing now in Java compare to what we did at the start of the year w/ Pencil Code?
- What do you think are the big differences between Java and Pencil Code?
- How is what you have done so far in Java different than what you did in the first part of the year in Pencil Code?
 - What is the same between Java and Pencil Code?
- Do you think the stuff you did in Pencil Code was helpful for what you're doing now?
 - If so, what and how has it helped?
 - Is there anything from Pencil Code that you think made Java harder?
- Are there any strategies for programming that you developed while using Pencil Code that you now use in Java?
 - Any strategies that are different in Java?
- You have worked in both java and in Pencil Code
 - Which format do you find easier to read programs in?
 - Why?
 - Which format do you find easier to write programs in?
 - Why?
- Next year do you think we should spent the first 5 weeks using Pencil Code? Why or why not?

15. Appendix E – Coding Manuals

This appendix includes the coding manual for all of the qualitative coding done as part of this dissertation. The coding manuals are in order of when they appear in the dissertation. For each set of code, the corresponding Figure and section of the dissertation is referenced. All codes were applied by a secondary coder with the inter-rater reliability scores included in the text where the Figure appears.

Pencil.cc vs. Java Comparison Coding Manual

This coding manual was used for Figure 5.1 and Figure 5.2 in Chapter 5.

Code	Description
Visual Layout	Response mentions the presence or absence of blocks or the blocks-like nature of the interface (note: this does not include the visual execution or the program)
Ease of Composition	Response refers to the drag-and-drop programming or explicitly mention how it is easier/harder to write programs in one modality. This code also includes the need to type in (or not type in) commands
Browsability	Response mentions the presence of the blocks in the palette and the ability to read through them. This also includes mentioning how commands do not need to be memorized or remembered
Prefabricated Commands	Response talks about how blocks can do more than a single command, or there not being a block for everything
Visual Outcomes	Response talks about the visual execution environment (i.e. moving a sprite or turtle compared to outputting text)
Syntax	Response references differences in syntax (i.e. semicolons or different keywords or a new language)
In-editor Help	Response references in-editor help features such as the help tip that appears when hovering over a command or the Quick Reference menu
Other	Response articulate other differences between Pencil.cc and Java not captured by the above codes

What do ____ do? And how are they used in programs? Coding Manuals

Variables Coding Manual

This coding manual was used for Figure 6.1 in Chapter 6.

Code	Description
Container	Response describes a variable as something that holds or stores things
Placeholder	Response describes a variable as a placeholder, a copy of something, or as a thing that references or refers to something else
Pointer	Response describes a variable as a thing that points to something else
Their Own Thing	Response describes a variable as its own thing (a value, a mini-program, a thing the computer uses) or is defined relative to itself

Note: All categories are mutually exclusive.

Conditional Logic Coding Manual

This coding manual was used for Figure 6.2 in Chapter 6.

Code	Description
Decides What Gets Run	Response says that conditional logic is used to control the set of commands that will be run.
Branching Logic	Response includes language saying that conditional statements can be used to make one set of command run or another (i.e. either/or feature explicitly mentioned)
Condition to Meet	Response include language saying that a conditional statement includes a condition that needs to be met or a condition that decides what will be run. It does not includes responses that use event-based language ("when x happens...")
Boolean Statements	Response explicitly includes the words true, false and/or Boolean
If and If/Else Discussed	Response defines both if and if/else
Misconceptions	Response includes misconceptions or incorrect statements about conditional logic. In particular using event-based languages (i.e. "if something happens then...")

Note: Condition to Meet and Boolean Statements are mutually exclusive.

Iterative Logic Coding Manual

This coding manual was used for Figure 6.3 in Chapter 6.

Code	Description
Define For Loops	Response includes a definition (or attempted definition) for what a for loop is (i.e. it runs a set number of times)
Define While Loops	Response includes a definition (or attempted definition) for what a while loop is (i.e. it runs until a certain condition is met)
Save typing & Convenience	Response includes language saying the loops are used to save typing, to make things easier, or to save the programmer from having to repeat

	code
Temporal Explanation	Response includes language suggesting that loops happen at a certain time (i.e. the mechanism is temporal as opposed to sequential)
No Repetition	Response does not include any language about repetition
Incorrect or Misconception	Response gives an incorrect definition of what either a for loop or a while loop, or show a fundamental misunderstanding of their behavior. Note: this is mutually exclusive with the temporal explanation category

Functions (metaphors) Coding Manual

This coding manual was used for Figure 6.4 in Chapter 6.

Code	Description
Instruction Sets	Responses talks about functions as if they are a set of instructions, collection of commands, set of actions. It is explicit that the function is made up of a collection of things (statements, commands, actions, etc.)
Equations	Response talks about how functions are like equations or expressions
Variables	Response defines functions as variable or a type of variable
Storage	Response talks about functions as a type of storage or a way to store things (i.e. store information not store lists of commands).
A way to Do Things	A function is a singular thing that can be used to do something in a program (i.e. an action, a task, a mini-program, etc.)

Note: All categories are mutually exclusive.

Functions (features) Coding Manual

This coding manual was used for Figure 6.5 in Chapter 6.

Code	Description
Modularization & Convenience	Response says that functions are used to make programming easier, to save the user from typing, can be called repeatedly with different inputs or produce different outputs, or to break the program down into pieces
Can be called	Response attends to the fact that functions are things that can be called or run
Take Inputs	Response includes the fact that functions can take inputs or have parameters
Have Outputs	Response includes the fact that functions can (or do) have outputs
Use Variables	Response talks about how functions use variables or a similar to variables
Like Equations	Response talks about how functions are similar to equations

Thing I Learned in Pencil.cc that Will be/Was Most Helpful in Java Coding Manual

This coding manual was used for Figure 8.2 and 8.3 in Chapter 8.

Code	Description
Specific Concepts	Response cites a specific concept or set of concepts (i.e. conditional logic, variables, etc.)
Programming Basics	Response references programming basics, either explicitly or by mentioning general programming components, like using blocks or commands
Syntax & Format	Response alludes to the syntax or format of code. This includes specific features like semicolons and curly brackets
Process	Response speaks to some process related to programming (i.e. figuring things out step-by-step)
Order & Sequence	Response speaks to the order in which code is executed or the relationship between sequential statements
Meta Programming Concepts	Response speaks to a meta-aspect of programming, like problem solving, figuring out the relationship between the code and an outcome, or knowing what you want a program to do
Note Sure	Response says they are not sure about what will be useful between the two environments or is unable to draw link between Pencil.cc and Java

16. Appendix F: Java Compilation Error Parser

This appendix presents the code that was used to categorize error reported by the Java compiler. The bulk of the logic in this program involves the use of regular expressions for pattern matching against standardized error messages generated by the compiler. This program was used in the analysis presented in the Frequency of Java Errors section of Chapter 8. This code was largely written by Connor Bain as part of an independent study organized by Uri Wilensky and lead by David Weintrop.

```
import re

def ErrorTypeIdentifier(theError):

    # Incorrect javac call (missing .java)
    if re.search("Class names, '.*', are only accepted if annotation processing is explicitly requested", theError):
        return "incorrect javac call"
    if re.search("javac: file not found: (.*)\\.java", theError):
        return "incorrect javac call"
    if re.search("javac: invalid flag: (.*)$", theError):
        return "incorrect javac call"

    # Class name does not match file name
    if re.search("class .* is public, should be declared in a file named .*\\.java$", theError):
        return "wrong file/class name"

    # Incorrect package import
    if re.search("package (.*) does not exist$", theError):
        return "wrong package name"

    # Missing parenthesis or bracket
    if theError == "\']\' expected":
        return "unmatched parenthesis or bracket"
    if re.search("\'\\" or \'\\[\' expected", theError):
        return "unmatched parenthesis or bracket"
    if re.search("\'\\)\' expected", theError):
        return "unmatched parenthesis or bracket"
    if theError == "'(' expected":
```

```

        return "unmatched parenthesis or bracket"

# Missing curly brace
    if re.search("reached end of file while parsing", theError):
        return "missing curly brace"
    if re.search("\'{\' expected", theError):
        return "missing curly brace"

# illegal escape characters
    if theError == "illegal character: '\\':
        return "illegal escape character"
    if theError == "illegal escape character":
        return "illegal escape character"

# copy and pasted smart quotes
    if theError == "illegal character: '\u201d'":
        return "used smart quotes"
    if theError == "illegal character: '\u201c'":
        return "used smart quotes"

# variable already defined
    if re.search("variable (.*) is already defined in method
(.*)$", theError):
        return "variable already defined"

# variable not initialized
    if re.search("variable .* might not have been initialized$", theError):
        return "variable not initialized"
    if theError == "variable not initialized":
        return "variable not initialized"

# Static / Context issues
    if re.search("Illegal static declaration in inner class
(.*)$", theError):
        return "static context issues"
    if re.search("non-static method (.*) cannot be referenced
from a static context$", theError):
        return "static context issues"
    if re.search("non-static variable (.*) cannot be referenced
from a static context$", theError):
        return "static context issues"

# Type issues
    if re.search("incompatible types: (.*)", theError):
        return "type mismatch"

```

```
if re.search("incomparable types: (.*)", theError):
    return "type mismatch"
if re.search("bad operand types for binary operator
'(.*)'$", theError):
    return "type mismatch"
if re.search("bad operand type (.*) for unary operator
'(.*)'$", theError):
    return "type mismatch"

# Method argument issues
if re.search("no suitable method found for (.*)$",
theError):
    return "wrong arguments"
if re.search("no suitable constructor found for (.*)$",
theError):
    return "wrong arguments"
if re.search("constructor (.*) in class (.*) cannot be
applied to given types;$", theError):
    return "wrong arguments"

return theError
```