The Pennsylvania State University
The Graduate School

SUPPORTING NOVICE YOUTH LEARNERS’ CODING AND
COMPUTATIONAL THINKING THROUGH PRODUCTIVE FAILURE-
BASED DEBUGGING AND TROUBLESHOOTING ACTIVITIES

A Dissertation in
Learning, Design and Technology

by

Sagun Giri

© 2023 Sagun Giri

Submitted in Partial Fulfillment
of the Requirements
for the Degree of
Doctor of Philosophy

August 2023
The dissertation of Sagun Giri was reviewed and approved by the following:

Gabriela T. Richard  
Associate Professor of Education  
Learning and Performance Systems  
Dissertation Advisor  
Chair of Committee

Ty Hollett  
Associate Professor of Education  
Learning and Performance Systems

Craig Campbell  
Assistant Teaching Professor of Education  
Lifelong Learning and Adult Education

Ashley Nicole Patterson  
Associate Professor of Education  
Curriculum and Instruction

Heather Toomey Zimmerman  
Professor of Education  
Director of Graduate and Undergraduate Studies  
Learning and Performance Systems
ABSTRACT

Past efforts to teach novices programming through pair programming and project-based learning utilizing different low floors, high ceilings and wide walls platforms have been successful. Building from related work, this study investigates the effectiveness of Productive Failure (PF) pedagogical design in supporting youth and novices when learning programming and fundamental computational concepts. I report on finding from an online synchronous remote workshop for participants from federal programs from low-income families during the Covid-19 pandemic. Participants engaged in code debugging without scaffolding followed by a consolidation phase. Findings suggest that PF design can introduce frustration and complexity, but when well designed and reinforced, it can aid learners’ comprehension and application of computational concepts and future troubleshooting skills. Students’ interest, motivation, and self-efficacy in programming were positively impacted by the PF design. However, some students lacked confidence in their abilities and tended towards problem avoidance. The iterative solution generation and trial and error were effective in learning code block functionalities, sequencing of scripts, and improving application accuracy of some computational concepts.

However, there is a need to design structured support structures in PF designs to manage failures, promote good programming practices, mitigate recurring error patterns and affective challenges, improve problem-solving approaches, and support interest and skill development in STEM, especially for vulnerable learners. Effective error location and debugging strategies may need to be modeled and scaffolded to support deeper conceptual engagement with fundamental computational concepts, ultimately improving their accuracy and efficiency in coding. The study suggests that PF design can be an effective approach to teach computational concepts when coupled with scaffolds and use of different debugging strategies. The study also provides insights into remote learning and delivery of online STEM focused workshops for vulnerable learners. Future work will explore how different designs can facilitate meaningful learning for marginalized learners, provide space for support, community building, and skill development in STEM fields.
# TABLE OF CONTENTS

**LIST OF FIGURES** ........................................................................................................................................... vii

**LIST OF TABLES** ............................................................................................................................................ x

**ACKNOWLEDGEMENTS** ...................................................................................................................................... xi

**Chapter 1** ......................................................................................................................................................... 1

Research Problem .................................................................................................................................................. 2

Research Purpose .................................................................................................................................................. 3

Research Questions .............................................................................................................................................. 3

Key Terms ............................................................................................................................................................. 3

**Chapter 2** .......................................................................................................................................................... 8

Role of Failure in Learning .................................................................................................................................. 8

  Historical lineage and arguments on failure-driven learning .............................................................................. 8

  Motivational, Attributional and Productive Effects of Failure ................................................................. 9

  Deficit Frames, Learned Helplessness, Negative Affect and Performance Effects of Failure .................. 14

  Dissertation Positionality on Role of Failure in Learning and Productive Failure .................................... 17

Productive Failure Design to Support Learning and Problem Solving .......................................................... 20

  Building from Failure Driven Learning design, Minimal Guidance, and Impasse Driven Learning ........ 23

  Implementation and Efficacy of Productive Failure Learning Designs .................................................. 26

  Productive Failure and Open-Ended Problems ......................................................................................... 32

  Summary of Productive Failure Literature ............................................................................................... 34

Supporting Novices Programming and Computational Thinking ................................................................. 35

  Novice Programming and Debugging ...................................................................................................... 35

Theoretical Framework ....................................................................................................................................... 41

  Constructivism ................................................................................................................................................. 41

  Heuristics .......................................................................................................................................................... 45

  Metacognition .................................................................................................................................................. 48

  Failure Associated Cognition ..................................................................................................................... 51

  Summary of Theoretical Framework ........................................................................................................ 51

**Chapter 3** ........................................................................................................................................................ 54

Design Based Research (DBR) .......................................................................................................................... 54

Mixed Methods .................................................................................................................................................. 55

Research Setting and Material ......................................................................................................................... 57

  Setting .......................................................................................................................................................... 57

  Participants ................................................................................................................................................... 57

  Instructors .................................................................................................................................................... 58

  Curriculum .................................................................................................................................................... 58

Research Design ................................................................................................................................................ 59
LIST OF FIGURES

Figure 1-1: Example code showing the sequence of blocks that have to be coded in Scratch. ..................5
Figure 2-1: Productive Failure Design ........................................................................................................26
Figure 2-2: PF design vs Design Instruction ................................................................................................27
Figure 3-1: Overview of participants’ demographic information .................................................................58
Figure 3-2: Virtual and Physical Debuggem- Narrative story for virtual Debuggem (top left); Provided codes and instructions (top right); Provided code block cutouts for Physical Debuggem (bottom left) and Solution example (bottom right) ...........................................................................62
Figure 3-3: Narrative story (left); Dragon incomplete code (middle); Dementors incomplete code (right) ........................................................................................................................................66
Figure 3-4: Percentage of questions per subcategories used in the survey ..............................................82
Figure 4-1: Bar chart of pre and post median scores (on all survey questions) for each participant ........100
Figure 4-2: Q5, Attitude Scale of Computer Programming Learning (ASCOPL) data box plot. Data analysis of 6 participants’ pre and post survey responses to 24 survey questions ..............................................................101
Figure 4-3: Q6, Computer Programming Self-Efficacy (CPSES) data box plot. Data analysis of 6 participants’ pre and post survey responses to 21 survey questions ......................................................................................101
Figure 4-4: Q7, Problem solving inventory (PSI) data box plot. Data analysis of 6 participants’ pre and post survey responses to 32 survey questions ........................................................................................................102
Figure 4-5: Stage for Debug 2 with the chicken and corn kernels sprite in a farm background (left); Given script missing code sequence to move the chicken sprite to the corn kernels and around the farm (right) ........................................................................................................................................107
Figure 4-6: Jamarcus’s iteration 1 code (top-left); chicken sprite continuously turning on the same spot due to the ‘Turn clockwise 90 degrees’ block inside the forever loop (top-right); Jamarcus’s iteration 2 code (bottom-left); chicken sprite oriented downwards due to the ‘Point in direction 180 degrees’ (bottom-right). ...............................................................................................................................108
Figure 4-7: Jamarcus’s iteration 3 code (left); chicken sprite moving to the corn kernels with the addition of ‘Move 30 steps’ inside the forever loop to complete task 1 of Debug 2 (right) ....................................................109
Figure 4-8: Stage for Debug 3 with pufferfish and Nemo sprites in a maze background (left), Given script with code sequence that setup initial starting position and appearance of the pufferfish sprite and incomplete directional codes (right) ................................................................................................................110
Figure 4-9: The direction system used in Scratch with 0 being the default sprite direction pointing the sprite rightwards (left); Nina’s iteration 1 code (middle); addition of ‘Move 10 steps’ block moved the pufferfish sprite rightwards even when up, down, and left keys were pressed (right) ........................................111
Figure 4-10: Nina’s iteration 2 code with improved directional movement and conditional to move the fish 10 steps back if touching pink boundary panel (left); Nina’s remixes included changes to the maze and addition of the ocean background (right) ........................................................................................................113
Figure 4-11: Stage for Debug 6 with the sprite Shantel in a room backdrop (left); Given script with sprite setup code on the left and incomplete code sequence on the right with dance animation codes, set rotation and movement styles missing event and forever loop for continuous dance moves (right) ........................................117

Figure 4-12: Delani’s iteration 1 code with ‘Go to X: 157 Y: -46’ motion block (left); Iteration 2 code with ‘Glide 1 secs to X: 171 Y: -29’ motion block (middle); Shantel’s position at the edge of the screen after Iteration 1 and 2 code initialization (right) ........................................................................118

Figure 4-13: Delani’s Iteration 3 code with multiple ‘Glide’ motion blocks and a ‘Go to X Y’ block paired with 0.3 secs wait times (left); Code initialization displaying asynchronous sprite movement and dance steps (right) ..................................................................................................................119

Figure 4-15: Debug 7 stage with Elvira, dragon and dementors sprites (left); Elvira’s code missing the ‘broadcast’ block to relay ‘expecto patronum!!’ (middle); dementors and dragon code missing event block to receive ‘expecto patronum!!’ (right) ..................................................................................................................121

Figure 4-16: Debug 7 Elvira’s code with ‘broadcast’ block added to relay ‘Expecto Patronum!!’ spell (left); Dragon’s code received the ‘expecto patronum!!’ spell to breathe fire towards dementors (middle); Dementors code received the spell and disappeared after dragon’s fire (right) ........................................122

Figure 4-17: Debug 9 stage with Jordyn and the ball sprite in a soccer field backdrop (left); Given ball code with an incomplete script with a glide block specifying destination coordinates at the back of the net (middle); Damari’s code for the ball with missing termination condition (right) ........................................127

Figure 4-18: Debug 8 stage with butterfly and Lulu the dog sprite in a soccer field backdrop (left); Given movement code for Lulu with missing code initialization and conditional scripts (middle); Given barking code with missing code initialization (right) ........................................................................131

Figure 4-19: Safiya’s iteration 1 solution for Debug 8 with ‘Say Come back’ dead code and ‘touching Butterfly1’ sensing Boolean without a container block ..................................................................................132

Figure 4-20: Safiya’s optimized Debug 8 solution with ‘touching Butterfly1’ sensing Boolean and ‘say Come back’ look block inside an ‘If-then’ conditional container block (left); initialized code shows execution of the condition (right) ........................................................................133

Figure 4-21: Safiya’s optimized code Debug 10 for Koko with score variable created and set to 0 (left); banana sprite code with a conditional sequence to increase the score by +1 if Koko touched a banana (middle); orange sprite code with a conditional sequence to increase the score by +1 if Koko touched an orange (right) ........................................................................135

Figure 4-22: Damari’s setup and movement codes for dino sprite for her Debug 14 game (left); code with a looped conditional sequence to increase the score by +1 if the dino touched an orange (middle); code with a looped conditional sequence to increase the score by +1 if dino touched a watermelon (right) ....139

Figure 4-23: Damari’s setup and movement codes for dino sprite for her Debug 14 game (left); code with a looped conditional sequence to increase the score by +1 if the dino touched an orange (middle); code with a looped conditional sequence to increase the score by +1 if dino touched a watermelon (right) ....140

Figure 4-24: Stage and sprites for Debug 10 (left); Provided codes for Koko which initialized Koko’s position and movements via mouse (middle); Provided codes for banana, orange, and red panels without scoring conditions and win state conditional code (right) ........................................................................143
Figure 4-25: Willow’s Koko codes had uninitialized events, incomplete score-setting and redundant conditional code for score changes (left); Solution for uninitialized events and incomplete score-setting through relocating the ‘Set score to 0’ block to backdrop and position setting code section (right) ........144

Figure 4-26: Willow’s Koko codes with uninitialized events, incomplete score-setting and redundant conditional codes (left); Solution to hide banana sprite once Koko ate them (middle); Solution for conditional score changes (right) .........................................................................................145

Figure 4-27: Optimal approach to update score for all orange sprites using custom blocks and conditional statements with sensing Booleans. .........................................................................................................................146

Figure 4-28: Nina’s Space Fruit game (left); Script for fruits in the game which hid after Ben touched it increasing the score by +1 (right) ........................................................................................................149

Figure 4-29: Script architecture for Ben and Beetles interaction. Ben’s code with directional controls, and redundant conditional codes for score changes (left); Beetles’ code with movement to track Ben, and a conditional code that reduced the score by -1 if touching Ben (right) ..............................................................150

Figure 4-30: Showing number of debugging problems solved individually, thorough facilitator feedback, problems that were not debugged and unattempted problems ..................................................................................................................156

Figure 4-31: Original Debug 4 code with missing initialization and loop for continuous backflips (left); Tuan’s completed code (right) .........................................................................................................................167

Figure 4-32: Debug 10’s original code lacked the score variable and a conditional (left); Jamarcus’ completed code added the score variable and size increase conditions for Koko (middle); Banana sprite code with score change conditional (right) ........................................................................................................170

Figure 4-33: Tuan’s Debug 14 game, Cho’gath’s Rampage, a retro version of his favorite game League of Legends (left); Tuan’s design of a retro version of the Summoner’s Rift map consisting of 5 levels (right) ........................................................................................................175

Figure 4-34: Cho’gath’s movement codes (left); Cho’gath’s initializing and level-up codes (right) ......176

Figure 4-35: Start button code with interconnected Event hats to set conditions to initialize each level (left); Parallely executed Level 2 codes initialized after ‘Level 2 start’ message was received (right) ...177

Figure 4-36: Stage (left); Given codes for Hope (middle); Given codes for Eto (right) .........................186

Figure 4-37: Safiya’s Lilo and Stitch quiz game (left); Initialization code to start the game (right) ........187

Figure 4-38: Question 1 code with conditions for score changes if the answer was correct or incorrect (left); Question 2 code which was a copy of Question1 with different answers and Event hat (right) ....188

Figure 4-39: Question 10 code with conditions for score changes and win-loss condition ....................189
LIST OF TABLES

Table 2-1: Frädrich et al’s (2020) catalog of Scratch error patterns.................................................................37
Table 3-1: Proposed design guideline for Debugging Mini Curriculum.............................................................68
Table 3-2: Implementation of the PF design with highlights of the activities each session..........................71
Table 3-3: Shows the types of data, methods and analysis used for each research question and sub questions .................................................................................................................................77
Table 4-1: Wilcoxon signed-rank test results (α = 0.05). Data analysis of 6 participants’ pre and post survey responses .................................................................................................................................98
Table 4-2: Excerpt 1, Jamarcus and facilitator discussing solution for Debug 2 and learning coding through trial and error..........................................................................................................................104
Table 4-3: Excerpt 2, Nina explaining her debugging process and discussing the helpfulness and frustration of learning computational concepts through trial and error during online interview ..........105
Table 4-4: Excerpt 3, Delani explaining the solutions that she generated for Debug 6 with the facilitator and what she learned in the process during Think Alouds and online interview ..................115
Table 4-5: Total number and percentage of recurring error patterns for all 9 participants. ......................123
Table 4-6: Excerpt 4, Safiya and facilitator discussing the solutions that she generated for Debug 8 during the Consolidation and Knowledge Assembly phase and her responses on her debugging process during the online interview. .................................................................................................................................129
Table 4-7: Excerpt 5, Damari explaining the game that she generated for Debug 14 and her responses on the challenges during the online interview. ..........................................................................................137
Table 4-8: Excerpt 6, Willow explaining the challenges she faced when working on Debug 10 during a Think Aloud session. ..............................................................................................................................141
Table 4-9: Excerpt 7, Nina explaining the challenges she faced when working on Debug 14 during a Think Aloud session. ..............................................................................................................................147
Table 4-10: Excerpt 8, Willow, Delani, Safiya and Jamarcus explaining the affective challenges they faced when debugging as well as positive affect from these challenges ..........................................................................152
Table 4-11: Students error locating strategies employed during debugging ..................................................164
Table 4-12: Excerpt 9, Students explaining their debugging and error-location strategies working on Debug 1-14 during Think Aloud and Interview sessions. .................................................................165
Table 4-13: Excerpt 10, Tuan explaining his bottom-up approach in Day 2 working on Debug 2-4 in the online interview and his switch to a top-down approach in Day 6 working on Debug 9 ..................172
Table 4-14: Excerpt 11, Safiya explaining her continued use of bottom-up approach from Day 2 to Day 6 .........................................................................................................................................................173
Table 4-15: Excerpt 12, Safiya explaining the challenges she faced while creating her game for Debug 14 during a Think Aloud session with the facilitator . ..........................................................................................184
ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, Dr. Gabriela T. Richard for her guidance, feedback, support, and encouragement throughout the research. I appreciate your mentorship in developing my skills as a researcher, for opportunities to present, publish and collaborate, and for the numerous life lessons along my journey that have helped me learn and grow. I would also like to express my appreciation to Dr. Ty Hollett for his constructive feedback, encouragement during difficult stages of this research, and for always batting for me. To my committee members Dr. Craig Campbell and Dr. Ashley N. Patterson, I extend my sincere gratitude for their time, expertise, and support. Dr. Campbell, thank you for answering my questions on methodology and teaching me to accept the messiness of qualitative research. Dr. Patterson, thank you for your encouragement and kindness. I will be forever indebted to Dr. Wesley Donahue for giving me the opportunity to work with him, his belief in my abilities, and his continued support for my research. I am grateful to Dr. Simon Hooper for providing me with the opportunity to pursue a PhD at Penn State and for his mentorship in my early years. Dr. Michael L. Commons, I deeply appreciate your guidance and the invaluable role you played in honing my research skills.

This dissertation would not have been possible without the support of Upward Bound Programs, Leo Conway, and Mickey Bellet. Thank you for working with me through multiple IRB modifications and the uncertainties of Covid-19 to conduct the coding, game design, and problem-solving workshop. Your support and partnership mean a lot. To the students who participated in the study and contributed to the completion of this dissertation, thank you for your enthusiasm, interest in the workshop, and for a fun teaching experience. I hope I have relayed your experiences and insights as best I can.

Nate Turcotte has been a great friend, brother, and colleague. He has always cheered for me, picked up my calls, pushed me, provided sound advice, and directed me to opportunities. We have developed partnerships with schools, given workshops together, and supported each other's accomplishments. He has been there for me through the good and bad times.
Will and Nicole have been friends who have encouraged and supported me. I value your friendship and positive outlook. Birajan, Namech, Charu, Bhabhana, Abha, and Rebecca for your moral support and providing much needed distractions with CSGO games, meals, and fun adventures.

I also want to thank Dr. Heather Zimmerman, Dr. Mindy Kornhaber, and many influential professors at Penn State who have helped me develop my identity as a scholar. My sincere thank you also goes to the admin staff at Penn State: Jenn, Darlene, Carol, Mike for consistently providing the help and support to navigate the various aspects of my academic journey.

My family are my biggest supporters and source of inspiration. Mamu, Buwa, and Aama thank you for teaching me the value of learning and for your unwavering support, countless prayers, and the unshakeable belief in my abilities. My brother Sarthak has been my best friend and someone that I can count on to bounce ideas off of, have deep conversations, and meaningful discussions. Finally, my incredible wife Aura Zhao. We met at Penn State, and you have been there with me every step of my journey. You have patiently listened to me talk about my dissertation countless times, proofread my work, and provided valuable feedback, supported me in difficult times, and believed in me at times when my belief was shaky. I could not have done this without you.
Chapter 1
Introduction

Learning to program is a useful and necessary skill for the technology focused workforce. The demand for programmers has increased rapidly due to rising automation and expanding service-based industries relying on smart algorithms to develop and train AI, support data driven decision making, marketing, and problem solving (Bureau of Labor Statistics, 2022; Bohr & Memarzadeh, 2020). Additionally, the lucrative salary, opportunity to solve real world problems and possibility of remote work has generated additional motivation for students to learn programming (Jenkins, 2001; Guzidal, 2015).

However, despite the interest, motivation, and support technologies, learning to program is a difficult task for a novice. Novices face difficulties in understanding and applying fundamental programming concepts such as conditionals, variables, loops, and operators (Gomes & Medes, 2007) and recurring syntax errors, code smells and general bugs (Frädrich et al. 2020, Meerbaum-Salant, 2013, Fraser, 2021). They also develop “bad” debugging practices by trying to solve buggy code through brute force (Nanja & Cook, 1987; Spohrer & Solloway, 1986), add high frequency errors (Griffin, Kaplan & Burke, 2012) and fixate on first solutions without analyzing the problem and conducting detailed error location or hypothesis testing (Chmiel & Michael, 2004)

Various efforts have been made to facilitate learning and address the difficulties faced by novice programmers. Computer programming has been integrated in education curriculum, after school camps and stem camps to support learning in different informal and formal settings (Adams, 2007, Werner et al., 2005; Maloney et al. 2008). Similarly, various “low floors, high ceilings and wide walls” (Resnick et al., 2009, p. 63) programming platforms such as LogoBlocks (Begel, 1996), Alice (Cooper et al. 2000), Scratch (Resnick et al. 2009), Snap! (Harvey & Monig, 2010) and Blockly (Fraser, 2015) were developed to support novices. Additionally, multiple instructional approaches such as pair programming (Umpathy & Ritzhaupt, 2017), project-based learning (Hsu, Chang & Hung, 2018) and constructionist gaming
(Kafai, 1995) have been developed to facilitate novice learning and understanding of computational concepts fundamental to programming.

The purpose of this study is to explore “Productive Failure” (PF) pedagogical design as an approach to support novices programming learning and understanding of fundamental computational concepts (Kapur, 2008). Productive Failure is a learning design that intentionally designs and uses failures for learning. Learning occurs through failures and generation of suboptimal solutions that conceptually engage learners in novel problems prior to instruction. PF design comprises two phases: an initial “problem-solving” Phase I where learners iteratively generate solutions to complex problems followed by an “instructional” Phase II where a teacher provides feedback on student generated solutions and shares canonical solutions to teach targeted concepts. Implementation of Productive Failure learning designs have largely occurred in formal STEM contexts where students engaged in learning through solving well-defined canonical mathematics and science problems (Kapur, 2008; Kapur & Bielaczyc, 2012; Kapur & Rummel, 2012). However, these highly structured activities limited the opportunity to understand the potential affordances of open-ended problems. This work contributes to an emerging body of research (Fields, Jayathirtha & Kafai, 2019; Litts et al. 2016; Searle, Litts & Kafai, 2018) on Productive Failure learning designs in informal learning environments. This chapter provides an overview of the research problem and purpose, research questions and key terms.

**Research Problem**

Research demonstrates that instances of failure can result in disruptions in learning, problem solving, task performance and deeper conceptual understanding for learners. Kapur’s (2008, 2012) Productive Failure research empirically showed that delayed instructions, designs for difficulty and initial absence of scaffolding was crucial for eventual success in problem solving. Productive Failure research and learning designs have largely focused on formal STEM contexts where students engaged in activities around well-defined canonical mathematics and science problems, missing the opportunity to study the
role of failure in open-ended problems. There are also gaps in understanding what the moments of failure, resolutions, and the process in between reveal about shifts in students' learning and conceptual understanding.

**Research Purpose**

The purpose of this DBR mixed methods study was to contribute to ongoing efforts and developing work on effective Productive Failure learning designs, particularly with respect to diverse open-ended problems. A PF design was implemented in the context of a virtual debugging workshop in Scratch with structured exploration and consolidation phases to understand how such a design can support learning of computational concepts. Based on quantitative analysis of students' pre-post test scores on subscales focused on problem-solving, computer self-efficacy and attitude and qualitative analysis of student interactions and artifacts, the study aimed at understanding learners’ failures, resolutions and shifts in learning and understanding of computational concepts.

**Research Questions**

This dissertation answers the following research questions:

- **RQ1**: How does productive failure design support learning of computational concepts when working with open-ended debugging problems in programming platforms such as Scratch?
- **RQ 2.1**: What kinds of challenges do students encounter, the resolutions they come up with and the debugging strategies they used when working through the debugging curriculum?
- **RQ 2.2**: What do these challenges, resolutions, process, and interaction patterns reveal about students’ evolving understanding of computational concepts?

**Key Terms**

Some key terms that will come up multiple times throughout the proposal are explained below.
- **Productive Failure**: Productive Failure (Kapur, 2008) is a learning design that entails designing learner driven, collaborative problem-solving conditions where students attempt to solve complex, ill-structured problems ahead of receiving direct instruction. Productive Failure design comprises two phases: Phase I, Exploration and Solution Generation and Phase II, Consolidation and Knowledge Assembly.

- **Solution Generation Phase**: Phase I of the PF design which entails students’ multiple solutions to ill-structured problems without scaffolding and direct instruction.

- **Consolidation and Knowledge Assembly Phase**: Phase II of PF design which entails facilitator feedback on student generated solutions and discussions of canonical solutions

- **Well-structured problems**: Problems that have specific goals. These problems have clearly specified problem space, path to solutions, and possible solutions. Information, resources needed to solve the problems are also clearly laid out. For example: finding the force with which a ball lands on a tennis court given the mass and acceleration of the ball.

- **Ill-structured problems**: Problems that do not have specific goals. These problems do not have clearly specified solutions paths, problem space and have multiple possible solutions. Hints can be provided to orient the students but due to the messiness and dynamic nature of the problems, clear information on solution paths cannot be specified.

- **Scratch**: Scratch is an open source, online, plug and play, block-based coding platform. Students can drag and drop coding blocks from the block’s palette onto the scripts area to create games, animations, interactive stories, tutorials, and classroom lessons. Users can choose from a variety of blocks such as motion, looks, sounds, events, data, control, etc. These blocks add different functionality and dimension to a user’s Scratch project. For example, participants can use “motion” blocks to add movement to a sprite (game characters or everyday objects), “looks” blocks to change the appearance of the sprites. They can also use “control” blocks such as repeat,
“if-then” to add conditions on a sprite’s action or “data” blocks to create their own variables.

These blocks represent one of the seven key programming or computational concepts.

- **Computational Concepts:** Computational concepts are programming structures that are common in “many programming languages” (Brennan and Resnick, 2012). These concepts are sequences, loops, parallelisms, events, conditionals, operators, and data. These concepts are fundamental to learning and understanding of programming languages. Brennan and Resnick (2012) identified seven computational concepts that students could use to create a variety of Scratch projects.

These concepts have dedicated Scratch blocks which allow users to program and execute specific instructions. See Figure 1-1 for an example code.

![Scratch Code Example](image)

Figure 1-1: Example code showing the sequence of blocks that have to be coded in Scratch.

- **Code Smells:** Code smells are hints or characteristics in the written code that indicate weak spots and deeper problems within the code (Beck and Fowler, 1999; Fraser et al. 2021). Smelly codes are indicators of code quality. The codes may be functional yet inefficient and unoptimized. Some code smells in Scratch include sequential actions, misordered code blocks, long scripts, redundant/duplicate codes, timing and synchronization issues and dead code.

- **Syntax Errors:** Syntax errors in programming are mistakes where programmers put an incorrect character or string in their scripts causing disruption in code execution. Common syntax errors
include missing parenthesis, semi-colons, undeclared, missing, or misspelled variables and incomplete or missing return statements.

- **Syntax Errors in Scratch:** In Scratch, typical syntax errors common in object-oriented programming do not exist. Fraser (2021) noted some exceptions. In Scratch, syntax errors include using incorrect reporter blocks such as “X position,” and “Answer,” inside Operator container blocks such as “_ + _,” “_ and _” as well as Sensing container blocks are considered syntax errors. Missing termination conditions, missing conditions in conditional container blocks, and deleting, writing incomplete codes are considered to be syntax errors.

- **General bugs:** General bugs are errors that can occur in any programming language. These bugs are results of conceptual misunderstandings, misconceptions, and unfettered use of button-up programming approaches. Common bugs in Scratch include missing or uninitialized loops, events, conditionals, and variables; interrupted loop sensing and non-functioning scripts/incomplete codes.

- **Forward Reasoning:** is an error location and problem-solving strategy that involves analyzing the existing code to identify the error source and work forwards to rectify it. Forward Reasoning strategies often follow a Serial Order or a Program Order (Katz & Anderson, 1978; McCauley et al. 2008).
  - Program Order: Forward Reasoning Program Order involves simulating the program’s execution as a computer would, to locate and debug errors.
  - Serial Order: Forward Reasoning Serial Order involves reviewing the existing code block by block to identify the error source and work forwards to rectify it.

- **Backward Reasoning:** Backward Reasoning is an error location and problem-solving strategy where the investigation and resolution of errors start with the incorrect behavior of the program, usually its output. It involves working backwards to the source of the error in the code (Katz & Anderson, 1978; Romero et al. 2007). Backward Reasoning consists of Simple Mapping and Causal Reasoning strategies.
- **Simple Mapping**: Simple Mapping is a debugging technique which involves mapping input and output data to identify errors. It can be used in Scratch programming to identify errors by mapping input and output data, such as issues with sprite movement in response to user input. By comparing the expected and actual movement outputs, Simple Mapping can narrow down and even pinpoint possible sources of errors such as missing code blocks or incorrect logic.

- **Causal Reasoning**: Causal Reasoning is a debugging technique that involves starting from the incorrect output and working backwards using their knowledge of programming concepts and syntax to locate the error (Fitzgerald et al. 2008; Romero et al., 2007). It also involves reviewing the program’s execution and utilizing information from the output to reason about possible causes of the bugs, eventually locating and resolving the error in their code.
Chapter 2

Literature Review and Theoretical Framework

In this chapter, I present the background of my literature and theoretical framework. I discuss prior literature on the role of failure in learning in an effort to lay out and understand the historical lineage of and cross-domain research on failure-driven learning. I highlight arguments and findings from both sides, i.e., one camp providing findings on the negative effects of failure on learning and another camp outlining the hidden efficacy of failure on learning and problem solving. This is followed by a discussion of my positionality and perspectives on the role of failure in learning debate and why this dissertation aligns with the productive failure literature and body of research that outlines the constructive effects of failure in learning. Manu Kapur’s work on Productive Failure (Kapur, 2008) and implementation of Productive Failure as a learning design in ill-structured math and science problems is discussed. Failure-driven learning and PF learning design was also examined within maker workshops and CSCL settings where problems are mostly open-ended (Fields et al. 2012; Griffin, Kaplan & Burke, 2012; Kafai et al. 2014; Litts et al. 2016; Lui et al. 2017). This body of work extended prior research on failure-driven learning and delayed instruction by developing design principles and implementation structures that need to be present in problem-solving contexts to bolster productive failure and to support learning from failure. This is followed by discussion prior research on debugging and programming for novices highlighting the challenges they face and frequent error patterns they struggle with. I end the chapter with my theoretical frameworks.

Role of Failure in Learning

Historical lineage and arguments on failure-driven learning

“Failure is instructive. The person who really thinks learns quite as much from his failures as from his successes.” - John Dewey
In the current career driven, tangible outcome-oriented society, academic success is important and expected (Kuh et al., 2006; York, Gibson & Rankin, 2015). Success is seen as the desirable outcome, a yardstick to measure performance, consistency, caliber, and competence. “Success defines the person, the organization, the culture. It is a clear goal for every initiative that has an outcome” (Loscalzo, 2014, p. 953). While success has consistently been celebrated, several scholars in the last three decades have studied and explored the role of failure in learning (Clifford, 1988; Kapur, 2008; Loibl & Rummel, 2015; Schwartz & Bransford, 1998; Van Lehn et al. 2003). In the following section, I will discuss this prior research outlining the history and practices around the role of failure in learning. I will also lay out the debate around the role of failure in learning and problem-solving and the efficacy of failure-driven learning and then discuss this dissertation’s positionality and alignment.

Motivational, Attributional and Productive Effects of Failure

Motivation theories and achievement motivation theories found that moderately difficult tasks, implying frequent failure as well as moderate probability of success, increased performance on those tasks and intrinsic motivation (Atkinson, 1964; Csikszentmihalyi, 1975; Deci, 1975; Lepper & Green, 1978). A second persistent finding of motivational and achievement motivation theories was that moderate risk-taking increased perception of competence, self-knowledge of abilities, task satisfaction, self-efficacy as well as enhanced persistence, concentration, and attention (Atkinson, 1964; Bandura, 1982; Csikszentmihalyi, 1975; Deci & Porac, 2015; Harter, 1978; Trope, 1979). Consistent with findings from motivation and achievement motivation theories, Clifford’s (1988, 1989, 1990) research on academic risk taking with grade school, high school and college students found that moderate risk increased performance, persistence, high task interest, and effort on tasks despite encountering multiple failures. Clifford’s theory of constructive failure also predicted and later supported that

“moderate risk-taking or a preference for optimally challenging tasks will be accompanied by relatively positive or constructive responses to failure (e.g., identification of the causes of failure, advice-seeking, strategy-change).” (Clifford, 1988, p. 15)
Clifford (1984, 1988, 1990) championed the role of failure in learning and argued for tolerance to error making, moderate academic risk taking and constructive responses to failure. She stated a need for changes in educational practices and noted that “many educators attempt to create error-proof learning environments” despite repeated findings showing the constructive effects of failure (Clifford, 1990, p. 23). The creation of error proof learning environments through reduced task difficulty, rewarding error-free performance and setting minimum criteria and standards deemphasized the importance of failed attempts in student learning and performance. Clifford’s theory of constructive failure proposed replacement of easy success with optimal challenges, coercive constraint-laden techniques with free exploration of problems and error-free performance with moderate academic risk taking. Through her work, she evidenced the role of failure in learning and argued for a learning environment tolerant of failure, supportive of attempts towards error correction and inclusive of learning activities with variable difficulty and payoffs. These principles are still reflected and implemented in problem-based learning environments and learning activities designed for productive failure.

Dweck (1985, 1986, 2008) found that if failure was seen as a challenge, then students put in additional effort and were on task to try to overcome these challenges. Dweck (1975) as well as Diener and Dweck (1978, 1980) identified and described major patterns of maladaptive or helpless behavior and adaptive or mastery-oriented behavior. The helpless behavior pattern in students was characterized by “avoidance of challenge and a deterioration of performance in the face of obstacles” while mastery-oriented pattern was identified as “seeking of challenging tasks and maintenance of effective striving under failure” (Dweck & Leggett, 1988, p. 256). Additionally, despite receiving the same tasks and producing the same task outcomes, students with helpless and mastery-oriented behavior patterns responded differently in the face of failure. Students with helpless behavior patterns “exhibited negative self-cognitions, negative affect, and impaired performance, whereas mastery-oriented students exhibited constructive self-instructions and self-monitoring, a positive prognosis, positive self-affect and effective problem-solving strategies” (Dweck & Leggett, 1988, p. 258).
This research was further supported by Locke and Latham’s (1990) work on goal setting and Heckhausen (1991) work on goal achievement. They found that if failure threatened goals that were prioritized by the learners, they would put additional effort to boost their task performance. In these instances, students did not feel low self-confidence and negative affect associated with failure but were more focused on goals at hand. Dweck’s research on fixed and growth mindset (1986, 2008) further showcased that failure can either be constraining or motivating depending on the mindset that people choose to foster, believe in, and develop. A person with a fixed mindset believed that intelligence and ability was static and pre-determined; “carved in stone” (Dweck, 2008, p.6). Failure, for people with fixed mindsets, was constraining. It was a constant reminder of their static ability and intelligence to the point that it could lead to negative emotions, giving up or inputting maximum and consistent effort (Dweck, 1986, 2008; King, McInerney & Watkins, 2012).

Alternatively, people with growth mindsets believe that “your basic qualities are things you can cultivate through your efforts…everyone can change and grow through applications and experience” (Dweck, 2008, p.7). People with growth mindsets saw failure as a “learning orientation,” i.e., a steppingstone to develop their skills and competence (Eppler et al., 2000, p. 354). This positive attribution of failure resulted in additional effort towards the task at hand and acted as a safeguard to negative emotions, low self-confidence, and task effort stagnancy (Aditomo, 2015; Dweck & Leggett, 1988; Hong et al. 1999). Duckworth’s (2007, 2016) work on grit further bolstered this perspective on growth mindset and resilient practices in the face of failure. Grit was defined as “trait level perseverance and passion for long-goals and showed that grit predicted achievement in challenging domains over and beyond measures of talent” (Duckworth & Quinn, 2009, p. 166). Duckworth’s research added a layer to learning from failure. She argued that a positive response to failure and resilient practices were important to learning, goal outcomes and ability development. However, what was more important to sustained goal achievements and growth was long term commitments and deliberate practice irrespective of failure experiences.
In addition to affective, attributional, motivation and achievement-oriented findings on the role of failure, there is also a growing body of educational research providing empirical evidence for the constructive role of failure (Kapur, 2008; Loibl & Rummel, 2015; Schwartz & Bransford, 1998; Schwartz & Martin, 2004; Van Lehn, 1988, 1999; Van Lehn et al. 2003; Westermann & Rummel, 2012). This body of work discussed the efficacy of delaying instructions and designing for optimal difficulties, both of which implies failure, on student learning and problem-solving. Impasse-driven learning is a “form of failure-driven learning” that explains how learning is facilitated by encountering and resolving impasses during problem solving conditions (VanLehn, 1988, p. 23). Impasse was seen as a form of failure or a period of stagnancy during problem solving conditions caused by students’ lack of knowledge, skills or familiarity with the task at hand. Students could resolve, repair the impasse on their own or seek outside help through tutors, reference materials. VanLehn et al. (2003) found that learning was more likely to occur when students reached an impasse during problem solving situations and then sought help to resolve the impasse rather than in situations where students did not reach an impasse. D’Melo et al. (2014) demonstrated that confusion as an impasse also worked to facilitate students’ learning if “appropriately induced, regulated and resolved” (p. 153). Additionally, Van Lehn (2003) found that delaying instruction until students make errors or reach a point of failure promoted learning and conceptual understanding rather than providing instruction from the beginning. Similar findings were demonstrated in Bradley and Noell (2018), Schwartz and Bransford (1998), Schwartz and Martin (2004), Roll et al. (2011), and Westermann and Rummel (2012).

Kapur’s (2008) work on Productive Failure (PF) added weight to the constructive role of failure and effectiveness of delayed instruction on learning and problem solving. PF learning design reversed the sequence of instruction that was well established and widely implemented in Direct Instruction (DI) designs. Rather than providing immediate instructions on target concepts and procedural steps required to solve complex problems, PF designs engaged students in unguided problem solving first. In this exploratory or generative phase I, students were given complex ill-structured math and science problems that would inevitably lead to failure and non-canonical solutions. Students could use their prior
knowledge, informal insights from their lives as well as incomplete conceptual understandings to generate multiple novels, semi-correct, “incorrect”\(^1\) or non-canonical solutions to those problems. Subsequent instructions would follow in the consolidation and knowledge assembly phase II along with discussions of canonical solutions and procedural steps to those ill-structured problems. Kapur and colleagues found that while students failed at generating canonical and optimal solutions to these problems, these failures, and the PF design itself was more productive than direct instruction to subsequent problem solving and learning science and math concepts (Holmes at al. 2014; Kapur, 2010, 2011, 2012; Kapur & Bielaczyc, 2012; Pathak et al. 2011). This research highlighted the productivity in failure and the efficacy of delayed instruction on conceptual learning and problem-solving.

This productivity in failure has since been confirmed and replicated in Computer Supported Collaborative Learning (CSCL) settings (Kapur, & Bielaczyc, 2012; Kapur & Kinzer, 2009). Similarly, STEM and Maker workshops involving digital and physical fabrication, tinkering and Bidirectional Design and making inherently value open-ended design activities with iterative design cycle and debugging where failure (defined in terms of generation of suboptimal solutions, rapid prototypes with progressive refinement) is expected and an inherent part of the design process (Blikstein, 2013; Kafai et al. 2014; Litts et al. 2016; Maltese, Simpson, & Anderson, 2018; Peppler 2013; Richard & Giri, 2017, 2018, 2019; Searle, Litts & Kafai, 2018). Exercises in failure in these settings played a “constant and prominent role in the overall learning process” (Litts. et al. 2016, p. 498). Blikstein (2013) found that by managing failures and working through iterative cycles on their fabrication projects, students created unique and complex designs that were also personally meaningful to them. Additionally, in these settings, failure also supported knowledge building, and promoted creativity, sustained task performance and innovation (Martinez & Stager, 2013; Martin, 2015; Ryoo et al. 2015).

\(^1\)Used in quotes as student generated solutions were rated as incorrect for math and science problems with fixed canonical solutions. However, in open-ended problems without canonical solutions, these solutions were determined to be sub-optimal and iterations rather than incorrect. See Searle, Litts and Kafai (2018)
More recently role of failure in learning has been studied in diverse domains such as business and entrepreneurial learning (Choo, 2008; Cope, 2011; Sarasvathy & Menon, 2013), organizational development and management (Cannon & Edmondso, 2006; Hodgkinson & Wright, 2002); healthcare (Edmonson, 2004; Macrae & Vincent; 2002) and learning sciences (Kapur, 2008, 2011, 2012; Sobel, 2014; Maltese, Simpson, & Anderson, 2018; Simpson & Maltese, 2017; Holmes et al. 2014; Litts et al. 2016). This literature highlighted the positive and productive effects of failure on learning, problem-solving, goal attainment, performance, students’ self-perception of their abilities, and their potential to achieve future success. However, despite these positive effects caution should be taken to generalize failure as an all-positive process in learning.

Deficit Frames, Learned Helplessness, Negative Affect and Performance Effects of Failure

Failure can also have devastating effects for learners because

....it may confirm preexisting negative beliefs or require downward revisions in self-image or goals. Repeated failures may lead to feelings of helplessness and depression. Children experiencing failure in schools may experience a decrease in self-efficacy and self-esteem. (Salkind, 2008, p. 393).

This analysis has been confirmed by another body of research that has highlighted negative affect, deficit frames and learned helplessness from failure. Seligman and Maier (1967) discussed the debilitating effects of failure through their theory and findings on learned helplessness. Learned helplessness was defined as an empirical phenomenon and a learned behavior that occurred when individuals go through repeated failures or uncontrollable, unpredictable events and experiences. Seligman and Peterson (2001) specifically referred to Learned Helplessness as “the maladaptive passivity shown by animals and people following experience with uncontrollable events” (p. 8583). Their research showed that repeated failures in given tasks resulted in individuals internalizing failures and succumbing to the mindset that future failures are inevitable. This perception or cognition of uncontrollable outcomes in one situation formed an expectation of uncontrollable outcomes in similar future tasks, rendering the individual helpless. Overtime, this expectation of helplessness was transferred and generalized to other situations resulting in
passivity and decrease in performance (Seligman & Solomon, 1971). Additionally, it was found that individuals attributed their helplessness to causes that were internal (me, my abilities, effort), external (luck, chance, fate, difficult), global (will apply to broad range of future situations), specific (will apply to narrow range of future situations), stable (long lived and recurring) and unstable (short lived and passing). These attributions impacted the degree and expectation of future helplessness and the kinds of deficits produced. In human studies, learned helplessness resulted in three major deficit frames: a) retardation of the perception of control (cognitive), b) reduction and delayed actions resulting from expectation of future failure (motivational) and c) low self-esteem and negative affect (emotional deficit) (Abramson & Seligman, 1978; Maier & Seligman, 1976).

Seligman (1975), Wortman and Brehm (1975), and Mikulincer (1994) found that multiple failures on tasks at hand resulted in feelings of helplessness and low self-confidence. Persistence of these failures further resulted in learned helplessness, lack of motivation and effort on future tasks. Additionally, thinking and obsessing over past failures and fear of failure resulted in reduced task performance and inaction, which further affected present and future performance (Dweck & Leggett, 1988, Dweck, 2008). More recently Heyd-Metzuyanim (2015) conducted a case study with a ninth grader who passed seventh grade mathematics with high scores but ended up failing ninth grade mathematics. She complained of severe mathematical anxiety which was further exacerbated by failures. These failures in turn resulted in increased anxiety.

Additionally, there are arguments cautioning against normalizing failure and minimizing the negative aspects of failure within the STEM and maker community because of the negative affect and deficit frames it can promote among individuals from low-income and historically marginalized communities (Eason, 2014; Vossoughi & Bevan, 2014; Vossoughi & Vakil, 2018). Even though failure is generally celebrated within the maker community and the wider maker movement, Vossoughi and Bevan (2014) argued that phrases like “celebrating failure” and “independent learning” that are frequently used in maker literature are “out of touch with the realities of schooling for students of color and can easily lend themselves to deficit frames” (p. 40). Eason (2014) argued that opportunity to fail and try again is a
privilege and stressed that the consequence of failure is different for individuals from different socio-economic and ethnic background

“In America, when you’re poor, and when you lack privilege, the consequences of failure are different…When you give yourself permission to fail, the implicit assumption is that you have the resources, materials, time, reputation/social capital to try again…When you are poor, you don’t have resources…Risk aversion is often ultimately the more practical option. The idea [in maker culture] is that you want to iterate, to improve, to fix things, that failure is how you learn. And it's not a bad idea, per se. But it's also not an idea that we can just assume is natural in every community. Because failure is for those who have resources and chances, who can make a mistake and not have to worry that it will reflect poorly on their entire race or ethnic group or neighborhood. (par. 4-6)

The historical association of failure with non-resilient behaviors and practices, the social stigma attached to the word coupled with stereotypes and restricted access to STEM for historically marginalized populations has also aided negative affect (lower self-confidence) and maladaptive attribution patterns of failure. This is especially true for women who are underrepresented and have restricted access and opportunities in STEM degrees and professions (National Science Foundation, 2021; Beede et al. 2011; National Research Council, 2007). While barriers to entry to STEM fields has decreased in recent years, there are still gender gaps around computer use and ownership as well as gender-specific stereotypes around STEM competency (Cooper, 2006; Cooper & Weaver, 2003; Li & Kirkup, 2007; Papastergiou & Solomonidou, 2005; Richard, 2015, 2017; Smith, Morgan & White, 2005). These factors along with lack of female role models in STEM, absence of historical investment on STEM education and careers for women and gender specific socialization from family, teachers and the wider community has contributed to women’s disadvantage in STEM, and impacted self-confidence, performance, and interest in the field (Beede et al. 2011; Dean & Fleckenstein, 2007; Richard, 2015; UNESCO, 2017).

Longstanding gender gaps in STEM and stereotype threat has also affected attribution patterns of failure. Dickhauser and Stienmeier-Pelster (2002) found that when men and women were asked to name the reason for computer related scenarios: imagine that you cannot open a file you have previously saved on a disk, men attributed the cause to external factors i.e., faulty disk while women’s attribution was internal i.e., lack of knowledge, ability. The authors found that this internal attribution led to lower levels of self-confidence and internalization of failure. Studies have also found that stereotype threat impacted
performance and women’s attributions of failure while working on math and on the computer tasks (Burke & Mattis, 2007; Koch, Müller & Sieverding, 2008; Smith, Morgan & White, 2005; Spencer, Steele & Quinn, 1999). Results showed that in negative threat conditions (where both groups were informed beforehand that men usually performed better in these tasks), women attributed the failure internally, blaming their ability while men depersonalized failure attributing failure externally.

Dissertation Positionality on Role of Failure in Learning and Productive Failure

The debate surrounding the role of failure in learning is long and inconclusive. One camp of research showed the productive, constructive, motivational, positive affect and attributional effects of failure (Aditomo, 2015; Dweck & Leggett, 1988; Heckhausen, 1991; Hong et al. 1999; Locke & Latham, 1990) This body of work focused on the hidden efficacy of failure in learning and problems solving and argued for tolerance to error making, and moderate academic risk-taking (Clifford, 1984, 1988, 1990). Designs for optimal difficulties, delayed instruction, unguided exploration of problems followed by guided consolidation were shown to support learning, problem solving, conceptual understanding and knowledge building (Kapur, 2008; Loibl & Rummel, 2015; Schwartz & Bransford, 1998; Schwartz & Martin, 2004; Van Lehn, 1988, 1999; Van Lehn et al. 2003; Westermann & Rummel, 2012)

Another camp of research and scholars presented findings and arguments on the negative effects of failure discussing negative affect (Heyd-Metzuyanim, 2015, Mikulincer, 1994; Salkind, 2008; Seligman, 1975; Wortman & Brehm, 1975), learned helplessness (Seligman & Maier, 1967) and deficit frames (Abramson & Seligman, 1978; Maier & Seligman, 1976). This camp also discussed that words such as fail early, fail fast and niche practices around celebrating failure within STEM and maker domain can lead to deficit frames, inequitable participation and promote negative affect and stereotype threat among members from low income and historically marginalized communities (Cooper, 2006; Eason, 2014; Li & Kirkup, 2007; Papastergiou & Solomonidou, 2005; Richard, 2015, 2017; Smith, Morgan & White, 2005; Vossoughi & Bevan, 2014). These communities are especially vulnerable to negative affect of failure, power relations in learning including internalization of failure as well as resignation and
expectation of failure because of the limited access to resources, inequitable participation in STEM degrees and professions in addition to historically and socially sustained stereotype and stereotype threat around STEM.

Highlighting both camps of the research, this dissertation will align with first camp of research delineating the constructive role of failure in learning while making efforts to attune to and address deficit frames, social stigma, connotations of inability and associations with non-resilient practices associated with the failure when designing and implementing the designed intervention. The reasons for alignment with constructive framing of failure including productive failure literature and designs are two folds. First, the camp of research literature discussing the productive effects of failure does not prioritize failure as a consistent requirement or an exclusive means to an end (increased performance, learning and deeper understanding). Rather these literatures discussed the latent efficacy in failure and delayed scaffolding that could be leveraged in certain conditions to support learning, problem solving and deeper understanding (Kapur, 2008; Loibl & Rummel, 2015; Schwartz & Bransford, 1998; Schwartz & Martin, 2004; Van Lehn, 1988, 1999; Van Lehn et al. 2003; Westermann & Rummel, 2012). In these designs, a short period of unguided or minimally guided problem solving resulting in suboptimal or non-canonical solutions was shown to be helpful to disclose knowledge gaps and help in student conceptual understanding. Kapur’s (2008) Productive Failure Designs showed that while period of problem solving and generation of solutions on their own could lead to failure, which was defined in terms of generation of suboptimal or non-canonical answers, this failure was temporary and was part of the process designed to help long term learning. The long-term goal was to support students' deeper understanding, learning of targeted concepts and future problem solving through knowledge assembly sessions in which a facilitator discussed and compared students’ solutions pointing out concepts that they already understood and could apply and clarifying their knowledge gaps that students could work on during practice sessions. Similarly, in maker domain failure was seen an inherent process and period of trial and error, debugging and rapid prototyping with consistent refinement supplemented by scaffolding and guidance to create artifacts and solve problems rather than a permanent stopping points in learning or negative performance index
by looking at failure as a productive rather than counterproductive dimension of learning, we can shift pedagogical directions as well as students; perceptions of such events that are inevitable in the design and learning process.” (p. 499).

Second, Reflexivity informs and shapes positionality (Homes, 2020). Reflections by researchers on their positions and views is necessary because researcher’s values, ethics, and cultural, social, and political context can influence the research process (Bourke, 2014; Creswell & Creswell, 2017; Savin-Baden & Major, 2013). As such my own biases along with my cultural and historical experiences, early socialization and educational experiences have shaped how I interpret failure and my alignment towards the positive role of failure in learning and productive failure literature. These biases have an impact on epistemological stance, theoretical lens and pedagogical and methodological leanings and inform my research design in this dissertation (Sikes, 2004; Rowe, 2014).

The first bias that I have is that I believe that failure can be instructional in learning, problem solving and can support long term growth and learning. This position comes from growing up in a military family where trying multiple times was encouraged (grit) and unsuccessful attempts were not ridiculed or rebuked. The position was reinforced from being brought up in middle and high school as well as working with my one of my mentors in Boston where iterative process of generating suboptimal, incomplete solutions (such as partial data analysis with few participants to see initial data trend, multiple paper drafts with sections left unwritten, sections not organized linearly and laden with track changes) was focused more than successful outcome on the first attempt. Similarly, student driven studies, creative ideas and new approaches to look at a set of collected data that may not be feasible or cost effective was encouraged rather than express instructions. This agency provided opportunities to fail and iterate with constructive feedback rather than negative performance evaluation or rebuke. I believe that this was instrumental to my growth as a researcher and independent thinker. Additionally, the position is further
informed and shaped by leaning towards a definition of learning as an active, contextualized and learner centered process of knowledge building and meaning making through discovery, inquiry, iteration, negotiation and by engaging in problem solving, collaborative activities and constructing personally meaningful artifacts through iterations and troubleshooting.

The next sections will discuss the literature on Productive Failure (PF), PF designs and application of PF designs for well-defined mathematical and science problems and for open ended problems in STEM and CSCL settings. Additionally, limitation in designing for Productive Failure in open ended problems discussed by emerging research (Litts et al., 2016; Searle, Litts & Kafai, 2018) will be presented. The section will end with a discussion of how this dissertation seeks to address these limitations and add to the Productive Failure design and implementation.

**Productive Failure Design to Support Learning and Problem Solving**

Productive Failure is a learning design where students first generate solutions to ill-structured problems (exploration and generation; Phase I) before receiving canonical instruction and scaffolding in consolidation and knowledge assembly; Phase II (Kapur, 2008). Absence of scaffolding in the early phase invariably led students to struggle and ultimately failure. Failure in this context meant that “students will typically be unable to generate and discover the correct solutions (canonical solutions) by themselves” (p. 5). Failure also meant the generation of suboptimal, non-canonical solutions which would be considered incorrect or failed based on traditional definitions of failure. However, this failure in the short term did not translate to lack of inability or learning potential. It was actually found to be productive towards students’ learning and conceptual understanding in the long term. This productive effect of failure in the long term was explained in part by the solution generation effect which referred to the direct correlation between generating solutions to long term learning, procedural fluency, and conceptual understanding (Kapur, 2015).
The absence of scaffolding in Phase I meant that students could only rely on their prior knowledge, incomplete understanding, heuristics, and informal insights from their lives to generate solutions to given ill-structured problems. This generation of incomplete, sub-optimal solutions was seen as evidence of students’ ability to conceptualize the problems in different ways and an understanding of some of the critical conceptual features. These solutions were also seen as an indirect measure of their knowledge gaps as well as activation and differentiation of prior knowledge (Kapur, 2012, 2014). The ensuing scaffolded and instruction driven Phase II provided an opportunity to build upon students’ prior knowledge, knowledge, and conceptual gaps. Facilitators led comparisons and contrasts among student generated solutions and student generated solutions with canonical solutions provided opportunities to address these gaps. These discussion sessions “afforded opportunities to attend to critical conceptual features that are necessary to develop deeper conceptual understanding” (Kapur, 2015, p.9).

The focus of Productive Failure is to design conditions where scaffolding is not available from the get go. This is in contrast to research and practice in learning sciences to design scaffolds in computer-supported collaborative learning (CSCL); Direct Instruction (Kirschner Sweller & Clark, 2006; Sweller, Clark & Kirschner, 2007) and problem-based learning settings (Davis & Miyake, 2004; Hmelo-Silver, 2004; Puntambaker, 2015). These scaffolds ranged from structuring the problem itself, metacognitive supports through reflection prompts, think aloud protocols, tools to support group discussions, designing activities to support task interdependence and collaboration, instructional materials, facilitation, on-time feedbacks, expert help and so on (Ge & Land, 2003; Hmelo-Silver, Duncan & Chinn, 2007; Kapur, 2009; Lin et al. 2012; Stahl, 2000). The argument for providing scaffolds or support structures centers around the Vygotskian (1978) argument on Zone of Proximal Development (ZPD). Briefly put, when learners are faced with complex problems that are beyond their existing knowledge and skill level, scaffolds, and presence of a knowledgeable other (expert, facilitator) can help learners solve those problems and aid their learning. The belief is that lack of scaffolds in problem solving, and collaborative settings may result in failure and hinder students’ learning potential. The efficacy of scaffolds is empirically shown and well documented in learning sciences (Brinkerhoff &
While this research showed that there was value in scaffolding ill-structured problems and collaborative problem solving, it did not imply that there was little to no efficacy in unscaffolded, ill-structured problem-solving processes. Kapur’s (2009) argument for Productive Failure advanced this possibility of latent efficacy in unscaffolded problem solving processes:

“I argue that in spite of knowing the kinds of help students need when learning something new, it may well be productive to withhold such help initially even if it leads to failure in the process…It is also reasonable to argue that in providing structure and support, scaffolds impose a certain amount of order on the problem and solution process and in doing so limit their exploration. [The argument for Productive Failure] is to allow for the concomitant possibility that under certain conditions, even ill-structured, complex, divergent, and seemingly unproductive processes have a hidden efficacy about them.” (p.22)

The premise of this approach was that scaffolding and structure in problem solving can limit the degree of freedom and learner’s exploration of the problems to procedural directions and thinking specified and encouraged through scaffold and structure. So, designing conditions for unscaffolded and student directed problem solving, could be effective for learning in the long term (Kapur, 2008, 2010, 2012, 2015; Kapur & Kinzer, 2009; Pathak et al., 2008). Productive Failure design empirically showed that this practice of leaving groups of learners to initially struggle with ill-structured problems could be a productive exercise in the learning process and could aid conceptual understanding of the problems given that this unguided discovery phase was followed up by scaffolded activities (instructor led discussions). It should be noted that PF design was not an argument against and an alternative to scaffolding and structuring problem solving and collaborative settings. But it was proposed as a design that could coexist with problem-based learning (PBL) and scaffolded learning designs.
Building from Failure Driven Learning design, Minimal Guidance, and Impasse Driven Learning

Learning designs that focus on delayed structure, designing for difficulties and impasses in the learning process and design are not new. Clifford’s theory (1998) of constructive failure suggested learning designs with optimal challenges in problem solving contexts with payoffs increasing with task difficulty. The risk to reward ratio on these designs resulted in an increase in student risk taking abilities, tolerance to making errors, motivation for multiple trials as well as maximized learning in the long run. Schmidt and Bjork (1992) found that designs that introduced desirable difficulties for the learners while training motor and verbal skills could enhance learning in the long term even though such difficult conditions resulted in less-than-ideal performance in the short term. These designed difficulties included variations in the way tasks were ordered for practice (random order), reducing and varying feedbacks and variability during practice. It was concluded that learning designs that supported optimal performance during training can be detrimental long term, however learning designs that are designed for difficulties supported learning in the long term.

In Schwartz and Martin’s (2004) Inventing to Prepare for Learning (IPL) cycle, groups of students were given ill-structured statistics problems to solve on their own without scaffolding. Student groups were asked to come up with their own solutions and representations to the problems. Students were then chosen at random to explain other groups’ non-canonical solutions, articulating their method of coming to the solution and how the solution addressed the problem. Teacher’s role during these presentations was to help students articulate and explain their solutions and provide feedback on the ways that students could come close to the canonical solution without describing the canonical solution. After student-driven presentations, the teacher led a direct instruction, class discussion where the facilitator described the canonical solution as well as knowledge gaps and effective articulation in each student generated solution. This was followed by subsequent practice with additional ill-structured problems. IPL cycle was found to be effective in students evolving understanding of the problem and knowledge of concepts associated with the problems. The invention phase, followed by direct instruction and
subsequent practice was found to be helpful to articulate their current knowledge, identify their knowledge gaps and build knowledge through iterative practice. It was also an initial proof of the latent efficacy in unscaffolded problem exploration and solution generation activities especially for subsequent learning when such activities preceded Direct Instruction.

There is also another body of research that discusses the role of designed impasses or failure to facilitate learning. Impasse driven learning theory explained how learning was facilitated in coached problem-solving settings when students reached an impasse (Jones & Van Lehn, 1994; Van Len, 1988, 1999; Van Lehn et al. 2003). Impasse was defined as a period of stagnancy during problem solving situations caused by students’ lack of prior knowledge, skills, conceptual understanding, familiarity with the tasks and level of uncertainty (Darabi, Arrington, & Sayilir, 2018; Van Lehn 1988). According to Van Lehn (1988) students could either repair the impasse on their own or seek help from tutors, reference materials and compare their solutions to their friends. Two interesting findings were evident in this research. First, it was found that learning was more likely to occur when students got stuck (impasse) while solving ill-structured mathematics problems and scaffolding i.e., on-time feedbacks, think aloud sessions, leading prompts were withheld until students reached an impasse. Students were found to generate incomplete and buggy solutions in the beginning; however this was helpful to develop understanding of the problem, procedural steps, and associated concepts. Second, subsequent instruction, scaffolding after students tried repairing the problems was found to aid learning because students could identify their knowledge gaps, missed procedural steps as well as how to apply the help to increase their future performance. Similar findings were reported in Bradley and Noell (2018), Disessa et al. (1991), D’Melo et al. (2014), Schwartz and Bransford, (1998), Schwartz and Martin, 2004, Roll et al. (2011), and Westermann and Rummel (2012). These studies represent a growing body of research that emphasized the latent efficacy in delaying scaffolds, Direct Instruction (DI) and the need to design conditions for difficulties during problem solving. These studies also emphasized the need for learning designs that provided opportunities for active solution generation and student driven repair efforts of those difficulties even though student efforts may not be initially correct.
Kapur (2008, 2012) addressed the need to design for difficulties, delay structure, scaffold and provide opportunities for exploration of the problems by proposing a Productive Failure learning design that embodied four core interdependent mechanisms. These mechanisms were a) activation and differentiation of prior knowledge in relation to the targeted concepts, b) attention to critical conceptual features of the targeted concepts, c) explanation and elaboration of these features and d) organization and assembly of the critical conceptual features into the targeted concepts (Kapur & Bielaczyc, 2012). These mechanisms were expressed in a two-phase design (See fig 2-1) where Phase 1 was a generation and exploration phase where students had opportunities to explore the problems, generate solutions in groups without scaffolding and investigate the affordances and constraints of those solutions. This was followed by Phase 2 which was a knowledge assembly and consolidation phase where facilitator led discussion sessions (scaffolding) were provided to address knowledge gaps and conceptual understanding in student generated solutions to move towards generation of canonical solutions. The design of both phases and subsequent creation of activities, collaboration, and learning environment (social surround) were guided by three core design principles. These design principles also embodied the four core interdependent mechanisms of the PF learning design

a) Design problem-solving environment where students work on complex problems that are challenging yet engaging and do not frustrate, where students can draw upon prior knowledge and problems that allow multiple solutions (embodied mechanism a and b)

b) Provide opportunities to elaborate and explain of solutions (embodied mechanism b and c)

c) Provide opportunities to compare and contrast the pros and limitations of “incorrect” or suboptimal solutions and opportunities to work towards generating canonical solutions

Implementation of these design principles in Phase I and II of the PF learning design is represented in the following table. The next section discusses the implementation of PF design in problem-solving contexts and discusses the efficacy and limitations of the PF design.
Implementation and Efficacy of Productive Failure Learning Designs

Implementation of PF design was completed in two phases. In Phase I, students were given ill-structured mathematics and science problems (finding Standard Deviation, average speed, Newtonian kinematics) that were challenging yet engaging, and activated and differentiated students’ prior knowledge on the subject (See Figure 2-2). This calibration of the “sweet spot” was done through multiple pilot testing, subsequent refinement and implementation as well as administering pre-tests to determine prior knowledge (Kapur & Bielaczyc, 2012, p. 50). Details on the results of iterative pilot testing, problem refinement and decisions for final selection of problems were not elaborated in PF literature. Students worked in groups to generate their solutions and representations for the problem. They had opportunities to explain, elaborate on individual solutions to the group, collaboratively critique, evaluate other answers and decide on solutions that were going to be shared with the class during phase II. Implementation of PF design also had a “social surround” component focusing on creating a safe space to work on exploration and solution generation (Kapur & Bielaczyc, 2012, p. 52). This included
facilitators assuring students that the expectation was to generate solutions collaboratively rather than get the correct answer and that it was okay to be stuck. Facilitators were encouraged to provide affective support but refrain from providing content related or cognitive support to create an environment where students had agency to be metacognitive, use their unique ways of thinking and reasoning about the problem and apply their prior knowledge, experience to generate solutions.

Phase II implementation of PF design consisted of providing instructional structure and scaffolding to students to help students understand primary concepts in those problems and assemble student-generated solutions into canonical solutions. This was achieved through a facilitator led whole class discussion where student groups shared their solutions to the problems. The facilitator summarized students’ solutions to draw attention to conceptual accuracy and knowledge gaps in student solutions, asked clarifying questions and encouraged other students to ask questions, and requested student groups to elaborate on some aspects of their solutions. Facilitator also shared the canonical way of solving and representing the problem, drawing comparisons and contrasts between the canonical and student generated solutions. The goal of this phase was for students to compare and contrast their solutions with the canonical solution to understand the affordances and conceptual limitations and knowledge gaps of their solutions. The goal was also to move students towards generating canonical solutions. This was done through practice sessions where students were given well-structured problems with just one canonical solution.

Figure 2-2: PF design vs Design Instruction

PF design was implemented in formal school settings in Singapore and India with ninth to eleventh grade students (See Kapur, 2008, 2010, 2012, 2015; Kapur & Bielaczyc, 2012). In India students were given two car-accident scenario problems based on Newtonian kinematics. In Singapore students were given data analysis problems where students were given a table showing the number of goals scored each year by three soccer players over 20 years. Students were asked to quantitatively determine the most consistent player with the canonical solution being that the students had to find and compare Standard Deviation of the three players to see how far away their numbers in respect to their average goals are. The problems were validated by pilot studies and with the help of teachers (subject matter experts) who validated the problems, checked for readability and language. To check the efficacy of the PF design, a comparison of learning from PF and Direct Instruction (DI) was conducted through a pre-posttest, quasi-experimental study.

Pre-test was conducted with multi-choice questions to check the extent of students’ prior knowledge on Newtonian kinematics (India) and Variance, SD (Singapore). Concurrent with PF design, students in the PF condition worked in triads to generate multiple solutions to the problems without any cognitive guidance or instructional support (Phase I). The consolidation phase consisted of student presentation of solutions, facilitator comparing and contrasting student generated solutions with each other, whole class discussion on those solutions with explanation of canonical solution and then individual practice sessions plus whole class discussion with new problems geared towards canonical solution generation. Students in DI condition were first given an instruction session where the facilitator explained the primary concepts associated with the problems (SD, variance, Newton’s 2nd law, kinematics) and also provided worked examples showing procedural steps to reach canonical solutions.

After each worked example, students then worked in triads to solve a similar problem followed by whole class discussion where teachers pointed out errors, knowledge, and concepts gaps. This was followed by another practice session and teacher led discussion of the canonical solution. Student groups then worked on the same problems that the PF group worked on in Phase I (exploration & solution
generation) followed by a final individual practice session (PF group received the same final practice problems) and whole class discussion.

When process findings were compared with their DI counterparts, PF groups were found to generate an average of six solutions but did not generate canonical solutions. Their performance by traditional standard of success i.e., getting the correct, canonical solution was very low both individually and in groups. DI groups in contrast performed well in problems given after each worked out example in the first instruction session as well as on the problems that were given to both PF and DI groups. Students in DI groups used only canonical procedural steps and were successful in consistently generating canonical solutions. However, it was noted that this performance success was due in part to close process monitoring by facilitator, showcased worked examples followed by discussion of step-by-step canonical procedures, and consistent scaffolding and feedback from facilitators. While PF groups did not generate canonical solutions, it was found that the generation of diverse non-canonical solutions was correlated with higher learning outcomes. The solutions generated by PF groups showed that their prior knowledge was activated. Similarly, the diversity of solutions showed that students in PF groups could use their prior knowledge to think about the problems in different ways to come up with additional solutions. Kapur (2015) argued that solution generation was an indirect measure of “knowledge activation and differentiation; the greater the number of such solutions, the greater the knowledge activation and differentiation” (p. 7). Third, analysis of pre-post performance between DI and PF groups on items measuring procedural fluency (ability to correctly and efficiently apply mathematical procedures/steps), conceptual understanding and transfer showed that students in PF groups significantly outperformed students in DI groups in conceptual understanding, transfer and did relatively well on procedural fluency. Number of solutions generated by PF groups was found to be a significant predictor of their conceptual understanding and procedural fluency and was referred to as the solution generation effect.

The efficacy of PF design was further replicated and reinforced by follow up and independently conducted studies with domain specific problems in mathematics (Kapur, 2010, 2011, 2012; Kapur & Bielaczyc, 2012) and science PF design with the unscaffolded problem solving and solution generation.
Follow up studies added to the efficacy of the PF design but also tested out questions and line of inquiry around PF design that was not addressed in seminal work (Kapur, 2015). One obvious line of inquiry was if the number of solutions generated by students is a significant predictor of their conceptual understanding and consequent learning from PF then wouldn't the math ability of students directly affect how many solutions students generate and their learning from PF. Kapur and Bielaczyc (2012) tested this line of inquiry in Singapore where students were sampled from three public, co-ed schools with varying math ability (high, medium, and low) determined by results on the standardized national exams. Students in each school were assigned to PF and DI and the condition of implementation was the same as earlier study (Kapur, 2010). The pre-post performance analysis however showed that PF students outperformed DI students. Solution generation effect was replicated (see paragraph above). Most importantly, a significant difference was not found between solution generation among students with different math ability. Also, students from all three ability groups were able to learn better from PF design than DI.

Two other lines of inquiry related to investigating the efficacy of PF designs were a) effect of guided vs unguided generation and b) effect of solution generation vs study and evaluation of failed solutions. The first inquiry tested the following question: if solution generation directly impacts conceptual understanding and learning from PF then would not providing cognitive guidance and facilitation result in moderating cognitive load resulting in more solution generation and subsequently more learning from PF? This was tested in a public co-ed school in Singapore with 109 grade 7 students (Kapur 2011). There were three groups; a PF group who did not receive guidance on Exploration and Generation Phase I, a guided generation group which was the same as the PF group with one exception-this group received guidance and support in the generation and exploration phase and third, a DI group. Other implementation conditions were the same as the previous study (Kapur, 2010). The guided generation group received cognitive guidance and facilitation i.e. facilitators clarified aspects of the problems drawing attention to primary concepts and parameters, hints on getting to correct solution generation steps, answered student questions, and provided prompts to encourage students to think aloud and explain their thinking and strategies. Findings showed that students PF group performed better than
DI and guided generation group on conceptual understanding, procedural fluency, and transfer. Similarly, students in the guided generation performed slightly better than the DI group but the performance difference was not statistically significant. This finding was replicated by Loibl and Rummel (2013).

Similar findings were reported in the second inquiry. The second inquiry tested the following question: is solution generation in PF crucial for conceptual understanding and procedural fluency or can student generated solutions from previous studies can be given to PF groups to study, evaluate and learn from other students failed or non-canonical problem-solving approaches (termed as learning from Vicarious Failure)? This learning from This was tested in two public co-ed schools in Singapore with 136 grade eight students in Singapore. Students were divided into two groups; a PF group who went through Phase I and Phase II of PF design in previous studies (Kapur, 2010, 2011) and a Vicarious Failure (VF) group who did not go through the generation and exploration Phase I but instead went through a study and evaluate phase. In this phase student groups were given six most student generated solutions by PF groups from previous studies to study, evaluate whether each solution addressed the problem and then provide explanation (conceptually) on their evaluation. The findings showed PF groups performed better than VF groups on conceptual understanding and transfer and did relatively well on procedural fluency.

These follow up studies added further weight to the efficacy of the PF design and learning that occurred due to student exploration and generation of solutions and the subsequent comparison, contrast of student solutions, presentation of canonical solutions, instruction on conceptual features of the problem and practice during the consolidation and knowledge assembly phase. The findings from guided vs unguided solution generation study reinforced that it is better to delay cognitive guidance and facilitation up until the consolidation and knowledge assembly because early scaffolding may steer students towards particular procedural processes and may not promote independent, group thinking about the problems, associated concepts. The findings from earlier PF design implementation (Kapur, 2010), unguided vs guided solution generation (Kapur, 2011) and PF vs VF (Kapur, 2013) also showed that generation of any kind of solution i.e. non-canonical, solutions that reflected incomplete procedural fluency, conceptual understanding and incorrect solutions provided opportunities to learn during the consolidation and
knowledge assembly phase. Generation and exploration phase in PF designs were also found to activate prior knowledge and draw attention to crucial conceptual features which were again crucial for learning and problem solving. The findings also pointed out that timing of scaffolding and support can be crucial to learning. This reiterated the premise that PF designs did not discount or call for replacement of scaffolding and supports actively implemented in Problem-Based Learning settings and CSCL settings with ill-structured problems. But PF design showed that a concomitant possibility existed where under certain conditions (detailed in PF mechanisms and findings), leaving learners to struggle and even fail (by traditional definition of failure) at problem solving complex problems without support can be productive for in the long run.

This concomitant possibility of unstructured problem-solving, role of solution generation and productive failure and its effect on activation of prior knowledge, learning and conceptual understanding, has also been explored in open-ended problem settings within informal learning spaces and Maker workshops.

Productive Failure and Open-Ended Problems

This section will outline research that explored productive failure and productive failure learning design in open ended design problems specifically within maker workshops and CSCL. Productive Failure learning designs embody two key aspects that are embedded in CSCL and makerspace related work: a) design conditions for students to make mistakes, and b) design conditions for multiple attempts despite initial inability (such as cognitive, conceptual, skill-based inabilities).

These aspects are embodied increasingly in pedagogy around making and CSCL groups (Brennan & Resnick, 2012; Blikstein, 2013; Kafai et al. 2014; Litts et al. 2016; Maltese, Simpson, & Anderson, 2018; Peppler 2013). Productive failure is embedded through computational practices such as testing and debugging, being incremental and iterative, abstracting, and modularizing tasks (Brennan & Resnick, 2012). These practices were seen to be incorporated universally in varying degrees while working with various maker toolkits (Kafai et al. 2013, 2014; Richard & Giri, 2017). Specifically, with Scratch, these
computational practices along with other dimensions of computational thinking (computational concepts and perceptions) have been heavily studied and implemented (Dasgupta, Hale & Hernandez, 2016; Fields et al. 2014; Kafai & Burke, 2013; Kafai, 2016; Lye & Koh, 2014). Computational practices while remixing Scratch projects were shown to support conceptual understanding of computational concepts, improve the quality of Scratch games that were produced and aid in expertise development in Scratch (Maloney et al. 2008; Monroy-Hernandez, 2012; Roque, Kafai & Fields, 2012).

Productive failure was also embedded in the affordances of the toolkits themselves. The design of the tools such as Scratch, Lilypad Arduino, and Makey Makey allowed for initial exploration and solution generation (iterative design, prototyping, troubleshooting, redesign) that led to failed solutions. The design of the toolkits as artifacts to construct and build upon encourages testing, debugging, experimenting, and iterating. The constructional build its aspect of the tools intuitively supported exploration, tinkering, iterative solution generation which form one of the core design principles of Productive Failure. These are aspects of productive failure design that were already embedded in the maker ethos and the toolkits and promoted learning (Litts et al. 2016). While productive failure is embedded in Scratch platform, Lye and Koh (2014) discussed the need to design a “problem-solving learning environment, with information processing, scaffolding and reflection activities” (p. 51).

Recently, design for productive failure has gained traction within makerspaces. Kafai et al. (2019) discussed the need to rethink debugging as Productive Failure. Fields et al. (2012), and Kafai et al. (2014) initially incorporated Debuggems as an assessment tool to gauge students' learning while working with open-ended e-textiles problems. These Debuggems were productive failure exercises where students were provided with e-textile designs consisting of faulty circuits and incorrect codes. Students were shown to encounter multiple initial failures as they navigated through these Debuggems to first identify problems in these e-textile designs and then figure out a way to correct these problems.

Emergent research explored productive failure designs in open ended e-textile based design problems. Litts et al. (2016) explored the potential of productive failure as a learning design to solve open-ended e-textiles challenges in an eight-week workshop with 16 high school freshmen. Their design
incorporated two exploration phases that covered two open-ended design problems pervasive in e-textile design: crafting with conductive thread and designing spatial circuity. However, this design did not include a guided consolidation and knowledge building phase. Additionally, learning outcomes were not collected for these open-ended problems. Rather, students’ perceptions and reflections on failure were reported.

This emergent research highlighted the need for Productive Failure learning designs in open-ended problem-solving settings to better understand how such designs can support learning in those settings.

Summary of Productive Failure Literature

In the previous section, I laid out the historical lineage tracing the role of failure in learning, laying out the body of research empirically showing the constructive role of failure as well another body of research outlining the debilitating effects of failure including deficit frames, stereotype threat, learned helplessness and negative affect. This was followed by the positionality of this dissertation on those two bodies of research and the reasons for alignment with research showing the constructive effects of failure on learning and associated productive failure research. I then outlined the Productive Failure (PF) literature highlighting the PF design, design mechanism and principles, implementation, and efficacy of PF designs in well-defined mathematical and open-ended problems in STEM and Maker settings. What was apparent from this literature was the latent efficacy in failure (categorized as generation of suboptimal, creative, and incomplete solutions as opposed to canonical ones) and delayed scaffolding during problem solving. Productive Failure learning designs showed that unscaffolded exploration and collaborative solution generation can be effective towards students’ conceptual understanding, procedural fluency and maximizing learning given that this phase was followed by scaffolded knowledge assembly and consolidation phase.

It was also apparent that there is a gap in design and implementation of Productive Failure learning designs in open-ended problem settings noted in recent literature in the field (Kafai et al., 2019;
Litts et al. 2016; Searle, Litts & Kafai, 2018). There is also a need to contribute to the growing understanding of how to design for Productive Failure in open ended problem settings and how such a design could support students' learning and problem solving. There have been ongoing efforts to address these limitations and this dissertation seeks to contribute to ongoing efforts and address the limitations by designing and implementing a debugging mini curriculum based on PF design to analyze the efficacy of such a design on students' learning and understanding of computational concepts.

Supporting Novices Programming and Computational Thinking

In this section, I will highlight prior research on debugging and programming for novices. I will outline the challenges novices face and frequent error patterns they struggle with, specifically in the context of Scratch, a block-based programming platform used in this dissertation. I will end the section by outlining how Productive Failure (PF) pedagogical design can be an effective approach to support novices learning and understanding of fundamental computational concepts.

Novice Programming and Debugging

Programming is a necessary skill for a technology focused workforce. However, learning to program is challenging for novices. Novices face difficulties in understanding and applying fundamental programming concepts. “Bad debugging practices”\(^2\) are an unintended consequence of novice programming (Meerbaum-Salant, Armoni and Ari, 2011). Due to the limited prior knowledge and solid conceptual grounding in programming fundamentals, novices relied heavily on first solutions, depth-first approaches to problem-solving and bottom-up, experimentation programming to generate optimal codes through brute force (Griffin, Kaplan & Burke, 2012; Li et al. 2019; Meerbaum-Salant, Armoni & Ari, 2011).

\(^2\) Considered debugging practices commonly made by novice programmers where programmers relied on fine grained programming and bottom-up solutions that often led to codes that were inefficient, difficult to scale up, and awkward. See Meerbaum-Salant, Armoni, and Ari, (2011), and Frädrich et al. (2020) for a more in-depth explanation of bad debugging practices.
Novices change their codes multiple times adding high frequency errors to an already buggy source code without conducting detailed error-location, developing test-cases and hypotheses (Chmiel & Michael, 2004). Additionally, recurring bugs, syntax errors and code smells are common struggles for novices (Frädrich et al. 2020, Fraser, 2021).

Frädrich et al. (2020) cataloged recurring error patterns in Scratch categorized into code smells, syntax errors, and general bugs. Table 2-1 shows the recurring error patterns novice programmers run into with examples. Code smells are hints or characteristics in the written code that indicate weak spots and deeper problems within the code (Beck & Fowler, 1999; Fraser et al. 2021). Smelly codes are indicators of quality and problems in a written code or generated solution for a programming problem. As such, code smells are not technically incorrect and the written code may even be functional, however the code is not optimized, is difficult to understand, remix and increases the likelihood of future bugs and failures (Martin, 2009; Yamashita & Moonen, 2012). Additionally, recurring code smells can also hinder the learning progress and performance of novice programmers. Hermans and Aivaloglou (2016) empirically showed that code smells in Scratch such as Long Scripts negatively impacts novice programmers trying to understand the game and systems logic while code smells like Duplications made it harder for them to modify, remix and scale up Scratch programs. Frequently occurring code smells included sequential actions, misordered code blocks, long scripts, redundant/duplicate codes, timing and synchronization, unnecessary conditionals and loops and dead codes. These recurring code smells indicated problems in understanding and applying fundamental code design principles and computational concepts. Additionally, the frequency of smelly codes also signaled the implementation of undisciplined or bad coding practices and debugging strategies specific to novice programmers. Specifically, these practices included bottom-up programming whereby students experiment with random blocks without attention to code comprehension, problem-analysis, error location and hypothesis testing.

Syntax errors in programming are mistakes where programmers put an incorrect character or string in their scripts causing disruption in code execution. Common syntax errors include missing parenthesis, semi-colons, undeclared, missing, or misspelled variables and incomplete or missing return
statements. In Scratch, however, typical syntax errors common in object-oriented programming do not exist. Scratch was designed to prevent syntax errors and provide low barriers to entry especially for novice programmers (Resnick et al., 2009). Snapping blocks together prevented syntax errors such as missing punctuation (quotes, brackets), typos and incorrectly spelled variables and statements that are common in object-oriented programming. However, there are exceptions. Fraser (2021) noted that syntax errors can arise from using incorrect reporter blocks such as X position, and Answer, inside Operator container blocks such as _ + _, _ and _ as well as Sensing container blocks are considered syntax errors. In addition, missing termination conditions, missing conditions in conditional container blocks, and deleting, writing incomplete codes also fall under syntax errors.

General bugs are errors that can occur in any programming language. Recurring occurrences of these bugs signal problems in the code. These bugs affect programmer’s problem-solving process and are the result of misunderstandings in programming concepts, overreliance on informal heuristics (anchoring, availability, first solutions, trial, and error) and misconceptions (Fraser et al. 2021). These pesky bugs are also the cause of frustration, discouragement, endless debugging attempts, and hinder students' enjoyment of programming and their learning trajectory (Frädrich et al. 2021). Common and most recurring bugs included missing or uninitialized loops, events, conditionals, and variables; interrupted loop sensing; and no working scripts/incomplete codes.

Table 2-1: Frädrich et al’s (2020) catalog of Scratch error patterns.

<table>
<thead>
<tr>
<th>Errors</th>
<th>Definition and Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Choosing random, incorrect code blocks</td>
<td>Chosen code blocks do not complete the intended actions.</td>
</tr>
<tr>
<td></td>
<td>E.g.: Task- Move sprite right when the right key is pressed.</td>
</tr>
<tr>
<td></td>
<td>![Correct code (left); Buggy code (right)]</td>
</tr>
<tr>
<td>Misordered code blocks</td>
<td>Code blocks are not ordered according to the intended code logic. As a result, the intended actions were either partially</td>
</tr>
</tbody>
</table>
completed or not completed. Representative of lack of understanding and application of Sequences.

E.g.: Task- Repeat this action 8 times → Move the sprite 30 steps, Wait, Make a sound

![Correct code (left); Buggy code (right)]

**Missing and Unused Initialization/Events**

Event blocks that initialize/run the code are missing. Additionally, Event blocks are added but follow up scripts are missing. Both result in Dead code.

E.g.: Task- If Space key pressed, get the sprite to move 30 steps and if touching the edge, move the sprite to a set location

![Correct code (left); Buggy code (right)]

**Sequential Actions**

Code blocks that repeat an action were duplicated and added on instead of adding a Forever or Repeat loop that optimized the written code. Representative of lack of understanding and application of Loops

E.g.: Task- Get the sprite to repeatedly glide to different locations, wait 1 second, switch costume

![Functional code (left); Non-functional code (right)]

**Missing Loop and Loop Sensing**

Scripts requiring sprite actions to repeatedly execute were not completed due to missing loops. Also, a script that is supposed to execute actions conditionally when an Event occurs, is missing the Forever or Repeat until loop. Without the loop, occurrence of the Event is only checked once and executed without the condition being met.

E.g.: Task- Get the sprite to Forever go to a specific location
If the sprite is touching the edge

Functional code (left); Non-functional code (right)

**Interrupted Loop Sensing**

One or more code blocks interrupt the checking of the condition in the If-Then when the sequence is executed. The condition may still occur, but the script does not work as written because the script is busy executing other actions in the code. Parallel scripts can resolve this.

Non-functional code (left); Functional code (right)

**Missing and Incomplete Conditionals**

Scripts requiring an action to be performed under a given condition were not completed due to missing conditional blocks such as If-Then, If-Else and Repeat until. Also, containers inside the conditional blocks defining the conditions were incorrect or incomplete.

**Missing termination condition**

Absence of termination conditions for Repeat Until blocks. Without the termination condition an infinite loop is created which hinders other code executions.

E.g.: Task- Get sprite A to move 10 steps. Repeat until the distance between the sprite A and B is less than 50. If the distance is less than 50 then get it to move -10 steps

Functional code (left); Non-functional code (right)
<table>
<thead>
<tr>
<th>Unused or Uninitialized Variables</th>
<th>Some scripts require variables to store data/numeric values. These variables were not created, added, used and/or initialized.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unused or Uninitialized Parallel actions</td>
<td>Parallel actions were not created, added, used and/or initialized.</td>
</tr>
<tr>
<td>Timing and Synchronization</td>
<td>Missing control blocks such as Repeat Until and Wait _ seconds that synchronize multiple sprite actions or control time between each action. Timing is not calibrated resulting in multiple actions executed before or after unintended times</td>
</tr>
</tbody>
</table>
| Other General bugs | **Incomplete code:** Code blocks lying around that have not been completed or initialized may hinder other code executions.  
**Missing code blocks:** Code blocks that complete an action or support other sprite actions are missing resulting in problems with code executions  
**Redundant codes:** Same code repeated which affects synchronization and code execution  
**Long Script:** Script is long which affects synchronization and execution of other scripts in the code.  
**Additional conditional or loops:** Adding loops and conditionals in scripts when unnecessary and which affects code execution and intended sprite actions. |

Past efforts to teach novices programming through pair programming (Umpathy & Ritzhaupt, 2017), constructionist gaming (Kafai, 2006) and project-based learning utilizing different “low floors, high ceilings and wide walls” platforms have been successful (Resnick, et al, 2009, p. 3). Novice-friendly programming platforms such as Alice, Scratch and Blockly have reduced the learning curve in addition to supporting novices' interest and engagement in programming (Maloney et al. 2008). Building from related work, I propose that Productive Failure-based learning designs can also be an effective approach to supporting youth and novices when learning programming and fundamental computational concepts. Productive Failure Learning Design can be used to reinforce effective debugging processes in novice
programmers. Engaging novice programmers in generating optimal solutions for open-ended debugging problems without initial scaffolding and direct instruction can support knowledge building and programming learning. Additionally, initial failures and iterative refinements of solutions followed by facilitator feedback may also support efficient debugging processes to encourage efficient code development.

**Theoretical Framework**

In the section below, I will discuss my theoretical framework that served as a foundation to define my research questions and understand the design and implementation of Productive Failure to identify knowledge gaps and design the debugging curriculum. Additionally, my theoretical framework also provided a lens to conceptualize, interpret, and analyze nuanced aspects of failure driven learning and understand the cognitive processes and strategies that goes into solving ill-structured problems.

**Constructivism**

Constructivism is a learning theory which posits that learning is an active process where learners construct new ideas, concepts, and knowledge based on prior knowledge and experiences. Learning is not a passive transmission and internalization of information acquired from knowledgeable others and the environment. Instead, learning happens in context through active student participation in problem-solving and critical thinking activities. In these activities, students are constructing knowledge and building on prior knowledge by testing and constructing hypotheses, using interpretations of their subjective experiences, cognitive structure (mental models, schema) and making meaning/sense of information, problems, tasks at hand (Bruner, 1966).

Piaget’s (1977) cognitive constructivist perspective framed knowledge as individual meaning making and making sense of the world through constant processing and integrating new information into evolving cognitive schemas, mental models. Learning was an active process of accommodation and assimilation of new information into existing cognitive structure. Learning was self-regulated, based on
self-discovery, dependent on developmental stages regulated by age (maturation) and the role of facilitators was to help students interact with their environment, new information.

Vygotsky’s (1986) socio-constructivist perspective framed knowledge as socially and culturally constructed and mediated by language. Learning was a social process of developing understanding, collaborative meaning making and knowledge construction through interaction with others and the environment and culture (Prawat & Floden, 1994). Learning preceded development with meaningful learning happening when learners engaged in collaborative activities, problem solving and then being internalized. The role of facilitator was to provide scaffolding, engage students in collaborative activity.

These two perspectives makeup the major strands of constructivism but constructivist perspective is not a binary perspective but a continuum where multiple definitions and interpretations of learning, knowledge, reality, role of memory, cognition, knowledge construction and conceptual change have been put forward. This dissertation will not go through all variations but will provide constructivist perspectives that have been crucial for my own conceptualization of learning, how learning happens in context, role of learners and teachers and the selection and development of pedagogies. My conceptualization of learning is guided by parts of Bruner’s (1961, 1973) constructivist perspective, cognitive constructivism, and socio-constructivist perspective. Bruner’s constructivist perspective framed learning as an active and social process where learners construct knowledge based on their current/past knowledge. In this view learners select, organize, transform, categorize, and integrate new information into “generic coding systems that permit one to go beyond the data to new and possibly fruitful prediction” (Bruner, 1957, p. 234). The coding systems were to be discovered and then developed by learners themselves rather than through external forces (facilitators). The concept of discovery learning implies that learning is autonomous, independent where students discover new principles and construct new knowledge as opposed to being taught directly. The role of facilitator was to provide the information that students need without structuring this information, provide minimal guidance while designing conditions to support discovery learning.
In my conceptualization, learning is an individual and social process of meaning making, constructing knowledge leveraging and building upon prior knowledge which is aided by designs and activities that provide students with agency to self-discover as well as supports, structure and scaffolding to aid students conceptual understanding and knowledge gaps. In this conception, I believe that prior knowledge can be crucial to integrate new information but can be detrimental to learning if students’ prior knowledge is limited, if students are faced with problems without foundational knowledge and if their prior knowledge consists of misconceptions and gaps. Similarly, I also argue that minimally guided and unguided discovery learning, prominent aspects of cognitive constructivist and Bruner’s constructivist perspective, can hinder learning. Proponents of direct instruction have made similar arguments against discovery learning citing reasons such as cognitive load on students, burden on working memory with trial and error, time consumption, generation of suboptimal solutions as detrimental to learning (Kirschner, Sweller & Clark, 2006; Sweller, 2010). Direct instruction proponents suggested that while “dealing with novel information, learners should be shown what to do and how to do it” (Kirschner, Sweller & Clark, 2006, p. 79). Acknowledging the extensive research showing the prominence of direct instruction over unguided discovery learning, it can be argued that discovery learning can be effective if it is guided and collaborative because it provides learners with the agency to construct, co-construct and add to their knowledge, develop, and compare solutions. Scaffolding and support can be effective when prior knowledge is limited and can help mitigate knowledge gaps as well as perpetual misconceptions.

This line of argument was supported by Mayer’s (2004) review of research comparing unguided, guided, and direct instruction problem-solving conditions. It was found that guided discovery was the “best method” for problem-solving and learning procedural steps and programming concepts because it helped students meet two crucial active learning criteria; a) activating prior knowledge or constructing knowledge to make sense of incoming information and b) integrating new incoming information with appropriate knowledge base (Mayer, 2004, p. 15). Unguided discovery with too much freedom was found to be detrimental in the first cognitive process of constructivist learning i.e., selecting relevant information. Direct instruction could, “in some cases, promote the cognitive processing needed for
constructivist learning, but in others, some mixture of guidance and exploration is needed” (Mayer, 2004, p. 17). Mayer concluded that a commitment to constructivist epistemology does not automatically equate to a commitment to unguided/pure discovery learning and associated methods of teaching:

“Overall, the constructivist view of learning may be best supported by methods of instruction that enable deep understanding of targeted concepts, principles, and strategies—even when such methods involve guidance and structure…There is increasing evidence that effective methods for promoting constructivist learning involve cognitive activity rather than behavioral activity, mix of instructional guidance and curricular focus rather than just pure discovery and unstructured exploration” (Mayer, 2004, p.14)

This view of constructivist learning is at the center of Productive Failure (PF) design which is the design basis for my debugging mini curriculum. In this constructivist pedagogy knowledge is built through initial participation in the exploration and generation phase where students generate solutions to ill-structured debugging problems collaboratively without scaffolding. Students rely on their prior knowledge (formal and intuitive) on programming, linear logic, problem-solving, metacognition and heuristics to generate multiple solutions. Domain information on Scratch and computational concepts will be provided similar to prior PF designs where students received basic domain information before going into the generation phase (Kapur, 2012). Affective support will be provided to help students persist without providing targeted content knowledge or key conceptual features. In the knowledge assembly and consolidation phase, the facilitator (the author) will compare and contrast student generated solutions with the best solution (quick, using key computational blocks, minimum code lines) pointing out key conceptual features as well as knowledge gaps in their solutions. Students then practice additional debugging problems with embedded helpful hints, metacognitive scaffolds, and facilitator support.

Learning happens through students’ active and collaborative exploration of the problems and solution generation leveraging their prior knowledge and then through scaffolded knowledge assembly and consolidation with the facilitator. This is where students’ solutions are compared and contrasted, attention is drawn towards crucial conceptual features necessary for each problem as well to knowledge and conceptual understanding and gaps in student generated solutions, best solutions to debugging problems are presented, and students are presented with scaffolded debugging sessions. Learning is also
aided by computational practice of debugging. Papert (1980) presented debugging as “a powerful addition to a person’s stock of mental tools”: a crucial part of the learning process where students are “thinking about their own thinking and learning about their own learning” (p. 155). Students discover gaps in their knowledge, conceptual understanding while iteratively solving problems, and fixing them. Deconstructing problems into smaller pieces allowed students to develop procedural knowledge and higher order thinking because it allowed students to understand processes for smaller parts of the problems which he argued could aid in reconstructing procedures to solve bigger problems.

Heuristics

“Heuristics are efficient cognitive processes, strategies that ignores part of the information, with the goal of making decisions more quickly, frugally (synonymous with effort reduction) and/or accurately than more complex methods” (Gigerenzer & Gaissmaier, 2011, p. 454)

Heuristics are approaches or strategies to problem solving and learning by discovery that focuses on plausible, provisional, useful, readily available, effort saving, fast but fallible and non-canonical solution generation (Feigenbaum & Feldman, 1963; Polya, 1954). The generated solutions may or may not be ideal, optimal or canonical but having incomplete solutions provides an avenue to iterate, revealing an individual’s prior knowledge of domain, knowledge, and cognitive biases. Simon (1955) suggested that people are limited by their cognitive abilities: computation abilities, working memory, recovery and recognition memory, information processing, differentiation, and organization. Additionally, they are also limited by the structure of task environments when making decisions or solving problems. Termed as bounded rationality, these cognitive limitations and structure of task environment with limits on time availability, information availability, processing, and computation forces individuals to use mental shortcuts i.e. cognitive heuristics to generate solutions and make decisions. This cognitive heuristic was termed as satisficing where individuals generate and pick solutions that they think are satisfactory in decision making and problem-solving contexts where optimal solutions cannot be determined or are elusive (Simon, 1955).
Heuristics were found to be “highly economical, useful and usually effective but they can lead to systemic and predictable error” and cognitive biases in certain task situations (Tversky & Kahneman, 1974, p. 1131). Other than cognitive limitations, individual emotions and motivations around certain problems, decision, and social pressure, one of the major contributing factors to these cognitive biases were the use of these mental shortcuts, specifically informal heuristics. Tversky and Kahneman (1974) noted three that were significant

- anchoring bias: tendency to over rely on the first piece of information to solve problems, make decisions
- availability heuristic: tendency to over value information that first comes to memory
- representative heuristic: tendency to make decisions or solve problems by over relying on rule of thumb strategy based on similar past experience or problems

Contrary to Tversky and Kahneman’s (1974) findings, Gigerenzer found that simple formalized heuristics rather than riddled with cognitive biases were “often more accurate than complex rational strategies even though they process less information” (Gigerenzer & Gaissmaier, 2011, p. 474). Known as the less-is-more effect, inverse-U- shaped relation was found between accuracy and amount of information, computation, and time. So, more information, computation ability and time were found to be more accurate until a point after which simple heuristics such as one-good-reason, recognition, take-the-first, fast and frugal heuristic were more accurate in decision making and problem solving. According to Gigerenzer’s (1999) theory of adaptive toolbox, individuals have a set of heuristics/cognitive mechanisms/tools hardwired by evolution; developed through individual learning, environmental structure (tasks, context, setting); and learnt through individual and social experiences in various contexts that are applied and adapted, unconsciously or consciously, to various decision making and problem scenarios (Gigerenzer & Gaissmaier, 2011). While these heuristics can lead to suboptimal solutions and biases, these heuristics also supported problem solving despite cognitive limitations of individuals. This was because these heuristics are fast and frugal, can be adapted and improved upon with subsequent applications, and can reduce cognitive demands (Czerlinski, Gigerenzer & Goldstein, 1999; Gigerenzer, 2008).
This theory of adaptive toolbox is embedded in Kapur’s (2008, 2011, 2012) PF designs which allows for informal and formal heuristics in the exploration and generation phase. In my debugging mini curriculum, based on PF design, heuristics play an important role in supporting deeper and durable learning. Students have the agency to employ heuristics that are fast and frugal, readily available, based on domain, procedural and/or complementary domain knowledge they already have on programming, logic, linear and critical thinking during the exploration and generation phase. These heuristics can mitigate the effects of cognitive and working memory limitations while activating prior knowledge. As a result, students are more likely to generate multiple solutions to problems despite these solutions not being optimal, canonical, and procedurally sound. However, as PF design blends heuristics, prior knowledge and formal knowledge, there are opportunities to address/remedy knowledge, conceptual and procedural gaps in the consolidation and knowledge assembly phase. This ensures activation and implementation of heuristics, prior knowledge, and formal knowledge (instruction, scaffolding driven) to support deeper and durable learning (Chen et al., 2018; Kapur, 2012). Problem solving, and solution generation before instruction, scaffolding (DeCaro & Rittle-Johnson, 2012; Van Lehn, 1989) and building on students’ knowledge frameworks (Linn, Davis & Eylon, 2013; Linn, 2006) have also been shown to support deeper knowledge building and conceptual advancement.

Heuristics are also embedded in and play a crucial role in the debugging process. It can be argued that since debugging in Scratch involves solving ill-structured problems, students can be limited or overwhelmed by information, restricted by incomplete conceptual understanding or prior knowledge, and faced with novel, multi-faceted problems. As such, heuristics can serve as guides to simplify the problems, reduce cognitive load (Sweller, 1983), and produce results with student’s available information, prior knowledge, experience, and cognitive limitations. It is possible that application of heuristics during debugging can be a productive, fast, and frugal way to activate prior knowledge and generate solutions during exploration and generation phase. Use of heuristics can also delineate conceptual understanding and gaps which can be addressed during the consolidation and knowledge assembly phase. Heuristics have been shown to be efficient and consistent at producing results in well-structured (Anderson, 1993;
Bransford & Stein, 1984; Newell & Simon, 1972), ill-structured problems (Chi & Glaser, 1985), and real-world problem scenarios (Gigerenzer, 2008). Heuristics in problem solving contexts has also been shown to produce comparable or even better results to complex statistical tools and algorithms which account for all available information, factors, data and employ rigorous computational processes (Gigerenzer & Gaissmaier, 2011).

Students can use informal heuristics (affect, anchoring, availability, familiarity, trial and error, social) and formal heuristics (recognition, similarity, debugging heuristics) to solve debugging problems. Strategies selected to debug Scratch problems “may be hardwired, based on what has been formally learned, based on social and cultural processes and norms, determined by the conditions surrounding the problem” (Csikar, 2018, p. 91). Novices without prior experience with Scratch may be limited in domain specific knowledge and therefore may not have complex cognitive maps to efficiently solve problems (Jonassen, 2000). As such, novices may rely on their prior knowledge, intuition, emotions to come up with solutions i.e., informal heuristics. Experts on the other hand typically have more complex cognitive maps built through experience and prior iterations with similar problems. It would be expected that they would rely on formal heuristics based on logical processes, higher level critical thinking, advanced reasoning strategies coupled with some reliance on informal heuristics (Perkins & Salomon, 1989. Jonasen, 1997).

Metacognition

Metacognition or metacognitive awareness refers to “second order cognitions: thoughts about thoughts, knowledge about knowledge, or reflections about actions” (Weinert, 1987, p. 8). Flavell (1979) defined it as “cognitive monitoring” (p. 906). Brown (1987) referred to it as an executive process. Kuhn and Dean (2004) defined metacognition as “awareness and management of one’s own thought” (p. 270). Metacognition can be generally referred to as awareness of one’s own procedural, declarative, and conditional knowledge, cognitive skills, and thinking to assess, monitor, manipulate and regulate their
cognitive processes to improve their understanding, performance and adapt their learning to new contexts and tasks (Flavell, 1992; Schraw & Dennison, 1994; Bransford, Brown & Cocking, 2000).

Schraw and Dennison (1994) reported two factors that are associated with metacognitive awareness a) Knowledge of Cognition and b) Regulation of Cognition. Knowledge of cognition is an awareness of one’s a) procedural knowledge: knowledge of how to perform a task displayed as heuristics/strategies (Schraw, 1998), b) declarative knowledge: content knowledge stated verbally and stored in memory. Also, knowledge about one’s skills, cognitive resources, abilities as a learner and factors that can affect one’s performance (Kuhn & Dean, 2004; Schraw, 1998; Schraw et al. 2006) and c) conditional knowledge: knowledge about when and how to use declarative and procedural knowledge (Garner, 1990). Knowledge of cognition is also awareness of one’s abilities, strengths and weaknesses, metacognitive strategies and when and how to use them to learn new information (Cross & Paris, 1988; Kuhn, 2000; Schraw et al. 2006). Regulation of cognition is continuous adjustments, improvements to one’s knowledge, actions, strategies through planning, monitoring, and evaluating (Cross & Paris, 1988; Lai, 2011; Whitebread et al. 2009). Planning involves identifying and selecting appropriate metacognitive strategies such as self-talk, goal setting, think aloud, re-reading, note taking, activating prior knowledge, time management, and others. Monitoring involves awareness of one’s task performance, self-testing, comprehension of tasks/problems and existing state of knowledge. Evaluation involves judging the effect of strategies on task completion and learning, reflecting on those strategies, and revising strategies used (Lai, 2011; Schraw et al. 2006;)

Metacognition is a crucial aspect and pre-requisite to solving and performing well on ill-structured problems (Aurah et al. 2011; Jacob & Paris, 1987; Shin et al. 2003). Ill-structured problems require metacognition because these problems are emergent, highly variable and may not have predictable, fixed canonical solutions (Jonassen, 2000). Similarly, these problems require various criteria to evaluate solutions and “success criteria depend on how the learner clarifies and reconciles competing solutions” (Aurah et al. 2011, p11). While well-structured problems can be solved and conceptual understanding reached with the application of established formulae, canonical procedural steps, and
repetitive practice, this is not the case with ill-structured problems. With ill-structured problems, students can improve their speed and accuracy with repeated practice. However, metacognition is needed to reach conceptual understanding and gain adaptive expertise. By employing metacognition students are not just working on problems specifically through trial and error but they are reflecting on their cognition, prior knowledge, differentiating and reflecting on solutions and adapting their approaches to the problems (Hatano & Inagaki, 1988; Hobden, 1998).

Metacognition is crucial to debugging ill-structured problems in this dissertation. Students first engage in ill-structured debugging problems in Scratch during the exploration and consolidation phase without scaffolding. Metacognition can be crucial because in this context awareness of one’s own thinking, active monitoring and regulation of cognitive process and application of heuristics can activate prior knowledge and help students devise strategies to generate solutions to attack those problems (Hennessey, 1999). Students cannot rely specifically on domain specific knowledge and as such metacognition on problem-solving processes becomes crucial (Xun & Land, 2004). This is true in my dissertation where students will receive information on Scratch platform, computational concepts, and various blocks associated with different computational concepts but will not receive domain specific knowledge on programming and logic. Students will have to rely on metacognition, heuristics, informal and formal prior knowledge to collaboratively generate solutions to debug the problems. Similarly, students can also reflect on and adapt their solutions based on feedback from the Scratch platform informing students whether the code is debugged or needs further work. Reflection on the problem-solving process and their own cognitive processes can lead to deeper understanding of the problem, awareness of their cognitive strengths and weaknesses, flexible approach to problem solving, diverse generation of solutions and subsequent development of knowledge base (Bransford, Brown & Cocking, 1999; Schmidt & Ford, 2003).
Failure Associated Cognition

“We punish students with bad grades when they offer the wrong answers on tests. Negative consequences give failure a bad name. For learning to take place, there should be expectation failure. If you do not fail when trying to accomplish something, you fail to learn” (Schank, 1997, p. 29)

According to the Schank’s (1997) notion of expectation failure, productive learning happens when students explore and generate solutions to problems with their current mental models and understanding expecting a successful outcome but receive an unexpected outcome. Expectation failure dictated that productive learning most likely occurred during initial unscaffolded problem-solving after which students were more likely to adjust their mental models because of failure outcomes. Environments which allowed for generation of creative, suboptimal solutions were found to be effective for learning because such environments exposed students to conceptual and knowledge gaps motivating students to evaluate and change their mental models and problem approach. This is concurrent with Chi’s (2013) theory of imperfect mental models where students draw inferences and audit their mental models after failure outcomes resulting in future learning and successful outcomes.

Exploration and generation phase before scaffolded consolidation and knowledge assembly design of Productive Failure embody aspects of Expected Failure principle. Exploration and solution generation phase provided initial opportunities to adjust their conceptual understanding and knowledge gaps when students encountered failures/generate suboptimal or non-canonical solutions when attempting ill-structured problems without scaffolding. Additionally, consolidation and knowledge assembly provided further opportunities when students could compare and contrast their solutions with others as well as compare it to canonical solutions provided by the facilitator.

Summary of Theoretical Framework

In the previous section, I discussed my theoretical framework which guided this study. I drew from theories from interdisciplinary fields such as developmental psychology and learning sciences to
conceptualize and understand the constructive as well as deficit-laden aspects of failure driven learning. Atkinson’s (1964) theory of achievement motivation, Dweck’s (1986) conceptualization of the growth mindset, and Duckworth’s (2007) identification of grit as a noncognitive factor guided my understanding of the motivational and positive attributional effects of failure. Clifford’s (1988) constructive failure, and Kapur’s (2008) productive failure highlighted the latent efficacy of failure in learning and problem solving to guide the development of the debugging curriculum with a focus on design for optimal difficulties of debugging problems. Conversely, Seligman and Peterson’s (2001) identification of learned helplessness, and Steele and Aronson’s (1995) conceptualization of stereotype threat guided my understanding and awareness of the deficit frames, inequitable participation, stereotype threat, and affective challenges that are especially detrimental among members from vulnerable and historically marginalized communities. As such, on-time feedback were built into the Solution Generation phase so that failures were not unchecked. Additionally, scaffolded practice and discussions of computational concepts, and ineffective debugging strategies were added to the Consolidation phase to support student learning.

To understand the design, structure, and implementation of Productive Failure, I relied on Bruner’s (1960), and Vygotsky’s (1987) Constructivism, Van Lehn’s (1988) work on impasse driven learning, and Schank’s (1997) failure assisted cognition. These bodies of work aided my understanding of the design considerations involved in integrating previously validated findings on delayed instruction and expectation failures into the two phases of PF design. Overall, constructivism promotes active learning, prior knowledge integration, and the use of heuristics metacognition, and expectation failure to support effective learning and problem solving. For example, the design of Productive Failure incorporates Schank’s (1982) principle of expectation failure. It allows students to explore and generate solutions without guidance, which ultimately leads to unexpected outcomes and failures. These failures then prompt students to adjust their mental models, and problem-solving strategies, resulting in productive learning. In a similar vein, Tversky and Kahneman’s (1974) work on heuristics plays a significant role in the process of debugging ill-structured problems during the Solution Generation phase. During this phase, students can use informal heuristics, such as trial and error, as a quick and effective approach to
iteratively solve problems. Utilizing heuristics can support activation of prior knowledge, reduction in cognitive load and facilitate identification of students’ conceptual understanding and gaps which can be addressed and further consolidated during the Knowledge Assembly phase.
Chapter 3

Methods

In this study, I used Mixed Methods and Design Based Research (DBR) approach to answer the following research questions

- **RQ 1**: How does productive failure design support learning of computational concepts when working with open-ended debugging problems in programming platforms such as Scratch?

- **RQ 2.1**: What kinds of challenges do students encounter, the resolutions they come up with and the debugging strategies they used when working through the debugging curriculum?

- **RQ 2.2**: What do these challenges, resolutions, process, and interaction patterns reveal about students’ evolving understanding of computational concepts?

In the following section, I will expand on the methods used for this study.

**Design Based Research (DBR)**

A Design-Based Research (DBR) approach was used for this study (Sandoval & Bell, 2004). The epistemic interest driving this study is to contribute to ongoing efforts and developing work on effective Productive Failure learning designs, particularly with respect to diverse open-ended problems. To this end, I focus on the implementation of a productive failure design in the context of a virtual debugging workshop in Scratch with structured exploration and consolidation phases to understand how such a design can support learning of computational concepts. As such DBR framework was selected to design and implement the debugging mini curriculum leveraging Productive Failure Design with ten high school students. DBR is an iterative, mixed method approach that aims to improve teaching and learning in authentic settings through iterative design, intervention, implementation, and analysis (Brown, 1992; Collins, 1992; Cobb et al. 2003; Edelson, 2002; McKenney & Reeves, 2013; Sandoval & Bell, 2004).
Each iterative cycle contributes to the development and improvement of the intervention, evolution of design principles as well as the development and creation of usable knowledge, practices and theories of teaching and learning (Anderson & Shattuck, 2012; DBRC, 2003; Herrington et al. 2007; Zheng, 2015). Therefore, DBR provides an opportunity to develop design knowledge, practices and advance theoretical knowledge of learning and cognition (Disessa et al., 1991; Bell, Hoadley & Linn, 2013).

The current study is the second iterative cycle of this DBR project and features the data from this cycle. However, I will explore themes such as mitigation of recurring bugs and changing debugging strategies with the data from the third iterative cycle at a later time. Three iterative cycles have been completed as part of this DBR project. The third iterative cycle was recently completed in July 2021 as an online synchronous workshop as part of a federal TRiO-supported after summer STEM camp for high school students from low-income and migrant communities in a rural part of Northeastern United States. The first iterative cycle was implemented as a pilot study in a Research Apprenticeship, LDT 594 class. IRB was not submitted for the pilot study. In the first iterative cycle I designed, and pilot tested two code debugging units (Virtual and Physical Debuggems) with graduate students. The goal was to explore the feasibility and effectiveness of interactive and self-directed code debugging in learning computational concepts. The first study highlighted applicability issues with the Physical Debuggem unit with a larger population and the need to redesign and expand the Virtual Debuggem unit to include problems that targeted all computational concepts individually and in combination. The design, implementation, and limitation of the first iterative cycle is discussed in detail in the Research Design section. The limitations in design and implementation in addition to participant feedback from the first iteration led to design decisions for the debugging mini curriculum.

**Mixed Methods**

A Mixed Methods approach was used to answer the research questions. The overarching research questions that the study aims to answer are: RQ 1. *How does productive failure design support learning of computational concepts when working with open-ended debugging problems in programming*
platforms such as Scratch? To answer this question, a 3-week curricular unit was first designed in Scratch Studio consisting of code debugging problems targeting fundamental computational concepts. The code debugging curricular unit was then implemented following the PF pedagogical design where 10 high school students first generated multiple solutions. This was followed by consolidating the concepts through facilitator led compare and contrast of student generated solutions, discussions of optimal solutions and scaffolded practice. The goal is to quantitatively understand the effectiveness of the designed intervention on students' learning and self-efficacy around programming and problem solving. Additionally, the goal is to also qualitatively understand aspects of PF design that support learning of computational concepts and how moment-to-moment discourse and interactions with the debugging problems in general mediate learning from failure. RQ 2.1 will unpack What kinds of challenges do students encounter, the resolutions they come up with and the debugging strategies they used when working through the debugging curriculum? RQ 2.2 will examine What do these challenges, resolutions, process, and interaction patterns reveal about students’ evolving understanding of computational concepts? With RQ 2, the goal is to qualitatively understand aspects of change in participants’ understanding and application of computational concepts that occurred as participants worked through various debugging challenges changing their debugging strategies and approaches to solution generation.

Approaching the study from a problem-oriented, research questions focused pragmatist research philosophy, it was logical to use a mixed methods approach employing both qualitative and quantitative methods to answer the questions (Creswell & Plano Clark, 2017; Johnson & Onwuegbuzie, 2004). A Convergent Parallel Mixed methods research design was used to answer the research questions (Jick, 1979; Creswell & Plano Clark, 2017). Convergent Parallel Design entails concurrent qualitative and quantitative data collection, independent analysis of the two types of data and interpretation of the results together (Creswell & Plano Clark, 2017).
Research Setting and Material

Setting

The Debugging mini curriculum was implemented as an online synchronous “Coding, Game Design and Problem Solving” workshop with high school students in collaboration with the Upward Bound program at Penn State University. Upward Bound is a federally funded program that conducts after school STEM and STEAM workshops and academic tutoring for high school students from low-income families, underrepresented and migrant communities in a rural part of Northeastern United States. The virtual workshop was conducted via Zoom in the fall of 2021 by Giri with an Upward Bound staff helping with attendance, technical and logistical support. The workshop was offered twice a week Tuesdays and Thursdays for 3 weeks where each session ran for two hours. Google Chromebooks were mailed to all enrolled students so that they could participate in the workshop and work through the code debugging curriculum created in Scratch Studio.

Participants

The online synchronous workshop enrolled 10 high school students from grade 10-12 between 15-18 years old. Demographically, there were 7 girls and 3 boys; 6 Latinx, 2 Asian, 2 Black (See Figure 3-1). Nine (9) students had no prior experience with coding, game design and Scratch. One student had prior experience with Scratch from participating in a summer Maker workshop organized by Richard (Richard & Giri, 2019) in collaboration with Upward Bound. Six (6) students participated in all 6 sessions of the workshop with 2 students participating in 4 sessions while 2 students attended 2 sessions.
Instructors

The researcher (Giri) was the instructor for the virtual code debugging workshop. An Upward Bound staff provided technical and logistical support and was present for all 6 sessions of the workshop. The staff member also took attendance, reminded students of upcoming sessions, and supported the instructor with workshop activities. The instructor conducted the workshop via Zoom leading all workshop activities, administering pre-posttest and online interviews after each session. This included leading the guided introduction session to Scratch and conducting think alouds during the Solution Generation phase where students were asked to discuss their solutions, debugging strategies and challenges encountered. Additionally, the instructor also conducted the Consolidation and Knowledge Assembly session where students’ generated solutions were compared with an optimal solution to address areas of conceptual understanding and conceptual gaps.

Curriculum

The debugging mini curriculum was developed iteratively by Giri over two iterative cycles of the DBR project. Individual lessons were based on *Debugems* (Griffin, Kaplan, & Burke, 2012) where students are presented with Scratch problems with strategically placed bugs to help students engage with programming in a playful way. However, unlike the Debugems, this curricular unit consisted of
debugging problems that targeted specific computational concepts individually and in combination to support students' incremental learning of computational concepts. For instance, problems in U2 targeted Events and Loops which consisted of missing sprite actions initialized by a ‘Forever’ or ‘Repeat 10 times’ blocks or Event blocks such as ‘When green flag pressed’ or ‘When space key pressed’ where students had to add in the missing blocks in sequence to debug the problem. The curricular unit consisted of 5 units with 14 debugging problems targeting fundamental computational concepts. The breakdown of the units was as follows:

- Unit 1 (U1); 3 problems = Sequences
- Unit 2 (U2); 3 problems = Events & Loops
- Unit 3 (U3); 3 problems = Conditionals & Parallelisms
- Unit 4 (U4); 3 problems = Operators & Data
- Unit 5 (U5); 2 problems = Design your own game

These problems were housed inside a debugging studio that was created for the Scratch classroom used for the workshop. Problems were presented as buggy games that participants had to troubleshoot to play. Bugs were strategically placed within the problems ranging from missing code blocks (events blocks in U2; if then, repeat until, etc. in U3) to misordered and incomplete code sequences. A narrative story was provided for each problem, which would provide a fun backstory for the challenge and a list of tasks to be completed. The design, implementation and workshop activities are discussed in detail in the next section.

**Research Design**

Convergent Parallel Mixed methods

I am approaching the research from a pragmatist research philosophy as it is pluralistic, problem-oriented, values both objective and knowledge and focused on answering research questions using both qualitative and quantitative methods (Creswell & Plano Clark, 2017; Johnson & Onwuegbuzie, 2004;
Maxcy, 2003; Teddie & Tashakkori, 2009). The goal is to understand the effectiveness of Productive Failure” (PF) pedagogical design as an approach to support novices programming learning and understanding of fundamental computational concepts. As such, I used a Convergent Parallel Mixed methods research design to carry out this study (Jick, 1979; Creswell & Plano Clark, 2017). Convergent Parallel Design entails concurrent qualitative and quantitative data collection, independent analysis of the two types of data and interpretation of the results together (Creswell & Plano Clark, 2017).

Following this design, for RQ1, I conducted a pre/post survey that participants completed at the beginning of the debugging workshop and at the end of the workshop. The validated survey instruments assessed participants' computer programming self-efficacy, understanding of computational concepts, attitudes towards computer programming learning and evaluations of their problem-solving. I analyzed pre-and post-test changes for each of the subscales through Wilcoxon matched paired tests to understand the impact of the designed intervention on students’ problem solving, understanding of computational concepts, and programming self-efficacy. I also used interaction analysis to unpack salient moments during think aloud sessions and semi-structured interviews to understand trends and trajectory of students’ learning of computational concepts and how the PF design supported participants’ learning of those concepts (RQ 1).

Interaction analysis was also used to understand students debugging strategies, moments of challenges and resolutions in addition to aspects of change in participants’ understanding of computational concepts that occurred as participants worked through various debugging problems (RQ 2). Artifact analysis was conducted in conjunction with interaction analysis to track students’ iterative attempts, changes, and edits in code sequences throughout the debugging curriculum that provided insights into their learning. Additionally, thematic analysis combining both inductive and deductive coding approaches was used to document, code and analyze salient moments of verbal and non-verbal (embodied) interactions. I focused on participants’ debugging challenges, solutions and debugging strategies to analyze their evolving learning and understanding of computational concepts.
DBR Project: Design and Implementation of Second Iterative Cycle

This section will describe the second iterative cycle of the DBR project. Design and implementation of the Debugging mini curriculum will be described in detail. In addition, this section will provide a summary of design and implementation of the first iterative cycle highlighting the lessons learnt that led to certain design and implementation decisions for the second iterative cycle.

First Iteration: Design and Implementation of Virtual and Physical Debuggems

The precursor to the current Debugging Mini Curriculum were Virtual and Physical Debuggems units that were I initiated and designed for sixth to seventh graders for Research Apprenticeship class, taught by Dr. Gabriela T. Richard (LDT 594) at Penn State University. The aim of those units was to support learning of fundamental computational concepts crucial for novice programmers through self-directed debugging process (Brennan & Resnick, 2012; Dasgupta, Hale & Monroy-Hernandez, 2016; Fields et al. 2012; Kafai et al. 2014; Kafai & Vasudevan, 2015).

Design and Implementation

Physical and virtual Debuggems were based on Debuggems (Fields et al. 2012; Griffin, Kaplan & Burke, 2012; Kafai et al. 2014). Debuggems are deconstruction kits that were designed to be broken and contain strategically planted bugs in a code sequence or e-textile design to allow students opportunities to explore and debug the problem individually or with peers. Similar to Debuggems, Virtual and Physical Debuggems were designed as a set of 4-5 coding challenges that students had to complete in online Scratch platform (Virtual) or solve using physical code block cutouts (Physical) (See Figure 3-2). However, unlike Debuggems, these problems were specifically designed to target specific computational concepts individually to support learning of computational concepts.
Students were given problems with code sequences for one or more sprite intentionally designed to be broken or buggy. Problems were presented as a narrative story. The narrative story would set the context for the challenge, instruct students of problems to be resolved and provide helpful hints to orient students’ thinking. Some code sequences had one or more missing code blocks for sprite i.e. a missing ‘If-then’ conditional crucial for sprite to perform actions when set conditions are met, a ‘Forever’ loop ‘to repeat sprite actions or a missing event such as ‘When green flag pressed’ to initiate an action. Students had to first identify the problems and collaboratively debug the problem.

The Virtual and Physical Debuggem were pilot tested in LDT 594 class with graduate students to gather feedback on the design and efficacy of the units. Graduate students in class were knowledgeable in Scratch so the units were directly implemented without initial guided and unguided exploration of Scratch
to get acquainted with the platform. Virtual *Debuggems* were administered first followed by Physical *Debuggems*. Students worked in groups to debug the problems and shared their solutions, debugging process and their feedback. Feedback on the efficacy of the units were collected through direct observation (Denzin & Lincoln, 1994), unstructured interviews and field notes (Huberman & Miles, 2002). Participants were given a printed copy of the units to make notes of the suggestions, changes, and comments to the design.

**Limitations, Lessons Learnt and Design Implications for Second Cycle**

Virtual and physical *Debuggems* were designed to support students learning and application of computational concepts by engaging in debugging through interactive and embodied approaches. By engaging in these units, I hoped that students will understand how specific code blocks were related to seven key computational concepts in *Scratch*. Similarly, I also anticipated that solving these incomplete code sequences within these two units could help deeper and contextual understanding of these computational concepts.

However, pilot testing revealed several problems with design and implementation of the units. These problems, feedback from participants and further exploration of the literature associated with constructive role of failure in learning and Productive Failure design guided and informed the second iteration. First, the units were short, and the problems did not encompass all seven computational concepts. The problems in Virtual and Physical *Debuggems* were centered around sequences, conditionals, events, and loops but did not incorporate multifaceted problems where students had to add a variable like scores, using Boolean logic with operators and data. This was identified as a limitation because it would be unreliable to gather information on learning and understanding of computational concepts when the actual debugging problems centered around some computational concepts but ignored others. Second, opportunities were not built in for knowledge consolidation which left learning of conceptual concepts ambiguous. Students were asked to attempt these problems without scaffolding and instruction during administration and a follow up discussion was not built in after students generated their
solutions. This left learning of computational concepts ambiguous because it was difficult to assess and address students’ conceptual understanding and knowledge gaps without a scaffolded follow up. The design did not incorporate a scaffolded consolidation and assembly of knowledge where facilitator could compare and contrast student generated optimal and suboptimal solutions, provide a model solution, and address conceptual gaps. Third, a concrete design to assess students’ learning and understanding of computational concepts was missing.

These limitations along with focus on relevant literature (Kafai et al., 2019; Kapur, 2008, Kapur & Bielaczyc, 2012; Litts et al. 2016, Searle, Litts & Kafai, 2018) to anchor the redesign and a need for deeper theoretical framing informed the design and implementation of the Debugging mini curriculum which will be described next.

**Second Iteration: Design and Implementation of Virtual and Physical Debuggems**

The limitations in design and implementation of the first iterative cycle informed the design and implementation of the Debugging Mini curriculum in the second iteration. Specifically, there were limitations in deeper theoretical framing on the learning design and assessment with the Virtual and Physical Debuggems. The units lacked a coherent learning design that leveraged the latent efficacy of failure driven learning while building opportunities for knowledge consolidation after multiple failed debugging attempts. As such, it was difficult to assess the trajectory of students' computational concepts learning and analyze the role of the designed units.

To address these limitations, conceptualization and implementation of the debugging mini curriculum followed the Productive Failure design (PF design) and design principles associated with PF design. This empirically based constructivist learning design consisted of 2 phases:

a) Exploration and Solution generation (Phase 1) and

b) Consolidation and Knowledge Assembly (Phase 2).

In phase 1 students generated multiple solutions to ill-structured problems without scaffolding which could lead to failure (defined as generation of suboptimal, non-canonical solutions). In phase 2 a
facilitator led a scaffolded knowledge assembly session. In this consolidation and knowledge assembly session, facilitator shared the canonical solution, compared, and contrasted student generated solutions with each other and with canonical solutions. Conceptual understanding of students as well as knowledge gaps were discussed before additional scaffolded practice sessions.

Recent literature from the learning sciences especially within the Maker domain have looked at implementation of Productive Failure designs in open ended e-textile design problems (Kafai et al., 2019; Litts et al. 2016). These studies discussed the fluidity of the exploration and generation phase and consolidation and knowledge assembly phase in open ended design problems and how it was difficult to target just one concept because of interdependence between concepts, problems and tasks when working with multi-tool e-textile designs. Students were reported to have difficulties generating multiple solutions because of the sheer volume of interdependent problems and difficulty prioritizing which problems to solve first. Searle, Litts and Kafai (2018) discussed the need to engage students in a more structured PF design such as structured debugging. This sort of PF design would have a clear Solution Generation and Consolidation phase which could support optimal solution generation and incremental learning. Kafai et al. (2019) and Litts et al. (2016) also discussed the need to contribute to the growing understanding of how to design for Productive Failure in diverse open-ended problem settings and how such a design could support students' learning and problem solving.

This study seeks to also address these limitations and contribute to ongoing efforts by designing and implementing a debugging mini curriculum based on PF design to analyze the efficacy of such a design on students' learning and understanding of computational concepts.

**Design**

The design of individual units for the Debugging mini curriculum was informed by *Debuggems*. *Debuggems* are programming or e-textile problems that contain strategically placed bugs that students had to identify and debug (Fields et al. 2012; Fields, Searle & Kafai, 2016; Griffin, et al. 2011; Griffin, 2016; Kafai et al. 2014). Five *Debuggem* units were designed for the Debugging Mini curriculum. Based on the
design feedback from previous iteration to encompass and target all computational concepts, the five units comprised a total of 14 open ended code debugging problems that targeted the seven computational concepts individually or in combination. The units were housed in Scratch studio. Problems were presented as buggy games containing code sequences for one or more sprites that had strategically placed bugs i.e., missing, or incorrect code blocks and incomplete code sequences. For instance, in Debug 7, the objectives were to

a) Get the dragon to appear 1 second after Elvira says “Expecto Patronum!!”

b) Bonus: Make all the dementors disappear after Elvira says “Expecto Patronum!!” and the dragon appears?

The given code for dragon was missing three code blocks: a) Event block to initialize the code, b) a control block for the dragon to wait 1 second, and c) a look block for the dragon to appear (See Figure 3-3, middle). Similarly, the dementors code had a missing look block that would make them disappear once the dragon was summoned and breathed fire (See Figure 3-3, right). Debug 7 focused on conditionals, parallelisms, and sequences. Students' task was to identify missing code blocks in a code sequence and add them in the correct order to debug the problem. To do so, they had to apply computational concepts that would complete the code sequence and solve the problem. Each problem was accompanied by a narrative story, which provided a fun way to engage students in the debugging process by providing a context for each problem (See Figure 3-3, left).

Figure 3-3: Narrative story (left); Dragon incomplete code (middle); Dementors incomplete code (right)
The units had a series of problems designed to gradually increase in difficulty, targeting specific computational concepts such as Sequences, Events, and Loops in earlier units and more advanced concepts like Conditional, Parallelism, and Data in later units. The problems required students to complete various tasks, including identifying missing blocks, writing, and completing code sequences, creating variables, and defining win conditions. In Unit 5, students created their own games and had to create game narratives, sprites, code sequences for each sprite actions and game progression but they also had to identify and debug emergent problems, which required an understanding of multiple computational concepts and their interconnections.

**Design Guidelines**

The design guidelines for Debugging mini curriculum were informed by Productive Failure design principles (Kapur, 2008; Kapur & Kinzer, 2009; Kapur & Bielaczyc, 2012). Table 3-1 provides an overview of these five design guidelines, along with supporting strategies and also accompanying design and theoretical principles that inform the specific design guideline.
Table 3-1: Proposed design guideline for Debugging Mini Curriculum. (Adapted from Zimmerman & Land, 2014; Land & Zimmerman, 2015)

<table>
<thead>
<tr>
<th>Design Guideline 1: Design lessons where students work on open ended problems that are challenging and engaging</th>
<th>Guiding Design Principles</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Design Strategies</strong></td>
<td></td>
</tr>
</tbody>
</table>
| ● Design problems with faulty codes, missing code sequences and code blocks that do not fit in the code sequence. | ● Productive Failure Design Principle 1.  
| | o Design problem-solving environment where students work on complex problems that are challenging yet engaging and do not frustrate, where students can draw upon prior knowledge and problems that allow multiple solutions. (Kapur & Bielaczyc, 2012) |
| | ● Design problems where students must add in sprites, change backgrounds, create entire code sequences for sprites, create games. |  
| | ● Design incrementally difficult and multifaceted problems where students will start by solving minor problems such as adding in missing sound and motion blocks, changing the wait time, direction of sprite. This will build to bigger problems with multiple tasks where students must not only add a missing block but write in the entire code sequence for a sprite, add new sprites, backgrounds, create win events and scoreboard. |  
| | ● Present problems in a narrative story format with instructions and task lists. |  
| **Design Guideline 2: Design problems and conditions for problem solving that allow students to use prior knowledge and allows many solutions** |  
| **Design Strategies** |  
| ● Design open ended problems such as adding a sprite and using “Repeat until” block to get it to perform an action, make a game that incorporates a background, two sprites and a soccer ball. |● Productive Failure Design Principle 1.  
| | o Design problem-solving environment where students work on complex problems that are challenging yet engaging and do not frustrate, where students can draw upon prior knowledge and problems that allow multiple solutions (Kapur & Bielaczyc, 2012) |  
| | ● Students will work on un-scaffolded Exploration and Solution Generation Phase where they can employ combination of prior knowledge, heuristics, metacognition to generate solutions. |  
| | ● FACT pedagogical design principle which focused on creating a mix of directed assignments and open-ended problems in Scratch. (Grover, Pea, & Cooper, 2015) |
Design Guideline 3: Design opportunities for students to elaborate and explain their generated solutions, methods, knowledge

**Design Strategies**
- In facilitator led whole class discussion during the Consolidation and Knowledge Assembly phase, students will explain their generated solutions, how they came to their solutions, prior knowledge, and cognitive approaches they used.

**Guiding Design Principles**
- Productive Failure Design Principle 2
  - Provide opportunities to elaborate and explain of solutions (Kapur & Bielaczyc, 2012)

Design Guideline 4: Design opportunities to compare student generated solutions and opportunities for scaffolded practice

**Design Strategies**
- In the Consolidation and Knowledge Assembly Phase, facilitators will compare and contrast student generated solutions highlighting conceptual understanding and knowledge gaps.
- During the Consolidation and Knowledge Assembly Phase, the facilitator will contrast student generated solutions with relatively ideal solutions (quick, using key computational blocks, minimum code lines) to aid learning and understanding computational concepts and through modeling and worked examples.
- Students will go through scaffolded practice sessions after the Consolidation and Knowledge Assembly Phase. Newer problems similar to ones in the Exploration and Generation Phase will be designed. Helpful hints will be provided.

**Guiding Design Principles**
- Productive Failure Design Principle 3.
  - Provide opportunities to compare and contrast the pros and limitations of “incorrect” or suboptimal solutions and opportunities to work towards generating canonical solutions (Kapur & Bielaczyc, 2012)

Implementation

The Debugging mini curriculum was implemented as a Virtual coding and debugging workshop with 10 high school students from grade 10-12 between 15-18 years old. The workshop was offered twice a week (Tuesdays and Thursdays) for 3 weeks as part of a federal Trio-supported after school program for students from low-income and migrant communities in a rural part of Northeastern United States. The virtual workshop was conducted via Zoom by the author (Giri) with one adult volunteer assistant to adhere to Covid-19 research guidelines. Workshops were mailed to all enrolled students so that they could participate in the workshop.
Implementation of the curriculum followed the Productive Failure learning design in which Phase I (Exploration and Solution Generation) involved exploring block-based programming through Scratch tutorials and then individually working on lessons U1 to U5, generating multiple solutions for each problem without scaffolding and direct instruction. Phase II consisted of guided Consolidation and Knowledge Assembly. In this phase, the facilitator (Giri) led a discussion session where students shared their generated solutions. Giri compared and contrasted these solutions with an optimal solution highlighting areas of conceptual understanding and knowledge gaps. This was followed by instruction on targeted computational concepts for problems in each unit. Consolidation phase was followed by scaffolded practice sessions.

Day 1 of the workshop consisted of administering the pre-test, facilitator guided introduction to the Scratch platform and exploration of Scratch through tutorials and remix activities. 9 out of 10 students were novice programmers without prior experience with Scratch or coding in general. As such, it was necessary to provide contextual information and general guidelines on Scratch platform before students worked on the debugging problems during the unscaffolded Solution Generation phase, facilitator provided a guided tour of Scratch using demo projects and public access YouTube videos to draw students’ attention to key features in Scratch such as main page, login, coding area, code blocks palette (motion, sounds, events, control, sensing, operators, variables), stage, scripts area, sprite library, costumes. Students then worked on exploring Scratch through tutorials and remixing two existing Scratch projects, DJ Cat, and Maze.

In Day 2-6, for the first half of the session, students worked individually on Debug 1-14 where in the Solution Generation Phase I, they generated multiple solutions to each problem saving each iteration as a copy in Scratch. Additionally, half an hour of the first session of each day was also devoted to whole class discussion and question and answer sessions that were not completed in the previous day. The second half of each session consisted of the Consolidation and Knowledge Assembly phase where the facilitator compared student generated solutions with an optimal solution and provided feedback on students' understanding and application of computational concepts. A 20-minute practice session was
conducted on Day 4 so that students could work on their coding, debugging strategies, and application of coding concepts. The facilitator administered Think Alouds during the Solution Generation phase and Consolidation Assembly phase. Students were asked to explain their progress, solutions, debugging strategies, challenges and understanding of computational concepts. Table 3-2 outlines the activities that students engaged in each session of the workshop.

Table 3-2: Implementation of the PF design with highlights of the activities each session

<table>
<thead>
<tr>
<th>Session</th>
<th>Day</th>
<th>Activities</th>
</tr>
</thead>
</table>
| 1       | Tuesday | Initial Setup (10 mins)  
• Facilitator (Giri) setup laptop.  
• Checked video, audio, internet connectivity, speed.  
• Login to Scratch Teacher Account. Student accounts setup ahead of time.  
• Login to Zoom. Invited students to the zoom room.  |
|         |        | Student Arrival, Pre-Survey (25 mins)  
• Students arrived at the designated zoom room. Facilitator turned on zoom recording.  
• Students who consented to workshop participation but not audio, video muted their microphone and turned their videos off.  
• General introduction/icebreaker.  
• Facilitator took attendance.  
• Students completed pre-survey. [15 mins]  |
|         |        | Introduction to the Workshop and Scratch (1 hr.)  
• Facilitator turned on Screen Sharing.  
• Students were introduced to the Coding, Game Design and Debugging workshop with Scratch.  
• Facilitator led students through a guided exploration of Scratch using PowerPoint slides and demo projects focusing on key features of Scratch such as such as login, code palettes, stage, scripts area, sprite library, costumes and provided a brief description of computational concepts.  |
|         |        | Exploring Scratch through Scratch Tutorials and Remixing (45 mins)  
• Facilitator provided students with usernames and passwords to their Scratch student accounts. Students changed their passwords. [5 mins]  
• Students explored Scratch’s functionality through two Scratch tutorials [20 mins]  
• Students explored Scratch through two Scratch remix projects: DJ Cat and Maze where they customized codes and added functionality). [20 mins]  |
| 2       | Thursday | Initial Setup (10 mins) |
• Facilitator (Giri) setup laptop.
• Checked video, audio, internet connectivity, speed.
• Login to Scratch Teacher Account.
• Login to Zoom. Invited students to the zoom room.

Student Arrival (10 mins)
• Students arrived at the designated zoom room. Facilitator turned on zoom recording.
• Students who consented to workshop participation but not audio, video muted their microphone and turned their videos off.
• Attendance taken by facilitator.

Share Remixed projects (15 mins)
• Students finished and shared remixing projects

Introduction to Debugging and Think Alouds (15 mins)
• Facilitator turned on Screen Sharing to introduce students to debugging curriculum through PowerPoint slides.
• Facilitator explained Think Alouds conducted throughout the workshop and encouraged to verbalize their problem solving and debugging process. Turn off screen sharing.

Setup (10 mins)
• Facilitator showed students how to access the lessons housed in the Scratch studio. Students were asked to remix each debugging problem so that they can individually work on the lessons.
• Facilitator showed students steps to save a copy of each solution they generate i.e. “Save as copy” function in Scratch using screen share.
• As a backup for the Scratch studio, facilitator will also send .sb3 files for direct access to the debugging units.

Solution Generation phase, Debug 1-3 (1 hr.)
• Students worked on debugging and generating multiple solutions to Debug 1-3 targeting Sequences.
• Facilitator took notes on salient moment: debugging processes, iterative attempt, and decisions on choice of code blocks, code arrangements, logic.
• Facilitator conducted Think Alouds with individual students asking questions about students’ progress, problem solving and debugging strategies.
• Students shared their screen during Think Alouds to show their codes for the debugging problems and explain their process.
• Facilitator answered questions and provided on-time feedback for students who struggled with affective challenges, multiple unsuccessful attempts.
• Students who completed Debug 1-3 could move on to subsequent units.

Consolidation and Knowledge Assembly phase, Debug 1-3 (30 mins)
• Students shared their screens and shared their solutions to Debug 1-3, their debugging process, and decisions that they made during solution generation.
• Facilitator compared/contrasted student solutions with an optimal solution.
3 Tuesday

Initial Setup (10 mins)
- Facilitator (Giri) setup laptop.
- Checked video, audio, internet connectivity, speed.
- Login to Scratch Teacher Account.
- Login to Zoom. Invited students to the zoom room.

Student Arrival (10 mins)
- Students arrived at the designated zoom room. Facilitator turned on zoom recording.
- Students who consented to workshop participation but not audio, video muted their microphone and turned their videos off.
- Attendance taken by facilitator.

Consolidation and Knowledge Assembly phase, Debug 1-3- continued (30 mins)
- Facilitator ran a whole class discussion session in Zoom providing feedback on student solutions and highlighting knowledge gaps in coding concepts.
- Facilitator answered questions on computational concepts, confusing items, general Scratch, coding related questions, debugging problems.

Solution Generation phase Debug 4-6 (1 hr.)
- Students worked on debugging and generating multiple solutions to Debug 4-6 targeting Events and Loops.
- Facilitator took notes on salient moment: debugging processes, iterative attempt, and decisions on choice of code blocks, code arrangements, logic.
- Facilitator conducted Think Alouds with individual students asking questions about students’ progress, problem solving and debugging strategies.
- Students shared their screen during Think Alouds to show their codes for the debugging problems and explain their process.
- Facilitator answered questions and provided on-time feedback for students who struggled with affective challenges, multiple unsuccessful attempts.
- Students who completed Debug 4-6 could move on to subsequent units.

Consolidation and Knowledge Assembly phase, Debug 4-6 (30 mins)
- Students shared their screens and shared their solutions to Debug 4-6, their debugging process, and decisions that they made during solution generation.
- Facilitator compared/contrasted student solutions with an optimal solution.

4 Thursday

Initial Setup (10 mins)
- Facilitator (Giri) setup laptop.
- Checked video, audio, internet connectivity, speed.
- Login to Scratch Teacher Account.
- Login to Zoom. Invited students to the zoom room.

Student Arrival (10 mins)
- Students arrived at the designated zoom room. Facilitator turned on zoom recording.
- Students who consented to workshop participation but not audio, video muted their microphone and turned their videos off.
• Attendance taken by facilitator.

Consolidation and Knowledge Assembly phase, Debug 4-6- continued (30 mins)
• Facilitator ran a whole class discussion session in Zoom providing feedback on student solutions and highlighting knowledge gaps in coding concepts.
• Facilitator answered questions on computational concepts, confusing items, general Scratch, coding related questions, debugging problems.

Practice session (20 mins)
• Students worked on practice debugging problems.

Solution Generation phase Debug 7-9 (1 hr.)
• Students worked on debugging and generating multiple solutions to Debug 7-9 targeting Conditionals and Parallelisms.
• Facilitator took notes on salient moment: debugging processes, iterative attempt, and decisions on choice of code blocks, code arrangements, logic.
• Facilitator conducted Think Alouds with individual students asking questions about students’ progress, problem solving and debugging strategies.
• Students shared their screen during Think Alouds to show their codes for the debugging problems and explain their process.
• Facilitator answered questions and provided on-time feedback for students who struggled with affective challenges, multiple unsuccessful attempts.
• Students who completed Debug 7-9 could move on to subsequent units.

Consolidation and Knowledge Assembly phase, Debug 7-9 (30 mins)
• Students shared their screens and shared their solutions to Debug 7-9, their debugging process, and decisions that they made during solution generation.
• Facilitator compared/contrasted student solutions with an optimal solution.

5 Tuesday

Initial Setup (10 mins)
• Facilitator (Giri) setup laptop.
• Checked video, audio, internet connectivity, speed.
• Login to Scratch Teacher Account.
• Login to Zoom. Invited students to the zoom room.

Student Arrival (10 mins)
• Students arrived at the designated zoom room. Facilitator turned on zoom recording.
• Students who consented to workshop participation but not audio, video muted their microphone and turned their videos off.
• Attendance taken by facilitator.

Consolidation and Knowledge Assembly phase, Debug 7-9- continued (30 mins)
• Facilitator ran a whole class discussion session in Zoom providing feedback on student solutions and highlighting knowledge gaps in coding concepts.
• Facilitator answered questions on computational concepts, confusing items, general Scratch, coding related questions, debugging problems.
Solution Generation phase Debug 10-12 (1 hr.)
- Students worked on debugging and generating multiple solutions to Debug 10-12 targeting Operators and Data.
- Facilitator took notes on salient moment: debugging processes, iterative attempt, and decisions on choice of code blocks, code arrangements, logic.
- Facilitator conducted Think Alouds with individual students asking questions about students’ progress, problem solving and debugging strategies.
- Students shared their screen during Think Alouds to show their codes for the debugging problems and explain their process.
- Facilitator answered questions and provided on-time feedback for students who struggled with affective challenges, multiple unsuccessful attempts.
- Students who completed Debug 10-12 could move on to designing their own games for Debug 13-14.

Consolidation and Knowledge Assembly phase, Debug 10-12 (1 hr.)
- Students shared their screens and shared their solutions to Debug 10-12, their debugging process, and decisions that they made during solution generation.
- Facilitator compared/contrasted student solutions with an optimal solution.
- Facilitator ran a whole class discussion session in Zoom providing feedback on student solutions and highlighting knowledge gaps in coding concepts.
- Facilitator answered questions on computational concepts, confusing items, general Scratch, coding related questions, debugging problems.

6 Thursday Initial Setup (10 mins)
- Facilitator (Giri) setup laptop.
- Checked video, audio, internet connectivity, speed.
- Login to Scratch Teacher Account.
- Login to Zoom. Invited students to the zoom room.

Student Arrival (10 mins)
- Students arrived at the designated zoom room. Facilitator turned on zoom recording.
- Students who consented to workshop participation but not audio, video muted their microphone and turned their videos off.
- Attendance taken by facilitator.

Solution Generation phase Debug 13-14 (1 hr.)
- Students worked on Debug 13-14 creating their own games targeting multiple computational concepts.
- Students created game narratives, wrote code sequences, and debugged emergent issues in their code.
- Facilitator took notes on salient moment: debugging processes, iterative attempt, and decisions on choice of code blocks, code arrangements, logic.
- Facilitator conducted Think Alouds with individual students asking questions about students’ progress, problem solving and debugging strategies.
- Students shared their screen during Think Alouds to show their codes for the debugging problems and explain their process.
Facilitator answered questions and provided on-time feedback for students who struggled with affective challenges, multiple unsuccessful attempts.

Consolidation and Knowledge Assembly phase, Debug 13-14 (45 mins)
- Students shared their screens and shared their games describing the challenges, debugging strategies, and ways to expand the game their debugging process.
- Facilitator compared/contrasted student solutions with an optimal solution.
- Facilitator ran a whole class discussion session in Zoom providing feedback on student solutions and highlighting knowledge gaps in coding concepts.

Post-Survey (15 mins)
- Students completed post-survey.

Review and Feedback on the workshop (15 mins)
- Facilitator asked feedback from students on the workshop.
- Students provided feedback on the format and presentation of the workshop, lessons, activities leading up to the lessons, conceptual understanding, and learning.

Data Sources
The data was collected from four main sources: a) Zoom audio recording of Think Aloud session and zoom video recording of students debugging progress, b) student generated artifacts, c) online semi-structured interviews and d) field notes taken by the researcher. Zoom recording was used to document salient moments of students’ debugging processes including unsuccessful and successful code debugs. Think aloud protocols developed by Richard et al. (2018) were used to encourage students to vocalize the thinking behind their generated solutions and to engage students in metacognitive processes related to their conceptual and procedural understanding of computational concepts. Semi-structured interviews were also administered online after each session to understand students’ debugging processes, moments of failure and learning of computational concepts. Additionally, student generated artifacts: their generated solutions to debugging problems and their created games were collected and analyzed to understand changes in students debugging strategies and their understanding, application and learning of computational concepts and programming.
Data Analysis

The data analysis conducted to answer RQ1 and RQ2 was based on three data sources. The Zoom recordings of students working through the debugging problems were used as the primary source of data for analysis. These Zoom recordings captured students' screens as they were working on Debugging problems during the Solution Generation Phase and facilitated discussions during the Knowledge Assembly Phase. The Zoom recordings also separately captured audio from Think Alouds (Ericsson & Simon, 1980; Charters, 2003) during the Solution Generation phase where students explained their debugging processes/strategies, the challenges they encountered and how they resolved those challenges. Digital artifacts i.e., students’ debugged projects, semi-structured interviews and validated survey instruments were used to support the Zoom recordings. The data was analyzed within the framework of interaction analysis (Jordan & Henderson, 1995) and guided by Derry et al (2010). Data analysis for the research questions and sub questions utilized different types of data, methods, and analysis. I used a mixed method approach to analyze RQ1 (Creswell & Creswell, 2017; Johnson & Onwuegbuzie, 2004). For RQ 2 and 2.1, I used a Qualitative approach where the data was analyzed within the larger framework of interaction analysis (Jordan & Henderson, 1995). This information is clearly tabulated in Table 3-3

Table 3-3: Shows the types of data, methods and analysis used for each research question and sub questions

<table>
<thead>
<tr>
<th>Research Questions</th>
<th>Data used</th>
<th>Method</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RQ 1</strong></td>
<td>Pre-post survey Video/audio Data</td>
<td>Mixed</td>
<td>Wilcoxon Interaction Analysis Artifact Analysis</td>
</tr>
<tr>
<td>How does productive failure design support learning of computational concepts when working with open-ended debugging problems in programming platforms such as Scratch?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RQ 2.1</strong></td>
<td>Artifacts Video/audio Data Interviews</td>
<td>Qualitative</td>
<td>Interaction Analysis Artifact Analysis Thematic Analysis</td>
</tr>
<tr>
<td>What kinds of challenges do students encounter, the resolutions they come up with and the debugging strategies they</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
used when working through the debugging curriculum?

<table>
<thead>
<tr>
<th>RQ 2.2</th>
<th>What do these challenges, resolutions, process, and interaction patterns reveal about students’ evolving understanding of computational concepts?</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ 1</td>
<td>How does productive failure design support learning of computational concepts when working with open-ended debugging problems in programming platforms such as Scratch?</td>
</tr>
</tbody>
</table>

Quantitative Data Analysis

To answer RQ1 How does productive failure design support learning of computational concepts when working with open-ended debugging problems in programming platforms such as Scratch? Pre/post surveys were administered before and after the debugging workshop. The pre-post survey consisted of items derived from three instruments with established reliability and validity measures. These instruments were a) Problem Solving Inventory, PSI (Heppner, 1988), b) Attitude Scale of Computer Programing Learning, ASCOPL (Korkmaz, & Altun, 2014) and c) Computer Programming Self Efficacy Scale, CPSES (Kukul et al. 2017). PSI had a reliability of high 0.80 ($\alpha > 0.80$) across several studies and with different population and cultures (Heppner, 1988; Heppner et al. 1994, 2002; Koormousi et al. 2016). The test-retest reliability ranged from $r = .80$ over a 2-week period, $r = .81$ over 3 weeks and 4-month period (Heppner, 1988). ASCOPL had established validity ($r = 0.671, n = 476$, p< 0.001) and reliability ($\alpha = 0.866$). Additionally, CPSES was also validated ($r = 0.966, n = 233$, p<0.01) and had high reliability measures ($\alpha = 0.95$).

The survey assessed participants’ computer programming self-efficacy and attitudes towards computer programming learning. The survey also measured participants’ evaluations of their problem-solving abilities and their attitudes towards failure and iteration in problem solving activities. All items on the pre-post survey were scored on a 6-point Likert scale with rating ranging from 1 Strongly Disagree to 6 Strongly Agree. I analyzed pre-and post-test changes for each of the subscales through the Wilcoxon signed ranked test.
Survey Items

The pre-post survey consisted of items from the previously reliable and validated instruments. I discuss these instruments in detail below.

Problem Solving Inventory, PSI ($\alpha > 0.80$ across several studies)

PSI (Heppner, 1988) assessed participants' evaluation of their problem-solving abilities and their attitudes towards failure and the iterative aspect of problem solving. PSI is a 32-item inventory that constituted three subscales/subcategories:

Problem Solving Confidence: These questions assessed participants’ levels of self-assurance when working on problems (PSC, 11 items, e.g.: “I am usually able to think up creative and effective ways to solve a problem.”)

Approach Avoidance Style: These questions measured participants’ tendency to avoid or approach problem solving activities (AAS, 16 items, e.g. “When I have a problem, I think up as many possible ways to solve it as I can until I can't come up with any more ideas.”).

Personal Control: These questions assessed participants’ self-belief that they are in control of their emotions and behaviors while solving problems (PC, 5 Items, e.g. Sometimes I do not stop and take time to deal with my problems, but just kind of push ahead.)

Attitude Scale of Computer Programming Learning ASCOPL ($r = 0.671$, $n = 476$ $p < 0.001$; $\alpha = 0.866$)

ASCOPL (Korkmaz, & Altun, 2014) assessed participants’ attitude towards computer programming learning. ASCOPL consisted of 20 items which could be grouped into three factors/subcategories:

Willingness: These items assessed participants’ positive attitude towards computer programming and inclination to learn computer programming. (WIL, 9 items, e.g. “I am confident I can learn programming,” “Given the chance, I would participate in computer programming courses.”)
Negativity: These items assessed participants’ negative attitude towards learning computer programming. (NEG, 6 items, e.g., “Programming is very difficult for me,” “I am not good at programming.”)

Necessity: These items assessed participants’ attitude on the importance of learning computer programming. These items encompassed both positive and negative attitudes. (NEC, 5 items, e.g., “Programming is an important skill to have.” “After graduating from school, I don’t think I will use programming skills.”

Computer Programming Self Efficacy Scale, CPSES ($r = 0.966$, $n = 233$, $p<0.01$; $\alpha = 0.95$)

CPSES (Kukul et al. 2017) assessed participants’ computer programming self-efficacy in addition to understanding and application of computational concepts with programs such as Scratch and Alice. CPSES consisted of 21 items that measured Programming knowledge, Programming Skills, and Cooperation. Items included questions such as “I know how to use loops instead of repeating the same code over and over,” “I know what conditionals like if- then, if-else, repeat until, wait until are and how to use them,” and “I can solve complex programming problems by separating them into smaller sub-problems.”

Item Revision

The pre and post survey used in the workshop included all 32 items from PSI and all 21 items from CPSES. However, the survey did not include all 20 items of the ASCOPL. Specifically, there were items under Willingness and Negativity that were taken out of the pre and post survey and additions, revisions were made to these subscales. These items were related to the attitude of willingness towards university programming courses, developing products and therefore did not focus on the general attitude of willingness to learn programming that were within the scope of my research questions. As such, the following items were taken out “Given the chance, I would like to participate in computer programming courses in different departments in my free time,” “I am sure I’m able to put-on high-level programming products” and “Computer programming courses are at the head of the courses that I enjoy the most.”
These items were replaced with the following items that conveyed attitudes of willingness towards programming in general: “I do research to be a good programmer,” “If I encounter a problem while programming that I cannot solve immediately, I do not give up until I solve it,” “Given the chance, I will participate in computer programming courses.”

Additionally, some item translations from Turkish, in which the items were originally written in, to English for Willingness and Negativity subscale were not cohesive and therefore revisions were made so that the items were clearly understood. Specifically, Willingness scale items such as “Writing a computer program is funny for me” was changed to “Programming is fun for me.” “I think that less time for lessons about programming skills” was changed to “Once I start working on a computer program, I try to finish it before everything else.” “I’m sure I can learn to computer programming” was changed to “I am confident I can learn programming.” Similarly, Negativity scale items were also added to and revised due to translation and scope issues. For instance, “I am afraid of computer programming courses,” “I’m not in computer programming,” and “In my spare time, writing a computer program does not deal with inside” were changed to “Programming is boring,” “Programming is very difficult for me,” “I am not good at programming,” “Programming-related activities makes me nervous” and “The very idea of code-writing makes me nervous.”

The final pre-post survey contained 77 questions which were divided into three groups: a) Q5-Attitude Scale of Computer Programming Learning, b) Q6- Computer Programming Self Efficacy Scale, and c) Q7-Problem solving inventory (See Figure 3-4).
The three groups each contain three subcategories of questions. The biggest subcategories in each of the groups are **Willingness** (13% of total questions) in Q5- **Attitude Scale of Computer Programing Learning** (ASCOPL), Programming skills (14.3% of total questions) in Q6- **Computer Programming Self Efficacy Scale** (CPSES), Approach avoidance style (20.8% of total questions) in Q7- **Problem Solving Inventory** (PSI). Hence the survey had a focus on not only participants’ programming skills but also different kinds of mentality on coding.

**Wilcoxon Signed Rank Test**

I analyzed the pre-and post-test changes for each of the subscales using the Wilcoxon signed ranked test. When answering survey questions, students gave a score of 1—6 (1: “strongly disagree”, 2: “disagree”, 3: “slightly disagree”, 4: “slightly agree”, 5: “agree”, 6: “strongly agree”). The data was therefore discrete. Also, they did not have normal distributions (Fig. 2-4). Thus, Wilcoxon signed-rank test was chosen instead of t-test which assumes that the data is normally distributed.
**Data transformation**

Among the 77 questions included in the survey, 51 questions were “positive” (the higher the score, the better the coding ability and/or more interest the participants had in coding). For example, “I can solve complex programming problems by separating them into smaller sub-problems” was a positive question. Meanwhile, 26 questions were “negative” (the higher the score, the worse the coding ability and/or less interest the participants had in coding). For example, “When confronted with a problem, I am unsure of whether I can handle the situation” was a negative question. Utilizing both types of questions allow researchers to garner more accurate and well-rounded participant responses. However, it creates issues for quantitative analysis such as the Wilcoxon test where scores for these two question types would cancel each other out to some extent.

To address this issue, I reverse coded the negative questions. Scores were transformed by using the following equation: \( x = 7 - y \), where \( x \) = New Score and \( y \) = Original score. Take the question "Programming is boring" as an example. The question was converted to “Programming is NOT boring”. And a score of 1 (strongly disagree) would be converted to 6 (strongly agree). This worked because "I strongly DISAGREE that programming is boring" was equivalent to "I strongly agree that programming is NOT boring".

**Qualitative Data Analysis**

Qualitative data analysis followed three levels of analysis. Level 1 analysis included the creation of content logs, transcription, and identifying salient moments of conversation/activity or “hotspots” (Jordan & Henderson, 1995). Level 2 analysis included transcription selection, writing multimodal transcripts (Mondada, 2019) and analytical memos (Strauss & Corbin, 1990). Level 3 analysis focused on detailed analysis of salient events through interaction analysis (Jordan & Henderson, 1995) and thematic analysis (Boyatzis, 1998; Braun & Clark, 2012).
**Level 1 Analysis**

Preliminary steps to Level 1 analysis included the creation of Content Logs to broadly log the video, audio data and artifacts. To create the content logs, the corpus of video and audio data were transcribed. In the transcriptions, I identified the time, speakers and verbatim report of the conversation that ensued in a particular time or instance. I noted relevant emotions, gestures, and actions that students and facilitators engaged in to explain their thinking, elucidate a point, and draw attention to particular code blocks or sequences. In addition, I also noted the speaker's tone, gaze, and action such as pointing, circling, selecting, and dragging, dropping, and moving code blocks and sequences. Following the transcription, I watched the videos where students worked silently through the debugging problems and listened to audio files of Think Alouds and the Knowledge Assembly session, at least two to three times. I divided the videos and audio recording into 2-minute instances or segments. In the content log I documented the time, speakers in the segment, segment name (i.e., Day7_02232021_001_cleaned.mp3), Lesson number (i.e., U1 Sequences; U2- Events and Loops), Debugging problem being referenced, a brief description of the content in the segment and justification of how aspects of the clips related to or answers the Research Questions.

Content logs were used as a guide to identify salient moments of conversation/activity or “hotspots” (Jordan & Henderson, 1995). For RQ1, these hotspots included instances where students directly explained how they learned computational concepts or understood some concepts better after going through periods of solution generation and knowledge assembly. Other hotspots included instances of successful debugging with Computational concepts they were struggling with in prior problems, correct application and explanation of computational concepts and error reductions in subsequent debugging problems. For RQ2, these hotspots included communication of challenges during solution generation, explanation of how they resolved those challenges including their debugging process, strategies and thinking behind different iterative attempts. These hotspots also included instances of frustrations and “aha moments” where students describe moments of clarity or understanding of computational concepts that they were struggling with in prior debugging problems and were not
correctly applying in their debugging attempts. Additionally, hotspots for RQ 2 also included silent screen recording (video) segments of students working on the debugging problems where specific actions were salient. These actions included choices of code blocks, real-time code edits, and code swaps during solution generation as well as unsuccessful and successful code sequences. These salient moments of frustrations, iterative attempts, resolutions, along with interactions and verbalization during, after successful and unsuccessful debugs provided insights into students’ learning and evolving understanding of computational concepts.

**Level 2 Analysis**

Level 2 analysis included transcription selection, writing multimodal transcripts (Mondada, 2019) and analytical memos (Strauss & Corbin, 1990).

**Transcript Selection:**

Content logs and identified “hotspot” or discourse rich moments provided a basis for transcript selections and qualitative analysis. To conduct Level 2 analysis, I listened to audio files and rewatched the videos again focusing this time on segments from the identified “hotspots” to further narrow the transcript selections. I used the timecodes recorded on the content logs to watch and listen to these specific salient segments. I then narrowed the transcripts again focusing on my research questions.

For RQ1 *(How does productive failure design support learning of computational concepts when working with open-ended debugging problems in programming platforms such as Scratch?)*, I focused on

a) instances during Think Alouds (audio) and interview where participants directly responded to how the PF design with its Solution Generation and Knowledge Assembly phases bolstered participants’ learning and understanding of computational concepts.

b) segments showing changes in understanding of computational concepts after going through multiple Solution Generation and Knowledge Assembly phases. I especially focused on instances of initial struggle with application and participants' explanation of specific computational concepts during iterative debugging attempts. I then sought out
segments which participants resolved, mitigated prior errors and discussed the changes in their understanding of computational concepts and how they applied those concepts in subsequent debugging problems.

For RQ 2.1 (What kinds of challenges do students encounter, the resolutions they come up with and the debugging strategies they used when working through the debugging curriculum?), I focused on

a) instances during Think Alouds (audio) and interview where participants articulated the challenges that they faced working through the debugging curriculum

b) instances where participants discuss the iterative troubleshooting that they did, the coding blocks and computational concepts that they used and the solutions that they came up

c) segments where they directly or indirectly shared their debugging processes and strategies (i.e., trial and error, forward and backward reasoning, problem simplification)

For RQ 2.2 (What do these challenges, resolutions, process, and interaction patterns reveal about students’ evolving understanding of computational concepts?), I emphasized,

a) instances where students directly or indirectly referenced their changing understanding and application of computational concepts. These included:

i. segments where participants articulated their learning about coding and computational concepts from facing and working through different types of challenges (i.e., choosing incorrect code blocks, multiple unsuccessful code debugs, code execution, timing, coordination)

ii. segments where participants expressed their learning about coding and computational concepts from their iterative attempts at debugging problems and through the solutions they generated throughout the curriculum

iii. segments where participants provided specific examples of computational concepts that they had problems understanding and applying in prior debugging
problems but now have a better understanding of those concepts. Some participants also explained how they came to this better understanding.

iv. instances where participants articulated computational concepts that they still have problems with providing indication of some persistent gaps in their understanding and application of those computational concepts.

v. silent screen recording (video) segments of students showing debugging decisions in real time that provided insights into their evolving understanding of computational concepts. These debugging decisions included choices of code blocks, real-time code edits, and code swaps during solution generation as well as unsuccessful and successful code sequences.

**Multimodal Transcripts:**

Once transcripts were selected using content logs and identified hotspots, multimodal transcripts (Bezemer & Mavers, 2011; Mondada, 2019) were prepared. The conventions for multimodal transcripts enable annotation of relevant details of verbal interactions and embodied actions. (Mondada, 2019, p.1). I created these multimodal transcripts to understand verbal and embodied actions such as gestures, gaze and object manipulation that occurred when participants were working on the debugging problems and when they were explaining their solutions during the knowledge assembly phase. Specifically, I noted the code blocks that participant pointed to, dragged from the code palette, moved around, used in their code sequences, and swapped out while debugging Scratch coding problems. In these transcripts, I also noted gestures to particular code sequences and code edits those participants made on screen during interactions with the facilitator. These transcripts with the inclusion of embodied actions provided additional context to conduct interaction analysis.

**Analytical Memos:**

Analytical memos were written for all hotspot moments that were selected for transcription. Analytical memos can serve as a tool to enhance the analytic process through reflection on the data.
analytical memos served two purposes. First, the memos helped me document and systematize my thinking, insights, observations, and reactions as a researcher and facilitator. Second, these memos served as a tool to conduct preliminary analysis on salient moments that I could use to develop into coherent explanations and answers to my research questions.

To write analytical memos, I rewatched all the videos and audio clips related to the identified hotpot moments that were transcribed. Descriptive memos prepared during data collection and the Multimodal Transcripts provided additional context to understand, expand on, reflect and record my thoughts and analysis of these salient moments. Using my research questions as a guide, I documented my thoughts about the participants’ engagement with the Productive Failure design and their learning of computational concepts as they worked through the debugging problems (RQ 1). I noted my analysis of the various challenges that participants faced, the types of debugging strategies that they used, and added my thoughts and questions about the solutions that participants came up with (RQ 2.1). I also analyzed the code sequences (artifacts) that participants wrote for each debugging problem, noting the code blocks that they used, the computational concepts that they applied and the overall operationality of the code sequences. Looking at participants’ code sequences, challenges, and solutions, I documented my thoughts on participants’ trajectory of learning and understanding of computational concepts during solution generation and knowledge assembly phase (RQ 2.2).

To maintain confidentiality of the participants’ I gave each participant a different name in the memo. Here is an example of the analytical memo I wrote *put the example below

*Level 3 Analysis*

In Level 3 Analysis, I focused on interaction analysis (Jordan & Henderson, 1995) and thematic analysis.
Interaction Analysis:

I used interaction analysis (Jordan & Henderson, 1995) to unpack RQ 1 and RQ 2. I went back to Level 1 analysis to the hotspots that I identified for my research questions and Level 2 analysis to the selected transcriptions to create case studies. These case studies would help me understand participants’ trajectory of learning of computational concepts and how the PF design supported participants’ learning of those concepts (RQ 1). In addition, these case studies would also support my understanding of the impact of different debugging challenges, participants debugging strategies and iterative solution generation had on their learning of computational concepts (RQ 2.1, 2.2).

To create these case studies, selected interactions identified in Level 1 between participants and facilitators during solution generation and consolidation phase were studied. Multimodal transcripts created during Level 2 analysis were used to understand the embodied actions that occurred in these interactions. In the transcript’s audio and video data, I identified the time, speakers, utterance, actions, and artifacts (code blocks, sprites) manipulated on screen. Utterances were documented verbatim to accurately represent speakers’ conversation during a specific segment. I also noted non-verbal, embodied actions such as posture, gaze, and emotions. Additionally, I noted the artifacts (code blocks, sequences, sprites) used and manipulated during an interaction through actions such as pointing, dragging-dropping, circling, looking, emphasizing, and inscribing. Verbal interactions were analyzed in conjunction with the gestures, gaze, and object (code) manipulation to gain insights about participants' debugging process and strategies, application of computational concepts and the learning that was happening through debugging.

For RQ 1: How does productive failure design support learning of computational concepts when working with open-ended debugging problems in programming platforms such as Scratch? I used interaction analysis to understand the nature of learning that was happening when participants went through the solution generation and knowledge assembly phase of the PF Design. I focused on understanding how moment-to-moment discourse and interaction with the debugging problems on Scratch mediate learning from failure. Here, I examined participants’ interactions, code manipulations
and generated solutions during iterative and failure prone cycles of debugging and after knowledge assembly sessions with the facilitator to understand changes in their application and understanding of computational concepts.

For **RQ 2.1: What kinds of challenges do students encounter, the resolutions they come up with and the debugging strategies they used when working through the debugging curriculum?** and **RQ 2.2 (What do these challenges, resolutions, process, and interaction patterns reveal about students’ evolving understanding of computational concepts?)**. I used interaction analysis to understand aspects of change in participants’ understanding of computational concepts that occurred as participants worked through various debugging problems. I looked at interactions where participants explained the different problems they faced while debugging, the various solutions that they came up with, and explained their process, codes and reasoning behind their solutions and code choices. I then created individual case studies. In these case studies, I documented and analyzed the nature of these challenges and generated solutions across the debugging problems to examine changes in participants’ understanding and application of computational concepts. I analyzed codes associated with participants' iterative solutions and moments of active debugging on screen: moment-to-moment debugging decisions, code block selections, code swaps/edits. I also examined the various debugging strategies that participants used and documented any changes in these strategies across problems that the participants attempted. This analysis of participants' codes and debugging strategies provided further insights into my understanding of participants’ changes in learning of computational concepts.

**Thematic Analysis:**

Thematic analysis is a method for “systematically identifying, organizing, analyzing and interpreting patterns of meaning (themes) within qualitative data” (Braun & Clarke, 2012, p. 57). Thematic Analysis allows researchers to identify relevant patterns and analyze meaning across a data set to answer a particular research question as well as to explore one particular aspect of a phenomenon in depth.
I conducted thematic analysis to unpack RQ 2.2 (What kinds of challenges do students encounter, the resolutions they come up with and the debugging strategies they used when working through the debugging curriculum?) and RQ 2.2 (What do these challenges, resolutions, process, and interaction patterns reveal about students’ evolving understanding of computational concepts?) I used thematic analysis to document, code and analyze salient moments of verbal and non-verbal (embodied) interactions. I focused on participants’ debugging challenges, solutions and debugging strategies to analyze their evolving learning and understanding of computational concepts.

I used a combined inductive and deductive coding approach. First, I conducted deductive coding. I went back to the Multimodal transcripts created for Level 2 analysis and completed open line-by-line and in-vivo coding. I created a coding scheme informed by Litts et al’s (2016) four stage coding process used to understand learning in open-ended e-textile design tasks through instances of iterations/challenges and how those challenges were resolved. These four stages of coding involved

- **Stage 1: Identifying instances of iteration/challenges.** I identified and coded for instances of
  
  - Verbalization of frustrations and interactions with peers, facilitators about the challenges
  - Iterations/Challenges identified by facilitator

- **Stage 2: Generating and applying a coding scheme of the type of challenges.** Litts et al. (2016) coded for challenges such as knots/tangles, polarity of LEDs, etc. for e-textile related challenges that participants faced. I generated and applied a coding scheme to document the types of challenges that participants faced while working through the debugging curriculum. I used Frädrich et al’s (2020) catalog of common bug patterns in Scratch as a framework to identify general bugs and recurring errors in all students generated solutions from Debug 1-14. Some of these challenges included
  
  - Snapping blocks that do not go together
  - Order of Code blocks/Sequence
  - Incomplete code sequences/Partially operational code
  - Trial and Error without understanding of code blocks and computational concepts
- Code Execution Timing, Coordination
- Fear Of Trying Blocks, Changing Code Sequences, Failure
- Understanding Debug Problem, Instructions, Objectives
- Technical (Internet, Saving, Zoom)
- Time (completing the debugs, Helping family, Work, Taking care of siblings during workshop)
- Redundant/Repeating/Competing code sequences

- **Stage 3: Challenge Resolutions, Problem Solving.** Litts et al. (2016) coded for how the challenges were resolved i.e., alone, with a peer, with a teacher, or not at all. I added to these coding schemes to encompass the various ways that participants resolved the debugging problems. These ways of resolutions included
  - Facilitator Feedback/Directions
  - Facilitator Prompts
  - Trial and Error
  - External Sources: Scratch projects, YouTube, Tutorials, Parents
  - Application, Understanding of computational concepts
  - Sequential ordering of code blocks, Getting the Logic right
  - Problem Decomposition
  - Synchronization/Coordination/Interactivity and Parallelism

- **Stage 4: Generate larger themes.** I used inductive coding for Stage 4 where I generated codes for larger themes that were emerging from the data. For instance, one of the themes that was emerging was the change in debugging strategies as participants progressed through the workshop. In addition to generating a coding scheme for debugging strategies, I also clustered codes for challenges and resolutions around the larger theme. For instance, I coded Trial and Error both as a type of challenge and an approach to resolution. But examining the frequent
occurrence of Trial and Error and the reasons for using it, I clustered it under the larger theme of Debugging Strategies that participants were using to solve the debugging problems. Some of these debugging strategies included

- Trial and Error
- Objective Focus
- Work around the problem
- Backtracking
- Bug Clustering
- Forward Reasoning
- Backward Reasoning

I also coded for shifts in learning of computational concepts that were emerging from the data and that I was seeing and from the challenges that participants faced, the debugging strategies that were used to resolve different debugging problems and the interactions around this process. This further helped me unpack RQ 2.2 (What do these challenges, resolutions, process, and interaction patterns reveal about students’ evolving understanding of computational concepts?).

**Researcher Positionality and Ethical Considerations**

In this section, I discuss my positionality as a researcher conducting this study and also discuss the ethical considerations. Although, I discussed my positionality at different points in chapter 2, I summarize and highlight key aspects of its various facets here. My position as a researcher in this study was first and foremost influenced by the longstanding and inconclusive debate on the role of failure in learning.
Two main camps of research exist in this debate. One camp emphasized the constructive effects of failure in problem-solving settings, highlighting its positive affective outcomes, attributions, and its potential as a motivating and learning tool (Clifford, 1988; Dweck & Leggett, 1988; Kapur, 2008; Van Lehn, 1988; Westermann & Rummel, 2012). This body of work argued for moderate risk taking, tolerance of impasses, and designing for optimal difficulties and unguided problem-solving followed by a guided consolidation. The other camp of research highlighted the affective challenges, learned helplessness, and deficit frames associated with unchecked failure (Heyd-Metzuyanim, 2015, Mikulincer, 1994; Richard, 2015, 2017; Vossoughi & Bevan, 2014). This body of work underscored the findings that promoting failure can perpetuate negative stereotype, and inequitable participation, particularly among historically marginalized population. In this study, I aligned with the camp that emphasized the constructive role of failure in learning while acknowledging the concerns raised by the other camp on the detrimental and negative attributional effects of repeated unchecked failures.

My positionality has been shaped by personal experiences, most notably by growing up in a military family. This upbringing instilled in me resilience and the value of learning from mistakes, continuously striving to improve through iterative attempts. My Nepalese roots and the formative years spent in progressive middle and high schools further shaped my positionality, emphasizing consistent self-improvement. Moreover, my experiences as an international student and the heightened awareness and sensitivity that may have come from being someone from a minoritized identity have informed my perspective. This reality compels me to continuously strive for opportunities, pursue ways to overcome setbacks and transform them into lessons in order to move forward towards a better future and fruitful opportunities. These lived experiences combined with my belief in the constructivist approach to education, which views learning as an active and learned-centered process, have influenced my stance on educational benefits of complex problem solving and learning effectively to navigate failure. To address personal biases in the study, reflexivity was employed throughout the research process. This involved framing and acknowledging opportunities to fail as a privilege and a learning process, as well as
emphasizing the optimal outcomes in debugging rather than viewing failure as a final result. By adopting this approach, I aimed to mitigate the impact of personal biases and foster a more objective understanding and framing of failure with vulnerable learners. Additionally, I also made efforts to address affective challenges and deficit frames associated with failures when designing and implementing the debugging curriculum. Specifically, on-time feedback were provided to novice programmers to during the Solution Generation phase to address affective and self-efficacy challenges that became prominent from unchecked failures. Additionally, during the Consolidation phase, I not only shared optimal solutions for debugging problems, but also addressed knowledge gaps, and offered targeted feedback on programming challenges and inefficient debugging processes. These measures were implemented to mitigate recurring failures, error patterns, and enhance subsequent problem-solving.

This research study adhered to ethical principles and guidelines to protect participants. Informed consent was obtained, ensuring voluntary participation and withdrawal rights. Confidentiality was maintained through data secure data storage and anonymization through use of pseudonyms. Additionally, privacy measures were implemented throughout the study. All collected data were stored on a password protected computer and two encrypted external hard drives. Risks and benefits were evaluated, minimizing harm. Similarly, IRB approval was obtained for the online synchronous workshop.
Chapter 4
Findings

This chapter aims at answering my research questions. I begin by answering RQ1: How does productive failure design support learning of computational concepts when working with open-ended debugging problems in programming platforms such as Scratch? I answer RQ 1 by analyzing the quantitative pre-post survey data and then analyzing case studies across the data set to support the quantitative findings. The pre-post test data was analyzed using the Wilcoxon Matched Pairs Signed Ranked Test. My findings showed that there was an overall increase in participants post-test scores from pre-test scores, $z = -10.52$, $p < 0.001$, $r = 0.35$. This increase in scores was statistically significant with a medium effect size ($r > 0.3$) showing that participants reported improvements in their problem-solving abilities, programming self-efficacy, and understanding and application of computational concepts. The findings indicated that implementation of Productive Failure design with open ended debugging problems supported participants' learning of computational concepts. To answer how PF design supported learning of computational concepts, I conducted artifact and interaction analysis. I analyzed students' iterative solutions to debugging problems and their interactions during the Solution Generation and Consolidation phase to understand aspects of PF design that supported students' learning. I used case studies to provide in-depth analysis of three aspects of Productive Failure design that supported learning of computational concepts and how these aspects of PF design supported learning. These cases were a) Learning Through Failure/Trial and Error, b) Suboptimal Solution Generation Supports Mitigation of Recurring Errors, and c) Facilitation Supports Solution Generation Through Changes in Debugging Strategies.

Second, I answer RQ 2.1: What kinds of challenges do students encounter, the resolutions they come up with and the debugging strategies they used when working through the debugging curriculum? To answer RQ 2.1, I used interaction analysis in conjunction with artifact analysis to unpack the challenges students faced working on the debugging problems, the solutions that they came up with and the debugging strategies that they employed. Guidelines provided by Braun, Clarke and Hayfield (2015)
and Saldana (2021) were used to generate, organize, analyze emerging themes and to select themes that could answer RQ 2.1. Based on the qualitative analysis, I uncovered

a) Four patterns of challenges: a) High Frequency of Syntax Errors and General Bugs, b) Recurring Smelly Codes, and c) Affective Challenges.

b) Four patterns of resolutions and debugging strategies: a) Problem Resolutions Through Individual Attempts and Facilitator Feedback, b) Trial and Error as the Default Debugging Strategy, c) Backward Reasoning Utilized Frequently as an Error Location Strategy but Forward Reasoning and Combination Strategy was More Successful, and d) Switch from Bottom-Up Bricolage Programming to Top-Down Approach

Third, I answer RQ 2.2: What do these challenges, resolutions, process, and interaction patterns reveal about students’ evolving understanding of computational concepts? To answer, RQ 2.2, I used interaction analysis and thematic analysis to examine and analyze changes in participants’ understanding of computational concepts as they progressed through the debugging curriculum. I analyzed Think Alouds, student generated artifacts and online interviews data to generate, analyze and select themes that could best answer RQ 2.2. Based on my analysis, I uncovered three patterns that provided insights into students changing understanding of computational concepts: a) Improved Application Accuracy of Some Computational Concepts, b) Issues with Combining Computational Concepts to Debug Problems and Write New Codes, and c) Influence of Error Mitigation on Code Efficiency.

Quantitative Findings

Wilcoxon Signed Rank Test

I used Wilcoxon Matched Pairs Signed Ranked Test due to the small sample size (n=6) and discrete data set that did not follow a normal distribution to analyze the pre-post scores. The Wilcoxon results (See Table 4-1) showed that overall post scores increased from pre-scores, $z = -10.52, p < 0.001, r = 0.35$, which was statistically significant with a medium effect size ($r > 0.3$). This suggests that after the
workshop, participants have reported improved programming skills as well as increased interest and confidence in coding.

Interestingly, Computer Programming Self-Efficacy (CPSES) saw a statistically significant increase in score with a large effect size ($r > 0.5$), $z = -9.25, p < 0.001, r = 0.58$. The workshop along with its exercises may have played a role in strengthening participants’ programming knowledge, skills as well as learning and understanding of fundamental computational concepts.

Additionally, in both Attitude Scale of Computer Programming Learning (ASCOPL), $z = -4.83, p < 0.001, r = 0.28$, and Problem-Solving Inventory (PSI), $z = -2.12, p = 0.034, r = 0.11$, the changes were statistically significant with small effect sizes ($r > 0.1$). This suggested that the workshop helped participants foster interest and willingness in programming and mitigate negative emotions around coding. Additionally, the workshop also positively impacted students’ problem-solving confidence, tendency to avoid or approach problem-solving activities and ability to control emotions specifically when faced with challenges.

Further Wilcoxon tests were conducted for each of the subcategories under Problem Solving Inventory (PSI). There was statistically significant increase in scores after workshop for questions related to Emotional/Personal Control, $z = -2.90, p = 0.004, r = 0.37$ (indicating a medium effect size) as well as questions related to Problem Solving Confidence, $z = -3.20, p = 0.001, r = 0.28$ (indicating a small effect size). The change in scores for questions regarding Approach Avoidance Style was not statistically significant, $z = -1.00, p = 0.317, r = 0.07$ (indicating a less than small effect size). This indicated that after the workshop, some participants mitigated their difficulty when confronting challenges in coding, both in terms of technique and mentality. Meanwhile, some participants still had similar levels of tendencies to avoid problem solving during programming before and after the workshop.

Table 4-1: Wilcoxon signed-rank test results ($\alpha = 0.05$). Data analysis of 6 participants’ pre and post survey responses.
<table>
<thead>
<tr>
<th></th>
<th>Pre (median)</th>
<th>Post (median)</th>
<th>p Value (2-tailed)</th>
<th>z</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Question Groups</td>
<td>4</td>
<td>5</td>
<td>&lt; 0.001</td>
<td>-10.52</td>
<td>0.35</td>
</tr>
<tr>
<td>ASCOPL</td>
<td>4</td>
<td>5</td>
<td>&lt; 0.001</td>
<td>-4.83</td>
<td>0.28</td>
</tr>
<tr>
<td>CPSES</td>
<td>2</td>
<td>5</td>
<td>&lt; 0.001</td>
<td>-9.25</td>
<td>0.58</td>
</tr>
<tr>
<td>PSI</td>
<td>4</td>
<td>5</td>
<td>0.034</td>
<td>-2.12</td>
<td>0.11</td>
</tr>
<tr>
<td>PSI--Emotion</td>
<td>3</td>
<td>4.5</td>
<td>0.004</td>
<td>-2.90</td>
<td>0.37</td>
</tr>
<tr>
<td>PSI--Avoidance</td>
<td>4</td>
<td>4</td>
<td>0.317</td>
<td>-1.00</td>
<td>0.07</td>
</tr>
<tr>
<td>PSI--Confidence</td>
<td>5</td>
<td>5</td>
<td>0.001</td>
<td>-3.20</td>
<td>0.28</td>
</tr>
</tbody>
</table>

**Median Scores**

Pre and post data was also analyzed for all 6 participants who completed both pre and post surveys and answered all 77 questions (See Figure 4-1). Four participants were from grade 11, one was from grade 10 and one was from grade 12. Here no difference in programming skills was observed in participants from different grades. Based on the median scores, all but two participants reported improvement in their programming knowledge. These participants also indicated a positive attitude towards programming after taking part in the workshop.
Box plots

Box plots on data from each question group are shown below. In Q5, Attitude Scale of Computer Programming Learning (ASCOPL), distribution of scores shifted upward suggesting improved computer programming attitude (See Figure 4-2). There was an increase in 25th percentile, median as well as 75th percentile. In Q6, Computer Programming Self-Efficacy (CPSES), scores saw the biggest increase, demonstrating enhanced programming self-efficacy (See Figure 4-3). In Q7, Problem Solving Inventory (PSI), post scores on questions related to problem solving inventory did not change significantly from pre scores in terms of median or distribution (See Figure 4-4).
Figure 4-2: Q5, Attitude Scale of Computer Programming Learning (ASCOPL) data box plot. Data analysis of 6 participants’ pre and post survey responses to 24 survey questions.

Figure 4-3: Q6, Computer Programming Self-Efficacy (CPSES) data box plot. Data analysis of 6 participants’ pre and post survey responses to 21 survey questions.
Figure 4-4: Q7, Problem solving inventory (PSI) data box plot. Data analysis of 6 participants’ pre and post survey responses to 32 survey questions.

Qualitative Findings RQ1

The formative assessment (pre-post survey) supplemented the qualitative findings for RQ1 to assess learning of computational concepts and how that learning occurred with the support of Productive Failure design. Learning was defined as participants correctly applying computational concepts and sequencing the code blocks to debug the problems (Brennan & Resnick, 2012; Grover, Cooper & Pea, 2014; Kafai et al. 2014). Learning was also determined as the improvement in understanding and application of computational concepts assessed as:

- reduction in the application errors during debugging
- reduction in the number of iterative attempts taken to solve the problem at hand
- operational code and
- added functionalities to the given code that showcased coding efficacy and understanding of computational concepts not targeted in the given problem
I used interaction analysis to unpack participants' discourse on how Productive Failure Design supported their learning of computational concepts. In order to do that, I created case studies that would help me understand the specific aspects of Productive Failure design that supported learning and how these aspects of PF design supported learning.

Artifact analysis was conducted in conjunction with interaction analysis to track their iterative attempts, changes, and edits in code sequences throughout the debugging curriculum that provided insights into their learning. For interaction analysis, I selected, categorized, and analyzed segments during the solution-generation and knowledge assembly phase where participants

- explained and showcased their learning from debugging failures.
- highlighted their changing understanding and application of computational concepts as they worked through the debugging curriculum.
- showed signs of change in their understanding and application of computational concepts.

To conduct the case studies, interactions were studied among participants and the facilitator (Giri) as well as participants and artifacts during the solution generation and consolidation phases. Multimodal transcripts were created to track, understand, and analyze participant’s interaction with and manipulation of artifacts. These artifacts were debugging solutions that participants created along with the code sequences that participants created, code blocks and sprites that participants were manipulating on screen during debugging activities. In the multimodal transcripts, I identified the time, speakers, utterance, actions, and artifacts (code blocks, code sequences, sprites) manipulated on screen. Within non-verbal, embodied actions, I noted their actions towards the artifacts during conversations as well as standalone artifact manipulations on screen. These actions included hovering over, moving, pointing, emphasizing, circling, inscribing and dragging-dropping code blocks, sprites, code sequences during debugging (solution generation) and debrief sessions with the facilitator (knowledge assembly. I analyzed these verbal and non-verbal interactions, code manipulations and generated solutions during iterative and failure prone cycles of debugging and after knowledge assembly sessions with the facilitator to understand participants’ learning and understanding of computational concepts.
Case I: Learning Through Failure/Trial and Error

Implementation of Productive Failure Design across the debugging curriculum meant that in Phase I: Exploration and Solution generation, students worked on debugging problems without scaffolding and direct instruction on computational concepts, coding blocks and their application. Students were task ed to debug the problems and, in the process, create multiple solutions to the problem. Without prior experience with coding and the absence of direct instruction as an intentional feature of the PF design, students generated solutions using Trial and Error. While this informal heuristics as a debugging strategy was a time sink and frustrating, the strategy turned out to be effective for students to explore the functionalities of the code blocks and figure out the correct application of computational concepts in their code sequences to effectively troubleshoot the problem. Excerpts 1 (See Table 4-2) and 2 (See Table 4-3) show how Trial and Error during the Solution Generation Phase I supported fundamental understanding and consistent learning of computational concepts. The discussion and figures (See Figure 4-5, 4-10) following the excerpts outline and analyze the solutions generated by Nina and Jamarcus using Trial and Error.

Table 4-2: Excerpt 1, Jamarcus and facilitator discussing solution for Debug 2 and learning coding through trial and error

<table>
<thead>
<tr>
<th>Turn</th>
<th>Speaker</th>
<th>Verbal and Non-verbal transcript</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Facilitator</td>
<td>Okay. That was great! So can you go back and share your screen and talk to me about what you did there? [points to Jamarcus’s Debug 2 codes]</td>
</tr>
<tr>
<td>2</td>
<td>Jamarcus</td>
<td>Yep. So basically what I did here was, I made it so that if it [the chicken sprite] was touching the corn kernel right? And it would glide to a random position. [points to Glide 1 second to corn kernels] A little bit like Move 10 steps here but it will glide, so it’d be smoother.</td>
</tr>
<tr>
<td>3</td>
<td>Facilitator</td>
<td>So what are some of the things that you tried and that did not work out?</td>
</tr>
<tr>
<td>4</td>
<td>Jamarcus</td>
<td>Some of the things that I’ve tried. I tried making the chicken turn around by like</td>
</tr>
</tbody>
</table>
90 degrees. But what would happen was that it would keep on turning forever. It will turn, and then just keep going straight. So it would be upside down. I also tried changing its angle But that didn’t work out either.

Facilitator So, how did you get to the right solution? How did you think about it?

Jamarcus Well, basically, I did trial and error. Basically, I found out the things that didn't work, so that when I found the glide button, it solved my problem, because then I could have him move around without having to turn.

Facilitator Right! Talk to me about trial and error. How did that actually help you? How do you feel about that process?

Jamarcus It's a little frustrating, but it's also fun, you know? Sometimes it's kind of disappointing, but I just have to keep trying cause every time I tried a code, I learn something. I knew what it did, where to put it and how I can use that later. Cause, that didn't work, so now it's like solving a puzzle. Like I said last time, sometimes if I tried a lot of things I might try to step back and analyze it again to see what worked and what didn't. So, I looked around the Events, the Controls and the Sensing blocks and found something that would do what I want.

Table 4-3: Excerpt 2, Nina explaining her debugging process and discussing the helpfulness and frustration of learning computational concepts through trial and error during online interview

<table>
<thead>
<tr>
<th>Turn</th>
<th>Speaker</th>
<th>Verbal and Non-verbal transcript</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Facilitator</td>
<td>How was working on the debugging problems? What was interesting, easy, difficult?</td>
</tr>
<tr>
<td>2</td>
<td>Nina</td>
<td>It is very interesting because I am still figuring out every concept in there, but when I was working on that particular game [Debug 3], I couldn't find the code I was missing.</td>
</tr>
<tr>
<td>3</td>
<td>Facilitator</td>
<td>What were some of the challenges, problems you faced while working on the problem?</td>
</tr>
<tr>
<td>4</td>
<td>Nina</td>
<td>I still struggle with some of the [computational] concepts. For example, I wanted to make the sprite move without touching the edge, so I put “Move” [Move 10 steps], but when I pressed the right or left click, it just moved right.</td>
</tr>
<tr>
<td>5</td>
<td>Facilitator</td>
<td>How did you go about solving these problems? Can you explain your debugging process in detail?</td>
</tr>
</tbody>
</table>
6 Nina I used motion, events and if-then [conditionals]. I chose one of the motion options [Move 10 steps] and I had to use the point in direction option [Point in direction 0] to make it go where I wanted it. I am still pressing every option [code block] because I still struggle with some of the [computational] concepts. I then used the if and then statement to make the Sprite go away from the edge [boundary panels]. Here, I used ‘If Touching Color Pink’ and ‘Move -10 steps’ to avoid the Sprite from touching the edge and in that way the sprite doesn’t go that way. When I just used the Move -10 Steps option, it did not move at all, and it was because I was still missing all these codes.

7 Facilitator How did you feel about trying out different solutions and testing to whether it worked?

8 Nina It can be very frustrating because I was not interested in coding at all before. But now that I have made some progress, it feels awesome!

9 Facilitator Do you think this process has helped you understand some coding concepts?

10 Nina Yes. Trying different solutions was helpful for me because I learnt the concepts that I was struggling with before like events, if-then and putting code in the correct place [sequence]. Now I know what each of these concepts mean and how to use it in my code. I am also learning how to use it with other code blocks to get the game to do more things. I think this process made me realize that some solutions that you think aren't going to work the way you wanted to. So it made me open my mind a bit, help me look at the other options. I have learned that I have to trust the process.

In both excerpts, students relied on Trial and Error to debug the problem at hand. Both Jamarcus and Nina tried out various blocks at random and in the process learnt the functionalities of those blocks and how to incorporate these blocks into their codes. The feedback loop built into Scratch also helped them locate errors in their code by examining the output to determine the Sprite actions that initiated or not. Through Trial and Error, they “found out the codes that didn't work” and edited their final code to remove these faulty blocks from the final code (See Table 4-2, Turn 6; Table 4-3, Turn 10). For example, Jamarcus experimented with motion blocks in his first two solution-generation attempts for Debug 2 before figuring out the optimal code sequence in his third iteration. For Debug 2, students were asked to a) get the chicken sprite to move to corn kernels, b) then saying “Yumm,” and c) move the sprite across the field. The given code consisted of a pecking animation sequence with a combination of look block ‘Switch costume to _’ and control block ‘Wait 0.2 seconds’ housed inside a Forever loop and initialized
when the chicken sprite was clicked (See Fig. 4-5, right). However, the given script was missing codes to move the chicken to the food source (corn) and around the farm.

Figure 4-5: Stage for Debug 2 with the chicken and corn kernels sprite in a farm background (left); Given script missing code sequence to move the chicken sprite to the corn kernels and around the farm (right)

In his first attempt, Jamarcus tried out a two Motion combo sequence placing ‘Turn 90 degrees clockwise’ and ‘Move 30 steps’ inside the ‘Forever’ loop and right after the pecking animation sequence (See Figure 4-6, top left). However, this code did not work out because once it was initialized i.e., when the chicken sprite was clicked, the chicken “would keep on turning forever. So, it would be upside down” without ever moving towards the corn kernels (See Table 4-2, Turn 4). The forever loop ensured that the code repeated after ‘Move 30 steps’ and the chicken continued turning 90 degrees clockwise hindering forward movement towards the kernels (See Figure 4-6, top-right). However, Jamarcus’s code would have worked if he took out ‘Turn 90 degrees clockwise’ allowing the chicken to continuously ‘Move 30 steps’ forward towards the corn kernels without turning. In his second iteration, Jamarcus took out the ‘Move 30 steps’ and ‘Turn 90 degrees clockwise’ blocks from iteration 1 code and added in a ‘Point in direction 180 degree’ (See Figure 4-6, bottom-left). However, this approach did not work either as ‘Point in direction’ block just set the orientation and direction of movement of the sprite but not the movement itself. For instance, ‘Point in direction 90 degree’ oriented the chicken sprite upright and facing front while ‘Point in direction 180 degree’ simply oriented the chicken to face downwards. Without directional movement blocks such as ‘Go to X: _ Y: _’ or ‘Glide’ blocks accompanying the change in degree of
directional orientation, the chicken stayed at the same spot pointing downwards but did not move towards the kernels (See Figure 4-6, bottom-right).

Figure 4-6: Jamarcus’s iteration 1 code (top-left); chicken sprite continuously turning on the same spot due to the ‘Turn clockwise 90 degrees’ block inside the forever loop (top-right); Jamarcus’s iteration 2 code (bottom-left); chicken sprite oriented downwards due to the ‘Point in direction 180 degrees’ (bottom-right).

Jamarcus debugged the problem in his third attempt. He deleted the ‘Point in direction 180 degree’ inside the ‘Forever’ loop as this code arrangement kept on turning the sprite. Rather, Jamarcus added the ‘Move 30 steps’ inside the loop after the pecking animation code. When initialized, the chicken moved continuously towards the corn kernels thereby completing task 1 (See Figure 4-7). To complete task 2: say ‘Yumm!’ to indicate chicken feasting on the corn, Jamarcus added a conditional ‘If-then’
paired with ‘Touching corn kernels’ sensing block and ‘Say Yumm!’ block (See Figure 4-7, left). He could have just added ‘Say Yumm!’ block below the ‘Move 30 steps’ to complete the second task, but he did more by adding a ‘If-then’ conditional whereby the chicken would say Yumm! only if it was touching the corn kernels which was a more realistic game effect to signify eating (See Figure 4-7, right). Jamarcus also completed task 3: get chicken to move around the field. He paired a ‘Glide 1 secs to random position’ with a ‘Forever’ loop after the ‘Say Yumm!’ block, which was effective as the chicken would move to different positions in the farm after eating the food.

Figure 4-7: Jamarcus’s iteration 3 code (left); chicken sprite moving to the corn kernels with the addition of ‘Move 30 steps’ inside the forever loop to complete task 1 of Debug 2 (right)

Trial and error also turned out to be effective in learning fundamental computational concepts such as events, conditionals, loops and putting codes in proper order (sequences). Without prior training on coding and knowledge of computational concepts, both Nina and Jamarcus were “struggling” with concepts such as events, conditionals and sequences and incorporating these concepts in their codes (See Table 4-2, Turn 4; Table 4-3, Turn 10). They tried out different permutations and combinations of conditionals (If-Then), Loops (Forever) and Event blocks in their iterations. Both Nina and Jamarcus found that while the process was “frustrating,” and at times “disappointing,” trying out different solutions helped them understand and apply these fundamental computational concepts in their code. They reported
that they were able to also figure out how to combine these concepts with other code blocks “to get the
game to do more things” (Table 4-3, Turn 10).

This learning was evident in their solutions as well. With each iteration, they located errors and
refined their code to debug the problem at hand and at times extend the game. For example, in her Debug
3 solution, Nina refined her iteration 1 solution to streamline the directional movement. She also extended
the game by getting the pufferfish sprite to move back 10 steps whenever the fish touched the pink
boundary panels. For Debug 3, students were asked to a) get the pufferfish sprite to move 10 steps in the
direction of arrow key pressed (U, D, L, R) and b) bonus task to get the fish sprite to say something when
it reaches Nemo at the end of the maze (See Figure 4-8, left). The given code consisted of incomplete
directional codes missing motion blocks that controlled the sprite’s movement (See Figure 4-8, right).
Additionally, the given code also setup sprite’s initial screen position with ‘Go to X: -172 Y: 110,
appearance with ‘Switch costume’ block and the rotation style, i.e., left-right, all around or don’t rotate, to
determine which direction a sprite can face throughout the game.

![Figure 4-8: Stage for Debug 3 with pufferfish and Nemo sprites in a maze background (left), Given script with code sequence that setup initial starting position and appearance of the pufferfish sprite and incomplete directional codes (right)](image)

In her first debugging attempt, she added ‘Move 10 steps’ motion block under the event blocks
i.e. ‘When up arrow key pressed,’ ‘When left arrow key pressed’ that initialized sprite’s directional
movements (See Figure 4-9, middle). This code did not function optimally as ‘Move 10 steps’ by default
always moves the sprite 10 pixel-length to the right. This is because Scratch uses an angle measurement
for direction where every degree added corresponds to 1 degree turn clockwise by the sprite. In this system, a direction of 0 represents Up where the sprite would point up while a direction of 90 means a sprite turns 90 degrees clockwise and would point right and a direction of -90 means that the sprite turns 90 degree anti-clockwise and would point left (See Figure 4-9, left). Additionally, in Scratch, the sprite’s default direction value is set to 90 degrees which represents no rotation meaning that a sprite will point and move right when motion blocks such as ‘Move 10 steps’ are used. Therefore, in Nina’s code, the addition of ‘Move 10 steps’ to the directional keys only moved the sprite rightwards even when up, down, and left keys were pressed (See Figure 4-9, right). The code would have worked if Nina had paired ‘Move 10 steps’ with ‘Point in direction__’ motion block whereby she could set the direction of movement for each arrow key press. For example, 0 for Up, 180 for down, -90 for left and 90 for right.

Figure 4-9: The direction system used in Scratch with 90 being the default sprite direction pointing the sprite rightwards (left); Nina’s iteration 1 code (middle); addition of ‘Move 10 steps’ block moved the pufferfish sprite rightwards even when up, down, and left keys were pressed (right)

In the second iteration, Nina optimized her iteration 1 code to streamline the directional movement and also built on the existing game by adding a game feature and remixing the background (See Figure 4-10). She continued to use trial and error “pressing every option” and tinkering with motion, event, and conditional blocks because as a novice programmer, she struggled with these computational
concepts (See Table 4-3, Turn 6). However, Nina also found that “trying different solutions was helpful” for her to learn to apply these difficult computational concepts correctly. She learned “what each of these concepts mean[t]... how to use it in her code” and how to pair it with other code blocks to add game features and extend the code (See Table 4-3, Turn 10). Nina also learned that she had to “trust the process.” She learnt that with trial and error, she needed to accept that some solutions would not work out but that she could learn from these failed solutions to think of and try a different code arrangement (See Table 4-3, Turn 10).

This learning was evident in her iteration 2 code. Nina optimized sprite’s faulty directional movements by pairing ‘Move 10 steps’ with ‘Point in direction _’ motion block and changed the degree of rotation inside the direction block to match sprite’s movement with the arrow key pressed (See Figure 4-10, left). For instance, for the Up-arrow key press, she set the degree of rotation inside ‘Point in direction _’ to 0 degrees which paired with ‘Move 10 steps’ moved the fish sprite 10 steps to the right and then upwards (See Figure 4-10, right). Similarly, Nina set the degree of rotation for Down arrow key press to 180 degrees, Right to 90 degrees and Left to -90 degrees. While this code arrangement got the fish to move Up, Down, Left, Right resolving Debug 3 task 1, the code could have been further optimized by reversing the order of ‘Move 10 steps’ and ‘Point in direction _.’ This reversal would set the direction of movement first before the actual movement happened which would have reduced the awkward ‘Move 10 steps’ that preceded before pointing the sprite to the direction of the arrow key press.

In addition to debugging the sprite’s movement problem, Nina also remixed the game by adding an ocean background, changing the maze setup and color of the boundary panels from blue to pink to personalize the game (See Figure 4-10, right). Moreover, Nina extended the game by adding a game feature whereby the pufferfish sprite moved back 10 steps when touching the pink boundary panels. Initially, she tried putting ‘Move -10 steps’ below the ‘Point in direction _’ block. However, this code sequence just moved the sprite 10 steps back disregarding the condition for the action: if touching pink boundary panel then move 10 steps back. Realizing that she was “missing all these codes,” she was started tinkering with different code blocks until she found the ‘If-then’ conditional (See Table 4-3, Turn
6). Nina paired the conditional with the sensing block ‘Touching pink boundary panel color’ and ‘Move - 10 steps’ and copied this code for all directional keys (See Figure 4-10, left). She also added ‘If on edge, bounce’ motion block to left arrow key code to prevent the fish from moving beyond the edge of the screen. Nina’s code showed her attention to gameplay and game mechanics to simulate an ideal maze game. Moreover, iteration 2 code indicated Nina’s improvements in the application and understanding of conditionals and sequences through trial and error.

Figure 4-10: Nina’s iteration 2 code with improved directional movement and conditional to move the fish 10 steps back if touching pink boundary panel (left); Nina’s remixes included changes to the maze and addition of the ocean background (right)

Nina and Jamarcus’s cases indicated that Productive Failure design supported learning of computational concepts and code optimization by providing opportunities to test out different code blocks and script arrangements and learn from their failures. Productive Failure learning design reversed the learning sequence whereby problem solving preceded direct instruction and scaffolding. While reversing the instruction sequence can seem counterintuitive, this reversal provided opportunities for learning and deeper understanding of fundamental computational concepts. Students overwhelmingly relied on informal heuristics and specifically on Trial and Error. While this approach was “frustrating” and “disappointing” at times with recurring failures, students also reported that the approach was also “fun,”
“helpful” to learn and apply computational concepts that they were struggling with and provided opportunities to “step back and analyze…what worked and what didn't” (Table 4-2, Turn 8; Table 4-3, Turn 6, 8, 10). Moreover, the process was economical, fast, and provided opportunities to iterate on solutions with different combinations and sequences of coding blocks. Through the iterative generation of sub-optimal solutions, students learned what these concepts meant and figured out the correct application of computational concepts such as events, loops, and conditionals in their code to not only debug the problem at hand but to extend their games. However, trial and error also had its drawbacks and led to systematic, predictable errors and cognitive biases noted by Tversky and Kahneman (1974). The most common problem was time sink whereby students spent a significant amount of time randomly incorporating different code blocks and combinations into their code sequence in the hopes of debugging the problem. Additionally, anchoring bias and availability bias also added to the overall debug time and time taken to move to another solution. Students fixated on first solutions overly relying on the first piece of information or set of code blocks that they thought would solve the problem.

Case II: Mitigation of Recurring Errors Through Suboptimal Solution Generation

The generation of suboptimal solutions minimized recurring errors and error patterns evidencing correct application and learning of computational concepts overtime. Student's’ initial solutions to debugging problems contained recurring errors such as misordered code blocks and choosing incorrect, random blocks that were reflective of the student's’ lack of understanding of computational concepts and coding. However, generation of suboptimal solutions minimized recurring errors over time and across the debugging curriculum. This was evidenced by the reduction in number of iterations needed to generate a functional code for a debugging problem and the subsequent mitigation of specific coding and conceptual error patterns as students progressed through the debugging problems. Minimization of these recurring error patterns showcased students progressive understanding and application of computational concepts. Additionally, it also showed the impact of the PF design with its solution generation and consolidation phases to support learning of computational concepts.
Excerpt 3 (See Table 4-4) presents an instance of students’ reported impact of suboptimal solutions on minimizing recurring sprite movement errors. Other students, such as Damari, noted improved understanding of computational concepts through suboptimal solution generation in addition to the error mitigation effect. Additionally, Table 4-5 tracked the progression of specific error patterns in students’ generated solutions as they progressed through the debugging problems. Findings from Table 4-5 indicated error mitigation and decreased occurrences in multiple error patterns. The findings provided support for improved understanding and increased accuracy in applying computational concepts such as sequences, loops, and parallelisms. However, the table also indicated error patterns where the frequency of errors stayed relatively constant or even increased in later problems indicating student’s’ persistent struggle with specific computational concepts such as variables, conditionals, and other general bugs.

Table 4-4: Excerpt 3, Delani explaining the solutions that she generated for Debug 6 with the facilitator and what she learned in the process during Think Alouds and online interview

<table>
<thead>
<tr>
<th>Turn</th>
<th>Speaker</th>
<th>Verbal and Non-verbal transcript</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Facilitator</td>
<td>You did a fantastic job moving Shantel from one place to another and getting the dancing done. Could you explain your process, what the problem was and how you debugged it?</td>
</tr>
<tr>
<td>2</td>
<td>Delani</td>
<td>Yeah. I did a couple of tries on this one. It was kind of confusing since it [the sprite] couldn't move anywhere. But what I did was I went onto &quot;Events&quot; and on this block [Shows ‘When this Sprite clicked’ block]. And for her to move from left to right, I looked at motions and found Glide and Go to X and Y.” I decided to test both of them to see which one worked better. I decided to put “Glide” [Points to Glide 1 secs to X: 171 Y: -29] instead since with “Go to X and Y” [Points to Go to X: 157 Y: -46 in iteration 1 code], she teleported instead of moving her slowly across the room.</td>
</tr>
<tr>
<td>3</td>
<td>Facilitator</td>
<td>So what do you think is happening right now? She's gliding from left to right. But how are you thinking of having her go left to right and dancing at the same time?</td>
</tr>
<tr>
<td>4</td>
<td>Delani</td>
<td>I might add another glide [circles Glide 1 secs to X: 171 Y: -29 in iteration 2 code] or just another motion block. I think putting different glide blocks which takes her to different points in the room could work.</td>
</tr>
<tr>
<td>5</td>
<td>Facilitator</td>
<td>That's great!</td>
</tr>
</tbody>
</table>
For her third solution, Delani added 3 different glide blocks to move Shantel to move her to various locations on screen, but she still would not dance and move at the same time. In the online interview below, Delani explained how she came to the correct solution and discussed the impact of creating and working through different solutions.

6 Facilitator What were some of the solutions that did not work? Why do you think that these solutions did not work?

7 Delani I tried different motion blocks at first. The block go to x: y: [Go to X: 157 Y: -46] did not work out for me since I wanted my character to move slowly from left to right, but this block teleported her from the left side to the right. Then I tried a glide [Glide 1 secs to X: 171 Y: -29] which moved Shantel left to right but did not get her to dance at the same time. Then I tried putting different glide blocks [Glide 1 secs to X: -116 Y: -35, Glide 1 secs to X: -21 Y: -25 and Glide 1 secs to X: 39 Y: -15] and then one go to x: y: block [Go to X: 157 Y: -12]. What I wanted was to move her [sprite] to different places [on screen] slowly from left to right and dance at the same time. But she would sometimes glide to one place and then dance or she would just dance. Then the teacher said that part of the code was just animation of her dancing. So I thought maybe if I get her to move just a little and play the animation over and over then that would work. I found Repeat and Forever [blocks] but I chose Forever because it made her dance without stopping. It worked in the end but it took a while.

8 Facilitator How did you feel about the process of coming up with different ideas to solve the problems, trying out different solutions and testing whether it worked?

9 Delani I felt like I was using my brain more. Finding solutions to this problem that worked out well while fixing the ones that didn't made me feel more active and a little frustrated. When a few of my solutions worked out I felt happy since the debugging problem was a little difficult but when it didn't I felt confused but I tried finding other ways to fix it. Trying different solutions told me what didn't work and I stopped doing that [making the same mistake]. When I tried those glide blocks and it didn’t work, I knew I had to think different. So I thought about making her move a bit and then do the dance animation forever and it worked.

In excerpt 3, Delani explained the impact of Solution Generation phase in minimizing recurring movement errors. Her Debug 6 solutions showed that iterative solution generation improved her conceptual understanding and application of coding blocks and computational concepts such as sequences and loops. With this improved understanding, Delani resolved the recurring movement error and generated a functional code in her fourth attempt. For Debug 6, students were tasked with a) getting Shantel (sprite) to move from left side of the room to the right doing some cool dance moves when the sprite is clicked and b) bonus task: adding some cool music and background to Shantel's routine (See
The given code consisted of a setup code initialized when Green flag was pressed which set up the sprite’s initial position with ‘Go to X: -176 Y: -57 and played two beats in a loop (See Figure 4-11, right). The given code also consisted of an incomplete code sequence with a dance animation sequence with a combination of look blocks ‘Switch costume to _’ and control block ‘Wait 0.3 seconds.’ However, this script was intentionally missing an event block and a loop to get Shantel to move from left to right while dancing simultaneously.

Figure 4-11: Stage for Debug 6 with the sprite Shantel in a room backdrop (left); Given script with sprite setup code on the left and incomplete code sequence on the right with dance animation codes, set rotation and movement styles missing event and forever loop for continuous dance moves (right)

Delani experimented with motion blocks in her first three iterations to complete the first task for Debug 6– get the sprite across the screen while dancing and moving at the same time (See Figure 4-12, 4-13). In her first attempt, Delani added the missing Event block ‘When this sprite clicked’ to initialize the given incomplete code (See Figure 4-12, left). Delani also added a ‘Go to X: 157 Y: -46’ motion block. However, this code did not complete Debug 6 tasks because once initialized, the ‘Go to X Y’ block teleported Shantel directly to the fixed coordinates at the edge of the screen rather than moving her steadily to the right while doing the dance moves concurrently (See Figure 4-12, right). In her second attempt, Delani tried the ‘Glide 1 secs to X:171 Y: -29’ block (See Figure 4-12, middle). However, she found that the glide block “moved Shantel left to right but did not get her to dance at the same time”
(Table 4-4, Turn 7). This code also did not work out because the dance code animation ran after the Glide block. As such, Shantel first glided rightwards to the specified coordinates at the edge of the screen rather than moving from left to right while doing the dance moves concurrently. For both iteration 1 and 2 codes, deleting the ‘Go to X: 157 Y: -46’ and ‘Glide 1 secs to X:171 Y: -29’ blocks then adding a ‘Forever’ loop enclosing the code sequence would have optimized the code by initializing simultaneous movement and dance.

Figure 4-12: Delani’s iteration 1 code with ‘Go to X: 157 Y: -46’ motion block (left); Iteration 2 code with ‘Glide 1 secs to X: 171 Y: -29’ motion block (middle); Shantel’s position at the edge of the screen after Iteration 1 and 2 code initialization (right)

Delani continued to experiment with motion blocks in her third attempt. She found that the ‘Go to X Y’ and ‘Glide 1 secs’ block moved Shantel left to right but did not get her to dance at the same time. As such, she tried putting different glide blocks [Glide 1 secs to X: -116 Y: -35, Glide 1 secs to X: -21 Y: -25 and Glide 1 secs to X: 39 Y: -15] and one ‘Go to X: Y’ block [Go to X: 157 Y: -12]. She also added wait times ‘Wait 0.3 secs’ and duplicated the dance animation codes i.e. ‘Switch costume’ blocks (See Figure 4-13, left). The idea was “to move Shantel to different places [on screen] slowly from left to right and dance at the same time” (Table 4-4, Turn 7). However, iteration 3 code also did not function as intended due to issues with code sequencing. When initialized, the sprite glided to the specified X and Y
coordinates on screen, performed the dance steps and moved again to a new position. However, Shantel’s motion and dance steps were not synchronized, appearing disjointed and badly executed (See Figure 4-13, right).

She ultimately debugged the problem in her fourth attempt. By generating and testing out three suboptimal solutions, she learned that “glide blocks didn’t work” and that she “had to think different” (Table 4-4, Turn 9). She mitigated recurring movement errors by deleting the ‘Glide’ and ‘Go to X Y’ motion blocks. Delani also fixed errors in code sequencing to optimize and synchronize Shantel’s motion and dance steps. She rewrote Shantel’s code pairing ‘Move 10 steps’ and the dance animation code (combination of ‘Switch costume’ and ‘Wait’ blocks) with ‘Forever’ loop (See Figure 4-14, left). This code got the sprite to move rightwards and simultaneously dance at the same time without the disjointed movement patterns that were persistent in the previous 3 iterations (See Figure 4-13, left).
Figure 4-14: Delani’s Iteration 4 code with ‘Move 10 steps’ and the dance animation code inside a ‘Forever’ loop (left); Code initialization displaying Shantel's synchronized rightwards movement and dance steps (right)

Delani’s case highlighted the impact of the Solution Generation phase in mitigating recurring errors through improved understanding and application of code blocks and computational concepts. With each iteration, she improved her understanding of different motion blocks and adjusted the code to refine Shantel’s movements. Delani also displayed improvements in code sequencing and understanding of loops to successfully synchronize Shantel’s movement and resolve Debug 6. Other students also reported similar experiences where they attributed iterative solution generation to improvements in their application accuracy of computational concepts. For instance, when asked about the impact of Productive Failure design on her understanding of computational concepts, Damari credited the iterative solution generation process as a means to better understand and accurately apply Events:

“I understand more with the Events concepts. This coding concept helped me with the dementors. In order for dementors to receive the message and disappear, I had to use the broadcasting block ['Broadcast expecto patronum!!'] in Elvira’s code, which is in Events [palette], in order to move them. Learning what I did with the broadcasting block will help me in the future when I need something to move or appear when a character says or does something.”
In Debug 7, students were tasked with a) getting the dragon to appear after the sprite, Elvira, said the spell ‘expecto patronum!!’ and b) making all dementors disappear after Elvira uttered the spell and the dragon breathed fire (See Figure 4-15, left). The given code for Elvira was missing the ‘Broadcast’ event block which relayed the message ‘expecto patronum!!’ and could activate scripts in other sprites with the matching event hat such as ‘When I receive ‘expecto patronum!!’’ (See Figure 4-15, middle). The code for dementors and dragon were missing the matching event hat to receive the message. Damari created two solutions for Debug 7 (See Figure 4-15, right).

In her first attempt, she added ‘When I receive expecto patronum!!’ event hat to both dragon and dementors code. However, without the ‘broadcast expecto patronum!!’ block added to Elvira’s code, the codes for dragon and dementor sprites did not initialize as the message ‘expecto patronum’ was not relayed and received by these sprites’ codes. In her second attempt, she added the ‘broadcast expecto patronum!!’ event block to Elvira’s code to resolve the code initialization error (See Figure 4-16, left). When ‘expecto patronum!!’ was received, the dragon sprite appeared (show and switch costume to dragon1-a) and breathed fire on the dementors (switch costume to dragon1-a) after a seconds wait (See
Figure 4-16, middle). Similarly, when ‘expecto patronum!!’ was received by the dementors, they disappeared from screen after the dragon had appeared and spit fire executing the spell’s effects.

Figure 4-16: Debug 7 Elvira’s code with ‘broadcast’ block added to relay ‘Expecto Patronum!!’ spell (left); Dragon’s code received the ‘expecto patronum!!’ spell to breathe fire towards dementors (middle); Dementors code received the spell and disappeared after dragon’s fire (right)

Damari reflected that iterative solution generation and tinkering with different code combinations improved her understanding and application of Events. She concluded that learning what she did with the broadcasting block would help her in future debugging when she needed a sprite to move or appear when another sprite says or does something. In other words, this learning mitigated errors related to conditionals and connected events in subsequent problems.

Recurring Errors in Students Suboptimal Solutions and Error Minimization Over Time

Mitigation of recurring errors was also seen across all students as they progressed through the debugging problems. I used Frädrich et al.’s (2020) catalog of Scratch error patterns as a framework to identify and categorize recurring errors in all students generated solutions from Debug 1-14. Frädrich et al.’s (2020) catalog tabulated frequent programming errors in Scratch encompassing general bugs, syntax errors, and code smells. As a result of this expansive and diverse bug patterns, I omitted error patterns related to custom blocks under ‘My blocks’, clones, ambiguous parameters, and advanced operators that
were outside the scope of the debugging problems in this study. Instead, I focused on frequent general bugs, syntax errors and code smells. I also added in an Other General Bugs section where I incorporated additional frequent bugs in student generated solutions such as redundant codes, missing and incomplete codes and long script among others. Some of these errors were already built into the problems that students had to debug such as Sequences (Debug 1-3), missing events and loops (Debug 4-6), missing conditionals and parallel code blocks (Debug 7-9) and operators and variables (Debug 10-12). However, I found that students ran into these same errors when they were working on their solutions and writing their own code sequences.

After cataloging frequent student errors, I calculated the total number and percentage of specific error patterns found in all students generated solutions for Debug 1 to Debug 14. The number of error patterns were calculated manually and through Litterbox. Litterbox is a static code analysis tool for detecting bug patterns and code smells in Scratch projects. However, Litterbox could not be used exclusively to identify and tabulate all error patterns in student generated solutions as it did not take into consideration content information about the debugging tasks. This static analysis of the code sequences meant that many solutions generated by the students were deemed as “bug free” when there were specific error patterns and code smells. Therefore, error patterns such as choosing random, incorrect blocks, misordered code blocks, missing loops, conditionals, and missing code smells such as additional loops, conditionals, redundant and incomplete codes were identified and counted manually. Table 4-5 shows the total number and percentage of recurring error patterns for all 9 participants.

Table 4-5: Total number and percentage of recurring error patterns for all 9 participants.

<table>
<thead>
<tr>
<th>Recurring Error Patterns</th>
<th>Debug 1-3</th>
<th>Debug 4-6</th>
<th>Debug 7-9</th>
<th>Debug 10-14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#</td>
<td>%</td>
<td>#</td>
<td>%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Category</th>
<th>Count</th>
<th>Percentage</th>
<th>Insertion</th>
<th>Execution</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Choosing random, incorrect code blocks</td>
<td>15</td>
<td>23.33</td>
<td>3</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>Misordered code blocks</td>
<td>14</td>
<td>31.8</td>
<td>1</td>
<td>4.54</td>
<td>0</td>
</tr>
<tr>
<td>Missed and unused Initialization/Events</td>
<td>8</td>
<td>16.67</td>
<td>4</td>
<td>22.22</td>
<td>3</td>
</tr>
<tr>
<td>Sequential actions</td>
<td>8</td>
<td>80</td>
<td>2</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>Missing Loop and Loop Sensing</td>
<td>12</td>
<td>28.12</td>
<td>3</td>
<td>9.37</td>
<td>8</td>
</tr>
<tr>
<td>Interrupted Loop Sensing</td>
<td>N/A</td>
<td>N/A</td>
<td>2</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>Missing and Incomplete Conditionals</td>
<td>4</td>
<td>7.69</td>
<td>4</td>
<td>30.77</td>
<td>4</td>
</tr>
<tr>
<td>Missing termination condition</td>
<td>3</td>
<td>14.29</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Unused or Uninitialized Variables</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Unused or Uninitialized Parallel actions</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Timing, Synchronization</td>
<td>6</td>
<td>27.27</td>
<td>1</td>
<td>9.01</td>
<td>1</td>
</tr>
<tr>
<td>Other General bugs</td>
<td>22</td>
<td>18.64</td>
<td>8</td>
<td>13.56</td>
<td>18</td>
</tr>
<tr>
<td><strong>Total Bugs</strong></td>
<td>102</td>
<td>46</td>
<td>22</td>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>
Table 4-5 shows that the frequency of error patterns associated with conceptual understanding and accurate application of computational concepts decreased as students moved throughout the debugging problems. Specifically, the occurrences of misordered codes, choosing random or incorrect code blocks, missing loops and loop sensing, timing and synchronization, sequential actions, unused or uninitialized parallel actions, missing and unused initialization/events decreased significantly (See Table 2-1 for definition and examples). Sequential errors (misordered codes) in student generated solutions decreased from 63.6% to 0% from Debug 1 to Debug 14. This decrease indicated that as students generated solutions and progressed through the problems, they increasingly ordered codes in the logical sequence directed in the programming tasks. Occurrences of missing loops and loop sensing also decreased steadily from nearly 37.5% to 25%. The sudden decrease in occurrences for Debug 7-9 to 9.37% was due to students’ lack of attempt at those problems and accurate application of loops by students who did attempt and debugged the problems. There was also a significant decrease in the occurrences of missing and unused initialization/events from 44.44% to nearly 17% and unused or uninitialized parallel actions from nearly 67% to 33.33% from Debug 1 to Debug 14.

Students also increasingly calibrated timing between code actions using ‘wait until’ and ‘wait 1 secs’ control blocks to synchronize multiple sprite actions or control time between each action. As such timing and synchronization errors decreased from 54.54% to 9.01%. Additionally, there was also a sharp decrease in students choosing random or incorrect code blocks in their solutions from 50% in Debug 1-3 to 10% in Debug 7-9 and back up to nearly 17% in Debug 10-14. Students' solutions and responses in interviews and think alouds (See Excerpt 3 and 4) show that this mitigation of error was due to students increasing understanding of the functionalities of different coding blocks. Additionally, the slight increase to 17% in Debug 10-14 was a result of students' first attempts at incorporating advanced computational concepts such as Data and Operators and the Variable and Operator coding blocks associated with those concepts.

However, there were error patterns where students did not show significant error mitigation which showed continued struggle with understanding and applying some computational concepts.
Additionally, frequency of some error patterns actually increased in later problems after an initial decrease. For instance, occurrences of **missing and incomplete conditionals** stayed at 30.77% with a slight decrease to 7.69% in Debug 4-6. Similarly, frequency of **termination conditions** (See Table 2-1 for definition and examples) also stayed at 42.86% in Debug 1-3 and Debug 10-14 with a slight decrease to 14.29% in Debug 4-6 and 0% errors in Debug 7-9. However, this sharp fall in occurrences of missing and incomplete conditionals in Debug 4-6 and termination conditions in Debug 4-9 was due to limited use of conditionals in their solutions rather than accurate application of conditionals resulting in decreasing errors. Students' solutions also showed that when students did use conditionals, the errors in writing complete conditionals with code for termination conditions still persisted. For instance, the given task in Debug 9 was to get the soccer ball to move to the goal post when the ‘Space’ key was pressed and if Jordyn (the sprite) touched the ball (See Figure 4-17, left). The given code for the ball sprite consisted of a script that set the initial position of the ball to a fixed coordinate on screen. Additionally, the given code contained an incomplete script with a ‘Glide 3 secs to X: 223 Y: -45’ that provided the destination coordinates for the ball at the back of the net (See Figure 4-17, middle). Willow added the missing event block ‘When space key pressed.’ She also added a conditional infinite loop by nesting the conditional to glide the ball towards goal if Jordyn touched the ball inside a forever loop (See Figure 4-17, right). However, Willow left the ‘Else’ part of the conditional blank which worked for this code because the ball stayed stationary if there was no contact with Jordyn. However, absence of a termination condition still affected overall code execution and optimization. The computer checked for the ‘Else’ condition every time and when the termination condition was not detected, the computer infinitely repeated the ‘If’ condition adding lag time for other codes to be executed.
Figure 4-17: Debug 9 stage with Jordyn and the ball sprite in a soccer field backdrop (left); Given ball code with an incomplete script with a glide block specifying destination coordinates at the back of the net (middle); Damari’s code for the ball with missing termination condition (right)

Additionally, for some error patterns like Other General Bugs and Unused or Uninitialized Variables, the error occurrences increased in later problems. The rate of Other General Bugs initially decreased from 37.29% in Debug 1-3 to 18.64% in Debug 4-6 and 13.56% in Debug 7-9. However, there was a sharp increase to 30.51% in Debug 10-14. The decrease in frequency of Other General Bugs in Debug 4-9 can be attributed to the low number of attempts for these problems. 4 students out of the 10 did not attempt these problems due to work and sibling care related absences on Day 3 and 4. Additionally, of the 6 students who were present and attempted Debug 4-9, the high occurrence of this error pattern still persisted as shown by the 19 errors in Debug 4-9. The error frequency was also back up to 30.51% in Debug 10-14. Students' solutions showed consistent occurrences of incomplete code blocks which were not initialized or completed and use of nested loops and conditionals when unnecessary which increased the likelihood of additional errors. Additionally, students had redundant codes which initiated the same action multiple times and affected the execution of other code sequences for particular sprites. The increased frequency of Other General bugs in later Debugs and consistent high occurrences throughout the problems indicated student's’ continued struggle with this error pattern.
To summarize the findings from Case II, the Solution Generation Phase of the PF design supported learning and understanding of computational concepts. Generation of suboptimal solutions minimized the occurrences of recurring error patterns that were persistent in students’ initial solutions. These recurring error patterns such as misordered codes, missing loops and loop sensing, timing and synchronization, uninitialized events and sequential actions reflected students’ limited understanding of computational concepts. Through iterative solution generation, students were able to identify, then eventually mitigate and even eliminate the occurrences of some of those error patterns in subsequent problems as shown in Table 4-5. This minimization of errors through suboptimal solution generation indicated students' increased understanding of computational concepts and the ability to accurately apply these concepts to generate an optimal code for debugging problems. However, it should be noted that not all error patterns showed significant reductions. For instance, error rates for the occurrence of **missing and incomplete conditionals** and **termination conditions** largely remained the same across Debug 1-14. Additionally, there was marked increase in error rates for the occurrences of **Other General Bugs** (i.e., incomplete, and missing code, redundant codes) and **Unused or Uninitialized Parallel actions**. This showed that while Solution Generation Phase I supported learning and improved understanding and application of some computational concepts such as Sequences, Events, Parallelism, and Loops, students continued to struggle with advanced concepts such as Variables, Conditionals, and Data (Operators).

**Case III: Influence of Facilitation on Solution Generation and Changes to Debugging Strategies**

Feedback during consolidation and knowledge assembly phase supported optimal solution generation and changes to students debugging strategies. Feedback on student solutions highlighted knowledge gaps in programming, inefficient debugging strategies and recurring errors in their solutions. Students acted on facilitator feedback to mitigate recurring errors and improve their knowledge of code blocks and programming concepts to generate efficient solutions. Facilitation also supported changes to efficient debugging strategies. Excerpt 4 highlights an instance of the impact of facilitator feedback in
Safiya’s understanding of Boolean and conditional container blocks and switch in debugging strategy from trial and error to forward reasoning.

Table 4-6: Excerpt 4, Safiya and facilitator discussing the solutions that she generated for Debug 8 during the Consolidation and Knowledge Assembly phase and her responses on her debugging process during the online interview.

<table>
<thead>
<tr>
<th>Turn</th>
<th>Speaker</th>
<th>Verbal and Non-verbal transcript</th>
</tr>
</thead>
</table>
| 1    | Facilitator | Safiya, do you want to go next? [Safiya: Yes]. Great! You are the co-host, and you can share your screen.  
[Safiya shares her screen and shows the solution that she came up for Debug 8] |
| 2    | Facilitator | Thank you Safiya. So, first things first, the “Touching” block [Touching Butterfly1], it can’t go inside the Forever loop because you see how the shape of that block is, it won’t fit into the Forever. It can only be used with something like “If then”, “If else”, “Wait until”, and “Repeat until”, so those are the only blocks that can take in the Sensing like the touching block. So if you can talk to me about what you were trying to do, then we can talk through the process. |
| 3    | Safiya   | So what I was trying to do was “When the dog touches the butterfly”, to get the dog to say, “Come back!” |
| 4    | Facilitator | Okay. In that case, you would have to use an “If then” statement. So you can use “Forever” there [pointing to the forever block she put underneath]. But you have to put the “If then” inside the “Forever”, what it would do is [doesn’t finish the sentence as Sofia starts putting in the “If-then” block inside the “Forever” and then puts the ‘Touching Butterfly1’ sensing block inside the “If then” and underneath that she puts the ‘Say Come Back’ block] |
| 5    | Facilitator | Can you show how that works? [runs the code that she just wrote which gets the dog sprite to move towards the butterfly and say “Come back” if touching the butterfly] Okay. That's great! And if you click on the butterfly, is there anything that you need to do there? |
| 6    | Safiya   | It says that it [butterfly’s code] doesn't need to be changed, but there is a bonus. |
| 7    | Facilitator | Okay. I want to record the dog's screen one more time, because I think you have got it. What does the task say here? |
| 8    | Safiya   | When you press the space key, get her to “Go to the butterfly” and “Say come back” and if she doesn't touch the butterfly she's supposed to “Switch to a
different costume”.

9 Facilitator Ok. But in your code right now you don’t have the second condition. Think about how to get the second condition in your code. I will share the solution I have but try to think of a way. Ok, now talk to me about the block on the right [barking code for the dog]. What is happening there?

10 Safiya Since it doesn't have an event block, nothing is happening.

11 Facilitator Okay what does the task up there say? What does it ask you to do?

12 Safiya When the space key is pressed, you're supposed to make her bark.

[Sofia adds a “When Space key pressed” event block on top. The dog is now barking and then walking towards the butterfly.]

13 Facilitator Yep! Ok there is just one more thing that is missing. So for dog's running code, If it touches the butterfly then it should say “Come back” which you have done. But the other condition was that if the dog is not touching the butterfly, then the dog should switch to “Lulu c” costume which is like a surprised look. So if you see in the solution I created, if you used a “If else” in place of If then, you could add this ['Switch costume to Lulu c’ look block] below Else to satisfy the second condition and complete all the objectives. But you did a great job just now by following hints to add in the If then conditional here.

Responses during online interview

14 Facilitator How did you go about solving these problems? What were some of the solutions that worked and some that didn’t work?

15 Safiya I got confused on what the task was and why certain blocks didn't go together. I tried different blocks, but it didn't work to just add the blocks while ignoring the hints and goals. But I got help with it and understood how to do it. Some solutions that worked were looking at the code and the problem and figuring out what blocks were missing instead of just trying different blocks. For example, the dog code didn't have a “When Space key” button and something that would make it do something if it touched a butterfly. Using if then statement I made the dog move to the butterfly and the butterfly move away and if it was touching butterfly to say, “come back.”

Excerpt 4 shows the impact of Consolidation and Knowledge Assembly Phase II of the PF design in improving students' solution generation through changes in their debugging process (See Table 4-6).

Safiya shared a solution that she wrote for Debug 8 during the Consolidation and Knowledge Assembly phase and worked on another while I (facilitator) was providing feedback on her solutions. In Debug 8, students were asked to a) get Lulu (dog sprite) to run towards the butterfly, when ‘Space’ is pressed; b)
get Lulu to say, ‘Come back!’ if she touched the butterfly or else get Lulu to switch to a surprised look costume; c) get Lulu to bark when ‘Space’ is pressed; d) bonus: get the butterfly sprite to change colors and fly around the field (See Figure 4-18, left). The given code for Lulu included two code sequences that controlled the sprite’s movements and barking. The given movement code for Lulu set the sprite’s initial appearance with ‘Switch costume to Lulu a’ look block, the initial position on screen with ‘Go to X: -82 Y: -17’ and a ‘Glide’ block that steadily moved Lulu to the butterfly sprite (See Figure 4-18, middle). However, this code was missing the event block to initialize the code when the space key was pressed. This movement code was also missing conditional scripts to get Lulu to say, ‘Come back!’ if she touched the butterfly or else get Lulu to switch to a surprised look costume. The given barking code for Lulu was also missing the ‘When space key pressed’ event block that initialized the code to get Lulu to bark (start sound dog1) repeatedly and change costumes every 0.2 seconds to animate her trotting across the field (See Figure 4-18, right).

Figure 4-18: Debug 8 stage with butterfly and Lulu the dog sprite in a soccer field backdrop (left); Given movement code for Lulu with missing code initialization and conditional scripts (middle); Given barking code with missing code initialization (right)

In her first debugging attempt, Safiya added the missing event ‘When space key pressed’ to Lulu’s movement code completing the first task for Debug 8. To complete the second task, she tried to place a sensing Boolean block ‘Touching Butterfly1’ inside the forever loop and added the look block ‘Say Come back!’ outside the code sequence (See Figure 4-19). Without a container block such as ‘If-
then,’ or ‘If-then-else’ to check if Lulu touched the butterfly sprite, the sensing block did not run. Additionally, the dead code ‘Say Come back!’ was not connected to Lulu’s movement sequence and therefore was not executed. It was evident from her code that Safiya did not know that sensing Boolean blocks needed to be nested inside a container block. Safiya’s interactions with the facilitator during the consolidation phase in Excerpt 4 and her responses during the online interview highlighted her novice programming expertise and her continued trial and error approach to debugging. During the online interview, she reflected that she was “confused on what the task was and why certain blocks didn’t go together” (See Table 4-6, Turn 15) Without a conceptual understanding of Boolean and conditional container blocks, Safiya “tried different blocks” to resolve the issue with code execution but she found that it “didn’t work to just add the blocks while ignoring the hints and goals” provided for Debug 8 (See Table 4-6, Turn 15).

Figure 4-19: Safiya’s iteration 1 solution for Debug 8 with ‘Say Come back’ dead code and ‘touching Butterfly1’ sensing Boolean without a container block

For students without prior experience in coding and Scratch, minor problems such as block placement, choice of relevant code blocks and code sequencing were recurring issues that hindered optimal solution generation. Additionally, absence of scaffolding during the Solution Generation phase, meant that novice programmers such as Safiya continued to rely on trial-and-error debugging. However, facilitator feedback on student solutions during consolidation and knowledge assembly phase improved novice programmers’ solution generation and optimization. Targeted feedback highlighting knowledge
gaps, error patterns, and paths to code optimization led to error mitigation and improvements in conceptual understanding and application of computational concepts. Similarly, feedback on inefficient debugging strategies resulted in some students changing their debugging strategies.

In Safiya’s case, facilitator feedback led to Debug 8 code optimization and a switch in debugging strategy from trial and error to forward reasoning (Katz and Anderson, 2009). Once, the facilitator (author) explained that sensing Boolean block such as “Touching [touching Butterfly1] block can’t go inside the forever loop because [of] the shape of that block” and that it can only be used with container blocks such as ‘If then,’ or ‘If else,’ Safiya optimized her Debug 8 solution (See Table 4-6, Turn 2, 4). She added a conditional container block ‘If then’ inside the forever loop pairing it with the Sensing Boolean ‘touching Butterfly1’ and the previously dead code ‘say Come back’ (See Figure 4-20, left). This code got Lulu to say ‘Come back’ if it touched the butterfly (See Figure 4-20, right). However, the new code did not resolve the second condition: get Lulu to say, ‘Come back!’ if she touched the butterfly or

**else get Lulu to switch to a surprised look costume.** Safiya could have fully completed the objective if she had added an ‘If-then-Else’ conditional container and put in ‘switch costume to Lulu C’ under the ‘Else.’ While Safiya did not completely resolve Debug 8, the feedback on code fit and the hint on conditional container blocks allowed her to optimize her code (See Figure 4-13, Iteration 2)

![Figure 4-20: Safiya’s optimized Debug 8 solution with ‘touching Butterfly1’ sensing Boolean and ‘say Come back’ look block inside an ‘If-then’ conditional container block (left); initialized code shows execution of the condition (right).]
Additionally, facilitator feedback also contributed to Safiya changing her debugging strategy to improve solution generation. She had been using trial and error “trying out different blocks” but switched to forward reasoning serial order “looking at the code and the problem and figuring out what blocks were missing...” (See Table 4-6, Turn 15). Forward reasoning is an error location strategy where programmers search for bugs by reviewing the existing code line by line (forward reasoning serial order) or by simulating the program’s execution (forward reasoning program order). Following facilitator's feedback on conditional container blocks and prompts to examine the task first (See Table 4-6, Turn 5, 9), Safiya read through the task and initialized her iteration 1 code. She examined the output, observing the actions that the code performed and did not perform from the Debug 8 objective list. Safiya then read and reviewed the entire code in serial order to understand the code and determine the missing codes and sprite actions. She determined that “the dog code didn’t have a ‘when space key’ button [block] and something [conditional] that would make it do something if it touched a butterfly” (See Table 4-6, Turn 15). Safiya added the missing ‘when space pressed’ event block and the conditional ‘If touching Butterfly1 say Come back’ to optimize the code in her second attempt (See Figure 4-20, left).

This change in debugging strategy continued in subsequent debugging problems. Safiya used forward reasoning serial order in Debug 9 and 10 completing set objectives and debugging the problems in her first attempt. Additionally, she created 2 Lilo and Stitch quiz games for Debug 13 and 14 using a combination of forward reasoning and incremental development strategy where she consistently read through her code in serial order, tested her code, examined the output, and optimized the code to incrementally scale up her game (See Excerpt 12). Her solutions showed improved understanding and accurate application of computational concepts such as conditionals, sequences, and advanced concepts such as variables and operators. For instance, in Debug 10 the given tasks were to add a score meter, change the score by +1 if Koko the monkey sprite eats a banana, and change the score by -1 if Koko eats an orange. To complete these tasks, Safiya first created the score variable and set it 0 at the beginning of the game (See Figure 4-21, left). She also wrote a conditional sequence whereby the score would go up by
+1 if the Koko touched any of the 10 orange sprites (See Figure 4-21, middle) and the score would go up by -1 if the sprite touched any of the 12 banana sprites.

Figure 4-21: Safiya’s optimized code Debug 10 for Koko with score variable created and set to 0 (left); banana sprite code with a conditional sequence to increase the score by +1 if Koko touched a banana (middle); orange sprite code with a conditional sequence to increase the score by +1 if Koko touched an orange (right)

Overall, facilitator’s feedback on student's’ solutions during the Consolidation and Knowledge Assembly phase became helpful and for some students critical in improving their subsequent solution generation. Second, students changed their debugging strategies from mostly unsuccessful and time-consuming Trial and Error to purposeful error locating strategies such as forward and backward reasoning and objective focused strategies like examining the output, backtracking, incremental development and testing, and bug clustering. Changes in debugging strategies also contributed to mitigate recurring errors and streamline their solution generation. Third, discussions on areas of conceptual understanding and knowledge gaps supported improved understanding and application of computational concepts in their codes.
Qualitative Findings RQ2.1

In this next section, I report on findings for RQ 2.1: *What kinds of challenges do students encounter, the resolutions they come up with and the debugging strategies they used when working through the debugging curriculum?* To answer RQ 2.1 I used interaction analysis and thematic analysis to unpack the nature and types of challenges that students faced as they worked through debugging problems, the solutions that they came up with and the debugging strategies that students used in their solutions. I selected, transcribed, categorized, and analyzed instances from Think Alouds (audio), video data and online interviews during the solution-generation and knowledge assembly phase where participants

a) listed and described the various problems that they faced when debugging
b) explained their solutions, the coding blocks, and computational concepts they applied and the reasoning behind their solutions
c) shared and explained their debugging process and strategies

Artifact analysis was also conducted to document and analyze students’ iterative solutions and moments of active debugging: code block selections, code swaps/edits, solution-to-solution debugging decisions. Artifact analysis provided additional insights into the specific problems and error patterns that students faced, the debugging strategies they utilized to generate solutions and changes in these strategies across generated solutions.

I coded and analyzed prominent moments of verbal and embodied interactions focusing on difficulties and recurring error patterns faced by the students, their iterative solutions and specific strategies students used to generate optimal solutions. Initial codes and axial codes were developed using guidelines from Saldana (2021) and informed by Litts et al’s (2016) four stage coding process focused on identifying instances of debugging challenges, types of challenges and strategies used to resolve those problems. Guidelines provided by Braun, Clarke, and Hayfield (2015) and Saldana (2021) were used to generate, organize, and analyze emerging themes and selecting themes that could answer RQ 2.1. I
identified the following patterns that summarizes the types of challenges, resolutions and debugging strategies that students used when working through the debugging problems.

Challenges-High Frequency of Syntax Errors and General Bugs

Syntax errors and general bugs were two of the recurring challenges that students faced when working through Debug 1-14. I used Frädrich et al’s (2020) catalog of error patterns in Scratch as a framework to identify and code recurring syntax errors and general bugs in all student generated solutions from Debug 1-14. However, I omitted error patterns focused on custom blocks, clones and advanced operators and parameters that were not within the scope of this study. Table 4-5 shows the frequency of these recurring error patterns.

Syntax Errors

The most commonly occurring syntax errors included termination conditions and missing or incomplete conditionals. Students’ lack of prior experience with Scratch coding and absence of on-time feedback and instructions on specific code block applications during solution generation phase also contributed to recurring occurrence of syntax errors. For instance, novice programmers like Safiya tried to snap blocks that did not fit together and had different shapes than the container blocks See Figure 4-19. These errors may not return an error message or hinder program from running but can result in bugs and sup-optimal codes that do not work as intended. Excerpt 5 (Table 4-7) shows an instance of students’ difficulty with syntax errors, specifically with missing/incomplete conditionals.

Table 4-7: Excerpt 5, Damari explaining the game that she generated for Debug 14 and her responses on the challenges during the online interview.

<table>
<thead>
<tr>
<th>Turn</th>
<th>Speaker</th>
<th>Verbal and Non-verbal transcript</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Facilitator</td>
<td>Tell me about the game that you created. What was the game about?</td>
</tr>
<tr>
<td>2</td>
<td>Damari</td>
<td>I created a similar version of a game that I used to play when I was younger.</td>
</tr>
</tbody>
</table>
The game is basically a dinosaur eating fruits all around the place.

Facilitator: Tell me about the codes that you wrote for your game? What did each code do?

Damari: The dinosaur's code has when the green flag is pressed, the dinosaur should go to a random place and eats the fruit. The watermelon's code is when the green flag is pressed the watermelon should go to a random place, but if it touches the dinosaur then the score should go up by 1. I wrote the same code orange as well.

Facilitator: What were some of the problems/challenges you ran into?

Damari: I wanted to change levels when the orange score was 10 and watermelon score was 15. I added a if-then and then equal operator block [ _ = _] and next backdrop. But I didn't know how to say If orange score equals 10 then change backdrop. Same for watermelon. I couldn't figure that out. Another one [challenge] was I wanted the score to go up but whenever I pressed the green flag the variables were not going up at all, and it was because I did use the if/then statement to make the variable go up by 1.

Absence of termination conditions and missing or incomplete conditionals were evident in many student generated solutions. Scripts requiring an action to be performed under a given condition were not completed due to missing conditionals such as If-then, If-then-Else and Repeat Until. Also, conditional container blocks which specified the condition for sprite actions were either incorrect or insufficiently defined. For instance, in Debug 14, Damari created an adventure game called “Dino Crave.” For Debug 14, the task was to create an adventure, strategy, puzzle, sports, or action game with at least 2 sprites/characters. Damari’s Dino Crave game was a tribute to a similar game that she used to play in her younger days (See Table 4-7, Turn 2). The objective of the game was for the dinosaur to eat oranges and watermelons each time it came into contact with them. The players earned points for each fruit consumed, with the goal of accumulating enough points to progress to the next level (See Table 4-7, Turn 4).

For the dinosaur code, she showcased her understanding of parallelisms with 3 parallelly running codes initialized by the event hat ‘When green flag pressed.’ The initial script launched the game with the sprite uttering the phrase ‘Let’s play a game.’ The use of a forever loop with a glide block caused the sprite to continuously glide for 0.5 seconds to random positions on screen (See Figure 4-22, left). Damari also created two score variables ‘Orange’ and ‘Watermelon’ set to 0 at the beginning of the game. The
variables kept track of the points earned by players by eating the fruits. To increase the score, Damari wrote two conditional scripts utilizing the ‘If-then’ conditional and sensing blocks which were nested inside a forever loop. When executed, the score would go up by 1 if the dinosaur touched the oranges or watermelons (See Figure 4-22, middle, right).

Figure 4-22: Damari’s setup and movement codes for dino sprite for her Debug 14 game (left); code with a looped conditional sequence to increase the score by +1 if the dino touched an orange (middle); code with a looped conditional sequence to increase the score by +1 if dino touched a watermelon (right)

Furthermore, Damari incorporated two parallel conditional scripts that were designed to change the game background, signaling a new level, if 10 oranges were eaten or if 15 watermelons were consumed (See Table 4-7, Turn 6). However, Damari “didn't know how to say ‘If orange score equals 10 then change backdrop. Same for watermelon. I[she] couldn't figure that out” (See Table 4-7, Turn 6). As a result, both scripts were missing termination conditions (See Figure 4-23, left and middle). The absence of the ‘Orange’ reporter variable being added to the conditional operator ‘If __ = 10’ and ‘Watermelon’ reporter variable being added to ‘If __ = 15’ meant that the conditions for advancing to the next level were not fulfilled. As a result, the code was initialized, but the ‘next backdrop’ command was not executed, even though the conditions for the new level had been met in the game.
Although Damari’s codes successfully enabled the movement of the dino sprite and kept track of score changes, it was lacking in important player interaction features. The lack of termination conditions prevented players from advancing to the next levels, and the use of glide blocks limited their control over the sprite’s movements, resulting in less immersive gaming experience. To optimize her code, Damari could have implemented the ‘Orange’ and ‘Watermelon’ reporter blocks and added a ‘Stop this script’ control block to ensure the scripts stopped running when the termination condition (such as eating a required number of fruits) was met (See Figure 4-23, right). She could also create scripts for gameplay and conditions in the next levels and allow players to control the sprite’s movements through directional inputs like arrow keys or joystick control. This would have enhanced the player experience and made the game more interactive and challenging.

Absence of termination conditions was a common issue among student solutions, revealing their limited understanding of conditionals. The frequency of termination conditions remained constant at 42.36% during the study with a slight drop in Debug 4-6 (See Table 4-5). It was common to see Instances of ‘Repeat until__,’ ‘If-then’ ‘Wait until__’ blocks left empty without termination conditions. In programming, termination conditions play a crucial role in controlling the flow of a program by signaling when a function should return or when a loop should stop. Nevertheless, for those new to programming,
understanding the concept of termination conditions and their significance is difficult. I observed that novice programmers like Damari often overlooked the importance of termination conditions, resulting in frequent errors in their code. They faced challenges in specifying conditions through conditional statements such as ‘If-then’ or ‘While’ loops, which evaluate a program’s outcome and determine when to end the program or loop based on the evaluation outcome. The absence of termination conditions in student solutions led to loops running indefinitely, causing the program to crash or hang and functions to not return the desired sprite actions or output.

**General Bugs**

General bugs were another set of recurring error patterns that were frequent in student generated solutions (See Key terms, p. 7 and Table 2-1 for definition and examples). Students faced a number of these general bugs while working through the debugging curriculum. While there was a reduction in occurrence rates for some of the general bugs as students progressed through the problems, the error patterns persisted over time (See Table 4-5). Common and most recurring bugs included missing or uninitialized loops, events, conditionals, and variables; interrupted loop sensing; and no working scripts/incomplete codes. While many of these bugs, i.e., missing, or uninitialized loops, events, conditionals, variables, and parallelisms, were built in default or provided codes for the debugging problems, these bugs consistently appeared in student generated codes for the problems. These errors also occurred in Debug 13 and 14 where the task was to create personalized games without any support codes provided to get students started. Occurrence of general bugs showed persistent issues with code application and challenges in understanding computational concepts to accurately apply it to debug problems at hand. Excerpt 6 (See Table 4-8) shows one of the instances of students’ challenges with general bugs and the resolutions that they came up with.

Table 4-8: Excerpt 6, Willow explaining the challenges she faced when working on Debug 10 during a Think Aloud session.
<table>
<thead>
<tr>
<th>Turn</th>
<th>Speaker</th>
<th>Verbal and Non-verbal transcript</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Facilitator</td>
<td>Can you tell me about what you are working on right now?</td>
</tr>
<tr>
<td>2</td>
<td>Willow</td>
<td>All right. Okay. I'm still working on this one [clicks “see inside”] and I'm having problems. I've been trying different things, so it's kind of a mess.</td>
</tr>
<tr>
<td>3</td>
<td>Facilitator</td>
<td>That’s ok. What are some of the problems you are facing?</td>
</tr>
<tr>
<td>4</td>
<td>Willow</td>
<td>So, so far what it does is every time it touches this specific orange right here [points to the orange on bottom left corner of the screen, next to Koko], the one next to the monkey, right now, it changes by negative two. So what I'm trying to do is I'm trying to get it to do that every time it touches all the oranges, not just that one.</td>
</tr>
<tr>
<td>5</td>
<td>Facilitator</td>
<td>Okay. And how are you planning to do that?</td>
</tr>
<tr>
<td>6</td>
<td>Willow</td>
<td>Well, either I have to code every single orange that is on here, which I think I'm going to have to do like you see how there's like all these oranges [pointing to code for Koko] and I have to change the score by negative 2 for all of them. Or maybe there's like a single block to change the score by -2 for all the oranges at once, but I'm not really seeing anything like that.</td>
</tr>
<tr>
<td>7</td>
<td>Facilitator</td>
<td>Mm-hmm. Maybe there is a way. Think operators. Also, think about setting the score meter to 0 when a player starts a game.</td>
</tr>
</tbody>
</table>

Willow attempted to complete the objectives of Debug 10, a simple 2D platformer single-player game requiring the addition of a scoring system and a win state (See Figure 4-24, left). The objective for Debug 10 were to a) introduce a score meter initialized at 0 at the beginning of the game; b) increase the score by +1 whenever Koko eats a banana, decrease the score by -1 whenever Koko eats an orange or touches the red bricks; and c) implement a bonus feature when the background changes to display the message "Congratulations!!!! You Won!!!" when the player’s total score reaches 10 points (See Figure 4-24, middle-right).
Willow was unable to complete the objectives of Debug 10 due to numerous general bugs in her solution, including missing conditionals, uninitialized events, and incomplete code. For instance, in order to initialize the score meter to 0 at the start of the game (Objective a), Willow created a “Score” variable but failed to properly initialize it with an Event block (See Figure 4-25, left). Upon inspecting the parallelly running codes, all initialized with the “When Green flag pressed” block, it appears that Willow intended to use this same event block to initialize the score to 0. However, the inclusion of an infinite loop in the code would have caused it to run indefinitely, resulting in the score meter perpetually set to 0. This, in turn, would have hindered the execution of the codes for Objectives b, c and d, which would update the scores by +1 or -1 based on Koko’s interactions with other game sprites. Therefore, further debugging and modifications were required to correct the issue and ensure that the score meter initialization worked as intended, enabling successful execution of codes for subsequent game objectives. To ensure the score meter started at 0 and was reset when the green flag was pressed, an efficient solution would have been to remove the forever block and move the ‘Set score to 0’ variable block within the code section that set the backdrop and Koko’s position (See Figure 4-25, right).
Willow’s Koko codes had uninitialized events, incomplete score-setting and redundant conditional code for score changes (left); Solution for uninitialized events and incomplete score-setting through relocating the ‘Set score to 0’ block to backdrop and position setting code section (right).

Additionally, Willow’s Koko codes contained redundant conditional code for scoring changes, which led to duplicate score changes when Koko interacted with the ‘Orange1’ sprite (See Figure 4-25, left). This redundancy occurred because of the pre-existing code for the ‘Orange1’ sprite, which decreased the score by -2 whenever Koko interacted with it (See Figure 4-26, left). To optimize the scoring system, it was necessary to write and replicate the conditional code with sensing Boolean ‘If touching Koko, change score by -2’ for all 10 orange sprites, while removing the redundant conditional code for scoring changes for Koko. Moreover, the Scratch code written by Willow also contained several instances of missing conditional statements, resulting in incomplete scoring changes for certain sprites. Willow added the ‘Wait until touching Koko’ conditional to hide the banana and red panel sprites, along with 9 out of 10 orange sprites (See Figure 4-26, middle). However, the corresponding code for score change necessary to complete the conditional statements was missing, such as the ‘change score by +2’ statement for banana sprites and ‘change score by -2’ block for red panels and orange sprites (See Figure 4-26, right). Willow did include the score change conditional code for ‘Orange1’, with a sensing Boolean block to modify the score by -2 when Koko consumed that specific orange sprite. However, this code was not replicated for the oranges and red panels, resulting in incomplete scoring changes for these sprites, leading to the partial fulfillment of objectives b and c.
During the study, Willow attempted to find a more efficient way to change the score by -2 for all orange sprites whenever Koko touched any of them. She explored the possibility of adding a single block to accomplish this but was unable to locate such a block (Table 4-8, Turn 4-6). Despite receiving a hint about the use of operators, Willow’s code remained unchanged (Table 4-8, Turn 7). However, Willow was correct in identifying that there existed an optimal approach to changing scores for all orange sprites whenever any of them interacted with Koko. This solution required the use of custom blocks and a higher level of abstraction, which was beyond the scope of procedural programming covered in this study. To implement this approach, the ‘My blocks’ feature in Scratch could have been utilized to create an algorithm and define the procedure for changing scores for orange sprites whenever Koko interacted with them. For example, a custom block named ‘Touching Orange’ could be created to define the procedure for score changes for all oranges using a series of conditional statements with ‘_ or _’ operators and sensing Booleans (See Figure 4-27). When Koko touched either Orange1 or Orange2 sprite, the scores would change by -2. The same conditional statements could be duplicated for all 10 orange sprites along with red panels to update the scoring system.
Novice programmers often encounter general bugs in their code due to their limited experience in programming. Lack of sufficient knowledge of debugging techniques and application skills of syntax logic, algorithm design and programming concepts such as conditionals, control flow and data types contribute to the occurrence of bugs in their code (Kelleher & Pausch, 2005; Resnick et al, 2009). Willow’s code was not immune to this phenomenon as it contained several instances of general bugs such as missing conditional statements for score changes for red panels and banana sprites and incomplete code for score changes for orange sprites, which could have been optimized using custom blocks. These issues highlighted the challenges that novice programmers faced in writing efficient and bug-free code during the solution generation phase, and the importance of providing them with appropriate guidance and support to improve their programming skills.

Challenges-Recurring Smelly Codes

In addition to syntax errors and general bugs, student-generated solutions in this study demonstrated frequent code smells. Frädrich et al’s (2020) catalog of error patterns in Scratch was used as a framework to identify these recurring issues, with patterns related to custom blocks, clones and
advanced operators excluded as they were outside the scope of this study. Frequently occurring code smells included sequential actions, misordered code blocks, long scripts, redundant/duplicate codes, timing and synchronization errors, unnecessary conditionals and loops and dead codes (See Table 4-5). The high frequency of these code smells indicated a lack of understanding and application of fundamental code design principles and computational concepts. It also pointed to use of undisciplined or bad coding practices and debugging strategies, such as bottom-up programming, where students randomly snapped blocks together without a clear plan or understanding of how they fit into the larger system. Although some students transitioned to a top-down approach, which involved problem decomposition, clustering bugs, and using a combination of forward and backward reasoning, many continued to struggle with approaching and thinking about problems from an algorithmic, design and systems perspective, as highlighted in Excerpt 7 (see Table 4-8).

Table 4-9: Excerpt 7, Nina explaining the challenges she faced when working on Debug 14 during a Think Aloud session.

<table>
<thead>
<tr>
<th>Turn</th>
<th>Speaker</th>
<th>Verbal and Non-verbal transcript</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Facilitator</td>
<td>Okay. Great. So, talk to me about the problems and challenges for this game and how did you debug it?</td>
</tr>
<tr>
<td>2</td>
<td>Nina</td>
<td>So, when I was trying to move Ben, I couldn't. When I was trying to make him go up, I put the move 12 steps in the up arrow, but every time I would click the up arrow, he was only moving to the right. He wasn't moving up. He wasn't going where I wanted him to go. I tried different blocks to see which one worked. And then I started looking through the motion blocks and then I remembered X and Y [coordinates], and I figured that that's what it needed in order for it to move up and down.</td>
</tr>
<tr>
<td>3</td>
<td>Facilitator</td>
<td>Great job! What other problems and challenges did you debug?</td>
</tr>
<tr>
<td>4</td>
<td>Nina</td>
<td>When I was making the storyline I had a problem, because they (Ben and the space bugs) would both speak at the same time and I wanted Ben to wait for the beetles to speak first, and that's how I added this, the “think... for three seconds” [points to the code initialized when Green flag is pressed with a ‘Think… for 3 secs’ look block]</td>
</tr>
<tr>
<td>5</td>
<td>Facilitator</td>
<td>Oh. So, the “think dot, dot, dot” is just Ben passing time?</td>
</tr>
</tbody>
</table>
Yeah. Because I could have also put the “Wait” block instead of the whole “Think” thing [block], but I just found it more useful, because the beetles are speaking in that moment and I just thought, "Well, those three dots could represent Ben processing everything."

Now you're thinking how a game designer would think. Micro actions. Very, very micro things that we normally do not even think about, it just happens automatically in games. It's awesome that you are thinking about these. Anything else?

Well, I was having the problem with the fruits not showing, but you helped me solve that. Now, this, right here [points to the code which changed the score by -1 if Ben touched a fruit] was a code that I tried, but failed. So, I'm not sure if I want to keep it there or not, but the whole point was for when the beetles touch Ben, for the score to go down by one, but it's not necessarily working.

Mm-hmm. Okay. We can talk about that.

Debug 14 required participants to create an adventure, strategy, puzzle, sports, or action game featuring a minimum of two sprites. Nina developed an interactive game called ‘Space Fruit’ where space bugs attempted to devour all the space fruits, and Ben had to collect them before the bugs caught him. The game featured directional controls and a score meter that increased by one whenever Ben collected a fruit but decreased by one if the bugs caught him (See Figure 4-28). She employed a bottom-up programming approach, dragging and snapping different blocks together to add game mechanics and sprite functionality, without intentionally designing and writing scripts to coordinate sequence and timing of sprite actions during gameplay (See Table 4-9, Turn 2, 6).
Despite the game’s functionality, numerous code smells persisted in the game, impacting the execution and gameplay as intended by Nina. Specifically, redundant codes, problems with timing and synchronization, and overuse of parallelisms were evident in Nina’s generated codes for Debug 14. Nina did resolve the timing and synchronization issue of overlapping dialogues between Ben and the beetles by experimenting with various wait times to synchronize their dialogues (See Table 4-9, Turn 4, 6). She updated the beetle’s narrative code by adding the command ‘Say We are the Bad Space Bugs!! For 3 seconds,’ as well as a Think… for 3 seconds’ look block to Ben’s code (See Figure 4-29). Both of these updates included an embedded wait time that could be adjusted as needed. By doing so, she made Ben wait for the beetles to speak first, indicating that he was processing information (See Table 4-9, Turn 6). The solution successfully resolved the timing and synchronization issue and in doing so, Nina demonstrated her game designer mindset by considering micro-actions, such as character processing information, in her coding.
Figure 4-29: Script architecture for Ben and Beetles interaction. Ben’s code with directional controls, and redundant conditional codes for score changes (left); Beetles’ code with movement to track Ben, and a conditional code that reduced the score by -1 if touching Ben (right)

However, other code smells persisted in the game, hindering the smooth execution of the Space Fruit gameplay as envisioned by Nina. One significant issue was the overuse of parallelisms, where 15 out of 19 scripts for all 9 sprites were initialized concurrently when the green flag was pressed. While concurrency is a crucial aspect of Scratch programming and enables the creation of complex and interactive gameplay, it can also present challenges for novice programmers, leading to unexpected and unwanted results (Meerbaum-Salant, Armoni, & Ari, 2011, p.171). In Nina’s Space Fruit game, the lack of proper synchronization of parallelly running scripts resulted in lag time between Ben and the beetles’ interaction. Lack of synchronization also caused delayed movements of the bugs and prevented the score meter from updating when the bugs touched Ben.

In addition to issues with parallelisms, Nina’s game also contained redundant code, resulting in inconsistencies in code execution and affecting gameplay. Specifically, Nina added score change scripts to both the beetle’s code and to Ben’s code, as well as 3 “Set score to 0” script to Ben’s code (See Figure 4-29, left). These redundancies caused inconsistent score to change and even prevented score change scripts from executing as the game progressed (See Table 4-9, Turn 8).
Nina’s solution for Debug 14 demonstrated her creativity and programming expertise with her effective use of events, variables, loops, and conditionals in her sprite scripts. However, her code exhibited numerous execution issues and code smells that adversely affected the game’s performance and user experience. While she was successful in addressing timing and synchronization issues, Nina’s overuse of parallelisms and code redundancy resulted in inconsistencies in score changes and delays in sprite actions, leading to a suboptimal user experience. This was due to her bottom-up programming approach and limited understanding of concurrency. While she pierced together various blocks to develop scripts for the sprites and add new functionalities, she did not consciously design and write scripts with scalability and integration in mind, leading to a lack of proper structure and organization, which further resulted in code smells. Furthermore, her bottom-up programming approach made it challenging to debug her code. Integration difficulties arose as separate pieces of scripts for Ben and beetles did not seamlessly fit into the larger system of the Space Fruit gameplay, leading to bugs and disjointed interactions between sprites. Debugging individual scripts in isolation also made it difficult to see the big picture and understand how each component fit into the overall system, leading to challenges in identifying the root cause of emergent bugs. Finally, high-level issues such as inconsistencies in score changes, delays in sprite actions, and timing and synchronization issues only became apparent to Nina once the entire system was integrated and tested as a whole, leading to a delayed discovery and the need for repeated debugging.

Nina’s case also emphasized the need for scaffolding for novice programmers to transition to and adopt top-down programming approach. While some students such as Tuan, Delani, and Willow transitioned to top-down approach, Nina, Safiya and Damari continued with a bottom-up approach leading to persistent issues with code smells and emergent issues while debugging. A top-down approach to writing programs and debugging would have ensured proper planning and organization while writing programs and minimize code smells and execution issues, resulting in optimal debugging. Additionally, top-down approach would have helped identify and address high-level issues such inconsistencies in score changes, delays in sprite actions, and timing and synchronization issues, before they became more
significant and challenging to fix. I discuss the implications of these findings in detail in Chapter 5-Discussions.

Challenges-Affective Challenges

Students often experienced affective challenges such as confusion about tasks and frustration when working on debugging problems. The challenges were particularly prevalent when students faced unresolved impasses during the Solution Generation phase. Following error returns and failed debugging attempts, students reported decision paralysis, fear of modifying code, and feelings of low self-efficacy and confidence as frequent challenges. However, some students also reported positive affect such as engagement, fun, and curiosity, seeing failures as opportunities to test unique code block arrangements and learn accurate applications of computational concepts. These findings suggest that affective challenges can support knowledge construction and assimilation, even though they can negatively impact performance after multiple failures. Figure 4-10 provides a collection of moments that highlight the affective challenges faced by three students and reasons behind those challenges.

Table 4-10: Excerpt 8, Willow, Delani, Safiya and Jamarcus explaining the affective challenges they faced when debugging as well as positive affect from these challenges

<table>
<thead>
<tr>
<th>Turn</th>
<th>Speaker</th>
<th>Verbal and Non-verbal transcript</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Facilitator</td>
<td>Okay. Let's talk about the process. You said you tried a lot of trial and error. How did you feel about that?</td>
</tr>
<tr>
<td>2</td>
<td>Willow</td>
<td>It's kind of frustrating because you want it to work. And sometimes it took me a while to figure out this whole “if touching Jordyn” thing. [hovers over conditional code ‘If touching Jordyn, glide 3 secs to X: 223 Y: -45] I was going to give up and just ask for help. I froze for a bit there because I didn’t want to keep on adding things that wouldn’t work. But then eventually I saw the little sensing circle here [Points to the drop-down menu to select a specific sprite to sense], and I'm like, &quot;Wow that’s it!&quot;</td>
</tr>
<tr>
<td>1</td>
<td>Facilitator</td>
<td>How was the process of coming up with different solutions and some of it working and some of it not working? How did you feel about it?</td>
</tr>
<tr>
<td>2</td>
<td>Safiya</td>
<td>It wasn't super bad, but it was still kind of frustrating. When you have to try</td>
</tr>
</tbody>
</table>
different blocks and see what they do and you try a block and you think it works but then it doesn’t do what you want to solve the problem. That was frustrating.

1 Facilitator How did you feel when some of the solutions worked out and some solutions didn’t work out while debugging?

2 Delani When a few of my solutions worked out I felt happy since the debugging problem was a little difficult but when it didn’t I felt confused but I tried finding other ways to fix it.

3 Facilitator How did you feel about the process of coming up with different ideas to solve the problems?

4 Delani I felt like I was using my brain more. Finding solutions to this problem that works out well while fixing the ones that didn’t made me feel more active and a little frustrated.

1 Facilitator How did you get to the right solution? How did you think about it?

2 Jamarcus Well, basically, I did trial and error. Basically, I found out the things that didn't work, so that when I found the glide button [emphasizes Glide 1 secs to random position], it solved my problem, because then I could have him move around without having to turn.

3 Facilitator Right! Talk to me about trial and error. How do you feel about the process?

4 Jamarcus It's a little frustrating, but it's also fun, you know? Cause, so that didn't work, so now it's like solving a puzzle.

5 Facilitator Okay. In trial and error, sometimes things don't work out right? How do you feel when things don't work out?

6 Jamarcus It's a little frustrating, but it's also fun, you know? Sometimes it's kind of disappointing, but I just have to keep trying cause every time I tried a code, I learn something. I knew what it did, where to put it and how I can use that [code block] later. Cause, that didn't work, so now it's like solving a puzzle. Like I said last time, sometimes if I tried a lot of things I might try to step back and analyze it again to see what worked and what didn't. So, I looked around the Events, the Controls and the Sensing blocks and found something that would do what I want.

Excerpt 8 shows the affective challenges that were part of the difficulty experienced by students during the Solution Generation phase. Willow, Safiya, Delani and Jamarcus all reported frustration and confusion when they faced an impasse despite multiple attempts at debugging. Students’ progress was also hindered by decision paralysis and feelings of low confidence and self-efficacy that became
pronounced and at times debilitating after persistent syntax and runtime errors. Willow reported that she “froze for a bit” after an unsuccessful debugging attempt where she was trying to get the ball sprite to sense Jordyn (Table 4-10, Willow, Turn 2). However, after discovering how to select Jordyn’s sprite from the drop-down menu on the Sensing block, she managed to complete the conditional statement such that the ball would glide to the back of the net if it made contact with Jordyn (See Figure 4-17, right). However, Willow expressed that she “didn’t want to keep on adding things that wouldn’t work” and that she “was going to give up and just ask for help.” (Table 4-10, Willow, Turn 2)

The absence of scaffolding and instruction on code construction and conceptual knowledge on computational concepts resulted in all 4 students attempting to debug the problems by “try[ing] different blocks and see what they do” (Table 4-10, Safiya, Turn 2). The trial-and-error debugging strategy contributed to multiple failures, recurring bugs and code smells triggering frustration, confusion, and confidence hits. Although all 4 students reported that the affective challenges, they faced had a negative impact on their performance at times, Delani and Jamarcus also recognized that these challenges, alongside the trial-and-error approach, facilitated code construction, learning and comprehension of computational concepts. Additionally, they noted that the process fostered curiosity, engagement, and fun. For instance, Delani explained that she felt cognitively sharp and engaged while using trial and error to explore alternative methods to correct faulty scripts. She described feeling “more active” and wanting to use her brain more during the process (Table 4-10, Delani, Turn 2, 4). Through trial and error, she familiarized herself with the various code blocks, solidifying her understanding with each unsuccessful attempt to refine her code construction and precise use of computational concepts, particularly conditionals and sequences. Likewise, Jamarcus found the process to be both “disappointing” and “fun,” likening it to solving a puzzle. He noted that beginning with trial and error improved his debugging skills, as repeated failures taught him to “step back and analyze” his code and the problem reflecting on what did and did not work (Table 4-10, Jamarcus, Turn 2, 4). Essentially, Jamarcus utilized error locating strategies during his debugging process. Initially, he employed Simple Mapping, a Backward Reasoning strategy, to examine the program output and locate the bug. However, he did not work backwards but instead used
Forward Reasoning Serial Order to review his code block by block and identify the error. To do this, he analyzed what worked and what did not work, added blocks incrementally to generate and test solutions to the debugging problems. Although Jamarcus still used trial and error, he was more purposeful in selecting and adding blocks to his code (Table 4-10, Jamarcus, Turn 6).

Novice programmers such as Jamarcus, Delani, Willow, and Safiya faced affective challenges such as frustration, confusion, decision paralysis and confidence hits when working on debugging problems. However, these challenges encouraged them to persist in generating solutions, which instilled a sense of curiosity and fun, and improved their code construction and learning of computational concepts.

My findings were consistent with previous research on computer programming education in some ways, but there were also some differences. Similar to D’Mello (2013) and Bosch and D’Mello (2017), I observed that students often faced affective challenges such as confusion and frustration, particularly when dealing with unresolved impasses despite multiple debugging attempts. However, unlike previous research, students did not report feeling bored while working through the debugging problems. Instead, they experienced decision paralysis, fear of modifying and remixing scripts, low self-efficacy and confidence hit following error returns and failed debugging attempts during the Solution Generation phase. Furthermore, some students had a positive attitude towards unsuccessful debugging attempts, seeing these failures as opportunities to experiment with unique code block arrangements, and learn about computational concepts such as variables, conditionals, and operators. This finding is consistent with VanLehn’s (1988) impasse-driven learning theory of Bosh and D’Mello’s (2017) study, showing that while affective challenges may hinder performance and become more pronounced after multiple failures, they can also facilitate discovery, knowledge construction, and learning.

Resolutions-Problem Resolutions Through Individual Attempts and Facilitator Feedback

Students generated optimal solutions to most of the debugging problems on their own through multiple iterations during the Solution Generation phase. However, some debugging problems required facilitator feedback and prompts. Figure 4-30 shows the number of debugging problems that students
solved individually, through facilitator feedback/prompts, problems that were not solved and problems that were not attempted. These numbers come from students’ responses during Think Alouds and online interview where students indicated the problems that they solved in each session and how they debugged those problems.

Figure 4-30: Showing number of debugging problems solved individually, thorough facilitator feedback, problems that were not debugged and unattempted problems

On average students solved 6 problems out of 14 (43%) individually without facilitator prompts during the Solution Generation phase. Out of a total of 14 debugging problems, Jonah (7), Nakiya (10), Safiya (9) and Tuan (11) solved most of the problems without scaffolding and external support. Among the four high performing students, Jonah was the only one with prior coding experience having participated in Maker workshops and coding club in school. Additionally, on average students completed 2 problems (14%) with facilitator’s feedback/prompts and did not debug 2 problems (14%) despite multiple attempts.
However, a glaring finding of students' debugging was that on average students did not attempt 5 problems (36%). Nevertheless, this high number of unattempted problems were not an indication of students’ inability, motivation or engagement but were reflective of logistical, technical problems and Covid-19 related realities (discussed in detail in discussion and limitations section). For instance, Damari, Jonah, and Jamarcus were rising seniors who had work outside of school. These students were invested in the workshop and at times were logging in from their work parking lot on their phone. However, they were not able to engage in the hands-on debugging problems compared to students who could switch between Zoom and Scratch to work on the problems and participate in Think Alouds. Similarly, Willow, Jonah and Delani were on childcare duty supporting their working parents. This added another level of complexity to being fully engaged in the fully virtual workshop activities and contributed to non-attempts for some debugging problems. In addition, the time spent on trial and error, which despite students’ successes in getting to the optimal solution, also contributed to non-attempts for some debugging problems.

As novice programmers, students had a high rate of success in completing the debugging problems individually during the Solution Generation phase with a 43% completion or 6/14 problems debugged. Number of factors contributed to this higher debugging success rate, some of which has precedence in prior literature. Most prominently, Scratch’s design has been shown to be novice friendly and have a shorter learning curve (Resnick et al., 2009). Purposeful design features such as visual programming interface, drag and drop coding, ability to design and remix games and immediate feedback on output have been shown to lower the barrier of entry to programming (Malan & Leitner, 2007; Weintrop & Wilensky, 2015) and ease cognitive load for novice programmers (Shin & Park, 2014). As a result, Scratch is effective in introducing novice learners to fundamental computational concepts, aiding learning of these basic concepts through hands-on application and supporting problem solving from the get go (Meerbaum-Salant, Armoni & Ben-Ari, 2010; Baytak & Land, 2011; Lewis, 2010). Additionally, Scratch’s success is well established in generating positive affective experience such as enjoyability, fun and engagement that supported novice programmers’ persistence in iterating and debugging which in turn
assisted their learning gains in programming (Malan & Leitner, 2007; Wilson & Moffat, 2010; Grover & Basu, 2017).

Similar to findings from prior research, Scratch’s accessible and novice friendly design supported students debugging whereby they could initially drag, drop, and plug different code blocks to explore functionalities. While this process of Trial and Error was time consuming, “frustrating” and “confusing” at times, it was also successful. Some students like Jamarcus, Delani and Willow also reported the process as “fun,” “enjoyable” and “like solving a puzzle” (See Table 4-10). Students could tinker/experiment with different blocks as well as try out various sequencing combinations and find out instantly through Scratch’s built-in feedback feature whether they completed the debugging objectives. In the process students figured out the functionality and application of different code blocks and accurate application of computational concepts to generate optimal solutions. This included accumulated knowledge on sequencing patterns to replace sequences of repeated blocks with a “Forever” loop, adding conditionals to initiate sprite actions based on predetermined actions to be completed beforehand and adding variables like “Score” to store data. I discuss the success of Trial and Error as the default yet successful initial debugging strategy in detail in the next section.

While students individually solved 43% of the debugging problems on average, they solved 14% (2 problems out of 14) of the debugging problems with facilitator’s feedback. This number was surprisingly low considering that 9 out of 10 students were complete novices to Scratch and programming in general. I observed that students were intent on solving the problems on their own, seeing it as a challenge. However, as students progressed to Debug 6 and up, the problems increased in difficulty with additional objectives that students had to complete to debug the problems. The problems also focused on more advanced computational concepts such as conditionals, operators, and variables where students had to create blocks to store values (i.e., Score) as well as coordinate and sequence blocks to meet conditions for specific sprite actions or for a change in the scoring parameter (i.e., for Debug 10 change score by -1 when Koko touches an orange and by +1 if Koko touches a banana). In these instances, failures increased despite multiple attempts and students actively sought out feedback. Previously successful Trial and Error
debugging strategies were not effective as these problems required deeper understanding of advanced computational concepts as well as accurate application and coordination of those concepts to debug the problem. Similarly, affective challenges (i.e., frustration, confusion, decision paralysis, low confidence) became prominent hindering students’ solution generation attempts (See Table 4-10). As a result of these observations on student difficulties, feedback was provided during the Solution Generation phase.

While this decision deviated from Kapur’s (2008) implementation of the PF design, where actionable feedback was held back until the Consolidation Phase, students benefited from the timely, positive, and problem specific feedback. Effective feedback has been shown to improve learning and performance in computer programming (Shute, 2008; Corbett & Anderson, 2001). Feedback that is positive, timely, non-evaluative and actionable can also ease affective challenges (Nordquist, 2007; Shute, 2008) and promote positive experiences such as engagement, feelings of enjoyment and self-efficacy (Lee & Ko, 2011; Mitrovic et al. 2013).

I provided feedback as prompts and helpful hints rather than providing actual solutions. Students reported that facilitator feedback clarified functionalities of code blocks and drew attention to conceptual errors in their scripts that were causing recurring code smells or adding bugs. Facilitator feedback for nearly there solutions also helped students debug the problems at hand and steered students away from recursive trial and error attempts with random blocks. Additionally, similar to prior studies (Biggers et al. 2008; Marwan et al. 2020), the positive experiences associated with solving a problem after multiple failures had a significant impact on these novice programmers’ intention to persist in generating solutions, solving problems, and learning programming.

Resolutions-Trial and Error as the Default Debugging Strategy

Trial and Error was students’ default strategy to solve the debugging problems. As discussed in the previous section, Trial and Error as an initial debugging strategy was expected for three reasons

- Implementation of PF design meant that during the Solution Generation Phase, students worked on debugging problems without direct instruction on block-based programming
and computational concepts. Without direct instruction and limited facilitator feedback, students used tools and strategies such as Trial and Error at their disposal to debug the problems at hand.

- 9 out of 10 students were novice programmers with no prior experience with Scratch, programming and debugging.

- Scratch and its accessible and novice friendly design features such as drag, drop, and snap blocks and immediate feedback on outputs supported tinkering and learning through exploration.

Implementation of the Productive Failure design reversed the learning sequence whereby self-directed problem solving preceded direct instruction and scaffolding. In the absence of instructional support as well as continuous feedback during the Solution Generation phase, novice programmers naturally turned towards Trial and Error to iteratively solve the debugging problems. Figure 4-5 also showed that on average students solved 43% of the problems (6 out of 14 problems) individually with many students reporting the use of Trial and Error as their primary strategy throughout the workshop. The strategy was economical and fast for novice programmers to explore the functionalities of code blocks and tinker with different sequencing permutations. Students also reported Trial and Error as a go-to strategy discussing its effectiveness to discover, understand and learn the correct application and coordination of computational concepts such as sequences, conditionals, and loops to debug the problems. This finding also has some precedence in prior literature where the intuitive nature of Scratch and the exploration through trial and error/tinkering was shown to increase competencies in computational concepts (Grover, Cooper & Pear, 2014; Kown, Lee & Chung, 2018).

Excerpt 1 (See Table 4-2) and Excerpt 2 (See Table 4-3) detail how Jamarcus and Nina utilized the trial-and-error approach to debug their code. For Debug 2, Jamarcus reported that he “basically did Trial and Error,” adding and testing different motion blocks until he found the ‘glide’ block, which allowed the chicken sprite to move seamlessly across the field (See Table 4-2, Turn 2-6). In iteration 1, Jamarcus used the ‘turn clockwise 90 degrees’ and the ‘move 30 steps’ motion blocks to turn and move
the chicken towards the corn kernel, but the chicken would only rotate in place due to the code being inside a forever loop (See Figure 4-6, top left-right). In Iteration 2, he attempted to fix this by changing the angle of the sprite using the ‘point in direction 180’ block, but this caused the sprite to orient downwards instead (See Figure 4-6, bottom-left). The issue was finally resolved in Iteration 3 when Jamarcus added ‘move 30 steps’ inside the ‘forever’ loop, enabling the chicken to continuously move towards the corn and complete the first task (See Table 4-7, left). For the second task, Jamarcus used an if-then conditional statement in conjunction with a look block and a sensing block, causing the chicken to say “Yumm!” when it touched the corn kernels, simulating the act of eating. Lastly, for the final task, he paired a ‘glide’ motion block with a ‘forever’ loop allowing the sprite to move randomly across the farm (See Table 4-7, left).

Jamarcus credited the Trial-and-Error approach for his understanding and application of different code blocks and computational concepts. He stated that every time he attempted a code, he learned something new, allowing him to understand its function, placement, and potential future uses (See Table 4-2, Turn 8). Despite the occasional frustration and disappointment, Jamarcus found the trial-and-error approach helpful in developing better debugging strategies. He learned from his multiple failures and tried to “step back and analyze it [the problem] again to see what worked and what didn't” (See Table 4-2, Turn 8). During later debugging attempts, Jamarcus used the Forward Reasoning Serial Order and incremental development methods, reviewing his code in serial order to locate errors after unsuccessful debugging attempts and progressively developing the code while frequently testing it before adding to it.

Nina, Wilma, Delani, Safiya and Damari also reported Trial and Error approach as their primary strategy to learn code block functionalities and apply computational concepts. Nina experimented with different combinations of motion, events, and conditional blocks to create a functional code for Debug 3, which would move the fish sprite in all 4 directions and across the maze after two iterations (See Figure 4-9 and Figure 4-10). She also added a conditional statement that caused the fish to move back ten steps when it touched the pink boundary panels (See Figure 4-10, left). Despite still struggling with some computational concepts such as events, conditionals, and sequences, Nina found that trying different
solutions was helpful for learning and correctly applying these concepts in her code (Table 4-3, Turn 6-8). Moreover, multiple failures and the iterative process of trial and error encouraged her to improve her debugging practices, and that she learned to trust the learning process.

Similarly, Safiya, Willow, Delani, and Damari also reported using the trial-and-error approach as their initial strategy to solve coding problems. Safiya employed a methodical approach by looking for available blocks and experimenting with various sequencing combinations to generate optimal solutions for Debug 1-4. Similarly, Willow and Delani tested their first-choice blocks that they believed were right or could solve the problem. Delani tested each block one by one until she found the correct one. Damari reported that she had to “press every block” until she found the one that could solve the problem. She compared the process to looking for a missing piece in a puzzle, where she sometimes had to erase the whole code to come up with a better solution. Despite encountering errors and failures, they found the trial-and-error strategy to be helpful in discovering code block functionalities and computational concepts. Damari equated failures/errors as moments when she “discovered another concept’s function.” This approach helped them make the game work without failure by progressively developing and testing the code.

Despite its effectiveness in solving coding problems, relying solely on trial-and-error approach as the primary debugging strategy can lead to bad programming practices among students. This was evident during the workshop, where students who continued to use trial and error primarily developed scripts with code smells, syntax errors, and general bugs as indicated by pattern of High frequency of syntax errors and general bugs (See RQ 2.1). These bad practices included bottom-up programming, where students dragged blocks that they believed were appropriate and tinkered with various sequencing combinations instead of approaching the problem from algorithmic and design level to incrementally solve the problem. Inefficient practices such as dead code, redundant/repeated scripts, and writing long scripts without proper decomposition using parallelisms, events and conditionals were also observed. Such practices had a negative impact on code optimization, code quality, and student's’ learning of programming and
accurate application of computational concepts. Therefore, while the trial-and-error approach can be helpful for novice programmers, it should not be relied upon as the sole debugging strategy. Scaffolds need to be developed to encourage students to adopt a more algorithmic and design-oriented approach to coding problems.

Resolutions—Backward Reasoning Utilized Frequently as an Error Location Strategy but Forward Reasoning and Combination Strategy was More Successful

In error location strategies, Backward Reasoning was more frequently used than Forward Reasoning. However, the combination of both Forward and Backward Reasoning was found to be more effective in locating and debugging errors. When utilizing Forward Reasoning, students typically used the Serial Order approach to review their code block by block and identify the source of the error before working forward to fix it. Investigation and reflection of errors began with the code itself in this approach. On the other hand, the Program Order approach involved simulating the program’s execution to identify and debug errors.

The Backward Reasoning strategy involved starting with the incorrect behavior of the program, usually its output, and working backward to locate the source of the error in the code. In Backward Reasoning, Simple Mapping was frequently used by students due to the immediate feedback provided by Scratch which helped to identify missing code and computational concepts. Simple Mapping was used to identify potential errors by mapping input and output data. For instance, Delani used Simple Mapping to identify issues with sprite movement in response to user input (code initialization) by mapping the expected movement output to the actual movement output and narrowing down the possible sources of errors, such as missing code blocks or incorrect logic. In addition to Simple Mapping, the Causal Reasoning approach was also frequently used to trace the chain of events that led to the error in the code from the output and refine the code. However, Causal Reasoning approach did not always lead to successful debugging.
During the Think Aloud sessions and in online interview questions, students explained their debugging processes, strategies, and solutions, which were analyzed based on Katz and Anderson’s (1987) error locating strategies. Table 4-11 shows the error location strategies used by each student, and Excerpt 9 provides examples of students’ explanation of their error locating strategies and debugging. To determine the dominant error locating strategy for each student, I first identified conversations with students about their debugging process, during the I then coded these conversations and written answers based on. However, it was not possible to determine the exact number of Debugs that each student solved with Forward or Backward Reasoning as this information was not made explicit in student conversations and could not be gleaned directly from their programming activities on screen. However, it was not possible to determine the exact number of debugging problems that each student solved with Forward or Backward Reasoning strategies as this information was not explicitly stated in student conversations and could not be determined from their programming activities on screen.

Table 4-11: Students error locating strategies employed during debugging

<table>
<thead>
<tr>
<th>Students</th>
<th>Forward Reasoning</th>
<th>Backward Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Program Order</td>
<td>Serial Order</td>
</tr>
<tr>
<td>Delani</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Damari</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Eliana</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Jonah</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Jamarcus</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Nakiya</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nina</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Safiya</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Tuan</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Willow</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Table 4-12: Excerpt 9, Students explaining their debugging and error-location strategies working on Debug 1-14 during Think Aloud and Interview sessions.

<table>
<thead>
<tr>
<th>Turn</th>
<th>Speaker</th>
<th>Verbal and Non-verbal transcript</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Forward Reasoning Serial Order</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Facilitator</td>
<td>How did you go about solving these problems? Can you explain your debugging process in detail?</td>
</tr>
<tr>
<td>2</td>
<td>Tuan</td>
<td>I went about the problems by picking out what I wanted to get done first and building upon that with the other things that needed to be done. Alright, so for this one [Debug 4] basically went over the problem, scanned the whole problem and code and figured out what I needed to do. So here it says this stuff doesn't have to be changed [Points to the given code that sets up the starting position of the sprite]. There's a bonus and the actual task [Clicks on ‘See Project Page’ and shows the instructions]. So basically went over to the actual task and identified the actual problem that you got to accomplish. So figuring out what the easiest thing to accomplish first is. So Whenever space key is pressed, just go and add the control function [Circles ‘forever’] to that [Points to the incomplete code with move 10 steps and turn anti-clockwise 15 degrees]. So when you put the space key [event block ‘when space key pressed’] and stuff and then you can press it [space key]. So I basically pressed it, and saw what it did. And it worked. And once you saw him do a flip already, then with the forever it would just keep on flipping.</td>
</tr>
<tr>
<td><strong>Backward Reasoning: Simple Mapping</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Facilitator</td>
<td>Okay. Great! So, talk to me about the problems and challenges that you faced and how did you debug it?</td>
</tr>
<tr>
<td>2</td>
<td>Nina</td>
<td>So, when I was trying to move Ben, I couldn't. I started out by just ... When I was trying to make him go up, I put the move 12 steps in the up arrow [points to the code that moved the sprite ‘When up arrow key pressed, move 12 steps’], but every time I would click the up arrow, he was only moving to the right. He wasn't moving up. He wasn't going where I wanted him to go. And then I started looking through the motion blocks and then I remembered X and Y [coordinates], and I figured that that's what it needed in order for it to move up and down. So I put this [Points to ‘change Y by 12’] instead and it worked. So I used the X and Y blocks for other directions.</td>
</tr>
<tr>
<td><strong>Backward Reasoning: Causal Reasoning</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Facilitator</td>
<td>What were some of the solutions that did not work? Why do you think that these solutions did not work</td>
</tr>
<tr>
<td>2</td>
<td>Delani</td>
<td>I tried different motion blocks at first. The block ‘go to x: y:’ did not work out for me since I wanted my character to move slowly from left to right, but this</td>
</tr>
</tbody>
</table>
block teleported her from the left side to the right. Then I tried a glide which moved Shantel left to right but did not get her to dance at the same time. Then I tried putting different glide blocks and then one ‘go to x: y:’ block. What I wanted was to move her [sprite] to different places [on screen] slowly from left to right and dance at the same time. But she would sometimes glide to one place and then dance or she would just dance. Then the teacher said that part of the code was just animation of her dancing. So I thought maybe if I get her to move just a little and play the animation over and over then that would work. I found Repeat and Forever [blocks] but I chose forever because it made her dance without stopping. It worked in the end but it took a while.

**Combination of Forward and Backward Reasoning**

1. Facilitator: Thank you. So Jamarcus, talk to me about your process of debugging, process of problem solving? What was your process?

2. Jamarcus: I read what they [problems] were asking for, right? For this one, [Debug 10] they [talking about the task and instruction] wanted you to make something like decrease the score or increase it [Change scores by -1 for oranges and red boundary panels and increase by +1 for bananas]. So then basically I could work off of that by finding blocks that had the function that I wanted. It could be like many different blocks that have the function that I want. Basically I did that. So if something didn't work, then I would just search for another thing [block] that would work.

3. Facilitator: Okay. So can you give some examples?

4. Jamarcus: Yeah. For example, with Coco, the problem was that the score wasn't increasing. So I had the idea of just changing the score for each of the sprites so that when they were touched [by Koko] the score would increase. And it worked.

5. Facilitator: Okay, great. What about some of the other games? Can you talk about some of those games as well.

6. Jamarcus: So, in the Finding Nemo one, the maze, it wasn't working correctly. When I hit the maze, it wasn't sending me back to my original position. So I just changed the color to Red. So it'd be easier for, I guess, the program to see that it was the color red, so it could detect that it was red and then it would take me back. I think it had something to do with the idea of opacity [saturation] being different.

Backward Reasoning was the primary error locating strategy used by students as opposed to Forward Reasoning. However, when they did use Forward Reasoning, they relied on Forward Reasoning Serial Order, which involved reviewing the code block by block to locate the source of the error then
working forwards to fix it. Tuan’s experience with Forward Reasoning Serial Order illustrates this approach. First, he examined the tasks and then “scanned the whole problem and code and figured what I[he] needed to do” (Table 4-12, Tuan, Turn 2). He identified the missing Event block for initiating backflips when the space key was pressed (See Figure 4-31, left). Tuan then added a forever loop inside the sequence ‘Move 10 steps’ and ‘Turn 15 degree anticlockwise’ to enable continuous backflips and successfully complete the debugging objectives (See Figure 4-31, right). Students found Forward Reasoning Serial Order was a successful method for locating and debugging because it was easier to execute. By reviewing the code block by block and then running the code, they were able to identify missing or uninitialized sprite actions and add the necessary code blocks to debug the errors. Compared to the Program Order, Forward Reasoning Serial Order allowed students to focus on specific areas of their code, rather than trying to understand the entire program execution flow. This made the process more manageable and efficient for students.

Delani, Damari, Nina, Safiya, and Jonah frequently used Simple Mapping in their use of Backward Reasoning (See Table 4-11). The frequent use of Simple Mapping can be attributed to the immediate feedback mechanisms in Scratch. This feedback enabled the students to quickly identify specific sprite actions that were not executed, providing them with clues regarding the missing code and
computational concepts that could be utilized to rectify the issue. Therefore, Simple Mapping became a convenient and readily available strategy for students to use. For instance, when Nina encountered an error in her Space Bugs game where the sprite was not moving up upon clicking the up-arrow key, she used Simple Mapping to locate and solve the erroneous line of code (Table 4-22, Nina, Turn 2). Working backward from this incorrect behavior, she edited her code to fix the errors. Nina replaced the ‘move 12 steps’ command with ‘change Y by 12’ which moved the sprite along the Y coordinate (See Figure 4-29, left). She applied the same fix to the Down arrow key changing Y by -12. The combination of Simple Mapping and Scratch’s immediate mechanism proved to be an effective tool for the students in identifying and correcting errors in their code.

The use of Causal Reasoning within Backward Reasoning was also observed among some students (See Table 4-11). They relied on this approach to trace and resolve errors in their code. By starting with the output of the code, they made predictions about the location and causes of the errors. However, this approach was not always successful in helping students successfully debug their code, as evidenced by the case of Delani in Debug 6. She encountered an error where her sprite, Shantel, teleported to the right side of the stage instead of moving slowly from left to right while dancing. Delani worked backwards from this output and traced the error back to the ‘go to X: 157 Y: -46’ motion block that she had initially added. However, her reasoning for the cause of the error resulted in additional errors. Delani believed that replacing the faulty ‘go to X: 157 Y: -46’ motion block with a ‘glide 1 secs to X: 171 Y: 29’ block (See Figure 4-12, middle) would resolve the sprite’s movement (Table 4-22, Delani, Turn 2). She later combined multiple glide blocks with the ‘go to X: 157 Y: -46’ motion block in iteration 3 (See Figure 4-13, left). However, the code output showed the glide block in iteration 2 moved Shantel from left to right but did not streamline the dance movement, and the combination of motion blocks in iteration 3 made the sprite’s movement more erratic (Table 4-22, Delani, Turn 2). Delani eventually solved Debug 6 with facilitator feedback in iteration 4 (See Figure 4-14). However, her case highlighted the inefficacy of causal reasoning in resolving errors, even when students were able to locate the errors.
This lack of efficacy may be attributed to the student's' limited programming knowledge and expertise to the student's' limited programming knowledge and expertise. Although they were able to identify errors in their code by examining the output of the code and making predictions about the possible causes of the bug, they struggled to effectively trace the error back to the code and rectify it. Mastery of programming concepts and logic is essential to effectively use the Backward Reasoning approach. Katz and Anderson (1987) found that Backward Reasoning strategies were frequently used by and yield positive results for experts rather than novices. They argued that an effective use of this approach requires knowledge of why an algorithm works in a particular way.

Although Backward Reasoning as a standalone approach had mixed success rates in locating and debugging errors, combining it with Forward Reasoning was effective. This finding aligned with Romero et al’s (2007) study on Java debugging for novices, which demonstrated that a dynamic and “multi-representational debugging environment” that displayed program code, program visualization, and output concurrently encouraged participants to use a combination of forward and backward reasoning (p. 12). By switching between examining the code (Forward Reasoning) and reviewing the program's output and visualization (Backward Reasoning), novice programmers were able to build a more comprehensive understanding of the program and integrate information from different external representations to locate and debug errors successfully.

Jamarcus, Tuan, and Willow applied this combined strategy successfully, as evidenced by Jamarcus’ approach to Debug 10. Initially, he used Forward Reasoning Serial Order to read through the code, task objectives, and narrative of the debugging problem (See Figure 4-32, left). Then, he examined the output (Backward Reasoning) and discovered the scores for the banana sprites were not displaying or increasing when Koko touched them (Table 4-22, Jamarcus, Turn 2). He experimented with different blocks, adding, and testing them to see whether they increased the scores. He also used Simple Mapping, where the output of the code helped him determine whether his experimentation with a code block or combination of blocks was effective in increasing the scores. When Simple Mapping failed to locate the error in Debug 10, Jamarcus switched to Causal Reasoning and hypothesized that adjusting the score for
each sprite so that it increased when Koko touched it would solve the problem (Table 4-22, Jamarcus, Turn 4). Drawing on this reasoning and information from the code output, he narrowed his search and eventually added a ‘Score’ variable, set it to 0 on Koko’s code (See Figure 4-32, middle), and added the ‘Change score by +1’ to the codes of all 12 banana sprites (See Figure 4-32, right).

Figure 4-32: Debug 10’s original code lacked the score variable and a conditional (left); Jamarcus’ completed code added the score variable and size increase conditions for Koko (middle); Banana sprite code with score change conditional (right)

Overall, combining Forward Reasoning with Backward Reasoning was more effective in locating and debugging errors than using Backward Reasoning alone. While Forward Reasoning was not used as frequently as Backward Reasoning, when used, students utilized Forward Reasoning Serial Order to review their code block by block and identify the error source (McCauley et al. 2008). In the Program Order, students simulated the program’s execution as a computer would, to locate and debug errors. In Backward Reasoning, Simple Mapping was frequently used to identify errors by mapping input and output data in Scratch. This technique provided immediate feedback to students and helped them pinpoint missing code and computational concepts. Students also frequently used Causal Reasoning. They traced the chain of events that led to the error by starting from the incorrect output and working backwards. Students reviewed the program’s execution and utilized information from the output to reason about possible causes of the bugs, eventually locating and resolving the error in their code. However, while
these strategies were successful in some cases, they did not always lead to successful debugging. Overall, the findings highlighted the importance of combining different strategies to identify and resolve programming errors.

Resolutions-Switch from Bottom-Up Bricolage Programming to Top-Down Approach

At the beginning of the study, students relied on a bottom-up or bricolage approach to debug their programs in Scratch. This method involved trial and error, tinkering, and experimentation with different code blocks in an attempt to solve problems. While this approach was effective for initial debugging and allowed students to explore the functionalities of different code blocks and learn about computational concepts such as conditionals, loops, and sequences, it was time consuming and led to unsuccessful iterations (See Table 4-2, 4-3). As a result, bricolage programming hindered their rate of completion and subsequent learning opportunities during the Consolidation phase (See Table 4-30). Additionally, the resulting code contained recurring code smells and syntax errors as the students did not approach problems from a comprehensive algorithmic, design, and systems-level perspective (See Table 4-5, 4-7, 4-9).

As students progressed through the debugging curriculum, some (Tuan, Delani, Jonah, Willow, and Jamarcus) refined their debugging strategies to include a structured, planning-oriented top-down approach or a mix of bottom-up and top-down approach. This approach involved decomposing problems into smaller units and incrementally developing the program while frequently testing their code. These students utilized the rapid experimentation and tinkering aspects of bottom-up programming while incorporating planning and design elements of top-down approach to avoid poor programming habits and practices. Excerpt 10 (See Table 4-13) provides insights and explanations regarding the shifts in students’ programming approach.

However, some students (Safiya, Damari, and Nina) continued to rely on the bottom-up approach despite its drawbacks. These students reported that the approach was useful in learning about the code blocks and refining their understanding of how these blocks could be used to debug problems. Over time,
they refined their bottom-up approach. They transitioned from randomly dragging and testing code blocks to deliberately selecting blocks based on the debugging objectives and testing them to choose the correct ones. Excerpt 11 (See Table 4-14) provides insights into why some students continued to use the bottom-up approach.

Table 4-13: Excerpt 10, Tuan explaining his bottom-up approach in Day 2 working on Debug 2-4 in the online interview and his switch to a top-down approach in Day 6 working on Debug 9

<table>
<thead>
<tr>
<th>Turn</th>
<th>Speaker</th>
<th>Verbal and Non-verbal transcript</th>
</tr>
</thead>
<tbody>
<tr>
<td>Day 2, Online Interview Responses</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Facilitator</td>
<td>How was working on those debugging problems? What was interesting, easy, difficult?</td>
</tr>
<tr>
<td>2</td>
<td>Tuan</td>
<td>I thought the problems were a good difficulty where they aren't too difficult or complex for someone as new to coding as me but aren't too easy as well, sparking creativity and problem-solving in a way where I think it more engaging than inside the normal classroom.</td>
</tr>
<tr>
<td>3</td>
<td>Facilitator</td>
<td>What were some of the challenges, problems you faced while working on debugging problems?</td>
</tr>
<tr>
<td>4</td>
<td>Tuan</td>
<td>Trying to find the exact command block to fit your needs was sort of a hassle but I'm sure that would dissolve as soon as I get more comfortable with the program.</td>
</tr>
<tr>
<td>5</td>
<td>Facilitator</td>
<td>How did you go about solving these problems? Can you explain your debugging process in detail?</td>
</tr>
<tr>
<td>6</td>
<td>Tuan</td>
<td>I went about the problems by picking out what I wanted to get done first and building upon that with the other things that needed to be done. Then I just used the commands that did the action directly, whether it be glide to a set of coordinates, turn 15 degrees, or say a message.</td>
</tr>
</tbody>
</table>

Day 6, Think Aloud Response to Debug 14 Process

<table>
<thead>
<tr>
<th>Turn</th>
<th>Speaker</th>
<th>Verbal and Non-verbal transcript</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Facilitator</td>
<td>Okay, Tuan, whenever you're ready. Show me the code first, I'll record that, and then we'll have a discussion, okay?</td>
</tr>
</tbody>
</table>
| 2    | Tuan       | So I changed Debug 10 to make my own game...I looked at Debug 10 and I did all of that, right? What was Debug 10 originally? So, it was Coco the monkey, right, eating bananas and stuff...You have a clear objective. It's just to eat the bananas and stuff...So, for me, with Coco [Debug 10], I was thinking, "It's cool" but what else eats things? As I said before, I'm mentally addicted to League of
Legends, right? So I changed Debug 10 into my game. It reminded me of Cho’gath, which is just a monster [Mage, Tank in League of Legends game] that eats Poros and gets bigger every time it eats. So, I incorporated that stuff into this character [Cho’gath’s] design. So, I made him eat little things called poros. They’re like balls of fur and horn. He [Cho’gath] gets bigger with everyone [poros] you eat. So, I made the objective of the original Debug 10 [to eat the bananas] and just morphed it into this weird thing [game] that I made.

Facilitator: Great, can you show the codes and tell me about the gameplay and your process.

Tuan: It’s just all over the place. I’ll go over to the movement part first, which is this. [Clicks on Cho’gath sprite and zooms in movement codes, See Fig 34] All right. There’s that. Then there’s this part over there [Zooms out and shows remaining codes for Cho’gath controlling access to next levels]. Then you have this one [Clicks on the music sprite]... That’s the music bot [Shows the code] There’s the start button. [Clicks on the Red Play button sprite and scrolls through the codes] It’s basically like this before the rest. And then I didn’t finish the first boss yet, but this is close enough. [Clicks on Warwick sprite and scrolls through the codes]

Facilitator: Okay. That was awesome! What are the problems that you ran into and how you debugged it?

Tuan: So, there were a ton of problems with this freaking thing. So, like I said, I had no past coding experience or game design. I just wanted to do this because I thought it was pretty fun. But the levels were hard to do because I had to add all the codes in correct order otherwise it wouldn’t work. So for example for this [points to all the Level 2 codes for Cho’gath] I had all this code under one [When I received Level 2 start] and so for some reason it didn’t work. Some codes would run like the variable would show but I don’t know like the size wouldn’t change, the laugh sound wouldn’t play and sometimes it wouldn’t say “Who will be Eaten first.” So then I broke it into different chunks and got each piece to run at the same time. I put the show variable and set [Points to ‘Show variable Poros eaten’ and ‘Set Poros eaten to 0’ blocks] with the movement and size [codes] together and the sounds together and then it worked. But it took a lot of moving things around and thinking about what each character needs to do and writing the codes in the exact order to do it. That took some time. And I also had to figure out how to do different levels but then I saw someone on Scratch using broadcast to run another code, and then it clicked. I could use broadcast to transition to different levels!!”

Table 4-14: Excerpt 11, Safiya explaining her continued use of bottom-up approach from Day 2 to Day 6

<table>
<thead>
<tr>
<th>Turn</th>
<th>Speaker</th>
<th>Verbal and Non-verbal transcript</th>
</tr>
</thead>
<tbody>
<tr>
<td>Day 2, Online Interview Responses</td>
<td>1 Facilitator</td>
<td>What were some of the challenges, problems you faced while working on</td>
</tr>
</tbody>
</table>
debugging problems?

2 Safiya Some challenges were not knowing exactly what a coding block meant.

3 Facilitator How did you go about solving these problems? Can you explain your debugging process in detail?

4 Safiya I looked at the available blocks and built them in a way that made sense. I also made sure that it worked. I tried different blocks from the palette and tried them out until I found the right one that solved the problem.

Day 6, Think Aloud Response to Debug 1-10 Debugging Strategy

1 Facilitator So you worked on 10 different debugging projects so far, right? First of all, can you talk about your process? How did you go about solving those problems?

2 Safiya So, to solve most of the problems I would just look at what the goal was and then look through the blocks that I thought would work and then just test it to see if they worked or not.

3 Facilitator Okay. What happened when you tested some blocks in it and some worked and some didn't, what would you do?

4 Safiya I'll either look at the code and see what worked and what didn't and then I looked at different blocks that I thought would work and test those [blocks] out. If that didn't work out I would ask for help with that [problem]. Sometimes I would think that I knew what I was doing, but then when I go to test it, it didn't work out the way I thought it would. So then I would have to do it over again.

5 Facilitator Why do you think that was? Why do you think that some blocks sometimes it worked the way you thought that it would and sometimes it wouldn't?

6 Safiya Probably because mainly with some of the blocks like the variables and operators, I didn't know what it did. So it took some time to try those and see what it did.

Excerpt 10 (See Table 4-13) highlights an example where students changed their programming and debugging approach from a bottom-up to a top-down approach. Tuan, a novice programmer with no prior experience with Scratch, initially used a bottom-up approach to generate solutions for Debug 1-4 on Day 2. He experimented with different blocks by adding them to his code and testing them with different sequencing combinations. However, he reported that trying to find the exact command block to fit his needs was time-consuming and a hassle (Table 4-13, Day 2, Turn 6).
As Tuan became more comfortable with Scratch and familiar with the functionalities of these code blocks, he switched to a top-down approach. He demonstrated this in creating his own game for Debug 14 on Day 6. He approached the task from a design, algorithmic/scripts, and systems level, designing and planning out the game narrative, sprite actions, and code scripts. He broke down the tasks into manageable chunks, incrementally remixing, writing, testing, and editing his codes to ensure optimal performance and mitigate the occurrence of code smells and bugs (See Table 4-13, Day 6).

Tuan approached Debug 14 by decomposing tasks into game design, programming, and debugging. He started out by remixing the existing Debug 10 game that he had already debugged but ended up completely redesigning it to make his own game called Cho’gath’s Rampage, a retro version of his favorite game League of Legends (See Figure 4-33, left). The objective of his game was to play as Cho’gath and go through the first 5 levels farming and eating the poros and ultimately face Warwick, the main boss, in level 6 (Table 4-13, Day 6, Turn 2). For game design tasks, he designed and planned each of the 6 levels, taking out the existing Debug 10 sprites along with the backdrops and codes, and replacing them with a retro version of the Summoner Rift’s map. Different parts of the map were assigned as backdrops for each of the 6 levels he designed (See Figure 4-33, right). He added new sprites such as Cho’gath, poros, Warwick and start button resizing, editing, and pixelating them using Scratch’s costume feature (Table 4-13, Day 6, Turn 4).

Figure 4-33: Tuan’s Debug 14 game, Cho’gath’s Rampage, a retro version of his favorite game League of Legends (left); Tuan’s design of a retro version of the Summoner’s Rift map consisting of 5 levels (right)
For programming tasks, Tuan wrote codes for one sprite at a time and incrementally built up the scripts testing the performance as he added new code blocks. Starting with Cho’gath, he broke down the codes by sprite actions, focusing first on directional keyboard movement and mouse movement (See Figure 4-34, left). Tuan then wrote the codes for each level meticulously and incrementally, writing out the conditions to go to the next level (See Figure 4-34, right).

He also wrote level-specific codes for each sprite, specifying the actions that sprites could or could not do at each level and the conditions for those actions to happen. To ensure smooth transitions between levels, Tuan used parallelisms and connected Event hats for levels so that actions completed in one level initiated and affected actions in other levels. For example, he used connected Event hats to set conditions to initialize each level. When the start button was clicked, the message ‘Level’ was broadcasted and the initial level was set to Level 2. Once the ‘Level’ message was received, the Event hat ‘when I receive Level’ set and initiated the conditions for all 6 levels to start. Additionally, a ‘Level start’ message was broadcasted to indicate the start of the next level (See Figure 4-35, left). When the message was received, parallel actions were initialized that Cho’gath performed at that level. For instance, when Cho’gath received ‘Level 2 Start’, a set of codes were initialized simultaneously to set Cho’gath’s size.
and initial position, set the ‘Poros eaten’ variable to 0, play music and set conditions for size increase and level 3 scripts to initialize (See Figure 4-35, right). By meticulously, planning and designing the game narrative, levels, sprite actions and incrementally testing and editing the codes, Tuan ensured optimal performance and mitigated the occurrence of code smells and bugs in his game.

Figure 4-35: Start button code with interconnected Event hats to set conditions to initialize each level (left); Parallelly executed Level 2 codes initialized after ‘Level 2 start’ message was received (right)

Tuan’s game was operational; however, it wasn’t without issues. He initially struggled with organizing and sequencing his Level 2 codes, leading to discrepancies in initialization and timing (See Table 4-13, Day 6, Turn 6). To overcome these challenges, he adopted a top-down approach. Tuan rearranged his Level 2 codes by breaking them down into smaller chunks and grouping them into actions specific sequences: music, movements, and levels. Tuan also utilized parallelisms and interconnected events to efficiently solve problems and optimize his game. His successful implementation of top-down programming exemplified the importance of code comprehension, task decomposition, error location, and incremental work towards creating optimal solutions. Other students also benefited from this shift from bricolage approach to a systematic top-down approach.

However, not all students switched to a top-down approach. Safiya, Damari and Nina continued using a bottom-up bricolage approach to program and debug problems. Although this approach was slower and more time-intensive, it helped them optimize their code and understand computational
concepts and coding blocks. Safiya’s example, in particular, illustrates how she relied on the bottom-up approach to program and debug problems. During an online interview on Day 2, Safiya discussed how she faced challenges in understanding certain code blocks and used a rapid plug-test-validate bricolage approach (Table 4-14, Day 2, Turn 2-4). This involved experimenting with different blocks until she found the one that solved the problem, a process that was iterative and creative in nature. Safiya continued to refine her approach throughout the workshop, selecting code blocks purposefully based on the debugging objectives and then testing them out to see if they worked (Table 4-14, Day 6, Turn 2-6). While the bottom-up approach was time-consuming, it helped students like Safiya, Damari, and Nina to develop their understanding of coding blocks and computational concepts such as variables, operators, and conditionals.

To summarize the findings, novice programmers in the workshop initially relied on a bottom-up bricolage approach to program and debug problems. This approach proved effective, particularly within the context of the Productive Failure design, which emphasizes trial and error, tinkering, exploration, and experimentation. Additionally, Scratch’s design and constructionist philosophy also supports programming by bricolage (Turkle & Papert, 1992; Resnick et al., 2009). However, the bottom-up approach was found to be time-intensive and affected students’ rate of completion of debugs. Additionally, as previous research has shown, bottom-up bricolage programming can promote poor programming habits, recurring code smells, code quality issues, and practices that hinder code optimization, quality, and learning (Aivaloglou & Hermans, 2016; Meerbaum-Salant et al. 2011; Moreno & Robles, 2014; Techapalokul, 2017). Students tended to drag and drop random blocks rather than working on problems from a planning, design, and algorithmic level. As a result, some students switched to a top-down approach, breaking down tasks into manageable chunks, and incrementally developing, testing, and editing codes to ensure optimal performance. Tuan’s example highlighted the success of this approach. However, other students like Safiya, Damari, and Nina, continued to use the bottom-up approach and refined it to be more purposeful in code block selection and testing. This helped them deepen their understanding and application of computational concepts. Overall, the findings highlights the
benefits and limitations of the bottom-up and top-down approaches in programming and debugging. While the bottom-up approach encourages creativity and exploration, it can also lead to poor programming habits and time inefficiencies. In contrast the top-down approach provides structure and promotes good programming practices but can limit tinkering and exploration. The findings also suggest that a combination of both approaches, tailored to the specific needs and goals of individual student's, can promote effective learning of programming concepts and practices.

**Qualitative Findings RQ2.2**

In this section, I report on the findings of RQ 2.2: *What do these challenges, resolutions, process, and interaction patterns reveal about students’ evolving understanding of computational concepts?* To answer RQ 2.2, I used interaction analysis, and thematic analysis to investigate how participants’ understanding of computational concepts evolved as they progressed through the curriculum. The data consisted of transcribed audio from Think Alouds, video recordings, and online interviews, where participants

a) discussed the coding challenges they faced, the solutions they developed, and their problem-solving process,

b) provided examples of computational concepts that they found difficult to understand and discussed their progress or lack thereof in applying these concepts to their solutions,

c) described and compared their understanding and learning of computational concepts over the course of the workshop.

To analyze the data, I used coding techniques informed by Saldana’s (2021) guidelines and Litts et al’s (2016) four stage coding process, which involved identifying instances of debugging challenges, types of challenges, and strategies used to resolve them. I also utilized Braun, Clarke, and Hayfield’s (2015) and Saldana’s (2021) guidelines to generate, organize and analyze emerging themes that could
answer RQ 2.2. Based on this analysis, I identified several patterns that shed light on the students’ evolving understanding of computational concepts.

**Improved Application Accuracy of Some Computational Concepts**

During the curriculum, students showed improved accuracy in applying computational concepts such as sequences, loops, parallelisms, and events, indicating better comprehension of these concepts. However, their solutions still contained recurring error patterns related to conditionals, data/variables, and operators, indicating their struggle with more complex computational concepts.

This study utilized application accuracy as an approach to assess learning and understanding of computational concepts. This approach is well established and has demonstrated efficacy in assessing computational concepts learning (Brennan & Resnick, 2012; Dasgupta et al., 2016; Fields at al. 2016; Griffin, Kaplan & Burke, 2012; Kafai et al., 2014; Richard & Giri, 2019). These studies analyzed program code, conducted portfolio analysis, focused on accurate application of computational concepts in debugging Scratch projects, and analyzed learning through remixing. In this study, analysis of application accuracy was based on

- occurrences of recurring errors associated with specific computational concepts
- analyzing students’ solutions (artifacts) and explanations of their solutions, challenges and debugging process

I utilized Frädrich et al’s (2020) inventory of bug patterns to categorize recurring errors in student generated solutions (See Table 4-5) and identified specific error patterns using Litterbox, a static code analysis tool for detecting error patterns in Scratch. Detailed information on recurring code smells and error patterns is discussed in the Case II findings for RQ1 and RQ 2.1.

Table 4-5 illustrates that there was a decrease in recurring error patterns associated with computational concepts such as sequences, loops, parallelisms, and events from Debug 1 to Debug 14. For example, misordered code blocks, which were an error pattern associated with Sequences, reduced from **14** occurrences or **63.6%** in Debug 1-3 to **0** occurrences in Debug 10-14. There was also an
improvement in students’ application accuracy with loops, parallelisms, and events showing a decrease in occurrences of the error patterns associated with these computational concepts.

In terms of Loops, occurrences of missing loop and loop sensing decreased from 12 occurrences, 37.5% in Debug 1-3 to 8 occurrences, 25% in Debug 10-14. However, the drop to 3 occurrences in Debug 7-9 was not due to accurate loop application by students, but rather because fewer students attempted those problems due to absences or time constraints. Additionally, the occurrences of Sequential action decreased from 8 instances, 80% in Debug 1-3 to 0 instances in Debug 10-14. Previously, students duplicated code blocks to repeat sprite actions. However, as they progressed in the curriculum, students utilized a Forever or Repeat loop, resulting in more streamlined and optimized code.

The frequency analysis showed a decrease in occurrences of error patterns and code smells associated with parallelisms and Events. Specifically, Unused or Uninitialized Parallel actions reduced from 4 occurrences, 66.67% in Debug 1-3 to 2 occurrences, 33.33% in Debug 10-14. Similarly, occurrences of Missing and Unused Initialization/Events decreased from 8 occurrences, 44.44% to 3 occurrences, 16.67% (See Table 2-1 for definitions).

Despite the improvement in these computational concepts, students continued to struggle with application and understanding of complex computational concepts such as conditionals, variables/data and operators. The occurrences of Unused or Uninitialized variables increased from 1 instance in Debug 7-9 to 4 instances in Debug 10-14. Additionally, there was a minor decrease in the occurrence of Missing and Incomplete Conditionals stayed the same at 4 occurrences, 30.77% to 1 occurrence, 7.69% in Debug 4-6. Students still struggled with writing multiple conditionals inside a script and coordinating the sprite actions such that the conditions for each sprite action did not overlap and the conditions were written completely to accurately initialize those conditions.

Although there was a decrease in the occurrences of error patterns associated with computational concepts, the reasons for improvement in application accuracy could not be solely determined through the frequency analysis. Therefore, an analysis of students’ artifacts and interactions during Think Alouds and online interviews was conducted. From this analysis, it was found that the improvement in
Application accuracy of some computational concepts could be traced to the iterative solution-generation phase of the PF design and changes in debugging strategies. I discuss this analysis in detail below.

**Application Accuracy through Iterative Solution-Generation Phase of the PF design**

The Solution Generation phase played a critical role in improving students’ application accuracy of some computational concepts. As per the Productive Failure Design, direct instruction and scaffolding were not provided to students on how to accurately apply computational concepts during this phase. However, it is important to note that the majority of students (9 out of 10) were novice programmers with no prior experience in programming or Scratch. This was reflected in their pretest scores on the Computer Programming Self Efficacy Scale (CPSES), which assessed their understanding and application of computational concepts. As a result, students had to rely on trial and error to iteratively generate solutions for the debugging problems. This process allowed them to develop a better understanding of the functionalities of code blocks and the accurate application of computational concepts. Through this iterative solution generation, novice programmers like Jamarcus and Nina were able to figure out how to accurately apply computational concepts such as Sequences, Events and Conditionals.

For example, Jamarcus struggled with code sequencing to solve the Debug 2 problem, which required students to move the rooster sprite to the corn kernels in a prescribed sequence: first, get the rooster sprite to move to corn kernels, then say Yumm, and move across the field. However, through iterative attempts at generating solutions and using trial and error, Jamarcus was able to generate the optional solution his third attempt. In his first attempt, he used a combination of two motion blocks, ‘Turn 90 degrees’ and ‘Move 30 steps’ inside the forever loop, which did not work as expected (See Figure 4-6, top left). However, he reflected that this sequencing did not work because the rooster sprite “would keep on turning forever. It will turn, and then just keep going straight. So it would be upside down” (See Table 4-2, Turn 4; Figure 4-6, top-right).

During Iteration 2, Jamarcus attempted to change the sprite’s orientation by adding ‘Point in direction 180 degree’ to face it towards the corn kernels (See Figure 4-6, bottom-left). However, this
sequencing did not work out because the sprite faced downwards, and without motion blocks such as ‘Move’ or ‘Glide,’ the sprite could not reach the corn kernels (See Figure 4-6, bottom-right). In his third attempt, Jamarcus found the optimal sequencing by adding a motion block ‘Move 30 steps’ and a conditional ‘If touching corn kernels then Say Yumm’ with another ‘Forever’ loop and motion block ‘Glide 1 second to random position,’ completing all Debug 3 objectives,(See Figure 4-7, left). The sprite moved towards the corn kernels, said Yumm, and moved across the field (See Figure 4-7, right). This iterative solution-generation approach allowed Jamarcus to develop a better understanding of computational concepts such as Sequences and Loops.

Similarly, Nina demonstrated an improvement in her application accuracy of computational concepts such as Events, Conditionals and Sequences through iterative solution generation for Debug 3. In her first attempt, she added a motion block ‘Move 10 steps’ to all directional Event hats but realized that it moved the sprite to the right when pressed the right or left arrow key (See Figure 4-9, middle-right). In her second attempt, she experimented with motion and control-conditional blocks to accurately move the fish sprite in all directions. Nina used ‘Point in Direction 90 degrees’ and ‘Move 10 steps’ to orient the sprite’s movement to the directional key press, completing the Debug 3 objective of moving the sprite 10 steps in all directions (See Figure 4-10, right).

The Solution Generation phase proved to be a valuable learning experience for students in developing their programming skills and improving their understanding and application accuracy of computational concepts. Through trial and error, students were able to generate multiple solutions and ultimately improve their application accuracy in solving debugging problems. The iterative approach allowed students to experiment with different code blocks and sequencing combinations, leading to a deeper understanding and accurate application of computational concepts such as sequences, events, and conditionals. Both Jamarcus and Nina reported that iterative solution generation was frustrating but ultimately helpful in their learning process (See Table 4-2, Turn 8; Table 4-3, Turn 10). As a result of this approach students gained knowledge not only on individual code blocks and concepts but also on how to combine them to create more complex programs. Overall, the Solution Generation phase proved to
be an effective method for teaching programming skills and enhancing student's’ computational thinking abilities.

Issues with Combining Computational Concepts to Debug Problems and Write New Codes

Student's encountered challenges when it came to integrating computational concepts to debug problems and create new code. These difficulties highlighted the challenges that novice programmers face in applying their knowledge, optimizing, and scaling their code, and tackling problems from a systems perspective. The debugging curriculum was designed to address this by initially focusing on a single computational concept in each problem, such as Sequences in Debug 1-3. Students were able to figure out the correct sequence for these problems without much difficulty by experimenting with different code blocks and sequences until they found the optimal sequence. This was demonstrated by Jamarcus and Nina’s iterative solutions in Debug 2 and 3 (See Table 4-2, Table 4-3).

However, subsequent debugging problems targeted multiple computational concepts in each problem, such as Events and Loops in Debug 4-6, Conditionals and Parallelisms in Debug 7-9, Operators and Variables in Debug 10-12, and game creation in Debug 12-14. This increased complexity required students to accurately apply computational concepts together to debug problems and create functional codes for their games. Excerpt 12 (See Table 4-15) highlights some of the difficulties that students faced when combining computational concepts to debug problems and write new codes. Figures, following the excerpt, provide examples of Safiya’s code to illustrate these challenges.

Table 4-15: Excerpt 12, Safiya explaining the challenges she faced while creating her game for Debug 14 during a Think Aloud session with the facilitator.

<table>
<thead>
<tr>
<th>Turn</th>
<th>Speaker</th>
<th>Verbal and Non-verbal transcript</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Facilitator</td>
<td>That looks awesome!! Talk to me about the first game. And also talk to me about the idea for the game.</td>
</tr>
<tr>
<td>2</td>
<td>Safiya</td>
<td>Okay. So, I really like Lilo and Stitch. I don't know why. I know about the movie and I started watching the show, so I based a game on it.</td>
</tr>
</tbody>
</table>
3 Facilitator So, talk to me about the game. How did you go about creating it? What was your process?

4 Safiya So, first I had to figure out how to make there be a correct answer which I got help with. But I asked the question, or I would say what question number it is. Okay. So, underneath the green flag code [Points to the short code initialized “When Green flag clicked”], I set the score as 0 and hid the variable “correct” so players don’t see the answer. Then I asked the questions. Up here where it says, "When one key pressed" [Points to another code with Event hat “When 1 key pressed”]. And then I would say, "Question one." And I would wait, and I would ask the question, and then set “correct” to “answer,” and I put what the correct answer was in the if-then-else bracket [Points to the rest of the code with If-then-else that set score conditions for correct and incorrect answer]... And this [Points to “Say ‘Hit the 2 key for the next question’ for 3 seconds at the bottom of the code initialized by “When 1 key pressed”] tells you how to get to the next question, basically. So, you hit the “2” key for the next question, then it would read question two, and it basically just repeats itself, but with a different question. So I did this for 10 questions.

5 Facilitator That’s very smart!! Talk to me about this game and the previous game. You can start with this game. What were some of the challenges, or some of the problems that you had to work on and how did you solve them?

6 Safiya I mean, the first problem was setting the correct [the variable “correct”] to “answer” which you helped me with. [Points to “Set correct to ‘Answer’ variable] First you need to create a variable "correct" then you can set "correct" to answer. Then put “if-then-else” and “say if answer = and put the correct answer in [Points to the rest of the code with If-then-else that set score conditions for correct and incorrect answer]. Then you can add 1 point and in else you can put -1 point or however many points you want players to lose or gain. And also, sometimes with the first game, I needed to figure out why it wasn't saying what the correct answer was when you got the incorrect answer. This was because the timing was off and went too fast to be able to say what the correct answer was

7 Facilitator Okay. What were other challenges? What were some other problems that you faced and how did you solve them?

8 Safiya So, for the first one [game], my main issue was writing the codes for the questions. I had all the questions in one code before and it was too long. So, I thought it would be more confusing ... if you have it all in one big line. So I split it up into separate parts [code for 10 questions], then I looked through “when blank key press”, then I saw that there were numbers [Points to the “When Space key press drop down menu to reveal numbers that can be selected as buttons] so I just had the numbers correspond with the question number. Writing the if-then-else bracket for the questions was hard too. You don’t know how to say things in code that you are thinking. I didn’t know how to say if this answer was correct then do this and if not do that. But once I found this block [Points to ‘_ = _ operator block], it wasn’t hard because I knew I had to write what to do for correct and incorrect answers.
On the final day of the workshop, Safiya demonstrated her programming skills by creating a unique game for Debug 14, which required students to create an adventure, strategy, puzzle, sports or action game with at least one sprite/character. Although a basic starter code for a soccer game with 2 sprites Hope and Eto (See Figure 4-36), was provided, Safiya chose to create her own unique game based on the Lilo and Stitch show. She created an interactive quiz featuring a Quiz master who asked questions and a score meter to track players’ performance (See Figure 4-37, left).

Figure 4-36: Stage (left); Given codes for Hope (middle); Given codes for Eto (right)

To start the game, Safiya wrote two parallelly running codes that were triggered when the Green flag was pressed (See Figure 4-37, right). One code set the score meter to 0 while hiding the ‘Correct’ variable that displayed answers. The other code initialized the quiz, with the Quizmaster giving an opening blurb before the quiz began. To sequence the intro text on screen, Safiya used a combination of look blocks, ‘Say for seconds,’ and control block, ‘Wait seconds.’ She calibrated the appear and wait times of the texts to avoid overlapping text. Safiya successfully implemented parallelism and variables to set up the intro part of the game.
After the quiz intro, players had to answer 10 questions by typing their responses and waiting for the correct or incorrect feedback. Each question was initialized with an Event hat ‘When __ key pressed,’ where the key press corresponded to the question number. For the first question, when the player pressed 1, the question was displayed on screen, and the variable, ‘Correct’ was set to ‘Answer’ (See Figure 4-38, left). Safiya then used an ‘If-then-Else’ conditional to determine the correct or incorrect answer. She also added an operator ‘Answer = Scrump’ for the first question to check the players’ answers against the correct one. If the player responded correctly, the score increased by 1, and a ‘Correct’ message was displayed on the screen. Conversely, if the answer was incorrect, the score decreased by 1, and an ‘Incorrect’ message was displayed (See Figure 4-38, left). The correct answer was displayed for 2 seconds and the players were then prompted to press 2 for the next question. She used the same code for questions 2-10, only changing the Event hat to initialize the code for subsequent questions (See Figure 4-38, right).
For the final question, Safiya added a sequence of look and control blocks that initialized the texts to end the game (See Figure 4-39). Lilo congratulated the player for finishing the game, and Safiya used a look block to display the player’s final score for 3 seconds. She also added an ‘If-then-else’ conditional to display different messages depending on whether the player’s score was greater than 6 or less than or equal to 6. Overall, Safiya’s code demonstrated her proficiency in implementing various computational concepts such as Events, Conditionals, Operators, Variables, and Parallelisms. She utilized these concepts effectively to create a game that was not only functional but also fun and engaging for the players.
Safiya’s road to developing the Lilo and Stitch game, however, was not easy. She encountered challenges when writing the code for questions in her game, specifically when combining computational concepts to write conditions for score changes based on correct and incorrect answers. She had difficulty setting the variable ‘Correct’ to ‘Answer’ and writing the if-then-else bracket for the questions (Table 4-15, Turn 6). Despite having a theoretical understanding of the concepts, she found it challenging to translate them into functional code (Table 4-15, Turn 8). These difficulties highlight a common gap between novice programmers’ knowledge of computational concepts and their ability to apply them practically. Prior research has shown that novice programmers tend to have surface-level knowledge, lacking a robust understanding of programming constructs or computational concepts to apply to a given problem (Winslow, 1996). They also tend to approach programming problems with a concrete and line-by-line code sequencing mentality, rather than starting at the system level to understand the problem’s components and then move on to writing the code scripts (Soloway & Spohrer, 1989).

Figure 4-39: Question 10 code with conditions for score changes and win-loss condition
Novice programmers also struggle with understanding and applying computational concepts to diverse problems and have deficits in planning, testing, and considering potential errors before writing the code (Soloway & Spohrer, 1989).

However, with the guidance of the facilitator, Safiya was able to overcome the challenges and implement the necessary computational concepts effectively to solve the problem (Table 4-15, Turn 6). Safiya’s development of the Lilo and Stitch quiz game highlighted the challenges that novice programmers face when implementing computational concepts to debug problems. Her difficulties with combining concepts and approaching emergent problems from a system level are common challenges faced by novice programmers. My findings highlight the importance of building robust foundational knowledge of computational concepts and problem-solving skills in novice programmers to help them overcome these challenges and bridge the gap between knowledge and practical application. These findings also reinforce the importance of tailored scaffolding, guidance, and practice to support planning and conceptualizing problems from a systems level and effectively apply computational concepts to diverse programming problems.
Chapter 5
Discussion

In this chapter, I will provide a detailed discussion of the findings and the implications of those findings. This DBR mixed methods study explored the potential of Productive Failure (PF) pedagogical design to support novice learners’ programming learning and understanding of fundamental computational concepts (Kapur, 2008). The aim of this study was also to contribute to the emerging body of work focused on the effective implementation of Productive Failure learning designs with diverse open-ended problems in informal learning environments (Fields, Jayathirtha & Kafai, 2019; Litts et al. 2016; Searle, Litts & Kafai, 2018). Additionally, the larger aim of this study was to investigate learning designs that support skill building, engagement, and interest in STEM for diverse youths.

I report on the findings from an online synchronous workshop designed around PF design where 10 high school students engaged in code debugging activities without scaffolding followed by a consolidation phase. Four theoretical frameworks behind the study were: a) Constructivism to understand how students learn by actively engaging with the PF design and solving debugging problems, b) Heuristics and c) Metacognition to understand students problem solving and debugging strategies and d) Failure Assisted Cognition to understand how suboptimal solution generation and failure supported learning of computational concepts. This study answered three research questions: a) RQ1: How does productive failure design support learning of computational concepts when working with open-ended debugging problems in programming platforms such as Scratch? b) RQ 2.1: What kinds of challenges do students encounter, the resolutions they come up with and the debugging strategies they used when working through the debugging curriculum? c) RQ 2.2: What do these challenges, resolutions, process and interaction patterns reveal about students’ evolving understanding of computational concepts?

Discussion of the research findings are provided below.
Improved Programming Attitude, Self-Efficacy and Application Accuracy of Computational Concepts

Findings showed that students' attitude towards programming and programming self-efficacy improved with Productive Failure Design. Additionally, students also reported improved learning and understanding of computational concepts. A Wilcoxon Matched Pairs Signed-Ranks Test indicated that post-test scores were statistically significantly higher than pre-test scores, $z = -10.52$, $p < 0.001$, $r = 0.35$ with medium effect size ($r > 0.3$). Moreover, the post-test scores were statistically significant for all three subscales assessing computer programming self-efficacy, attitudes towards computer programming learning and student evaluations of their problem-solving abilities. All items on the pre-posttest were scored on a 6-point Likert scale with ratings ranging from 1 Strongly Disagree to 6 Strongly Agree. Post-test scores were statistically significantly higher than pre-test scores for Computer Programming Self-Efficacy (CPSES), $z = -9.25$, $p < 0.001$, $r = 0.58$, with a large effect size ($r > 0.5$). CPSES assessed programming knowledge and skills with items that specifically evaluated students’ self-efficacy in understanding and applying computational concepts. Similarly, post-test scores were statistically significantly higher than pre-test scores for both Attitude Scale of Computer Programming Learning (ASCOPL), $z = -4.83$, $p < 0.001$, $r = 0.28$, and Problem-Solving Inventory (PSI), $z = -2.12$, $p = 0.034$, $r = 0.11$, albeit with a small effect sizes ($r > 0.1$). However, post-test scores were not statistically significantly higher than pre-test scores for Approach Avoidance Style within PSI, $z = -1.00$, $p = 0.317$, $r = 0.07$ indicating a less than small effect size. This indicated that while some students better mitigated their difficulty when confronting programming and problem-solving challenges following the workshop, others struggled with tendencies to avoid problem solving which may have impacted their programming learning and understanding of computational concepts.

The findings indicated improvements in students’ confidence to program and belief in their ability to accurately apply computational concepts after participating in the workshop. From pretest to post-test, students self-reported ratings of their programming knowledge and ability in CPSES increased...
from a median rating of 2 (Disagree) to a median score of 5 (Agree). Statements related to application of computational concepts such as operators (I know what the operators +, -, *, /, >, <, = mean in programming and how to use them), loops (I know how to use loops instead of repeating the same code over and over) and conditionals saw improved student ratings of 4 (Slightly Agree), 5 (Agree) and 6 (Strongly Agree). Similarly, ratings on statements related to the ability to debug problems, decompose problems into amenable chunks and improve code efficiency also saw marked improvements. However, there were also areas where students’ confidence in their ability remained constant. Specifically, ratings on statements related to testing the codes, and explanation of steps towards an optimal solution remained the same. For instance, students rated 2’s (Disagree) and 3’s (Slightly Disagree) on statements such as I can test and operate the program I developed and I can explain my idea for a computer program step by step.

Confidence in programming, and improvements in self-reported programming self-efficacy was also apparent in the students Think Alouds and Online interview responses. Students expressed increasing confidence in their ability to program and accurately apply computational concepts during later sessions 3-6 when compared to initial sessions. For instance, during the initial two sessions, Jamarcus, and Willow both expressed confusion and frustration with finding correct code blocks and correctly applying conditionals and sequencing their codes. During the final day of the workshop, both students reported increased confidence in their abilities to write optimal codes to debug programming problems and precisely use sequences and conditionals. My findings indicated that familiarity with Scratch contributed to increasing confidence and reported self-efficacy in programming. Familiarity with Scratch and code blocks contributed to successful debugs with fewer iteration and failures which added to improved confidence in students’ ability to program and their perception of their programming self-efficacy. Additionally, solution-generation effect (i.e. more solutions = learning from failed solutions = more learning) which is well established in Productive Failure literature (Kapur, 2008, 2015) also supported improvements in programming self-efficacy and confidence although initially, it resulted in confusion,
frustration and feelings of inability. I discuss the solution-generation effect and its role in improved application accuracy and understanding of computational concepts in the next section.

Similarly, findings also indicated that students’ attitude to programming also improved where they were able to manage initial difficulties in programming, negative emotions during failures and mitigate problem avoidance. While students were able to manage negative emotions and some even reported programming and debugging as fun, engaging, and supporting curiosity, affective challenges still persisted. From pretest to post-test, students self-reported ratings of their programming knowledge and ability in ASCOPL increased from a median rating of 4 (Slightly Agree) to a median score of 5 (Agree). Statements related to willingness (i.e. I am confident I can learn programming, and Given the chance, I would participate in computer programming courses) and necessity of programming (i.e. Programming is an important skill to have, and Programming improves your problem-solving skill) saw improved ratings from 3 (Slightly Disagree) to 4 (slightly Agree) and 5 Agree). Similarly, students disagreed with statements related to negative attitudes towards programming (i.e. Taking a programming course is a waste of time for me, and It does not matter for my future to be successful in programming). Qualitative data showed similar findings with Willow, Delani and Jamarcus reporting willingness to program and showing persistence in iteratively working towards optimal solutions despite multiple failures. However, affective challenges such as confusion and frustrations about the debugging problems, and decision paralysis still remained. These affective challenges were expected with novice programmers who had no prior experience with programming, debugging and Scratch and who reported low confidence in their ability to write functional programs, and solve programming problems. Additionally, implementation of Productive Failure Design also meant that students were thrust into solution generation without prior instruction and scaffolding which added to confusion, frustration, and feelings of low efficacy.

My findings were similar to prior research that focused specifically on the affective experiences of novice programmers. Similar to D’Mello (2013) and Bosch and D’Mello’s (2017) findings, I found that confusion and frustration were dominant affective states faced by students resulting from unresolved impasses during debugging, multiple failure attempts, and absence of optimal solution due to knowledge
gaps in students programming knowledge. However, while prior research findings reported that affective states such as boredom were frequent, my findings showed that boredom was not frequently reported as an affective challenge. Rather, decision paralysis in modifying and remixing scripts due to fear of not achieving operational solutions after multiple attempts and feelings of confidence hit and self-efficacy were reported frequently. Additionally, while D’Mello (2013) and Bosch and D’Mello’s (2017) found that positive affective states were rarely reported by novice programmers, in my findings, some students frequently reported positive affect such as engagement, fun, and curiosity in addition to the frustration, confusion and disappointments of unsuccessful debugging attempts. Safiya, Delani and Willow reported that while they were frustrated when their solutions did not work out, they were also persistent and engaged in finding ways to debug the problems (See Table 4-10). Similarly, Jamarcus reported temporary frustration and disappointment with failures but correlated these unsuccessful attempts as fun puzzles and error outputs as opportunities to learn accurate applications of computational concepts and test unique script sequences. Similar to impasse-driven learning studies (Newell, 1990; VanLehn, 1988), my findings showed while frustration, confusion and disappointment were pronounced after multiple failures leading to temporary hindrance in debugging performance, these affective challenges also supported engagement with the debugging problems. Similarly, for some students, initial disappointment and frustration were also coupled with a sense of fun and curiosity that reinforced persistence with solution generation, improved participants’ code construction and learning of computational concepts.

In summary, findings showed that students' interest and willingness to program and self-efficacy in applying computational concepts improved with Productive Failure Design. However, pre-post scores on Approach Avoidance Style subscale indicated that while some students mitigated their difficulty when confronting debugging challenges, others struggled with tendencies to avoid problem solving which impacted learning. In these instances, within-task feedback in addition to affective support can aid changes in problem solving approaches and persist past failures.
Learning Supported by Productive Failure Design: Solution Generation Effect

The primary finding of the study showed that the Solution Generation Phase of the Productive Failure design supported learning of programming and fundamental computational concepts for novice programmers. However, the Solution Generation phase was frustrating for novice programmers who had to initially rely on trial and error to debug the problems. This resulted in students spending a significant amount of time on figuring out optimal solutions for the initial debugging problems which in turn hindered solution generation attempts for subsequent problems that targeted advanced computational concepts such as parallelisms, operators, and data. As such, learning, understanding and application accuracy for these advanced computational concepts suffered.

The Solution Generation phase was effective for students to explore the functionalities of the code blocks and figure out the correct application of computational concepts. In the absence of direct instruction and scaffolding, and prior experience with programming, trial and error became the go to informal heuristics applied by the students in their attempts to generate optimal solutions for the debugging problems (Tversky & Kahneman, 1974). My findings showed that trial and error was effective for novice programmers as an initial strategy to explore code block functionalities, learn accurate application of computational concepts, and figure out optimal script sequencing to debugging problems. However, the strategy was frustrating with a significant time sink to be viable for optimal solution generation and to support long-term knowledge and skill development in programming. Similarly, anchoring bias (relying on the first piece of information, set of codes) and availability bias (overestimating the significance of available information on problem solving) hindered rapid optimal solution generation (Tversky & Kahneman, 1974). As such, 7 out of 10 students did not attempt or complete on average 7 of 14 debugging problems with only 3 students completing on average 11 of 14 problems (See Figure 4-30). Problems targeting advanced complex computational concepts such as operators, variables and conditionals were not attempted affecting learning and understanding of these computational concepts.
Additionally, Case II findings showed that iterative attempts at debugging problems during the Solution Generation phase contributed to improved programming self-efficacy and application accuracy of some computational concepts. Application accuracy of computational concepts was apparent in the frequency of recurring error patterns as students progressed through the debugging curriculum. Specifically, frequency analysis of recurring error patterns showed marked reductions in general bugs and code smells related to sequences, loops, parallelisms, and events (See Table 4-5). Frequency of general bugs such as use of incorrect code blocks and missing or uninitialized parallel actions, loops, events decreased from Debug 1-14. Additionally, occurrences of code smells related to Sequences (Misordered codes, Sequential action), loops (Loop Sensing) and other code quality issues such as Redundant codes, Timing and Synchronization also decreased significantly.

In addition to the frequency analysis, improvements in computational concepts were also apparent in students’ debugging solutions showing mitigation of frequently occurring error patterns such as code smells, syntax errors and general bugs. Additionally, the built-in feedback in Scratch supported application accuracy of computational concepts by aiding error location and code refinement. When students ran their code, Scratch let students know specific sprite actions that weren’t executed thus providing hints towards missing code and computational concepts that they could apply. However, my findings also showed that the occurrences of error patterns related to conditionals, variables/data and operators still persisted showing students continued struggle with advanced computational concepts. Specifically, occurrences of unused or uninitialized variables and operators increased from Debug 1-14 and frequency of missing and incomplete Conditionals stayed constant (See Table 4-5).

My findings showed that the Solution Generation phase supported learning of computational concepts by encouraging novice programmers to iteratively create and optimize their solutions to debugging problems. This Solution-Generation effect on learning is well documented in Productive Failure literature with canonical STEM problems (Kapur, 2008, 2015); and open-ended design problems with e-textile wearables (Litts et al. 2016; Searle, Litts & Kafai, 2018; Fields & Kafai, 2021). Application accuracy and understanding of computational concepts improved when novice programmers generated
sub-optimal solutions experimenting with different code blocks and sequencing combinations, encountered failures, and learned from these failures to optimize their solutions and debugging strategies.

However, struggles with advanced computational concepts such as Boolean logic (operators), variables and conditionals persisted. While novice programmers were still able to generate sub-optimal solutions, these concepts are more abstract, and required understanding the relationships between computational concepts and using these concepts in combination in their code. Other studies have found similar results where Boolean logic, variables, parallelisms, and conditionals required the ability to see and understand code structures, relationship between computational concepts as well as additional instruction, modeling, and practice (Meerbaum-Salant, Armoni & Ari, 2010; Grover & Basu, 2017; Baytak & Land, 2011).

Learning Supported by Productive Failure Design: Impact of Facilitation on Debugging Strategies and Conceptual Understanding

Learning of computational concepts was also supported by facilitation during the Consolidation and Knowledge assembly phase of the Productive Failure Design. I, as the facilitator, provided feedback on student solutions highlighting areas of conceptual understanding and problems in their codes. Findings indicated that facilitator feedback during the Consolidation and Knowledge phase supported code optimization and application accuracy of computational concepts by encouraging student reflections on their solutions and changes in debugging strategies. Students reported the effectiveness of these feedback in helping them streamline their own solution generation and improve their debugging process. However, findings also indicated that on-time feedback is needed earlier during solution-generation for novice programmers struggling conceptually and procedurally.

Facilitator feedback and subsequent reflections on their solutions brought students’ attention to limitations of their debugging strategies, specifically with trial and error. Some students changed their debugging strategies from time consuming strategies like trial and error to purposeful error locating
strategies such as forward and backward reasoning and objective focused strategies like examining the forward and backward reasoning (See Table 4-11). Changes in debugging strategies also contributed to mitigate recurring errors and optimize future solutions. Additionally, students learnt from optimal solutions shared by the facilitator. By comparing their solutions with the optimal one, students were able to identify code application and sequencing issues to improve application accuracy of computational concepts.

Case III findings showed an instance of how facilitator feedback supported Safiya’s understanding of conditionals and container blocks. Facilitator feedback also contributed to changing her debugging strategy from trial and error, to forward reasoning. Changes in debugging strategy was also apparent in students’ subsequent solutions. Safiya continued using forward reasoning for subsequent problems, generating optimal solutions for Debug 9 to 10, and creating two playable Lilo and Stitch quiz games.

While feedback during the Consolidation and Knowledge Assembly phase was helpful to support changes to better debugging strategies and programing learning, findings also indicated that early and on-time feedback was necessary earlier, during the Solution Generation phase, for novice programmers. Some students who were struggling with debugging problems conceptually and affectively. For instance, in one of the think aloud exchanges during the debugging activities, Willow expressed her frustration during the Solution Generation phase. She explained that she froze up frequently when solving Debug 9 and did not want to add code blocks that did not solve the problem. She was on the verge of giving up and asking for help when she eventually found the Sensing Boolean block ‘Touching Jordyn.’ She added the Sensing Boolean block inside the ‘If-Then’ container along with the motion block ‘Glide 1 secs to X: 223 y: -45’ to complete the conditional that would glide the football sprite towards goal if Jordyn touched the ball (See Table 4-10). Novice programmers continued to struggle with debugging problems especially with problems (Debug 7-14) targeting advanced computational concepts such as conditionals, operators, and variables.
Providing support structures early on during solution generation may support novice programmers in mitigating affective challenges to persist past failures and timely change in inefficient debugging strategies. Specifically, within-task feedback on code block functionalities, clarifying computational concepts and providing on-time feedback on students generated solutions can also support programming self-efficacy and application accuracy of computational concepts. Additionally, early support structures may also initiate early shifts to better debugging strategies that may support optimal solution generation and conceptual learning.

**Challenges Endemic to Novice Programmers**

My findings showed that the challenges faced by students when working on the debugging problems were native to novice programmers. Three error patterns were prominent a) syntax errors b) code smells and c) general bugs. While Scratch was designed to eliminate syntax errors that plague text-based programming, there are exceptions. Common syntax errors in visual based programming languages like Scratch included using incorrect reporter blocks inside a container block, missing termination condition, and missing conditions in conditional container blocks. My findings showed that the most common syntax errors included termination conditions and missing or incomplete conditionals. Student solutions also showed the presence of code smells such as sequential actions, long scripts, redundant/duplicate codes, timing and synchronization, and dead codes. General bugs such as missing or uninitialized loops, events, conditionals, and variables and interrupted loop sensing were frequent in student solutions. Additionally, student solutions also included recurring code smells such as sequential actions, long scripts, redundant/duplicate codes, timing and synchronization, and dead codes. Code smells are hints in the code that indicate problems with code quality and optimization (Beck & Fowler, 1999; Fraser et al. 2021). Smelly codes can be operational and are technically correct but the code is not efficient.

Recurring occurrences of syntax errors, general bugs and code smells indicated gaps in students’ understanding and application of computational concepts as well as over-dependence on informal
heuristics such as trial and error and anchoring bias. On the one hand, the frequency of these error patterns contributed to frustration, decision paralysis, feelings of low confidence and self-efficacy, disappointment, and discouragement-affective challenges that at times hindered student progress and enjoyment of programming and debugging activities. However, on the other hand, my findings also showed that these recurring errors also contributed to a sense of fun, curiosity and challenge and supported learning through persistent solution generation.

**Implications of Findings**

**Design Implications: Provide support structures early to support long term programming**

My findings suggest a need to provide structured support structures for novice programmers during the Solution Generation phase in addition to scaffolds provided during the Consolidation and Knowledge Assembly phase. I argue that providing support structures early on during solution generation may support novice programmers in mitigating affective challenges to persist past failures and timely change in inefficient debugging strategies. Specifically, within-task feedback on code block functionalities, clarifying computational concepts and providing on-time feedback on students generated solutions can support programming self-efficacy and application accuracy of computational concepts. As evident in the findings, being able to solve a problem with facilitation after struggling through multiple failures had a significant impact on these novice programmers' intention to persist in the curriculum and learn programming. Helpful hints and guiding prompts for nearly there solutions also helped students debug the problems at hand and steered students away from recursive trial and error attempts towards efficient error location and debugging strategies. Feedback that is positive, timely, non-evaluative and actionable has been shown to ease affective challenges (Nordquist, 2007; Shute, 2008) and promote positive experiences such engagement, feelings of enjoyment and self-efficacy (Lee & Ko, 2011; Mitrovic et al. 2013).
While this revision deviates slightly from Kapur’s (2008) original conception of Productive Failure design with problem-solving preceding facilitation, it does not minimize or discount the solution-generation effect or the benefit of failures on student learning (Kapur, 2008, 2015). However, this revision challenges Kapur’s (2011) findings where he concluded that students assigned to the Productive Failure condition outperformed students assigned to Guided-Generation and Direct Instruction conditions on conceptual understanding and procedural knowledge. While these findings might hold true for mathematics and science problems with canonical solutions, my findings showed that on-time feedback during solution generation improved conceptual understanding and application accuracy of computational concepts such as sequences, loops, events, and parallelisms. Additionally, in open-ended programming problem settings absent a canonical solution and with multiple pathways to “optimal” solutions, it may be necessary to provide scaffolds to draw novice programmers’ attention to recurring code smells and bugs, conceptual errors, and inefficient programming practices early on. Providing opportunities for novice programmers to ask clarifying questions in instances of decision paralysis, frustration and conceptual ambiguity can contribute to students’ feelings of programming self-efficacy and sustain their programming interests and learning long term.

Remote Learning and Delivery of Online workshops

This study provided insights into remote learning especially for vulnerable learners and delivery of online STEM focused workshops. The 3-week online synchronous code debugging workshop was administered during the Covid-19 pandemic and was designed to support participants in federal programs for low-income families. The workshop presented opportunities and challenges that can serve as considerations for scholars interested in understanding the dynamic aspects of remote learning and delivery of online technology mediated workshops.

First the workshop represented one of the few remote learning opportunities – and the only enrichment opportunity - readily available for TRiO participants. Pandemic restrictions, low-income background, limited technology access, and PA rural schools’ lack of resources to pivot rapidly towards
creating and delivering remote extracurricular activities had limited these students to just attending online schools. By delivering Google Chromebooks with necessary software installed, using open-access programming platform (Scratch) and redesigning the workshop to be delivered remotely, this online synchronous code debugging workshop mitigated these issues to provide a space for community building, learning and enrichment. My findings suggest that it is critical for researchers to understand and plan for limiting factors (technology access, cost, modality, time) and also be mindful of equity, participants background and access before delivering remote learning opportunities. Remote learning can be accessible but without understanding and working to mitigate access, equity and technology issues, remote learning can also promote inaccessibility and inequitable participation (Bansak & Star, 2021; Katz & Rideout, 2021).

Second, the workshop provided insights into realities and complexities of student participation that scholars may need to consider when designing and conducting remote workshops. 3 rising seniors, Damari (Latinx, 18 yrs.), Jonah (Latinx, 16) and Jamarcus (Latinx, 16), all had jobs and were on call throughout the week leading to some absences. At times, these students logged in to the workshop Zoom from their phones in their work parking lots or their cars. However, they were not able to simultaneously engage in debugging activities on Scratch and participate in Think Alouds and Consolidation phase on Zoom as they did not have a secondary device to access Scratch app while attending Zoom from their primary device (phone). This was not an issue for students with Google Chromebook logging in from homes who could switch between Zoom and Scratch. As a result, despite a high level of interest from students, full participation in the workshop and learning activities were curtailed. Similarly, Willow (Latinx, 16), Jonah (Latinx, 16) and Delani (Black, 16) were on childcare duty supporting their working parents. During workshop time 5-7 pm, students were also helping parents with dinner and other household chores. Realities of work, family, and technology added another level of complexity to being fully engaged in the remote programming workshop activities and contributed to non-completion of some debugging problems. Understanding these realities, in the final two sessions, these students were given
opportunities to work on the debugging problems before the next session and email the facilitator (Author) with questions for asynchronous feedback on debugging problems.

From a design and implementation perspective, researchers must extend remote learning environments around these realities rather than controlling for it and risk promoting inequitable participation and inaccessibility. Providing opportunities to access workshop activities, software, technological devices outside of the workshop space and during absences may support sustained participation in remote learning, additional learning opportunities and also mitigate some negative effects of distractions and access on performance (Gottfried, 2011). Additionally, adaptive teaching practices such as pre-recorded workshop materials (Pilkington & Hanif, 2021), asynchronous feedback (Litts et al., 2016) and technology training and preparation for using digital tools (Bozkurt et al., 2020) can also support motivation and mitigate digital inequities and inequalities for vulnerable learners and disadvantaged communities.

Theoretical Implication: Productive Failure Design in Open-Ended Problem Settings

My findings suggest a need to reconceptualize productivity of failure during the Solution Generation phase and reframe the latent efficacy in unscaffolded problem solving process. I argue that recurring failure can hinder long term learning, conceptual understanding and building self-efficacy if left unchecked. Specifically in diverse open-ended problem settings, with multiple pathways to an “optimal” solution but without an established canonical solution, recurring failures may promote deficit frames, affective challenges, and inequitable participation especially for vulnerable learners (low-income, migrant, and marginalized).

I argue that failure in its current conceptualization in the Productive Failure literature does not take into consideration a body of work cautioning against normalizing/celebrating failure and minimizing the negative aspects of failure. Persistent failures can lead to learned helplessness (Seligman & Maier, 1967), and negative affect that hinder present and future learning and performance. Additionally, recurring failure can promote deficit frames among individuals from low-income and historically
marginalized communities (Vossoughi & Bevan, 2014; Vossoughi & Vakil, 2018). The historical association of failure with non-resilient behaviors and practices, the social stigma attached to the word coupled with stereotypes and restricted access to STEM for historically marginalized populations also aid negative affect (lower self-confidence) and maladaptive attribution patterns of failure (Cooper, 2006; Li & Kirkup, 2007; Papastergiou & Solomonidou, 2005; Richard, 2015, 2017; Smith, Morgan & White, 2005). Additionally, the opportunity to fail and try again is a privilege and the consequences of failure are different for individuals from different socio-economic, racial, and ethnic backgrounds (Eason, 2014).

Findings in this study indicated a need to reconceptualize failure during the Solution Generation phase as a means to initiate independent problem-solving and conceptual engagement but not as an end to learning and conceptual understanding. Failures during the Solution Generation phase supported some novice programmers’ interest and willingness to program, procedural fluency, and improved self-efficacy in applying computational concepts (Kapur 2008; Kapur & Bielaczyc, 2012; Litts et al. 2016). However, recurring failures also resulted in problem avoidance, feelings of low self-efficacy and confidence in their ability to learn programming. While some students mitigated their difficulty when confronting challenges, others did not feel fully confident in some aspects of their abilities veering instead towards problem avoidance. Additionally, absence of scaffolds and exclusive reliance on trial and error also hindered understanding and application of abstract computational concepts (conditionals, operators, variables). To mitigate these affective challenges and deficit frames, feedback and scaffolds were provided during the Solution Generation phase to engage students conceptually and provide helpful hints for students with recurring failures. Findings showed that recurring failures when monitored and addressed through support structures aided students’ comprehension of computational concepts and effective future troubleshooting efforts.

Based on my findings, I reframe failure as independent exercises in problem-solving that novices can engage in to support knowledge building but also a process that must be monitored and supported to support long term learning and interest development. This reframing can contribute to the field and emerging body of research (Fields, Jayathirtha & Kafai, 2019; Litts et al. 2016; Searle, Litts & Kafai,
2018) on Productive Failure learning designs in informal learning environments in two ways. First, this framing recognizes that failure is necessary but not a sufficient condition for knowledge building and learning. It allows for learner agency in problem-solving. But this framing also encourages support structures to be designed to monitor and scaffold failures for learners of various skill levels so that recurring failures does not lead to or promote internalization of failure and maladaptive attribution patterns. Specifically for complete novices, design of support structures that engineer conceptual engagement with the facilitator to think through their solutions and approaches may be beneficial for learning. My findings and prior studies also showed that novices required instruction, modeling and practice for effective application and understanding of computational concepts (Meerbaum-Salant, Armoni and Ari, 2010; Grover & Basu, 2017; Baytak & Land, 2011).

Second, this framing does not normalize failure. This reconceptualization encourages scholars to explore revisions to Productive Failure designs that can support learning especially for marginalized learners. It is especially crucial for Productive Failure designs for open-ended problems as these work support interest and skills development in STEM fields where marginalized learners have faced historically and socially sustained stereotypes and stereotype threat (Beede et al. 2011; Cooper & Weaver, 2003; National Science Foundation, 2021). This reconceptualization also calls on scholars to design learning activities and scaffolds attuned to the needs and experiences of marginalized communities who are vulnerable to negative affect of failures, power relations in learning and inequitable participation (Burke & Mattis, 2007; Koch, Müller & Sieverdung, 2008, Richard, 2017).
Chapter 6

Conclusion

Findings show that Productive Failure learning designs can introduce frustration and complexity that, when well designed and reinforced, can aid in learners’ comprehension of computational concepts and effective future troubleshooting skills. Specifically, I found that students’ interest and motivation to learn programming, as well as their self-efficacy in applying computational concepts, were positively impacted by the Productive Failure Design. However, it was also evident that while some students were able to effectively manage and overcome challenges, others may not have felt fully confident in some aspects of their abilities and instead tended toward problem avoidance. The findings also highlighted aspects of the Productive Failure design that supported learning and understanding of computational concepts. Specifically, the use of iterative solution generation and trial and error as an initial debugging strategy was found to be effective during the Solution Generation phase in learning code block functionalities, sequencing of scripts, and improving application accuracy of some computational concepts. However, relying solely on trial and error as the primary debugging strategy was time-consuming and resulted in recurring code smells and bugs. Novice programmers also struggled with integrating coding concepts, optimizing, and scaling code. While some students such as Tuan, Delani, Jonah, Willow, and Jamarcus adopted a more structured top-down approach decomposing problems into smaller units, testing incrementally, and avoiding poor practices, many programmers stuck with the bottom-up approach despite its drawbacks. To encourage students to adopt a structured top-down programming approach focused on solving problems from an algorithmic, design and system level, scaffolds need to be developed that promote good programming practices, optimize code quality, and establish a solid foundation in computational concepts and problem-solving skills to bridge the gap between knowledge and practice.
Novice programmers need structured support during the Solution-Generation phase to address emotional challenges, improve debugging strategies, and programming skills. Similarly, timely feedback and guided facilitation after failures improve self-confidence while debugging prompts can support transition from trial and error to structured debugging strategies. Additionally, support structures are needed so that failure is not unchecked. Recurring error patterns and affective challenges need to be addressed and mitigated especially for vulnerable learners to support interest and skill development in STEM. Additionally, effective error location and debugging strategies may need to be modeled and scaffolded during the Solution Generation phase to support deeper conceptual engagement with fundamental programming concepts. While this proposed approach to provide structured scaffolds and on-time feedback during Phase I differs slightly from Kapur's (2008) Productive Failure design, it still recognizes the value of unscaffolded iterative attempts and failures and in learning. However, my findings challenge Kapur's conclusion. In programming problems without a single correct solution, providing scaffolds is necessary. Scaffolds address code issues, conceptual errors, affective challenges, and inefficient practices to support long-term learning. Therefore, ongoing work will be focused on designing optimal support structures for novice learners.

Moreover, the workshop provided insights into remote learning especially for vulnerable learners and delivery of online STEM focused workshops. The workshop represented one of the few remote learning opportunities and the only enrichment opportunity available for vulnerable learners during the pandemic. My findings emphasized the need to consider factors such as technology access, equity, and participant backgrounds when delivering remote workshops. Challenges with student participation, such as work obligations, students’ childcare responsibilities to support working parents, and limited device access, impacted engagement and sustained participation in workshop activities. To support inclusive and equitable participation, researchers should accommodate students’ realities and provide additional support, such as access to materials and asynchronous feedback. Adaptive teaching practices and technology training can be provided to address digital inequities.
Overall, the findings suggest that Productive Failure Design can be an effective approach to teaching computational concepts when coupled with scaffolds and the use of different debugging strategies. The design allows for trial and error, tinkering, exploration, and experimentation, but it is essential to develop scaffolds that promote good programming habits, incremental development and focus on code optimization. By combining different approaches to error location and resolution, novice programmers can develop a deeper understanding and application of fundamental computational concepts, ultimately improving their accuracy and efficiency in coding.

Limitations

One of the primary limitations of this study was the online modality and technology. While redesigning the workshop to deliver it fully remote provided a unique opportunity to provide learning and community building space for students feeling the effects of isolation and monotony of online class, there were challenges in implementation of the study. Specifically, logistical challenges affected data collection and quality of data at times. Breakout room feature in Zoom could not be implemented during the Solution Generation phase to conduct Think Alouds due to personnel issues. As such, Think Alouds during Solution Generation were conducted in the primary Zoom room where students shared their screens to show their and explained their solutions and debugging process individually while other students in the same room worked on the debugging problems. This arrangement may have contributed negatively to data quality because it opened up opportunities to cheat. While I did not find evidence of cheating, I acknowledge that students’ solutions may have been influenced by their peers which could have resulted in them revising their solutions to debug the problem at hand. In a future study, I plan to investigate similarities and differences in students' solutions to analyze the impact of open Think Alouds on students’ learning and application accuracy of computational concepts.

Additionally, carryover effects cannot be discounted during the Solution Generation phase (Lin & Lin, 2014). Specifically, findings showing mitigation of recurring errors in later debugs and improvements in application accuracy of computational concepts such as sequences, loops, parallelisms,
and events could have been influenced by practice effects. There is a possibility that students may have performed better in later debugging problems just because they had a chance to practice different sequencing combinations and code blocks through iterative attempts during the Solution Generation phase. Similarly, fatigue effect may have also influenced students’ performance during the Solution Generation phase where they performed worse in subsequent iterative attempts because they were tired and unmotivated as a result of recurring failures.

Similarly, learner centered challenges and inequitable access to technology exacerbated in marginalized communities and uncovered by the pandemic also impacted participation and completion of all workshop activities (Bansak & Starr, 2021; Beaunoyer et al., 2020; Hall et al., 2020). All of the participants were from low-income families who did not have access to high-speed internet. Unstable connection, bad audio, and video quality in addition to video-audio lags and echo impacted participation, completion of debugging problems and at times quality of Think Alouds and discussions during Consolidation phase. Additionally, students' unpredictable work schedules, helping parents at home with chores and sibling care also affected full participation in workshop activities resulting in attrition with 6 participants leaving after the first session and some students not attempting or completing all 14 debugging problems. Attrition impacted significance levels of Wilcoxon signed ranked test and non-completions impacted deeper analysis of students shifts in learning and understanding of computational concepts.

Additionally, data quality was also impacted by practices in place to safeguard participants’ privacy. During the workshop, student videos were turned off as non-consented minors may be present. Students were at times Zooming in from public places. This privacy practice also extended to Think Alouds and Consolidation sessions, where simultaneous video and audio recording was not permitted. This meant that students would share their screen and show their solutions and current progress. They would then turn off screen sharing to explain their debugging approach, challenges, and solutions. This impacted transcription and artifact analysis as students' explanations and solutions were not in sync as they went back and revised the solutions, they explained during Think Alouds and added more solutions.
Future Work

Future work on supporting novice programmers in learning programming and improving application accuracy of fundamental computational concepts will be focused on designing optimal support structures such as progress monitoring tools, addressing digital inequities faced by marginalized learners, and fostering community support. These efforts aim to track iterative attempts and learning trajectories, provide scaffolds, and offer adaptative feedback to promote equitable learning opportunities for diverse learners, and create a supportive programming community.

Primarily, future work will explore designing scaffolds during Solution Generation to support novices struggling with conceptual and affective challenges. Design of optimal support structures can help students persist past recurring failures but still encourage and promote independent conceptual engagement, solution-generation and problem-solving. In the context of open-ended programming problems, these support structures can include the use of static analytic tools such as Hairball, Litterbox and Quality hound to automatically detect code smells and bug patterns (Fraser et al., 2021) and then provide feedback and hints on code to clear misconceptions and improve code quality (Gusukuma et al. 2018; Rivers & Koedinger, 2017). Other support structures include Intelligent Programing Tutors (IPT) like iSnap for personalized next-step hints for block-based programming environments (Price & Barnes, 2015) and metacognitive support system including an array of supports such as procedural, conceptual, metacognitive scaffolds, reflective prompts, self-assessment, self-questioning, and self-directed strategies (Rum & Ismail, 2017). In the context of open-ended programming problems, I envision a design consisting of two primary components. First, a progress monitoring tool that can track students’ progress in the debugging curriculum including iterative attempts and types of failures (conceptual, code smells, bugs). The progress monitoring tool will also be designed highlight recurring code smells and error patterns in novices code and provide digital scaffolds and adaptive feedback in the form of helpful hints on programming logic, sequencing, and coding concepts to support optimal code construction. Second component would be procedural, conceptual, and metacognitive scaffolds provided by the facilitator based on students' struggles.
Future work will also explore how different designs can facilitate meaningful learning for especially marginalized learners, while also providing space for support and community building, particularly in the context of remote learning opportunities. Findings showed that novice programmers from low-income communities struggled without support structures. Although affective challenges such as confusion, frustration, and decision paralysis encouraged persistence and a sense of curiosity among some students, for many students these challenges negatively impacted their programming self-efficacy, and performance. While some students mitigated their difficulty when confronting failures and managed to iteratively generate and refine their solutions, others were not confident in their programming abilities. These students avoided debugging problems targeting advanced computational concepts and did not refine their debugging strategies beyond trial and error. As such, revisions to the Productive Failure design (Kapur, 2008) are needed to address and mitigate historical negative attributional patterns towards failure while also supporting equitable participation, interest, and skill development in STEM. Specifically, I propose three revisions focused on building adaptive and structured support to aid novice and vulnerable learners programming knowledge and skill development:

- structured support and facilitation will be provided throughout the curriculum as opposed to just during the Consolidation and Knowledge Assembly phase with just affective feedbacks provided during Solution Generation phase.
- scaffolds on coding concepts and optimal debugging strategies will be provided during the Solution-Generation phase to support iterative attempts while mitigating affective challenges from persistent failures. Introduction to coding concepts, Scratch, and debugging strategies will be followed by demos, debugging ill-structured programming problems, and practice sessions.
- adaptive feedback, practice sessions, structured scaffolds on students debugging strategies will be provided during both phases to support transition to top-down debugging strategies. Facilitated discussions during consolidation phase will encompass students with optimal solutions sharing their process and strategies in addition to facilitator’s feedback.
These revisions will be implemented as iterative cycles of the current DBR study with redesigns following each implementation and analysis of the impact of these redesigns on novice learners' programming, understanding of coding concepts, and use of optimal debugging strategies.

References


impacts and mitigation strategies. *Computers in Human Behavior, 111*, 106424


Csikar, L. (2018). Decision Making in the Sciences: *Understanding Heuristic Use by Students in Problem
Solving (Order No. 13419700). [Doctoral Dissertations, Old Dominion University]. ProQuest Dissertations Publishing


That Make Us Smart (pp. 3-34). New York: Oxford University Press.


Gusukuma, L., Bart, A. C., Kafura, D., & Ernst, J. (2018, August). Misconception-driven feedback:


Heppner, P. P., Pretorius, T. B., Wei, M., Lee, D.-g., & Wang, Y.-W. (2002). Examining the generalizability of problem-solving appraisal in Black South Africans. *Journal of Counseling...


Hsu, T. C., Chang, S. C., & Hung, Y. T. (2018). How to learn and how to teach computational thinking:


https://www.newamerica.org/education-policy/reports/learning-at-home-whileunderconnected/


problems: A reconstruction kit for debugging e-textiles. In *Proceedings of the 7th Annual Conference on Creativity and Fabrication in Education* (pp. 1-8).


publications.


technology in computer science education (pp. 156-160).


https://doi.org/10.1007/s11187-011-9412-x


Searle, K. A., Litts, B. K., & Kafai, Y. B. (2018). Debugging open-ended designs: High school students’ perceptions of failure and success in an electronic textiles design activity. *Thinking Skills and


Smith, J. L., Morgan, C. L., & White, P. H. (2005). Investigating a measure of computer technology


children, 23(3), 34-41.


Weintrop, D., & Wilensky, U. (2015, June). 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).


Wilson, A., & Moffat, D. C. (2010). Evaluating scratch to introduce younger school children to programming. In J. Lawrance & R. Bellamy (Eds.), *Proceedings of the 22nd annual psychology of programming interest group* (pp 1-12).


Sagun Giri

EDUCATION
The Pennsylvania State University  
Ph.D. in Learning, Design and Technology  
University Park, PA  
August 2023
Caldwell University  
B. A. in History and Political Science, Pre Law (minor)  
Caldwell, NJ  
May 2012

PROFESSIONAL EXPERIENCE
Instructional Designer and Teaching Assistant  
Workforce Education & Development, Penn State, University Park, PA  
2018-2023
Research Associate  
The Playful Learning Research Group, Penn State, University Park, PA  
2017-2023
Dean’s Graduate Research Assistant  
Avenue PM, Penn State, University Park, PA  
2015-2017
Manager  
Core Complexity Assessments, Cambridge, MA  
2014-2015
Managing Editor  
Behavioral Development Bulletin, American Psychological Association  
2013-2014
Research Associate  
Harvard Medical School and Dare Association Inc. Cambridge MA  
2012-2015

SELECTED PUBLICATIONS

