UNIVERSITY OF CALIFORNIA,
IRVINE

Exploring How Novice Programmers Pick Debugging Tactics When Debugging: A Student's Perspective

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Software Engineering

by

Felix Chan Lee

Thesis Committee:
Professor James A. Jones, Chair
Professor André van der Hoek
Professor Cristina Lopes

2019

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my appreciation to my committee chair and advisor, Professor Jim Jones, for guiding me throughout my years as a Master's student. Thank you for giving me the freedom to pursue different interesting debugging topics and for encouraging my curiosity no matter how strange the subject.

I would also like to thank my committee members, Professor Crista Lopes and André van der Hoek for taking the time to review my thesis and provide feedback. Thank you to the Spider Lab for creating a great atmosphere that motivated me completing this thesis.

Lastly, I would like to thank my mom and sister for all that they have done to help me reach this point in life.

Abstract of the Thesis

Exploring How Novice Programmers Pick Debugging Tactics When Debugging: A Student's Perspective

By

Felix Chan Lee

Master of Science in Software Engineering

University of California, Irvine, 2019

Professor James A. Jones, Chair

Novice developers use a variety of debugging tactics to debug. However, how they select a tactic still remains unclear. Many studies in Software Engineering describe programmers using tactics like adding print statements, but only a few studies hint at factors such as knowledge and habits, social environment, and experience that may influence these decisions. To help us understand how novice programmers select debugging tactics, we turned to Information Foraging Theory (IFT) to analyze this decision-making process through the lens of a cost-benefit analysis. We conducted a qualitative study that explored how novice programmers describe their decision-making process when deciding which debugging tactics to use when debugging. We found that novice programmers use a variety of debugging tactics including testing code, searching for help, and taking notes on paper. Furthermore, we reported activities where novices leverage their past experiences, adapt to their task environments, and anticipate future risks and rewards to decide among a variety of tactics to pursue. From our results, we offer suggestions to educators to explicitly teach the value and costs of using certain tactics so that novice programmers may select the optimal tactic in any debugging situation. Furthermore, we suggest future research to explore novice debugging behaviors within non-computing environments to gain a holistic understanding of the debugging process.

Chapter 1: Introduction

University students taking introductory software programming courses grapple with the difficulties of learning to program and debugging their programs [19]. In parallel, programming educators face difficulties in teaching debugging effectively [15]. Important literature in the debugging education field dating as far back as the 1970's have provided a wealth of knowledge on the debugging processes that programming students follow [15]. Within these models of the debugging process lie an often overlooked subprocess that describes how novice programmers select debugging tactics. Among the many debugging studies that explore programmers' debugging processes, Gould's model explicitly highlights the selection of a debugging tactic as an initial step in the process [8]. In contrast, other studies focus on other aspects of the debugging process and take our understanding of how programmers select their debugging tactics for granted [10][21]. Likewise, subsequent studies that explore the debugging tactics that novice developers use fail to explain why they pick those tactics in the first place. This gap in our understanding contributes to findings where researchers and educators are puzzled by debugging behaviors that novice programmers exhibit, such as when they stubbornly pursue debugging tactics without success and use sub-optimal tactics [5][16]. Without understanding why novice developers use the debugging tactics that they do, educators lack the context to teach debugging tactics effectively to students such that these skills persist long after they complete their courses.

Accordingly, we were interested in how novice developers describe their decision-making process when deciding which debugging tactics to use when debugging. To study this phenomenon and expand on previous studies that have studied the debugging tactics that novice programmers use, we focused on the following two research questions:

- What debugging tactics do novice programmers describe using to debug?

- How do novice programmers describe the activities involved in selecting a debugging tactic?

Chapter 2: Literature Review

This chapter discusses literature in two areas of Software Engineering that informs our understanding of how novice programmers pick their debugging tactics: the debugging education and Information Foraging Theory (IFT) literatures. Because the debugging education literature is sparse in descriptions of how novice programmers select debugging tactics, we leverage IFT as a possible framework for understanding how novice programmers decide among various debugging tactics to use. Pirolli and Card first conceptualized this theory based on Optimal Foraging Theory, which describes how wild animals forage for food in their environments by weighing the cost of foraging in exchange for the value of consuming the food source [18]. While Pirolli and Card use this theory to predict end-user behavior navigating the World Wide Web, others have operationalized the theory to study debugging behaviors successfully [13][17]. We leverage both of these research efforts since they describe and explain the debugging tactics that novice programmers use.

The same debugging tactics that novice programmers use in their Software Development Environments (SDE) appear as enrichment strategies in IFT. These enrichment strategies are a subset of foraging strategies that specifically describe actions where the programmer or "information seeker can deliberately modify the environment to either improve the density of useful material in a patch or speed travel between patches" [13]. Accordingly, debugging tactics where programmers execute programs with added print statements and comments to generate program output are effectively enrichment strategies that create additional useful information in the environment. Rearranging windows closer together in the SDE to facilitate efficient access to different information sources would also be considered an enrichment strategy. That both debugging tactics and enrichment strategies describe the same behaviors motivate us to leverage both subfields of Software Engineering to understand decision-making process novice programmers go through to select their tactics. Informed by findings from both of these subfields, we justify the need for a study to explore how novice programmers make decisions to pick their debugging tactics.

Debugging tactics

In reviewing the debugging education literature, we find that novice programmers use a variety of debugging strategies. These studies describe debugging strategies in two distinct ways: (1) high-level plans that describe how novice developers' approach debugging and (2) debugging tactics or low-level actions within their debugging environments that facilitate their high-level debugging strategies. For example, Katz and Anderson's work describing the process novice programmers use to debug exemplifies the first kind of high-level debugging strategies. They describe novice programmers approaching their debugging problems by using forward and backward reasoning where programmers either simulate their program's execution or work backwards from their program's output to reason the cause of a bug [10]. While this and other similar studies are useful to understand the general approaches that novice programmers follow when debugging, our study focuses on the low-level debugging tactics that facilitate these high-level approaches. For example, we are interested in Benander and Benander's description of how novice programmers produce diagnostic program output, read error messages, and hand trace in their COBOL development environments to debug [1]. Other studies like Romero et al.'s explore both high-level debugging strategies and low-level debugging tactics [20]. However, we focus on debugging tactics, since we can leverage the IFT literature to understand these tactics as a form of enrichment strategies and thereby make sense of how novice programmers use these strategies [13].

Many studies in the debugging education and IFT literatures describe a number of debugging tactics that novice programmers use to debug. In an early study of novice developers' debugging tactics, Benander and Benander report five tactics used to debug COBOL programs: (1) using COBOL debugging verbs to produce output, (2) reading messages from the system, (3) reading error messages, (4) tracing by hand, and (5) asking other programmers for assistance [1]. This study is unique in its reporting of asking other programmers as a debugging tactic and suggesting that collaboration, or lack thereof, between students could influence the choice of debugging tactic that students select.

As SDE's evolved into sophisticated systems, newer tactics that leverage the new features in the SDE emerged. Fitzgerald et al. describe how novice developers use tracing in several ways, including mentally tracing the code, tracing on paper, tracing with print statements, and tracing with the breakpoint debugger. Their study also reports novice programmers using external resources like JavaDocs, rewriting code, and testing their code with sample input [5]. In addition to these same tactics, Murphy et al.'s study also report tactics where novice developers add comments and apply the undo button in their SDE's to debug [16]. We find similar debugging tactics from the IFT literature in the four domains of enrichment strategies that Piorkowski et al. report: (1) searching for information using the SDE's code searching features; (2) writing a to-do list on a piece of paper; (3) using an online search engine to look for information; and (4) testing, which involves manipulating the code and running it to create new sources of information in the environment to facilitate debugging [17].

Below, we give an overview of the low-level debugging tactics found in the existing literature that our debugging study focuses on:

- generate program output (i.e. print statements) [1] [5] [7]
- tracing by hand [1] [5]
- asking other programmers for help [1]
- searching for external resources (i.e. documentation) [5][7]
- rewriting code [5]
- testing code with sample input [5]
- adding comments [6]
- applying undo button in SDE [6]

- searching for code with SDE [7]
- writing a to-do list on paper [7]

Selecting debugging tactics

In addition to exploring debugging tactics, we also leverage both fields of Software Engineering to gain a broader understanding of the decision-making process that novice programmers apply when selecting debugging tactics. Both fields are saturated with studies that describe debugging tactics novice programmers use within their SDE's. Yet, the rationale behind why novice programmers employ those tactics or how they come to select those tactics remain unclear. As McCauley et al.'s literature review of the debugging education literature shows, the debugging education landscape does not adequately address how novice programmers pick their debugging tactics—hardly any studies explore the decision-making process explicitly [15]. Furthermore, the few studies that suggest factors that may influence debugging tactic selection focus on experienced programmers rather than novices. In his study of two professional programmers, Gould assumes that programmers select tactics according to factors such as their knowledge, habits, and experiences during the debugging experiment [8]. In another similar study, Gould and Drongowski suggest several additional factors that influence the debugging strategies programmers use such as the information available to the programmer, the time available to complete the debugging task, and the programmers' own motivation [9]. While these studies incorporate experienced programmers rather than novices, they offer possible implications for how novices may select their debugging tactics as well. Benander and Benander's study focuses on novice programmers and they point to social factors in the environment, such as the difference between a classroom and a non-academic setting as possible influences for whether novice programmers will decide to ask other programmers for help when debugging [1]. From another perspective, studies within the IFT field leverage the theory's concept of scent to explain how programmers select and adapt their debugging strategies to meet the constraints of their environments. By following scent, programmers choose actions that will maximize the ratio of value gained from consuming a piece of information and the cost of foraging for it [13]. Thus, IFT provides another possible explanation of how novice programmers pick debugging tactics: they perform cost-benefit analyses to determine which tactic to deploy given the conditions in their current task environments.

Chapter Summary

Both the debugging education and IFT literatures provide ample evidence of the debugging tactics that novice programmers use, but a limited number of explanations for how they select them. Those few studies that do attempt to describe the decision-making process that programmers follow often lack the descriptive power to provide concrete evidence of the activities involved in making these decisions. Furthermore, these studies primarily use quantitative methods and only employ the occasional use of interviews and observations to confirm their quantitative results. Consequently, this motivates us to follow a qualitative design to provide the depth and richness necessary to describe how novice programmers pick their debugging tactics from the student's perspective.

Chapter 3: Methodology

The purpose of this qualitative study is to explore how novice developers at UCI decide on which debugging tactics to use when debugging. Literature on software debugging education and information foraging theory describe novice developers using these debugging tactics to gather information to locate and fix bugs. Yet, it remains unclear how novices decide on which strategies to pursue. Thus, we conducted this study to gain insight on their decision-making process and to inform future studies on the possible factors involved. Our study addresses two research questions to understand this decision-making process: (1) What debugging tactics do novice programmers describe using to debug? and (2) How do novice programmers describe the activities involved in selecting a debugging tactic?

The rest of this chapter describes our research methodology, including the following topics: (1) rationale for choosing a qualitative research design, (2) research sample, (3) overview of the information needed, (4) research procedures, (5) data collection process, (6) data analysis process, (7) issues of trustworthiness, (8) limitations of the study, and (9) summary of this chapter.

Rationale for Qualitative Design

While both qualitative and quantitative approaches have been applied successfully in Software Engineering research, our study adopts a predominantly qualitative approach. Quantitative methods test hypotheses by understanding the relationships between well-defined, quantifiable variables. Our approach differs from their quantitative counterparts in that the goal in qualitative studies is not to test specific hypotheses, but to understand and provide a holistic and rich account of complex social phenomena [14]. Since our study seeks to uncover the possible variables involved with selecting debugging tactics rather than test these unknown variables, a qualitative approach is most fitting. Furthermore, the exploratory nature of our study lends itself to the interpretivist stance we adopt, as our study seeks to capture and understand our participants' unique experiences when choosing their debugging tactics. Rather than generalizing and abstracting these experiences, we leverage each student's perspective to elicit a detailed account of how novice programmers pick debugging tactics.

Research Sample

We recruited twelve UCI students with programming experience through a mixture of homogeneous and snowball sampling procedures. These purposeful sampling methods are appropriate for qualitative studies, because they allow us to "identify our participants and sites [...] based on places and people that can best help us understand our central phenomenon" (Creswell, 2012). The flexibility and variability in purposeful sampling techniques allowed us to pick a heterogeneous group of students with varying skill levels and experience, but at the same time have all taken the introductory Python programming courses at UCI. As a result, we distributed recruitment flyers to professors teaching first- and second-year programming courses covering different programming languages like C++, Java, or Python.

Purposeful sampling also allowed us to target students with programming experience outside the classroom. For example, we leveraged social media and posted on the university's Facebook group page for the Association of Computing and Machinery (ACM). Snowball sampling filled the gaps where homogenous sampling either did not generate enough recruits or if we felt a participant sparked interesting insights that we wanted to hear more about. In these situations, we would ask participants who completed the study to refer their friends and peers who might be interested in the study. Our inclusion criteria included the following:

- A first- or second-year UCI student studying Software Engineering, Computer Science, or a related field where they obtained experience coding and debugging software.

- Participants must have experience writing small- to medium-sized programs in Python or a similar high-level language

Of the twelve participants we recruited, eight had freshman or sophomore standing, while the other four had junior standing despite being second-year students. All participants had at least two years of programming experience from UCI or another education institution or from programming on their own. We also ensured that participants were at least comfortable programming with Python and had experience in integrated development environments (IDE's), since our study procedures relied on Python and the Eclipse IDE. The demographics of our participants are summarized in Table 3.1 below.

Table 3.1: Participant Demographics

Participant	Class standing	Held degree or pursuing	Number of programming courses taken at UCI	Yrs. programming	Yrs. programming for large software projects
Participant 1	Freshman	Computer Science	3	3	0
Participant 2	Freshman	Computer Science	1	3	0
Participant 3	Sophomore	Computer Game Science	4	2	0
Participant 4	Junior	Computer Science	6	3	0.5
Participant 5	Sophomore	Computer Science	4	2	0
Participant 6	Junior	Computer Science	5	2	0
Participant 7	Sophomore	Computer Science / Applied Physics	4	3	0

Participant 8	Sophomore	Computer Science	5	3	0
Participant 9	Sophomore	Computer Science	4	2	0
Participant 10	Sophomore	Computer Science	5	3	0
Participant 11	Freshman	Computer Science	4	5	1
Participant 12	Junior	Software Engineering	7	3	0

Information Needed

To answer our two research questions, we collected participants' demographic information and perceptions of their decision-making process through questionnaires and interview questions that included inquiries into class standing, pursued, and academic and technical experiences. To collect perceptual information, we used direct observation and interviews to elicit the debugging tactics that novices use when debugging. We asked them directly why they chose certain strategies over others. This breakdown of the information we needed are summarized in Table 3.2 below.

Table 3.2: Overview of Information Needed

Type of Information	Description of Requirements	Method
Demographic	Information regarding participant's academic background and technical experience.	Questionnaire, interview
Perceptual	Descriptive Information of participants' experiences choosing different debugging tactics.	Observation, interview

Research Question 1: What debugging tactics do novice programmers describe using to debug?	Descriptions of debugging tactics participants used during the study and in the past	Observation, interview
Research Question 2: How do novice programmers describe the activities involved in selecting a debugging tactic?	Descriptions of the activities that participants describe taking place when selecting debugging tactics	Interview

Research Procedures

Prior to data collection, we submitted an IRB application for human subject research and obtained IRB approval. This ensured that our research procedures would protect our participants' data and well-being.

After the IRB process, we began recruiting. We contacted professors teaching introductory programming courses and asked them to forward our recruitment flyers to their students. During the latter half of the research study, we modified this recruitment procedure to include social media outreach and direct referrals from participants. We modified our approach to gain access to students that may provide us with different perspectives from those that had already completed the study. We screened potential participants, and those who passed were given informed consent forms to sign and return to us before officially taking part of the study. Shortly after receiving their informed consent, we scheduled one-on-one sessions with participants on campus in a quiet conference room, where all study procedures would proceed.

In preparation for the meeting on campus, we installed the necessary software on two MacOS laptops, one for the participant and the other for the researcher. On the participant's laptop, we installed the latest version of Python (v3.7.2), Eclipse (64-bit (v4.10.0)), and PyDev plugin (v7.1.0), so that participants could debug their Python tasks on Eclipse. On both laptops, we installed the latest version of Slack to provide audio and text chat capabilities between researchers and participants, as both parties would be in separate rooms during the debugging session. All other software used in the study, such as Screen Sharing and QuickTime Player, were already available and came pre-installed on both Macs. We set up Screen Sharing between the two laptops before the participant's arrival and began recording the participant's screen when they began their debugging sessions.

Participants received their laptops shortly after arriving to the study location, and we welcomed them with a short presentation about the study and the day's agenda. We briefed each participant that they would complete a preliminary questionnaire; a debugging session; and a retroactive, semi-structured interview to conclude. After answering any clarifying questions, we asked them to complete the online questionnaire.

Upon completing the questionnaire, we gave a brief overview of the debugging tasks they would be completing. This high-level overview included going over the instructions, namely that nothing would be off-limits and that participants were encouraged to speak aloud while debugging. We also gave a walkthrough demonstrating how to run the Python programs and explained to participants how to interpret the test cases included with the programs. After answering any questions participants raised, we initiated a Slack audio call between both laptops, and we exited to the room next door. Our participants would begin debugging for the next 50 minutes while we observed their screens through our laptops and noted any interesting observations in a log.

We would occasionally remind participants to speak aloud if they remained quiet for more than five minutes; however, we ceased the reminders if participants remained quiet after the initial reminder. We interpreted this behavior as a display of discomfort, and we opted to allow participants to debug however they felt most comfortable. Such observations and conversations during the interviews, where several participants mentioned the awkwardness of "thinking aloud" while debugging, motivated us to modify our procedures. With future participants, we decided to make the "think aloud" aspect of the debugging sessions optional. We preferred observing students debugging with more freedom and in a comfortable environment over having them debug while feeling awkward about speaking out their thoughts. As interviews would serve as our primary mode of data collection, we justified losing the ability the hear participants' thoughts as a tradeoff that would yield more interesting and accurate observations.

After the debugging sessions, we reentered the room and ended the Slack call. We did not give prior warning before ending the debugging sessions so that participants would not feel the need to alter their behavior to accommodate for time pressures. Participants were given an optional 10-minute break before we began the retrospective interviews. The majority of the participants, however, opted to skip the break. All interviews lasted approximately 50 minutes or until the total time allocated for the study was spent. We wrapped up the interviews by thanking the participants and compensating them with $50 Amazon gift cards.

After each session with a participant, we would write a one-page summary of our first impressions and highlights that stood out from the session. This summary was a form of preliminary data analysis, and these notes would be useful during the actual coding and analysis phases. We manually transcribed each interview to allow us to further familiarize ourselves with the data and perform some initial note taking and analysis. After we transcribed all the interviews, we openly coded them and performed a thematic analysis on the content.

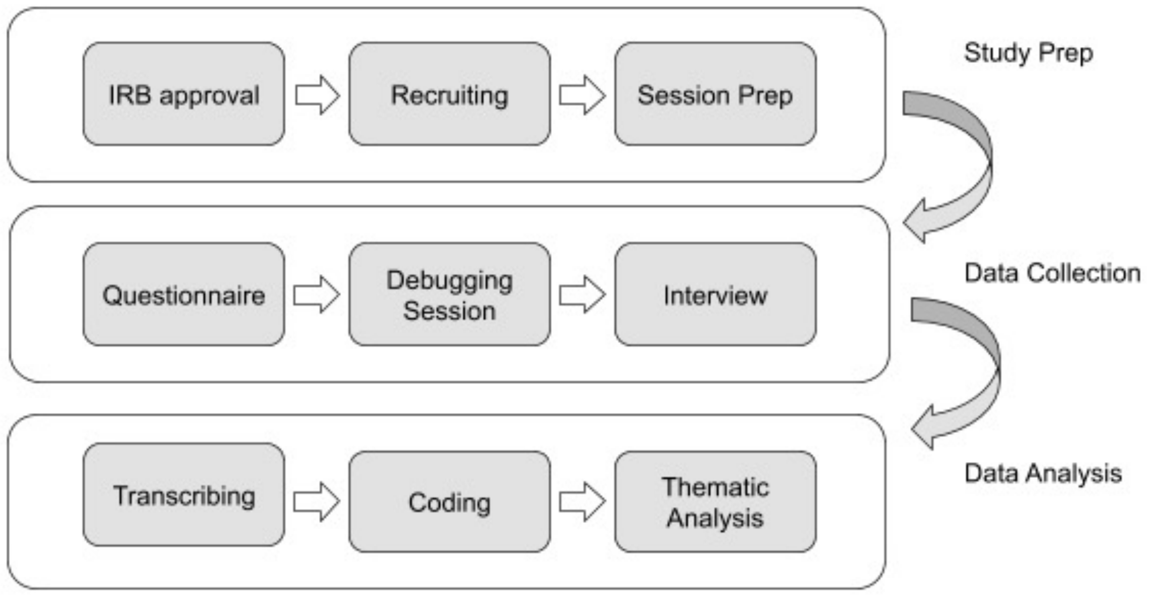

Figure 3.1: Flowchart of the research procedures from initial preparations to data analysis

Data Collection

Questionnaire

We distributed a preliminary questionnaire to collect demographic and other background information about our participants. We incorporated Feigenspan et al.'s questionnaire, which measures programming experience among undergraduate students, since the demographics of our participants were similar to those the questionnaire was intended for [4]. While borrowing pre-designed instruments is not typical in qualitative studies, the measures gave us a glimpse of the technical makeup of our participants and helped us understand their interview responses with respect to their technical backgrounds [3]. We included an open-ended question more typical of qualitative studies that asked about debugging tools they have used in the past. This question would help us triangulate any debugging tools or strategies that we observed in the debugging sessions or heard from the interviews.

Observation

We observed participants debugging five versions of the same Python program written by previous first-year UCI programming students. We borrowed these student programs from a UCI professor who teaches one of the first-year programming courses. Since these programs contained real bugs introduced by other first-year students, we would be observing students debugging programs containing bugs that their peers and students like themselves would make. Furthermore, because the Python program was the product of an “in-lab” assignment given to first-year programming students to complete and submit in class, it fitted well with our time-constrained study. It was a challenging and yet doable exercise for the majority of our participants.

Of the 264 student programs that we received from the professor, we had to exclude 70 due to them not containing any bugs. From the remaining 194 programs, we randomly sampled five to include in the study. We picked five to ensure there was ample work for participants to utilize the full 50 minutes. This also gave us the freedom to randomly assign a different starting task to each participant and generate enough variability and overlap among our twelve participants. We debugged these programs ourselves to record the location of the bugs and the source of their failures. This preliminary analysis also helped us determine that the five randomly sampled programs were well-varied in the type and number of bugs, which ranged from one to four.

We did not make any modifications to the programs except to rename them such that we could include them all within the same Eclipse project for our participants to run. The debugging assignment itself involved implementing a class that models a list of dictionaries, and that contained various rules on how its operations should function. The assignment included a class file for participants to write their implementations, a driver program to run the program, a text file called the “self-batch test” to test the driver program, and some helper module files to help the program run.

We did not expect participants to finish debugging all five programs, since we aimed to observe their debugging behaviors for as long as possible. We recorded our participants' screens and logged live notes in a spreadsheet as participants debugged.

Semi-Structured Interviews

While we created an interview guide to structure our interviews, the actual questions we asked were formed over the course of the study in iterative fashion. Interviews with participants were conversational and we would first venture into interesting topics that arose from observing the debugging sessions. We would then branch out into topics that participants' themselves brought up. To ensure our research questions were answered, we also tailored questions around the motivation of participants' choice of strategies. Since we were interested in uncovering participants' motivations for using certain debugging tactics, we incorporated an interview laddering technique that serves just this purpose [12]. As we interviewed more participants, the interview questions homogenized around topics that were interesting and consistently brought up during conversation. Thus, question formulation was iteratively refined with more and more interviews being conducted.

Data Analysis

We coded our interview transcripts to make sense of the large volume of data, and our choice of thematic analysis as the preferred method of data analysis influenced the manner in which we coded. For instance, to code our interviews to answer our first research question regarding the debugging tactics that novice developers use, we followed a "codebook" approach as defined in Braun et al.'s conceptualization of thematic analysis [2]. For coding our interviews to answer our second research question asking how novice developers describe their experiences deciding between debugging tactics, we adopted a "reflexive" approach. Unlike the codebook approach, where "some if not all themes are determined in advance of full analysis, and themes are typically conceptualized as domain summaries," in the reflexive approach "coding is an organic and open iterative process; it is not 'fixed' at the start of the process (e.g., through the use of a codebook or coding frame)" [2]. Since the literature has already defined certain categories of debugging tactics, we could leverage this using a codebook approach to summarize and extend the known categories that define debugging tactics. The literature, however, does not clearly define the patterns that describe how novice developers pick debugging tactics. Thus, it was more appropriate to adopt a reflexive approach to analyze our data and "provide a coherent and compelling interpretation of the data, grounded in the data" [2]. For both processes, coding was an interactive activity involving email and in-person discussions surrounding our interpretations of the data.

Issues of Trustworthiness

This section discusses the degree to which our qualitative study establishes trustworthiness by addressing issues of credibility, transferability, dependability, and confirmability. These four criteria are analogous to the issues present in quantitative studies, which include internal and external validity, reliability, and objectivity. Due to the qualitative nature of our study, the former is the more appropriate criteria to establish trustworthiness [14].

Credibility

Credibility is the degree to which our study's findings are found to be credible and accurate. We addressed issues of credibility by conducting the following activities that increase the likelihood of our findings and interpretations to be credible: triangulation and member checks [14]. For example, we collected data using both interviews and observations to triangulate our data by comparing observations from participants' debugging sessions with the responses they gave during the interviews. In addition to the multiple methods we used to collect data, we also performed member checks during the interviews by “playing back” participants’ answers and actions recorded on video and confirming if our interpretations accurately captured their actions and responses. These two activities served to increase the credibility of our study.

Transferability

The issue of transferability in qualitative studies and external validity in quantitative studies differ significantly between these two research approaches. While in quantitative studies the researcher uses confidence intervals and statistics to determine whether a study’s findings can be generalized to other populations and contexts, in qualitative studies the responsibility of determining whether a study’s findings can transfer to a different context lies with the reader aiming to make use of a study's findings [14]. Nevertheless, the qualitative researcher must address their study’s degree of transferability through thick descriptions and by “providing the widest possible range of information for inclusion in the thick description” [14]. We increased the degree of transferability in our study by using purposeful sampling to cast a wide net for capturing different student perspectives. Furthermore, we included detailed information about our research procedures, data, and discussion of the data.

Limitations of the Study

Procedural issues affected several areas of the study, including its degree of transferability, dependability, and confirmability of our findings. While we picked a debugging task that typical introductory programming students at UCI complete, the specific structure of the program limits the extent to which our findings could transfer to another context. Because the test cases that were included with the program are very specific to the style that the instructor structures their assignments, they could have affected the strategies that participants used, and in other contexts, this program’s style may not be as common.

The setting in which participants debugged also limits our degree of transferability. Because participants debugged in an environment in which they described as similar to a testing environment, our findings may not apply to contexts where students are debugging without testing pressures.

Lastly, our study addressed limitations to its degree of dependability and confirmability by ensuring that our coding and data analysis procedures were well-documented. Furthermore, we used Google Docs version control system ensure transparency in how our codes and themes evolved over time. We also kept memos and notes explaining updates to the coding schemes and our rationale for our interpretation of the themes. Thus, we left a transparent audit trail to provide evidence of our thought processes when coding and analyzing our data despite not conducting an audit ourselves.

Chapter Summary

This chapter described our research methodology, including a discussion of our research design, a detailed description of the procedures, and a discussion of the study's trustworthiness and limitations. We selected a qualitative design due to the nature of our research questions, which sought to explore novice developers' experiences picking debugging tactics to debug. We utilized questionnaires, observations, and interviews to collect data in understanding this phenomenon. We interpreted our data using a reflexive, thematic analysis approach. We discussed how we addressed issues of credibility and transferability by applying member checking, triangulation techniques and providing rich, detailed descriptions of our study whenever possible. Lastly, we concluded with a discussion of the limitations of our study, namely that the study's superficial setting restricted the types of debugging tactics and behaviors that we could study. Furthermore, while we left an audit trail for others to judge the dependability and confirmability of our findings, we ourselves did not perform this audit.

Chapter 4: Results

In this chapter, we present our findings on how novice developers pick their debugging tactics when debugging. We answer our first research question by detailing the debugging tactics that participants described using during their debugging sessions and throughout their academic careers. To answer our second research question, we describe the activities involved in selecting these debugging tactics. Our results derive primarily from interviewing our participants about their debugging processes. We leverage our direct observation of participants debugging to design our semi-structured interview protocol and support the results that we captured from them.

For our first research question, we determined three types of debugging tactics that novice developers use to debug:

1. Participants ran code in their IDE's to generate new sources of information.
2. Participants sought help from external sources by searching on the Internet or asking people in their social networks.
3. Participants took notes or sketched on scratch paper to trace their code.

For our second research question, three themes emerged that describe the activities novice developers perform when selecting debugging tactics to pursue:

1. Participants leveraged their past experiences to determine which tactic to use or avoid.
2. Participants adapted to their task environments and chose tactics they believed were most suitable for those environments.
3. Participants anticipated future risks and potential rewards to determine which tactics to pursue.

This chapter is organized by the research question and themes that emerged. We include key quotations from participants that capture the underlying concept that each theme encapsulates. Furthermore, whenever necessary we augment these quotations with additional context to offer rich descriptions of the phenomena.

Research Question 1

We explored the debugging tactics that novice developers use in our first research question. These debugging tactics focused on the enrichment strategies that the IFT literature describes. Thus, in all of the debugging tactics described below, the participant acted within the task environment with the intent to gain more information from their environments or to make finding information in that environment more efficient.

Generating information by testing code

The first theme that emerged involves debugging tactics where participants ran code to generate more information in their IDE's. As demonstrated in Figure 4.1, participants would test their code in different ways to generate useful debugging information about the program, such as running their code with and without modifications and running debugging tools. Below, we describe participants performing the following debugging tactics: (1) running the code and (2) using debugging tools.

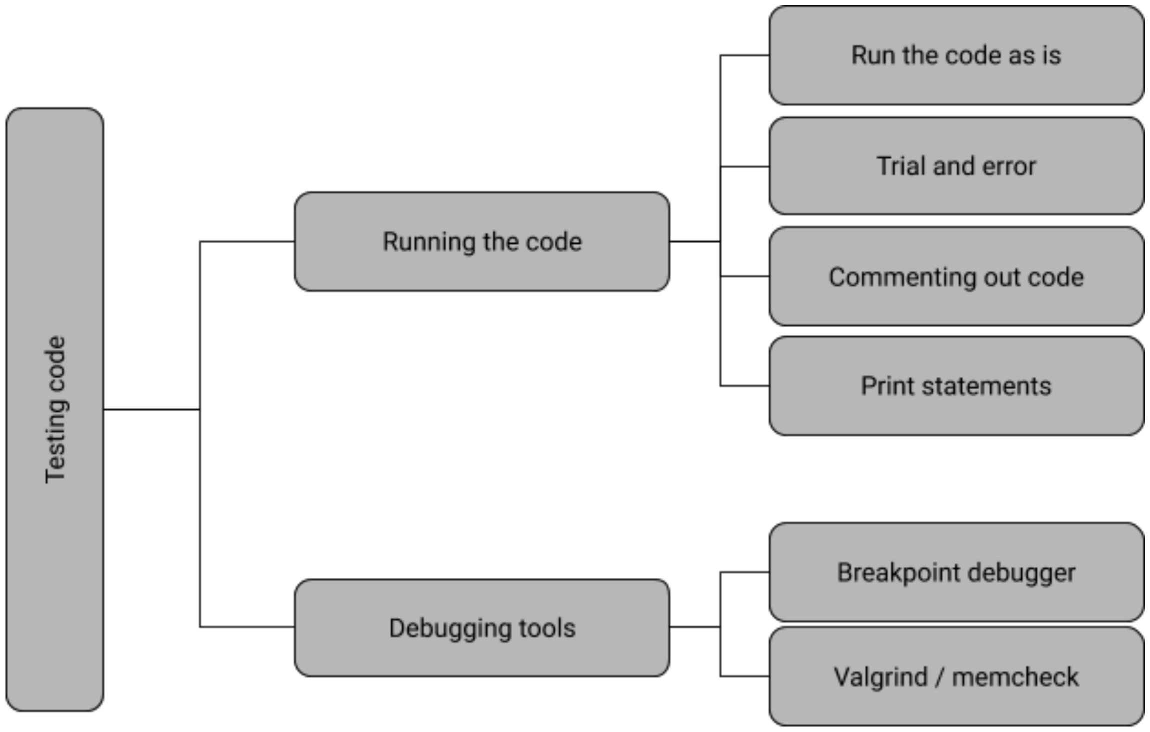

Figure 4.1: Codes and examples of debugging tactics falling under the "Testing code" theme

Running the code.

A strategy that participants used involved running the code to create program output that they could then process and gain valuable information. For instance, when asked about their general approach with tackling the study's Python program, Participant 9 thought to first "[run] the batch self-check and see what the errors were." This participant leveraged the test code in the study's program to verify if expected and actual outputs matched. Other participants described using similar strategies even when debugging programs outside of this study. Participant 1, for example, mentioned the following when asked how they would debug under normal circumstances not specific to the study: "I'll run it, and then if I see the error [...] I go to the line and see what's wrong." Thus, participants described running the code as a means to check if the program produced any specific errors.

Other participants described strategies involving adding test code or test inputs before running the program to observe how their programs would behave. Participant 6 described his approach to finding inconsistencies in the program by running the code with modifications:

> *"I probably don't have the best method. Basically, trial and error. See if [...] I can figure it out. But I try to reason. I try to look and see what it would produce and if that doesn't align with what I think it should then I try to just change it and see what happened [sic] [...] how it responds.*

Participant 1 described using a similar strategy involving trial and error: "I put a bunch of random inputs and then I see which one doesn't have the right output." Like Participant 1, Participant 8 too described running the code with their own test code to check for errors and other information that the IDE generates: "I would write my own unit tests and if something is going wrong I will literally initialize the class and give it some test values like maybe an edge case, some edge cases, and then check individual [...] [a]ttributes."

To produce program output to compare with actual output, participants also added diagnostic print statements to their code before executing it. For instance, Participant 9 stated, "I'll see at what point what went wrong. So, I'll probably do some print statements of... inside that function. See what the variables are. [sic]" Alternatively, Participant 4 used print statements to gain an understanding of the program itself:

> *"I went printing out everything that those two functions used. Like set item, I kind of went and put print statements, because every chunk of code you kind of know what it's doing. Like this code, this chunk right here 46 through 52, it's adding something to the assert list. I don't know what it was adding but it was*

> *adding something. So, I printed out just to see what happened. Like, why did it do that?"*

To Participant 5, print statements helped them visualize and understand the code better: "I feel like just me personally I have a really hard time visualizing what's happening unless I print every single thing out and I write it myself."

In addition to adding test code before running the program, participants also ran the code after using comments to remove code to gain insight into the state of the program. Participant 6 stated, "I just didn't understand the purpose of those [changes]. That's why I was like what is it doing? So, I commented it out hoping it would fix it." Similarly, Participant 2 described their approach when dealing with code they did not understand: "I'll just delete it or I'll comment it out and try to rewrite it if I don't really understand what's going on."

Using debugging tools.

While participants described using print statements as a debugging tool to diagnose symptoms in their code, these built-in language constructs do not apply to the "using debugging tools" category. This category refers to standalone tools like built in breakpoint debuggers in IDE's that the user can run to diagnose their code. Nevertheless, participants used debugging tools in a similar fashion to print statements; they used the debugging tools to generate output that detailed the program's internal state and inspected that information for errors and other specific information.

Participants described using different types of debugging tools depending on the environment they were in. The majority of participants described using breakpoint, step-by-step debuggers commonly found in IDE's such as Eclipse. In Participant 9's case, they used MIPS, an IDE for assembly language. In describing their debugger usage, Participant 9 said, "Well, I'm still being introduced into assembly. So, I'm not really sure how other people debug. But yeah just specific to assembly. I use the debugger, because I don't have any better tools basically."

Participants also used other versions of the breakpoint debugger like gdb. Participant 8 describes their general workflow with gdb as so:

> *"Here's the GDB workflow in my opinion. You go and see your code in the line, so you can get the line number or you can just memorize the line number and then set a breakpoint there. And then you run the code. And then when you get to that breakpoint you say, oh now it's time to run one at a time, and one at a time, and like that is all overhead to me"*

For debugging memory leaks in a C++ environment, Participant 7 indicated that they would turn to tools like Valgrind and memcheck, which were specific to the problem.

While similar in purpose as the print statements, using debugging tools like the breakpoint debugger was not a universal strategy. Eleven participants indicated that they had experience using breakpoint debuggers due to course requirements. However, many participants expressed that they did not use breakpoint debuggers frequently in their workflow but, rather, reserved them as a last resort. For instance, comments like those below highlight the distinct differences in usage between print statements and breakpoint debuggers.

Participant 6:
On print statements: "I use those [print statements] a lot."
On breakpoint debuggers: "Back when I was in class I would sometimes use the built-in debugger."

And for Participant 8:
On print statements: "I used to just print statements all the time. Just print. That would be my quote, unquote breakpoints stuff for it's like okay I got here."
On breakpoint debuggers: "I'll use the debugger a lot more than what I used to when I was in [Informatics & Computer Science] 33 when I never used it."

Searching for help

The second theme involves strategies where participants sought help from external sources rather than trying to solve the problem on their own. As shown in Figure 4.2, participants sought help through online search engines like Google and from people in their social networks. To gain access to these repositories of information and elicit useful information from them, participants filtered the information they received by narrowing down their search queries or the questions they asked to generate meaningful results. We described these two debugging tactics below: (1) searching for online resources and (2) seeking help through social networks.

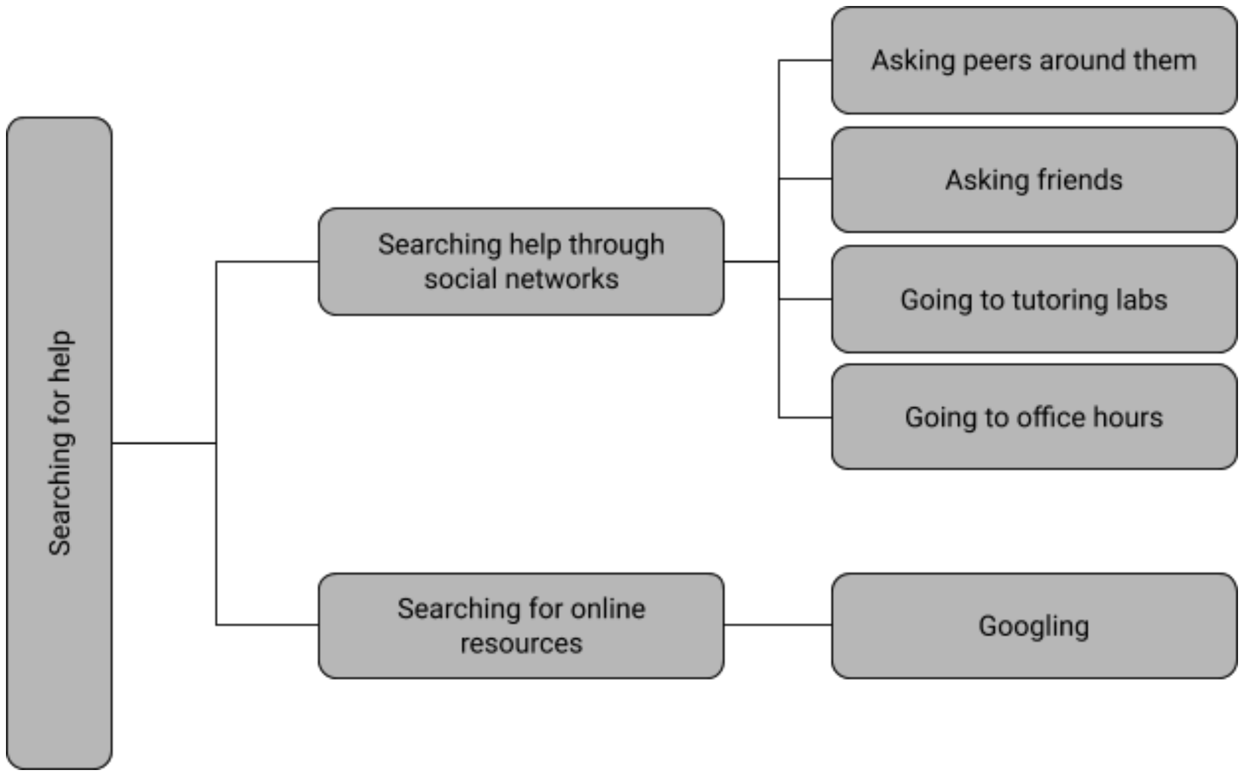

Figure 4.2: Codes and examples of debugging tactics falling under the "Searching for help" theme

Searching for online resources.

Participants searched for information online whenever they were unsure about particular concepts. When searching online, participants described using search engines like Google and continually searching until they found relevant and useful information. For example, when describing their online searching strategies Participant 7 said the following:

> *"[I] f I get really stuck, I try out a lot of different things involves [sic] a lot of Googling obviously. I usually for like minor problems that I just don't know because of my lack of skills or whatever [sic]. I just go to StackOverflow and sooner or later I will find something that's similar or at least will give me an idea of what I want to do."*

Participant 12 described relying on the internet heavily for certain types of problems. For instance, they mentioned that they would online if they were "getting errors that [they didn't] understand" or if they were "looking up legal syntax or just general C++ or programming knowledge."

Elaborating on how they search for information online, Participant 11 highlighted the importance of narrowing down searches:

> *"Going online is actually the most useful resource I think for debugging [...] But if you're able to narrow down your search as well, well then I think online is generally the best option for debugging."*

Seeking help through social networks.

While participants described searching online for help, they also indicated that they would seek out other people to help them debug. For Participant 5, they would seek help from an array of people and responded with a list when questioned about their debugging options:

"Asking other people who didn't write it. Maybe they have [a] better perspective [...] So first I try it myself, then I go online then I ask [...] someone who's sitting near me, and then I ask for help from the person who wrote it or some authority kind of like a tutor."

Like Participant 5, Participant 7 also leveraged access to authority figures in their institutions to ask them for help: "I go to office hours. That happens very rarely, but I did go to office hours once when I was really, really stuck... Worse case I go to lab even worse case I go to the office hours." For Participant 11, asking teachers and those around them was a strategy that started during high school. When asked about how their current debugging approaches evolved, Participant 11 stated the following:

"In high school I was one of those people who would constantly pester other people, asking them [...] how to solve the recent lab problem. Most people in my high school were pretty unhelpful if I'm being honest. And then when I asked the teacher, the teacher was actually pretty good. But the thing is, I would ask them too many questions, which, as you know, is pretty annoying."

Asking friends through social media was another strategy that participants described. They mentioned they would ask their friends for help and suggested the type of questions they would ask:

"I always have a group chat with me... [S]ometimes if I've been stuck on something for days and nothing is working, I'd ask a friend and I tell them, hey, have you tried this? And what happens if this? And usually they'll help. They'll usually take the time to help me to figure out what's going on."

A couple of participants also delved into the process they used to ask questions. For example, Participant 5 stated the following with regards to how they ask authors of the original code for help:

> *"Well, I mean first I see if I can figure out what I'm doing. And then otherwise I'll be like, hey. I would just tell them to run through the whole thing. Like what they were intending to do and then I'll tell them where the error is at."*

This approach is similar to how Participant 4 debugged code with the benefit of being able to ask the original author; both participants would leverage having the original author's knowledge of the program to elicit information that would help them comprehend parts of or the program. Participant 4 worked as a tutor at UCI and had to frequently debug other students' code. When asked about their debugging process while helping their tutees debug their own code, they described the following process:

> *"It's almost unfeasible for me to read everything on the spot and just figure [the code out]. I just can't know everything just by reading the code. Just like sometimes on my own code I can't just read it and [...] know what's happening. Because that's often not the case. I would have to tell them, 'hey, what does this do?' I'll ask him, 'what does this line do? What do you think it does?' And I'll tell them, 'hey, print it out. Is it really what you think it was doing?' And oftentimes that's the problem."*

Participants asked a variety of people for help when debugging, including original authors of the code they were debugging, friends or peers working on the same project, people not associated with the project, and authority figures like tutors and professors. With regards to the questions they formed when asking others for help, each described a different questioning process.

Taking notes on paper

The third theme captured strategies where participants wrote notes on paper to create new sources of information that would reveal more fruitful information to aid in debugging. As pictured in Figure 4.3, participants traced code and sketched diagrams on paper. Essentially, all participants who used paper did so to visualize the program in a non-digital medium way.

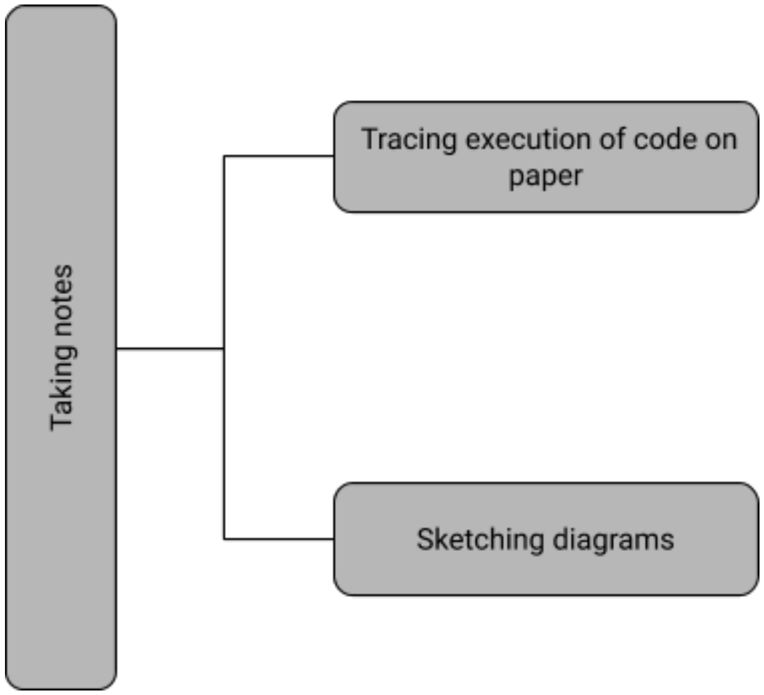

Figure 4.3: Codes and examples of debugging tactics falling under the "Taking notes" theme

Participant 9 described using paper in the following manner to execute the program by hand: "I drew out what was in the batch self-check and went along. [I] kind of did the program on paper." Other participants like Participant 3 and Participant 1 leveraged paper, because it helped their mental state when debugging. For Participant 3, when asked if using paper helped them understand complex code, they gave the following response: "Either that or when I feel lost [...] to just clear out my mind." For Participant 1, they described using paper to as a means of staying focused on the problem: *"I'm not very focused. I'm just like oh it doesn't work. [...] [I]f I can't think of* *anything then I'll be like oh gosh I actually have to work. And then I'll pull out a piece of paper and actually go line by line."* Participant 3, also stated the following with regard to using paper to debug:

> *"If the program tells me nothing, I'm going to trace the program on paper. […] Because […] sometimes [when] we write a program I can draw diagrams. [A]nd those I feel like those diagrams really help me understand […] because we can cross reference and stuff. And for object oriented programming I think it's pretty helpful."*

Like Participant 3, Participant 9 relied on drawing diagrams extensively to visualize the program better. They described their process and motivation using paper in the following quotation:

> *"I'm a really visual learner [sic]. So, I think it helps a lot when I actually trace it on paper, because then I can get a better understanding of exactly what the method is supposed to do. So, for example, this one was like you're supposed to update the most recent dictionary. So, to do it by hand like I saw it […] I trace. I look for 'b.' Not here. Not here. I don't want there. So, when you do it by hand […] it helps me a lot."*

Participants described using paper to debug as a strategy that helped them mentally visualize and understand the code by representing it in a more accessible medium. Thus, by using paper, participants create new sources of information that they can consume while debugging.

Section Summary

Participants described using debugging tactics that previous literature have reported. Table 4.1 below compares the debugging tactics that we saw and those in the literature. In describing instances when they ran the code to help them debug, participants demonstrated how modifying code and running code were highly coupled activities that ultimately allowed participants to gain valuable information from the generated program output. When seeking help through search engines or their social network, participants described tactics where they narrowed down their search queries or formulated and structured their questions to elicit the most useful responses. Lastly, in describing how they use paper when debugging, participants described the usefulness of creating new representations of the program on paper to visualize and understand the code.

Table 4.1: Comparison of the debugging tactics found in the debugging and IFT literature and our study

Debugging Tactic	Existing Literature	Our Study
Generate program output (i.e. print statements)	[1] [5] [7]	✔
Tracing by hand	[1] [5]	✔
Asking other programmers for help	[1]	✔
Searching for external resources / documentation	[5][6][7]	✔
Rewriting code	[5][6]	✔
Testing code with sample input	[5][6]	✔
Adding comments	[6]	✔
Debugging tools (e.g. debugger)	[5][6][7]	✔

Searching for code with SDE	[7]	
Writing a to-do list	[7]	
Applying undo button in SDE	[6]	

Research Question 2

To better understand novice developers' thought processes when making decisions about their debugging tactics, we asked what kind of activities they perform leading up to these decisions.

Leveraging past experience

Participants made decisions about their debugging tactics by accounting for information gained from past experiences with these strategies. Figure 4.4 shows the following activities that were involved in their decision making: (1) past value gained by using a strategy, (2) old debugging habits they have formed, and (3) progress they have made on the current debugging task.

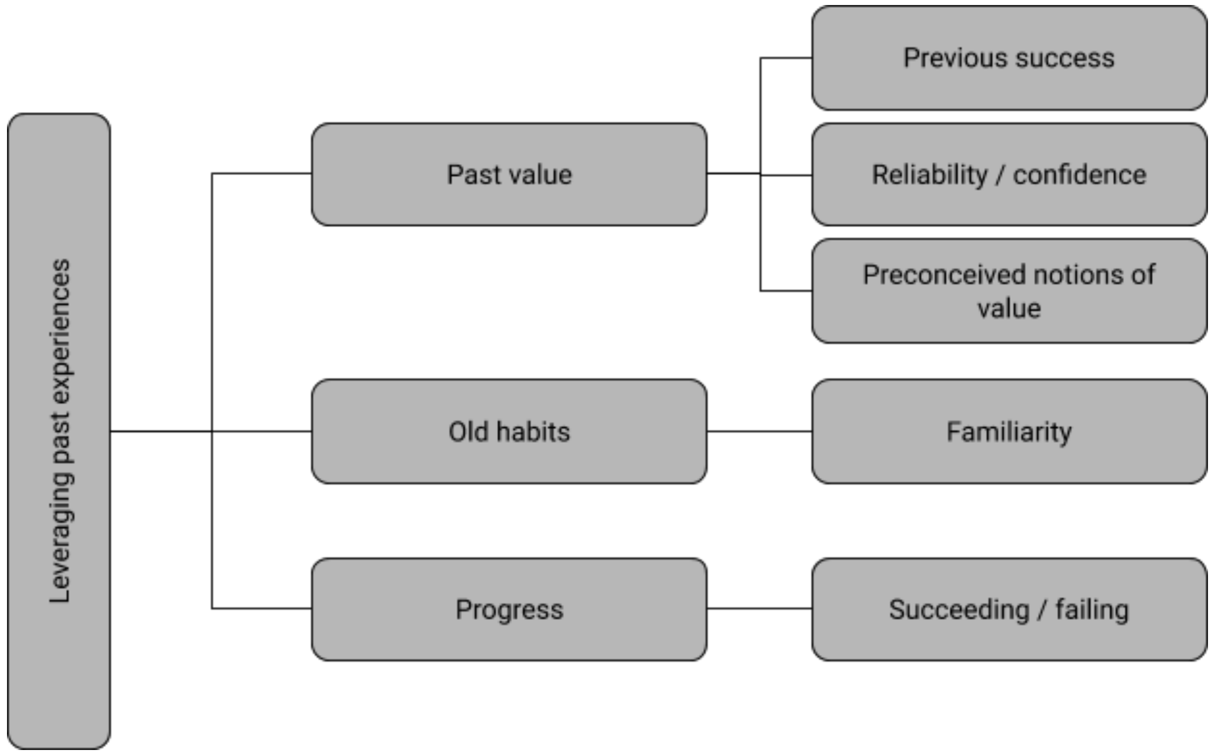

Figure 4.4: Codes and examples of activities in selecting debugging tactics that fall under the "leveraging past experiences" theme

Past value.

When considering which debugging tactics to use, participants leveraged their past experiences using the particular strategies and the value they gained from using them. For instance, Participant 10 described using debuggers on complex problems due to their previous success with it in their classes. They said:

> *"It was a concept I thought of myself. It worked. I mean I passed all my classes with it. I did fine with all my classes with it and I still learned the material. There's nothing specifically bad about doing that, it's just mainly time-wise that's debatable."*

Participant 6 even acknowledged that their trial and error strategy was only sometimes effective. Yet, they continued using it, making the argument that "it's got [them] this far" in their academic career.

Other participants also indicated a sense of confidence when using certain strategies, especially when they knew they could retrieve certain information using the strategy. For example, Participant 1 described their confidence in using debuggers to find the line number of the bug:

> *"If I didn't know how it was doing that or what line it was doing that I wouldn't be able to [...] that's why I went to the debugger because then once it screwed up, I'd be like, oh it messed up in that line."*

Likewise, for Participant 12, when asked about their preference in using the internet and print statements to fix bugs, they hinted at the strategy's reliability in solving very easy problems: "[I]t's easy. [...] [I]t's reliable [...] If it's a very easy problem and error, then I can fix it right away." Participant 3 gave a similar answer, explaining why they choose to use adding test inputs as their first debugging strategy: "I feel like most problems can be solved by try inputs."

Yet, another reason that participants gave for why they choose certain strategies over others involves preconceived notions of value they may gain from using those strategies. These preconceptions built up throughout their academic careers, as Participant 3 described about their usage of debuggers: "I feel like a lot of people use it. People say it's a great tool. Maybe I should try to use it more to save more of my time."

For Participant 8, despite having learnt the strategy in class, they did not think they would need it until personally witnessing their friend using it:

> *"So [Professor Z] teaches debugger [sic] and then I decide I don't need that. I'm just going to go through all of [Informatics & Computer Science] 45 without a debugger and do print statements all the time. Then [Informatics & Computer Science] 46 comes along and [...] it's the first project and friend shows me that he or his friend has a problem and he uses GDB [sic]. And I'm like oh holy crap [...] It's like, oh cool. I'll never need that and then now I need it."*

Old Habits.

Participants described mixed experiences concerning using strategies that they were taught in class. Participant 8 did not initially use the debugger despite being taught it, while other participants did use what they learned in the classroom and stuck to the debugging habits they had developed. For instance, Participant 9 indicated preferences for strategies that they learned early on in their academic careers: "Yeah, I mean printing is just like that's taught to you [...] the first day you program hello world, and using print statements to debug. It's like I have the tools so I use."

For instance, Participant 9 indicated preferences for strategies that they learned early on in their academic careers: "Printing is taught to you [...] the first day you program Hello World and [so is] using print statements to debug. It's like I have the tools so I use [them]"

> *"I think it may be because I didn't learn the debugger at first. Because in [Informatics & Computer Science] 32, I think we learned to write unit tests, but we didn't learn that much about debugger. [...] We didn't even use Eclipse. We used IDLE. In [Professor Y's class], we didn't learn about debugger that early too [sic]. We used it later in the quarter. We learned later in the quarter. Maybe because I was introduced to it later, that's why I feel more distant to it. And when I was learning debugger I was already [...] using those print statements for a long time. That's why I feel like oh, those are going to help me more."*

When discussing why they did not use debuggers, Participant 7 discussed similar problems of familiarity as Participant 9:

> *"I do want to use it, because I think it's useful. But I don't think they got us really into that kind of method. [...] Nobody ever showed us why is this super useful [sic] and how to make it useful. Because for me it's because I'm a newb, I don't*

know how to use it. It's definitely going to take up a lot more time for me to use. And, if they had a demo of it or something, that would have been great."

For Participant 12, their old habits ultimately determined which strategies they used: "I was taught it, but it's just the old habits die hard. For me it's like I've been printing so why not just keep doing it?"

Progress.

More recent experiences also motivated participants to choose certain strategies over others, especially when they were not making progress using certain strategies. For example, many participants indicated switching strategies when their original strategy fails. For Participant 8, they described switching to the debugger when they could not understand the bug from using print statements alone:

> *"So that would be the [...] ultimate reason I would pull out GDB – the print statement is giving me something that's [...] not deceptive, but if the print is [...] giving me something wrong and I don't know why. [For example,] I have no clue why and I can't figure it out with print statements."*

Similarly, Participant 4 also indicated switching strategies when not making progress:

> *"I guess at some point I wouldn't say I would give up, but at some point maybe I'll stop and maybe try something else. Because there's often different ways to go at it [sic]. Maybe I'm trying to implement something that's kind of impossible."*

A majority of participants stated that debuggers were their last resort when all else failed.

Adapting to the task environment

Novices also considered information about the current task environment to decide on the tactics that they thought would be most appropriate to use. Figure 4.5 conceptualized this activity and described some information in the environment that participants considered while deciding which debugging tactic to use: (1) the type of debugging problem they were trying to solve, (2) the requirements or constraints in their task environment, and (3) how negatively they were feeling at the moment.

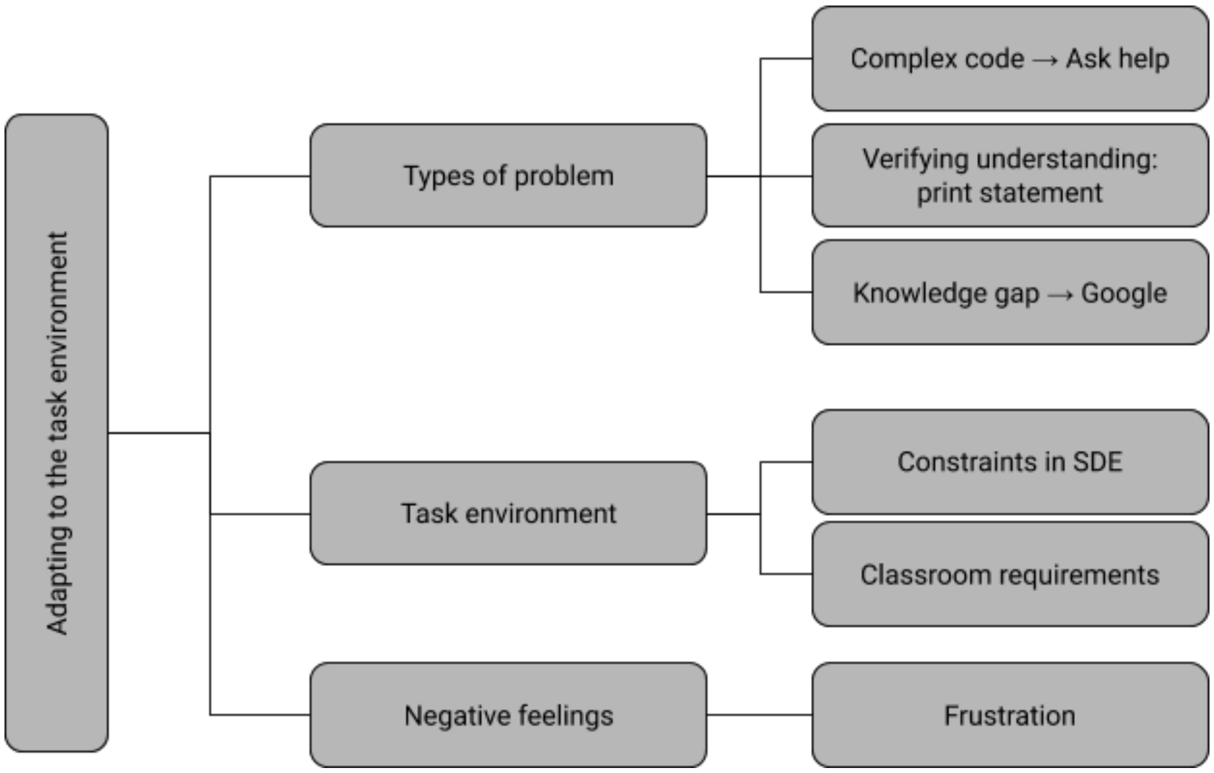

Figure 4.5: Codes and examples of activities in selecting debugging tactics that fall under the "adapting to the task environment" theme

Type of problem.

Participants described using strategies to address certain specific problems and information goals. For example, when trying to understand difficult code, Participant 5 described their preference for asking other people: "If [...] the code is too convoluted and messed up, and I feel like asking someone so that I understand the code would be better than me just using the debugger [sic]." In comparison, Participant 1 described using print statements when they believed they already had a good understanding of the code:

> *"Maybe it's more of me trying to understand what the line is supposed to be doing, and then print statement is I already understand and it's for me to verify if it's doing what it's supposed to be doing. That's why I go to print statements first, because I believe that I already understand what is going on."*

For other information goals, such as trying to bridge some conceptual knowledge gap, novice developers like Participant 7 described using online resources: "If I have some missing knowledge, I'm trying to fill in that gap with Geeks for Geeks, for example, or StackOverflow."

Task environment.

The environment in which participants debug also contributed to tendencies to use certain strategies over others. For example, Participant 12's strategy depended on the programming language they were debugging in. When asked about what differences between Python and C++ motivated them to use print statements more often in C++, they gave the following answer:

> *"Just the fact that C++ is static or [in] C++ you have to deal with memory and also clean up. Just those two mainly. [...] I think I did use print statements in Python, but it was like you said. It wasn't that big of a deal for me compared to C++, because I didn't have to deal with dynamic memory."*

Participant 9 also mentioned having to specifically use a certain strategy when programming in an assembly programming environment: "when I'm actually debugging I just use print, but if we're talking about assembly then I have to use the debugger [...] because [...] printing [is] a pain."

While participants picked certain strategies based on the programming language and IDE, their social environments also determine what strategies they used. In particular, the class requirements that participants felt compelled to abide by motivated their choice of strategies. Participant 11 described their experience using strategies for their school assignments:

> *"Because the way schools designed their assignments, it's done so to force you to [...] do it in a specific way. And also, the answer isn't readily available most of the time. So, [...] that's why I normally rely on print statements and debuggers for school assignments, because the teacher wants you to do it a specific way and it's not online often."*

These class requirements varied according to the class participants were in. Participant 7 mentioned this point when asked when they used the debugger:

> *"I'll say that [Professor Y] was the only one who asked us to use that. So in all of the rest of my courses it either wasn't even allowed or [...] because [Informatics & Computer Science] 45C and stuff like that you're not really allowed to use any outside debuggers. We only have what's provided to us."*

Similarly, Participant 3 described how they had to practice and use the debugger often due to the testing environment of their lab courses: *"Because there are some programs that we have to go to lab tests and debug part of the program. And that's when I use debugger the most, because we have to learn about that. Because I try to practice it."*

Formal environments like lab courses promote certain strategies, but less formal settings also invite other strategies. For example, Participant 7 described their experience going to the tutoring labs searching for help from tutors:

> *"I love going to lab because the TA's and the tutors are closer to my age and it's like, it's okay I was dumb at one point. You know? Also, I was tutoring for one of the classes so I know how they relate. That's a you know less formal setting so I like to use that."*

In contrast, they described feeling anxiety about looking dumb when going to formal office hours seeking help from professors: "It's just like I guess super anxious, and I have to like go to the professor and be like, 'hello. I'm dumb.'"

Negative Feelings.

In addition to anxiety, participants indicated other negative feelings like frustration that motivated them to choose certain strategies over others. Frustration, for example, would indicate and precede a change in strategy. For instance, Participant 8 described their experience with frustration while applying print statements: "So that would be if it's frustrating me to no end, and [...] I can't figure out in my head with the arithmetic why I'm getting right print statements, but not correct calls from [...] another part of the code. That's a gdb thing for me." Others like Participant 6 also mentioned frustration as the "trigger" for using the debugger: "Probably frustration. I feel like that triggers the debugger. Like this isn't working. Like, kind of opening the debugger and literally just every line of code [hitting] next, next, next, next."

Feeling lost or hopeless also preceded a change in strategy. Participant 3 described this when they said: "When I actually feel lost, like I don't know what's going on, then that's when I go debugger." With a very similar response, Participant 5 described when they would start asking for help: "When I'm completely, completely lost.

[...] When I've exhausted all my options and made as many print statements as I can and I don't understand what's happening."

Anticipating future risks and rewards

Participants selected debugging tactics by considering anticipated risks as well as short- and long-term rewards. As summarized in Figure 4.6, we described the following risks and rewards that participants anticipated when deciding to use a debugging tactic: (1) time investment, (2) expected immediate value, (3) future value, (4) future work, (5) manual effort, (6) mental effort, (7) risk of failure, and 8) power.

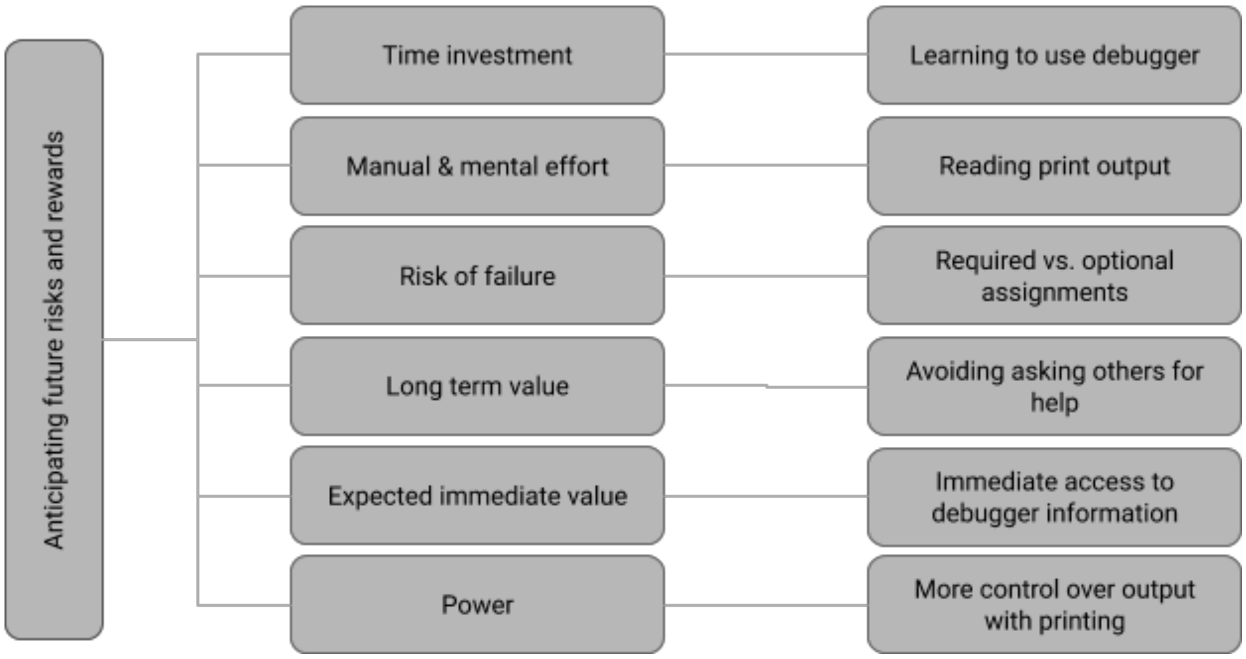
Anticipating future risks and rewards
Time investment
Manual & mental effort
Risk of failure
Long term value
Expected immediate value
Power
Learning to use debugger
Reading print output
Required vs. optional assignments
Avoiding asking others for help
Immediate access to debugger information
More control over output with printing

Figure 4.6: Codes and examples of activities in selecting debugging tactics that fall under the "anticipating future risks and rewards" theme

Time investment.

Participants considered the potential time necessary to apply certain strategies. For example, Participant 12 described having to spend time taking out a piece of paper as a barrier to them taking notes on paper as a strategy: *"For me, I don't want to have to spend the time to* *get a paper and draw it. You know, go back and forth. I know it's very useful, but if for easy problems, I'm just like, well what if I just printed out this and let's see where it takes me."*

Other participants also raised concerns about underestimating the amount of time they needed. According to Participant 4, he does not expect to spend an hour fixing a small error and "if it takes an excessive amount of time, at some point [he will] just stop." Alternatively, the potential to save time also motivated participants to pursue certain strategies. Participant 9, for example, described adding comments as a strategy that would save them time in the future:

> *"So, commenting is basically abstracting away from the code and just saying what it does. So instead of rereading, 'Oh okay. What does this line do? What does this line do?' I just look at comments and be like, 'Oh, yeah. Okay. So, this block does this. [...] And when I look back that will save me time.'"*

Manual and Mental Effort.

In addition to time, participants worried about the manual and mental effort needed to apply certain strategies. Participants mentioned the tediousness involved when applying the debugger. For example, Participant 6 stated, "[Y]ou have to figure out where exactly it would be useful, the breakpoint, and stepping through. And it's just kind of tedious." Participant 1 described similar sentiments about the extra work involved with debuggers: "I don't know. I feel it's just extra steps. You have to figure out what line you want to do it and then step through it."

In other environments, such as MIPS for assembly, print statements were the culprit rather than debuggers. Participant 9 described the overhead involved with setting up print statements for debugging in assembly: "It's [...] too much of a hassle, because instead of Python and Java, where you can just print, you need to load into a register. [...] Basically, the sycalls are just a pain."

In addition to the setup costs involved with applying certain strategies, the effort involved with processing large amounts of data also made certain strategies less appealing. Participant 11 described the inconvenience of using the debugger to investigate large for-loops due to having to repeatedly run through many iterations of the loop: *"It doesn't run all the solutions at once. It runs only one at a time. So that makes it a bit more inconvenient [...] which is why for for-loop statements typically I don't really rely on breakpoints."*

Participant 6 highlighted the mental effort involved in keeping track of changes when applying the debugger to large volumes of data: "I remember when I would use it just stepping through and seeing if every variable would change how I wanted it to and it's just a lot to keep track of." Meanwhile, for smaller datasets, Participant 2 described favoring print statements: "And then the print statements [...] I use those if it's small data. If it's just one or two, three lines that you can print out and kind of see what issue is happening where."

Another barrier that participants described involved learning how to use strategies they were not familiar with. For Participant 1, the effort involved learning a new process when they could fall back on habits. They stated, "I'm just used to typing things and printing things but debuggers is like a whole [other process] Again, I'm lazy. It's a whole other process. Something that I'm not familiar with."

Risk of Failure.

When faced with potential risks such as failure, participants described using strategies that they normally would not use if not forced to. For example, when asked whether they have had to use debuggers throughout their academic career, Participant 5 answered, "Not really. Unless it was required for an assignment, I didn't really use it." Participant 1 gave a similar answer and highlighted the risk of affecting their grades as the primary factor for why they would use the debugger:

> *"If I had to debug code that was already written and it was by my own free will, then no. Because I'm lazy. So, I'm not going to falter, I would just give up. But if it was say for a grade, right? Or, for some other external reason, and I had to do it [..] and I was stuck, then I would pull out the debugger."*

While participants used the strategy that their classes required, this did not mean they incorporated these strategies into their debugging habits. In fact, Participant 9 described how being forced to learn a strategy actually make them not want to use it outside of class, because they ended up not seeing the value after learning the new strategy:

> *"All right, so I think our first assignment [...] was literally jump right in Eclipse to use the debugger. [The Professor] told us the very first day to go into the debugger. I feel that just being an assignment kind of maybe made me not want to use the debugger. [...] Because I can see how the breakpoints can be useful: you stop and see what's going on at that point. But I think with print statements I can get the exact information I want."*

Participants also described other elements of their social environment, such as the time pressures and deadlines involved with completing their assignments, as reasons for using certain strategies. For example, Participant 10 described using debuggers during lab exams: "I did debuggers on some of [Professor Y's] lab exams simply because I thought I had little time. And then I was like, maybe I could implement a breakpoint to see how that works."

We also saw the effects of time pressures during our own study when participants remarked how the study setting felt so much like a lab exam that they felt compelled to not use strategies like searching online. For instance, Participant 7 remarked how they would use StackOverflow "normally, but here [they didn't] know if it was first allowed to use StackOverFlow and stuff like that."

Long term value.

In addition to anticipating the costs of using certain strategies, participants also described evaluating the longer-term benefits of using strategies as deciding factors for picking them in the first place. Participant 4 described learning and using a strategy only if they received a return on investment in future debugging sessions: "It's almost like if it's not worth learning then I'm not going to. I would rather learn something else worth learning or fix the problem outright if I'm never going to look at this ever again." Participant 8 expressed the same sentiments after learning to use a debugger in class: "I just threw out everything I learned. I'm not using this debugger ever in my life."

Participant 5 described refraining themselves from using certain strategies, because they felt they could learn more: "For [Informatics & Computer Science] 31, 32 or whatever, I didn't really ask for help because [...] I just felt I could probably learn more by solving it myself."

Expected immediate value.

Other participants expressed expecting immediate value when using certain strategies as reasons they chose to use them. For example, Participant 9, as a visual learner, described how they expected value from sketching their code on paper: "Pointers are literally the words, but if you draw it out like with arrows, it helps me a lot to see they're actually pointing to something." As another example, Participant 11 described expecting to see a detailed view of their program's internal state whenever they opened the IDE's debugger:

> *"So in Java it's very convenient because it opens a separate menu and in that separate menu it goes to the code line-by-line. And also it shows the arrays. Each different object is a different window. So, it shows you in depth descriptions of each different object."*

Participant 2 discussed how they expect certain strategies to behave, such as how they help format the program output they produce and how considering this they prefer the one that produces neater output: “the print statement, [...] unless you format it [...] [in] a very neat way, you can dump data onto the screen and rather than formatting it’s definitely way better to just use the debugger where they give you the information.”

Power.

Participants also anticipated gaining power, the sense of control and freedom to manipulate the code or environment, when using certain strategies. For instance, Participant 3 described possessing a sense of control when taking notes on paper: “Maybe it's just my personal preference, but when I write the stuff out I feel I actually do feel I'm in more control.” When describing sketching on paper while debugging, Participant 9 also described similar feelings: “You're free to do anything on the paper. You can draw whatever. I don't really see how anything else can give you this freedom—just writing anywhere.”

Participants also described gaining a sense of control when using both print statements and debuggers. When asked why they preferred using print statements rather than debuggers, Participant 5 described having a sense of control: “I think there's probably more sense of control when you're using a print statement.” Similarly, Participant 9 favored print statements due to the sense of freedom and control that it afforded:

> *“Because [the] debugger lists all the variables, so you have to pick out exactly which one you want. [For example,] ‘Oh, I want to see this variable.’ But print statements is more concisely. It has exactly the information that I want. It’s more catered and well, you can custom say what you want.”*

Section Summary

Participants described their experiences making enrichment decisions that were based on their prior experiences, their present environment, and their immediate and distant future. They would use their prior experience and present environment to determine how suitable certain strategies may be. These elements would factor into their perceptions of future risks and value associated with using a strategy, and their expectations ultimately lead to some decision.

Chapter Summary

This chapter explored how novice developers describe their decision-making process for picking debugging tactics by answering two questions: (1) What debugging tactics do they use? and (2) What activities are involved in selecting a debugging tactic to use?

In answering the first question, we identified three categories of debugging tactics. Participants would run their code to generate new sources of information in the form of program output or views that displayed internal state, such as the debugging views. Moreover, this strategy involved modifying code either through print statements or custom code to test certain program behaviors. The second type of debugging tactic involved seeking help from online resources as well as from other people. This strategy described how novice developers filter for information in both computing and social environments by formulating queries that can narrow down their searches. This strategy saw instances where strategies branched out from the IDE, which many debugging tactics derive from. Lastly, we found that novice developers would take notes on paper to improve the usefulness of the information they gather in other mediums, such as the IDE. Again, this strategy demonstrated how some strategies cross over to other environments.

For our second research question, our findings identified three activities that novice developers do to select debugging tactics. They leveraged past experiences, information they sensed in their current environments, and expectations of the future to inform their enrichment decisions. When leveraging their past experience, they considered previous encounters with strategies and their impressions from those experiences to decide whether or not they should use them. Novice developers also took into account more recent events like their current progress on the debugging task, as well as aspects of their surrounding and computing environments to motivate their decisions. Lastly, novice developers acted in anticipation of the future by deriving estimates for how valuable certain strategies may be or how risky, and then determining whether they should pursue them. Thus, the decision-making activities we identified in this chapter revolved around elements of the past, present, and future.

Chapter 5: Discussion and Implications

This chapter discusses how our results relate to previous studies describing what debugging tactics novice programmers use and what factors may affect their selection. While we find that novice developers use a variety of tactics, all of which already detailed in the literature, we highlight the various environments in which novices execute them. Furthermore, we discuss the significance of understanding how novice programmers debug within different settings and environments and how they make decisions across time throughout this process. We conclude by describing the implications of our findings for educators and future research to better understand novice programmers' debugging processes.

Employing tactics across environments

Unsurprisingly, our results report many of the same debugging tactics that the literature describes. For example, we find that novice programmers trace their code using print statements and breakpoint debuggers, modify their code to test different inputs, and test their code for different execution paths [5][6][16]. All of these debugging tactics fall under our "testing code" theme, and the tactics involve performing actions within the programming environment.

Yet, our results also show that novice programmers employ debugging tactics beyond their programming environments. Our "searching for help" and "taking notes on paper" themes conceptualize tactics where novice programmers utilized their physical and social environments to help them search for information to debug. Like previous studies report, we find tactics where novices look online for information and seek help from other people around them. Moreover, novice programmers who were more visual learners favor using pen and paper to gain additional insight from tracing their code, taking notes, and sketching out their ideas by hand [1][5][6][16].

When applying these tactics, novice programmers switch contexts to the respective environments where they deploy these tactics. For example, they leave their SDE's and navigate to their general computing environment, where they can open their browsers to search for debugging information online. When using pen and paper to debug, they switch contexts to their physical surroundings to leverage physical mediums that allow them to better visualize their code compared to their SDE's screens. Likewise, when they fail to make progress using tactics within their SDE's, such as with print statements and debuggers, they search for knowledgeable people to ask questions within their social surroundings. They leverage their social networks in their search for information, often relying on classmates, friends, and formal resources like lab tutors and professors at their institutions. Thus, novice programmers leverage their SDE's as well as their non-computing environments to debug.

Yet, previous studies overlook the role that non-computing environments play in how novice programmers choose their debugging tactics. Much like the debugging education literature, the IFT literature has not fully explored the debugging tactics that novice programmers apply outside of the conventional programming environment. Debugging studies that leverage IFT often focus on foraging behavior within SDE's and implicitly in the physical environment, as Piorkowski et al. do with their cursory mention of the to-do listing as an enrichment strategy [13][17]. Even Pirolli and Card, who first conceptualized IFT, do not operationalize the task environment beyond the computing and physical environments [18]. Our results demonstrate that novice programmers apply debugging tactics within non-computing environments as well, and therefore, they should be emphasized.

As our results show, novice programmers make cost and benefit decisions informed by elements of their task environments. For instance, novice programmers make inquiries to a variety of sources, including their peers, friends, tutors, and professors. From an IFT perspective, novice programmers are the predators seeking information, the prey. This information resides with people who are the information patches that make up the social environment. Novice programmers navigate through this social network of people in search of information and weigh certain costs and benefits associated with asking other people for help. We can analyze their behavior of asking questions as a form of enrichment strategy where they formulate their questions precisely like they do with search queries on Google to elicit the most useful and least costly information for their information goals. In our own study, some novices describe asking authorities like professors in their social networks as costlier than asking their peers, since the act induces more anxiety and fear of looking incompetent. Thus, expanding our operationalization of IFT's environment construct to include non-computing environments may yield insight into how novice programmers debug.

By exploring how novice programmers debug within other settings, such as their classrooms and universities, we gain a holistic understanding of the debugging process. Consequently, there are practical incentives in understanding how novice programmers debug within these non-computing environments. Novice programmers learn and practice debugging within their classrooms and institutions and outside of class in their homes and on the Internet. Understanding how they debug in these various environments may point to areas where these environments could change to successfully facilitate the debugging tactics that novice programmers use.

Making decisions across time

In addition to the debugging tactics that novice programmers deploy in various environments, our results also describe the activities involved with selecting these tactics and they expand on existing literature that describe this process. Our results confirm Gould's suspicion that the professional programmers in their study selected debugging tactics according to pre-existing factors such as their knowledge, habits, and present experience during the debugging session [8]. Despite the differences in study samples, Gould's description of how professional programmers select their debugging tactics is consistent with our findings for novice programmers. For example, we find that novice programmers use certain strategies that they have learned early in their careers and that have become mainstays in their debugging toolbelts to the point that they tacitly select them out of habit. Novice programmers also consider how amenable certain strategies may be to their task environments and they adapt accordingly. For example, when considering using print statements in an SDE for the Assembly language, novice programmers opt for the built-in debugger instead. They explain that this behavior avoids the unnecessary overhead of writing several lines of code to simulate print statements in a low-level language such as Assembly, and they gain the same benefit of adding single-lined print statements in a high-level language like Python. Lastly, our results show that novice programmers make debugging decisions by anticipating future risks and rewards. For instance, novices estimate how much time they have to invest to use print statements compared to the debugger, and then they pick the least costly option. They also consider social factors such as the class requirements that their professors set and whether using certain strategies may yield a higher or lower grade. These behaviors align with IFT's concept of scent that describes how programmers evaluate the value and cost of certain foraging behaviors when pursuing a piece of debugging information [10].

Unlike previous studies, however, we highlight the role of time, as all three activities that describe how novice programmers select their debugging tactics revolve around decision-making across different periods of time. When picking a debugging tactic in the present, novice programmers consider their previous experiences using a particular strategy. For example, some participants mention experiencing high costs or low rates of return when using certain strategies, and consequently they lower their expectations when reconsidering those options in the future. Thus, when they decide on debugging tactics, they opt for tactics that have proven to work well in the past. This includes strategies that may not be the optimal choice for their present task but are elevated in status because of their past success. Similarly, we also find that some novices stick to their old habits when debugging by choosing strategies that they have consistently used. They indicate a preference for tactics that they find natural and familiar even if the tactic would not be the most effective.

Consequently, time plays an important role in shaping how novice programmers make debugging decisions. For novices to make decisions that accurately align their expectations of the cost and value of using certain tactics with reality, we suggest that educators explain to novice programmers the costs and benefits of using each particular debugging tactic. As some of our participants mention, they did not adopt certain tactics earlier, because they did not realize the value of certain tactics until experiencing it themselves. Thus, demonstrating this value in the classroom may help novice programmers develop skills earlier in accurately assessing the effectiveness of certain debugging tactics for their present debugging problems. As Fitzgerald et al. suggest, educators could teach these concepts through heuristics or as general advice given as debugging best practices [5]. Novices may fail to choose the optimal debugging tactic simply because their expectations of cost and value in using those tactics are misaligned with reality. Focusing on ensuring that novice programmers gain the necessary experience early in their academic careers to make these debugging decisions will be vital for them to develop better debugging abilities.

Chapter 6: Conclusion

This thesis presented a study that describes the debugging tactics that novice programmers use and the activities involved in their selection. We focused on understanding the debugging tactic selection process due to inadequate descriptions of this process in the debugging education literature. Motivated by the lack of material on this topic, we chose a qualitative study to explore this phenomenon and provide rich descriptions. We leveraged heavily on semi-structured interviews that were supported and inspired by direct observations of novice programmers debugging and responses from a preliminary questionnaire eliciting information about participants' educational and technical backgrounds.

Our results showed that novice programmers employ a variety of debugging tactics across various environments. These debugging tactics included testing their code, taking notes on paper to process information more effectively, and asking people in their social networks for help. Furthermore, novice programmers would employ these tactics across various environments including their general computing environments as well as non-computing environments such as their physical and social environments. To make decisions on which debugging tactic to use, we found that novice programmers consider their past experiences using particular debugging tactics and their experiences during their debugging sessions. These activities along with their expectations of future risks and rewards in using particular tactics would inform their decision making.

Accordingly, we discussed the potential benefits of teaching the value and risks of using certain debugging tactics to novice programmers. As novice programmers develop their debugging skills over time, so too do their perceptions of the usefulness of certain debugging tactics. Explicitly teaching the cost and benefits of debugging tactics builds experience that novice programmers can leverage in future debugging situations to pick the optimal debugging tactic for their problem. Programming educators may teach novice programmers how and when to use various debugging tactics, but students ultimately decide whether or not to use them.

This work paves the way for future studies in IFT to expand on the operationalization of the environmental construct. As debugging studies in IFT often focus on the SDE it may be worthwhile for future work to consider that novice programmers leverage debugging tactics in non-computing environments as well. Like previous works, our study did not anticipate the significance of debugging in non-computing environments. We recommend a future study on novice programmers in their natural debugging settings, like their classrooms and homes to describe more realistic debugging behaviors. Future studies that utilize a similar artificial lab setup as ours could organize multiple participants debugging in the same room and observe their interactions to capture the social debugging elements involved. By expanding the scope of future studies to include various computing and non-computing environments, we may gain greater insight into how novice programmers debug.

References

[1] Benander, Alan C., and Barbara A. Benander. “An Analysis of Debugging Techniques.” *Journal of Research on Computing in Education* 21, no. 4 (June 1989): 447–55. https://doi.org/10.1080/08886504.1989.10781893.

[2] Braun, Virginia, Victoria Clarke, Nikki Hayfield, and Gareth Terry. “Thematic Analysis.” In *Handbook of Research Methods in Health Social Sciences*, edited by Pranee Liamputtong, 843–60. Singapore: Springer Singapore, 2019. https://doi.org/10.1007/978-981-10-5251-4_103.

[3] Creswell, John W. *Educational Research: Planning, Conducting, and Evaluating Quantitative and Qualitative Research*. 4th ed. Boston: Pearson, 2012.

[4] Feigenspan, J., C. Kastner, J. Liebig, S. Apel, and S. Hanenberg. “Measuring Programming Experience.” In *International Conference on Program Comprehension(ICPC)*, 73–82, 2012. https://doi.org/10.1109/ICPC.2012.6240511.

[5] Fitzgerald, Sue, Gary Lewandowski, Renée McCauley, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. “Debugging: Finding, Fixing and Flailing, a Multi-Institutional Study of Novice Debuggers.” *Computer Science Education* 18, no. 2 (June 2008): 93–116. https://doi.org/10.1080/08993400802114508.

[6] Fitzgerald, S., R. McCauley, B. Hanks, L. Murphy, B. Simon, and C. Zander. “Debugging From the Student Perspective.” *IEEE Transactions on Education* 53, no. 3 (August 2010): 390–96. https://doi.org/10.1109/TE.2009.2025266.

[7] Fleming, Scott D., Chris Scaffidi, David Piorkowski, Margaret Burnett, Rachel Bellamy, Joseph Lawrance, and Irwin Kwan. “An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks.” *ACM Trans. Softw. Eng. Methodol.* 22, no. 2 (March 2013): 14:1–14:41. https://doi.org/10.1145/2430545.2430551.

[8] Gould, John D. “Some Psychological Evidence on How People Debug Computer Programs.” *International Journal of Man-Machine Studies* 7, no. 2 (March 1, 1975): 151–82. https://doi.org/10.1016/S0020-7373(75)80005-8.

[9] Gould, John D., and Paul Drongowski. “An Exploratory Study of Computer Program Debugging.” *Human Factors* 16, no. 3 (June 1, 1974): 258–77. https://doi.org/10.1177/001872087401600308.

[10] Katz, Irvin R., and John R. Anderson. “Debugging: An Analysis of Bug-Location Strategies.” *Hum.-Comput. Interact.* 3, no. 4 (December 1987): 351–399. https://doi.org/10.1207/s15327051hci0304_2.

[11] Ko, Andrew J., Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. “An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks.” *IEEE Trans. Softw. Eng.* 32, no. 12 (December 2006): 971–987. https://doi.org/10.1109/TSE.2006.116.

[12] “Laddering: A Research Interview Technique for Uncovering Core Values :: UXmatters.” Accessed December 4, 2018. https://www.uxmatters.com/mt/archives/2009/07/laddering-a-research-interview-technique-for-uncovering-core-values.php.

[13] Lawrance, J., C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming. “How Programmers Debug, Revisited: An Information Foraging Theory Perspective.” *IEEE Transactions on Software Engineering* 39, no. 2 (February 2013): 197–215. https://doi.org/10.1109/TSE.2010.111.

[14] Lincoln, Y. S., & Guba, E. G. (1985). *Naturalistic inquiry*. Beverly Hills, Calif: Sage Publications.

[15] McCauley, Renée, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. “Debugging: A Review of the Literature from an Educational [Perspective.” *Computer Science Education* 18, no. 2 (June 2008): 67–92. https://doi.org/10.1080/08993400802114581.

[16] Murphy, Laurie, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. “Debugging: The Good, the Bad, and the Quirky – a Qualitative Analysis of Novices’ Strategies.” In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, 163–167. SIGCSE ’08. New York, NY, USA: ACM, 2008. https://doi.org/10.1145/1352135.1352191.

[17] Piorkowski, David J., Scott D. Fleming, Irwin Kwan, Margaret M. Burnett, Christopher Scaffidi, Rachel K.E. Bellamy, and Joshua Jordahl. “The Whats and Hows of Programmers’ Foraging Diets.” In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI ’13*, 3063. Paris, France: ACM Press, 2013. https://doi.org/10.1145/2470654.2466418.

[18] Pirolli, Peter, and Stuart K. Card. “Information Foraging.” *Psychological Review* 106 (1999): 643–675.

[19] Qian, Yizhou, and James Lehman. “Students’ Misconceptions and Other Difficulties in Introductory Programming: A Literature Review.” *ACM Transactions on Computing Education* 18, no. 1 (October 27, 2017): 1–24. https://doi.org/10.1145/3077618.

[20] Romero, Pablo, Benedict du Boulay, Richard Cox, Rudi Lutz, and Sallyann Bryant. “Debugging Strategies and Tactics in a Multi-Representation Software Environment.” *International Journal of Human-Computer Studies* 65, no. 12 (December 2007): 992–1009. https://doi.org/10.1016/j.ijhcs.2007.07.005.

[21] Vessey, Iris. “Expertise in Debugging Computer Programs: A Process Analysis.” *International Journal of Man-Machine Studies* 23, no. 5 (November 1, 1985): 459–94. https://doi.org/10.1016/S0020-7373(85)80054-7.