UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

DEVELOPING CODE READING SKILLS IN ENTRY-LEVEL COMPUTER SCIENCE COURSES

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

DOCTOR OF PHILOSOPHY

By

KEERTI BANWEER

Norman, Oklahoma

2025

DEVELOPING CODE READING SKILLS IN ENTRY-LEVEL COMPUTER SCIENCE COURSES

A DISSERTATION APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Dr. Deborah Trytten, Chair

Dr. Amy McGovern

Dr. Dean Hougen

Dr. Amy Bradshaw

Acknowledgements

I would like to express my deepest gratitude to my advisor, Dr. Deborah Trytten, for her unwavering support, guidance, and encouragement throughout my journey in the Ph.D. program. Her generosity with her time and expertise helped me shape my research around my passion for teaching and improving instructional methods in academia. I am sincerely thankful to my dissertation committee members—Dr. Amy McGovern, Dr. Dean Hougen, and Dr. Amy Bradshaw—for their continuous support, thoughtful feedback, and encouragement during the most challenging phases of my doctoral studies. Their insights and mentorship have been invaluable to the development and completion of this research.

I would also like to acknowledge the School of Computer Science, the Gallogly College of Engineering, and the staff at OU's Graduate College for their support in facilitating my research and providing assistance whenever needed. Special thanks go to Dr. Quiroga Allison, Dr. Brian McSkimming, and Dr. Casey Haskins for their help in collecting student feedback and consent, which was essential for the successful implementation and evaluation of the **Code Insight** activity. Their collaboration and support played a crucial role in the data collection process.

Lastly, I extend my heartfelt thanks to my family and friends. Your patience, encouragement, and unwavering belief in me have been my foundation throughout this journey. It is with your love and support that I have reached this milestone.

Contents

List of Tables

List of Figures

Abstract

Code review and debugging are fundamental practices in software development, essential for ensuring code quality, consistency, and knowledge sharing. With the growing use of AI-based tools for code generation, these skills have become even more critical. While large language models (LLMs) can accelerate code writing, they also introduce new challenges related to code correctness, maintainability, and integration with existing systems. As a result, effective code review and debugging are increasingly important in both industry and education.

In computer science (CS) education, these skills are equally vital for student success. In large-scale programming courses, where autograders are commonly used, students must be able to identify, locate, and fix errors independently. However, despite their importance, code review and debugging are often neglected in early programming instruction. This is largely due to the complexity and resource demands of integrating these practices into the curriculum, both for instructors and students. Without explicit instruction, students often rely on trial-and-error methods, which can lead to frustration, reduced confidence, and poor learning outcomes.

This dissertation addresses this instructional gap by examining the challenges students face in learning debugging and code review at different stages of a CS program. Using qualitative methods, the study explores how students develop debugging skills in a CS program at the University of Oklahoma and how the absence of structured guidance affects their learning. The findings from this study indicate that even students with intermediate to advanced programming experience struggle with debugging when not explicitly taught in instruction, and that unguided practice can have a negative impact on their learning experience.

To address these challenges, this research introduces *Code Insight*, a novel instructional intervention to be implemented as an in-class activity, grounded in the principles of active learning. Designed for integration into classroom lectures, Code Insight simplifies the teaching and learning of code review and debugging practices in introductory programming

courses. This intervention emphasizes code analysis over code creation and is structured to minimize grading in instructional burden. Importantly, it is intentionally designed to minimize excessive cognitive workload on novice programmers in a CS program, ensuring that the activity supports learning without introducing unnecessary complexity and additional workload on students as well as instructors.

The effectiveness of **Code Insight** intervention was evaluated through anonymous student feedback collected across two semesters. The results indicate that the students felt more confident in their ability to review and debug code after participating in the activity. By identifying and addressing common difficulties in learning to debug a program, this dissertation presents a practical and scalable solution for enhancing programming instruction starting from introductory courses in a CS program. It highlights a critical gap in integrating code review and debugging practices in CS education and offers a pedagogical approach that supports the development of essential programming skills and enhances students' motivation and self-efficacy when working with complex programming tasks.

Chapter 1

Introduction

Artificial Intelligence (AI) has made remarkable progress and is now deeply embedded in various aspects of daily life, from AI-powered medical diagnostics to autonomous vehicles [64]. In the software industry, AI-based tools that can automatically generate code are transforming the landscape of programming practices among developers. These tools have the potential to enhance developer productivity by generating code segments and reducing the time required to write a program from scratch [23]. However, this shift also places greater emphasis on the programmer's ability to critically review and debug AI-generated code by analyzing and evaluating the code's functionality to ensure its correctness, maintainability, and consistent integration with existing systems.

This chapter discusses the importance of code review and debugging skills among students in the CS program and how this dissertation addresses the existing gaps in the literature on peer code review, tackling the challenges associated with integrating these complex and time-consuming yet crucial and essential skills as part of explicit instruction.

1.1 Beyond Writing Code: The Role of Review and Debugging

Writing code is only one part of a software developer's responsibilities. Equally important are tasks such as *code review* and *debugging*, which are essential for maintaining software quality, ensuring consistency, and adapting to evolving technologies. These practices have long been integral to the software development process and remain crucial in the era of AI. As developers increasingly rely on AI-generated code, they spend more time reviewing and debugging to ensure that the output meets functional and quality standards [109].

Code review serves multiple purposes beyond quality assurance. It supports knowledge sharing, promotes consistency, encourages exploration of alternative solutions, and fosters team collaboration [97]. Whether AI tools are involved or not, code review remains a cornerstone of professional software development, providing several benefits for developers maintaining software systems.

1.1.1 AI in CS Education: Opportunities and Challenges

In CS education, the use of AI-powered tools by students is becoming increasingly common and inevitable due to their accessibility and ease of use. While these tools offer benefits such as faster code generation and real-time assistance, and may have the potential to improve the learning experience, the prevalence of AI in CS education also raises serious concerns. One of the most significant challenges is the increased risk of plagiarism, as students may submit AI-generated code without a thorough understanding of it, which can negatively impact learning outcomes and academic integrity.

To address concerns about plagiarism, one suggested possibility is for CS researchers to explore innovative ways to integrate structured code review and debugging activities into programming courses, thereby improving the learning experience of students in the presence of AI-based tools by encouraging them to effectively analyze and review the code [26]. These

practices not only promote deeper skill development but also serve as a pedagogical response to the ethical and educational challenges posed by AI tools in CS education. Strategic and systematic integration of these activities can help students engage more meaningfully with code and reduce overreliance on AI-generated solutions.

1.1.2 Instructional Gaps in Current Practice

Despite their importance in industry and academia, code review and debugging practices are often part of the hidden curriculum in programming courses. These skills are rarely taught explicitly, and previous efforts to integrate them into instruction have faced multiple challenges. For instance, peer code review activities often suffer from low student engagement [100], while debugging instruction is hindered by students' low self-efficacy and frustration tolerance [113]. Additionally, the effective use of digital tools for these practices requires technical infrastructure, instructor training, and institutional support —resources that are often limited [11].

This dissertation contributes to the growing body of research in CS education advocating for innovative instructional strategies that incorporate essential skills such as code review and debugging into CS programming courses to improve the learning experience of the students and prepare them for their future in industry. The goal of this research is to equip students with essential programming skills while minimizing additional responsibilities and resource requirements for instructors and excessive cognitive load on learners. Specifically, this research examines strategies that foster the development of code review skills among novice programmers in introductory courses, while enhancing their self-efficacy when working on complicated programming tasks. It aims to enhance educational experiences, foster critical thinking, problem-solving, and collaboration, and better prepare students for the evolving demands of the software industry.

1.2 Motivation

Although code review and debugging are critical in industry, they are often underemphasized in CS programming courses. Students are frequently expected to develop these skills independently through the practice of class material and trial and error rather than through structured instruction. As a result, these practices become part of the hidden curriculum, leading to inconsistent skill development among programmers at different levels within the CS program, which impacts their learning experience and the development of essential skills.

1.2.1 The Impact of AI and Large Class Sizes

The growing use of AI-assisted tools in education further complicates this issue. These tools can obscure students' understanding of programming concepts and raise concerns about academic integrity [26]. At the same time, increasing enrollment in CS programs has led to larger class sizes and increased instructional demands for programming-intensive courses [22]. To manage this growth, many institutions rely on autograders, which provide immediate feedback to the students and reduce grading workloads for the educators [49]. However, these tools are only practical if students can interpret autograders' feedback and debug their code, skills that are often underdeveloped and one of the key challenges among beginner programmers, leading to frustration and low self-efficacy [72].

1.2.2 Addressing the Gap

Existing code review frameworks are often too complex for introductory programming courses in a CS program and are found to be challenging for beginner programmers. They can place an extensive cognitive load on novice programmers and increase workload through grading burdens for instructors [3, 105]. Students may also lack motivation to engage in additional assignments, especially when they struggle with foundational programming concepts and program writing skills [5, 66]. These challenges are even more pronounced in intermediate-

level courses, where students are expected to work with multiple programming languages within a short timeframe while addressing the challenges of evaluating and debugging the program to identify and resolve any issues.

This research addresses a gap in the literature by proposing a simplified and adaptable in-class activity grounded in the principles of integrated learning theories, such as constructivist, active, and situated learning theories, to encourage the development of code review and debugging skills for students starting from beginner-level programmers in a CS program. This in-class activity is designed to integrate code review and debugging practices into introductory programming courses in a way that is scalable, manageable, and pedagogically sound. It aims to improve student engagement, enhance motivation, and support the development of essential programming skills without increasing the workload for instructors or learners [43, 113].

1.3 Research Focus and Objectives

This research addresses the challenges of integrating complex and time-consuming code review and debugging activities into introductory programming courses. It focuses on supporting the development of essential programming skills among beginner students. Specifically, the study investigates two key research problems in CS education related to the lack of explicit instruction in these practices:

1. Why is it important to teach code review skills to novice programmers?

2. How can a code review activity be designed and integrated into instruction to enhance student learning, by improving motivation, engagement, and self-efficacy, without increasing instructors' workload or students' cognitive load?

To address the first research problem, this study adopts a qualitative approach to examine how students with intermediate to advanced programming experience approach programming

tasks. It investigates the challenges that arise when code review and debugging are not explicitly taught, as outlined in the research question presented in Table 1.1. The study examines the strategies students employ to write and evaluate code independently, as well as how they develop debugging skills through hands-on experience in coursework. It also discusses the difficulties students face as they progress through a CS program, particularly as programming tasks become more complex.

To address the second research problem, this dissertation introduces an in-class, active learning intervention called **Code Insight** to be integrated into lectures and implemented during lectures on the programming concepts by providing a systematic, guided process by the instructors. Designed for use during lectures and classroom activities, this intervention helps students build code review and debugging skills by analyzing peer-written code. This research problem is explored through two research questions, also detailed in Table 1.1. The goal is to support students in acquiring essential skills without adding a significant workload for instructors or placing excessive cognitive demands on students new to programming concepts. Using both qualitative and quantitative methods, the study evaluates the impact of integrating **Code Insight** on students' understanding of programming concepts, their ability to review and debug code, and their performance in programming tasks such as projects and exams.

This dissertation addresses the challenges of integrating the complex process of code review into introductory programming courses while guiding students to analyze and evaluate the code. It also tackles the common issues of low student engagement and motivation among novice programmers by simplifying the code analysis process through a strategically guided integration within classroom instruction. The following research objectives guide this study:

1. Identify common programming practices and problem-solving strategies used by students at different levels in a CS program (Chapter 3).

2. Design and implement an active learning activity for in-class integration to enhance students' learning experiences and engagement (Chapter 4).

3. Evaluate the impact of structured code review activities on students' ability to review and debug code, as well as their overall programming performance and learning experience with this intervention during class lectures (Chapter 5).

This dissertation uses both qualitative and quantitative methods to achieve these objectives. It highlights the need for practical ways to integrate code review and debugging into CS courses to enhance student learning and support the development of essential programming skills. By proposing a scalable and adaptable instructional in-class intervention, this dissertation contributes to the ongoing efforts in CS education to improve teaching practices for essential skill development and address the challenges posed by the increasing use of AI tools in programming courses.

Table 1.1 outlines the research questions addressed in this dissertation and summarizes the key contributions of this study to CS education. Each chapter listed in the table corresponds to a specific research question and provides a detailed discussion of the methodology, implementation, and findings.

Table 1.1: Research Questions and Summary of Contributions.

Research Question	**Contributions**	**Chapter**
*	**Table 1.1 – Continued on the next page**	*

Research Question	Contributions	Chapter
What challenges do intermediate to advanced CS students face throughout their undergraduate program at University of Oklahoma, and how do they navigate these difficulties?	• Investigate common programming and problem-solving practices used by students with intermediate to advanced experience. • Explores why students adopt specific debugging strategies in the absence of formal instruction. • Highlights the limitations of treating code review and debugging as part of the hidden curriculum in CS education. • Identifies key challenges and obstacles encountered by students as they progress through increasingly complex programming tasks. • Examines how students overcome these challenges through both formal and informal learning strategies without systematic guidance.	3

* **Table 1.1 – Continued on the next page** *

Research Question	Contributions	Chapter
How can an in-class active learning framework be designed and implemented to effectively support the development of code reading and debugging skills among novice programmers?	• Introduces *Code Insight*, an active learning framework designed to systematically guide students in analyzing and evaluating code inspired from peers from previous semester, while minimizing cognitive load for learners new to the complex tasks of programming and reducing grading effort for instructors. • Addresses common challenges such as low engagement and motivation in code review activities among beginner programmers. • Provides empirical evidence on the effectiveness of structured code analysis in improving students' understanding of programming concepts, as well as their code reading and debugging skills.	4

* **Table 1.1 – Continued on the next page** *

Research Question	Contributions	Chapter
How does the *Code Insight* framework influence the code writing skills of novice programmers in an introductory programming course?	• Examines the impact of the *Code Insight* active learning framework on the development of code writing skills among novice programmers. • Investigates how structured code review activities contribute to students' understanding of programming logic, structure, and style.	5

1.4 Contributions

To provide a comprehensive analysis of the research questions outlined in Table 1.1, this dissertation begins with a qualitative study involving interviews with CS program majors. The study focuses on students with intermediate to advanced programming experience, examining their learning experiences related to essential skills such as code review and debugging. It aims to identify the challenges students face when developing debugging skills independently through a hands-off approach and highlights the importance of explicitly incorporating code review and debugging activities into instructional practices.

The findings from this study underscore the urgency of integrating code review and debugging into programming courses. Building on these insights, the dissertation introduces the design and implementation of the in-class intervention, **Code Insight** activity, which aims to address gaps in the literature regarding the integration of peer code review in programming courses from a CS program. This activity is evaluated for its impact on students'

learning experiences and their development of foundational programming skills, with a focus on code review and debugging. The contributions presented in the following chapters provide evidence to support further research in CS education and encourage the adoption of instructional strategies that promote these essential skills.

1.4.1 Learning Experiences and Challenges of Intermediate to Advanced Programmers in a CS Program

The research presented in Chapter 3 contributes to understanding how students in a CS program develop debugging skills, particularly when these skills are not explicitly taught in programming courses. This chapter presents findings from a qualitative study that explores how students with intermediate to advanced programming experience approach programming tasks and problem-solving while tackling complex debugging tasks.

The study investigates the common strategies students use to write, test, and debug code, and findings suggest that they often rely on trial-and-error methods. The findings generated from the qualitative research provide in-depth insights into how students develop programming habits and debugging strategies without any explicit instruction. The analysis further presents patterns to understand why they tend to adopt specific ineffective strategies to debug the program. These findings reveal patterns in how students learn to debug through experience and follow ad hoc strategies when writing a program.

This chapter also identifies the key challenges students face as they progress through increasingly complex programming tasks. These include difficulties with systematically debugging the program and managing frustration during the debugging process. Without any strategic guidelines, the findings suggest that students apply ineffective debugging strategies to identify and fix errors in their programs, increasing the time spent on program development.

Notably, the research highlights the limitations of treating code review and debugging as part of the hidden curriculum. The results from this study emphasize that students often re-

tain the practices they develop early in their learning journey in the CS program. Therefore, the study presents arguments for integrating structured code review activities, starting from introductory programming courses. Providing strategic and explicit support can reduce reliance on informal methods, leading to improved learning outcomes and increased confidence in programming among students.

1.4.2 Developing In-class Code Review Activity

This contribution is based on the research presented in Chapter 4, which introduces a novel and structured in-class activity called **Code Insight**, implemented as a intervention during class lectures. This study introduces and evaluates the **Code Insight** intervention to support the development of code review and debugging skills among novice programmers. This in-class activity is designed to strategically guide students through the process of analyzing and evaluating code inspired by their peers responses from previous semester exams. It is intentionally designed to be a simple and adaptable activity, minimizing both additional excessive cognitive load for students and grading overhead for instructors. The simplicity of its structure makes it feasible for integration into large, introductory programming courses without requiring significant additional resources.

The in-class intervention that utilizes active learning, directly tackles common issues faced in early programming education, such as low student engagement and motivation when working on code review activities independently. By embedding the activity within regular classroom instruction and using peer-generated code, **Code Insight** fosters a more interactive and relatable learning environment. The in-class implementation encourages active participation and helps students see the relevance of code review to their learning.

Through a combination of qualitative and quantitative methods, the study provides some empirical evidence of the effectiveness of structured code analysis in improving students' understanding of programming concepts. The findings demonstrate that students who engage with **Code Insight** intervention mention greater confidence in analyzing and writing code.

The findings suggest that student feel more confident when debugging their program, suggesting a contribution towards improving self-efficacy. These results support the value of integrating structured code review into early-stage programming curricula.

1.4.3 Evaluating the Impact on Students' Learning and Skill Development

The research presented in Chapter 5 examines how structured code review activities, implemented through the **Code Insight** intervention, influence the development of code writing skills among novice programmers.

This study analyzes students' performance data to assess the direct impact of participating in **Code Insight** activities on their ability to write code. In particular, it focuses on students' performance in programming problems during exams and investigates how these outcomes correlate with their engagement in the code review tasks.

Based on the empirical findings, this dissertation provides practical, evidence-based recommendations for incorporating structured code review into early-stage programming curricula. These recommendations are designed to support the development of essential programming skills in a way that is both pedagogically effective and feasible for implementation in large or resource-constrained classroom settings.

1.5 Organization of the Dissertation

The remaining chapters of this dissertation are organized as follows:

Chapter 2 provides an overview of peer code review in both academic and CS education contexts. It includes a review of prior research on peer code review practices. It discusses the theoretical framework that informs the design and implementation of the in-class code review activity for introductory CS courses.

Chapter 3 presents a qualitative study exploring how CS students develop debugging skills throughout their academic journey. This chapter examines the impact of the absence of explicit instruction in code review and debugging, highlighting the challenges and obstacles students encounter in CS courses.

Chapter 4 describes the design and development of the **Code Insight** activity. It outlines the in-class implementation of the proposed in-class intervention integrated during class lectures, guided by the integrated theoretical model. This chapter also presents a qualitative evaluation of students' learning experiences, focusing on changes in their self-efficacy and comfort with debugging tasks.

Chapter 5 presents a mixed-methods evaluation of the impact of the **Code Insight** activity on student performance. It analyzes students' performance in code-writing tasks and investigates whether participation in the activity correlates with improvements in programming skills and learning outcomes.

Chapter 6 summarizes the key findings of the dissertation and discusses their implications for CS education. It also describes possible future work, including ways to expand the study into longitudinal research to explore the development of essential skills among students and explore how the **Code Insight** intervention works in different classroom settings and with greater integration of group discussion among students.

Chapter 2

Background and Related Work

This dissertation introduces a novel in-class intervention designed to support the development of essential programming skills among novice programmers in a CS program and enhance their overall learning experience. This chapter provides background and related work that inform the design of this intervention to be integrated into instruction, with a particular focus on the importance of skills such as code review and debugging. These skills are critical for understanding and improving code, yet they are often challenging for beginners. By addressing these challenges, this research aims to support students in building confidence and competence when working with complex programming tasks, such as analyzing, evaluating, and debugging code.

To inform the design of an in-class code review activity for introductory programming courses, this chapter reviews prior research on the difficulties faced by beginner programmers. It also examines the role of code review in both academic and professional settings, highlighting its value in improving code quality and fostering collaboration. The chapter discusses how incorporating code review into instruction can enhance student learning and engagement.

Finally, this chapter identifies key gaps in the existing literature on code review in CS education and presents the theoretical framework that guided the development of the proposed

activity. This in-class intervention supports the goal of helping beginner programmers develop foundational programming skills through structured, peer-based learning experiences.

2.1 Foundational Skills in Programming: Code Review and Debugging

Code review and debugging are essential practices in the software development process, playing a crucial role in ensuring code quality, reliability, and maintainability. Code review functions as a collaborative quality assurance process that helps enforce coding standards, detect defects early, and facilitate knowledge sharing among team members [1, 7].

Debugging, in contrast, is a cognitively intensive activity that involves identifying, analyzing, and resolving faults in a program. It requires developers to form hypotheses, test them, and iteratively refine their understanding of the program's behavior [33]. These skills are critical in professional software engineering and should be developed during CS education among students.

The focus of this dissertation is to encourage the development of applicable code review skills among new programmers by teaching them how to critically analyze and evaluate existing code, identify potential issues, and explore different conceptual strategies for resolving them. By engaging in these practices, students receive much-needed practice on essential skills, which will allow them to gain confidence in reading and understanding code, ultimately improving their learning experience and code-reading and debugging abilities.

2.1.1 Code Review Practices in Professional Software Development

In industry, code review is a standard and essential practice used to improve software quality, support team collaboration, and facilitate knowledge sharing by providing multiple approaches to improve the code. It enables the early detection of defects, promotes consistent coding practices, and facilitates efficient knowledge transfer to new developers by exposing

them to organizational standards and common programming practices for consistency and shared knowledge of the existing system [85, 8, 87].

Software developers in an industry often dedicate a considerable portion of their time to reviewing code written by their peers and providing constructive feedback by analyzing and evaluating the code in detail. For instance, a study of open-source software (OSS) projects found that developers spend an average of six hours per week reviewing or preparing code for review. In many organizations, code review is a formal part of the development workflow [13]. At Microsoft, approximately 50,000 developers engage in code reviews each month. To support this process, companies have developed internal tools to streamline code review activities, such as Microsoft's CodeFlow [14].

Recent research has further highlighted the evolving role of modern code review in professional settings. A large-scale survey by Yang et al. [114] synthesized findings from over 230 studies, identifying key benefits of modern code review, including improved code maintainability, enhanced team communication, and better onboarding experiences for the new developers in the team. The study also noted several challenges, including reviewer fatigue and inconsistent review quality, which highlights that code review is a complex and time-consuming task.

Code reviews contribute to defect detection early in the development process, proving effective in enhancing customer satisfaction and reinforcing their value as a quality assurance mechanism [45]. Similarly, another empirical investigation found that modern code review practices, even when lightweight, have a measurable positive impact on software quality and team productivity [68].

These findings provide insight into the integral role of code review in contemporary software engineering, not only as a technical safeguard but also as a social and educational process that supports continuous learning and collaboration within development teams. Due to the lack of explicit instruction on code review practices, many new graduates are underprepared for this complex and cognitively demanding task in the industry and should be

prioritized as part of the explicit instruction in programming courses in CS education [86].

2.1.2 Peer Review in Academic Research and Practice

Peer review is an effective process by which scholarly or professional work is evaluated by individuals with similar qualifications or expertise, providing meaningful feedback to ensure its accuracy, quality, and alignment with disciplinary standards [98, 63]. Peer review has contributed to an enhanced learning experience for students in higher education, where students develop skills to effectively review and evaluate their peers' work, providing constructive feedback to assist in improving their peers' work and acquiring critical thinking and problem-solving skills in the process. As a collaborative learning activity, peer review encourages students to engage with different perspectives and ideas of their peers and critically assess alternative approaches, contributing towards the improvement of their peers' work [38]. This process not only supports the development of critical thinking and communication skills but also fosters a sense of academic community and shared responsibility for learning.

Peer review proves to be equally beneficial to instructors by providing students with timely feedback and enhancing their learning experience, allowing them to reflect on their work and gain knowledge from different perspectives offered by their peers. It also promotes active learning by encouraging students to take ownership of their learning process. Research shows that receiving feedback from peers helps students iteratively improve their work, which can enhance both academic performance and self-regulated learning [89, 107].

Recent studies have further emphasized the pedagogical value of peer review. Wei and Liu [108] conducted a systematic review of 60 studies and identified a wide range of benefits, including cognitive, social, and metacognitive gains. Their findings also highlight the importance of designing structured peer feedback processes to overcome common challenges such as inconsistent feedback quality and student discomfort with critique.

Peer review, when integrated with effective feedback strategies, can greatly improve stu-

dent motivation, engagement, and performance [111]. Similarly, research on peer assessment design stresses the importance of training, multiple review iterations, and flexibility in assessment formats to enhance the student experience and learning outcomes [29].

Considering the positive impacts on student learning and engagement, peer review continues to gain popularity across disciplines in higher education to foster deeper learning and meaningful collaboration [76].

2.1.3 Implementing Peer Code Review

Peer code review offers valuable educational benefits by enabling students to learn from one another's perspectives in both academic and professional settings [58, 35]. Recognizing the importance of collaborative learning, numerous studies have investigated how peer review, often referred to as *code review* or *peer code review*, can be effectively integrated into programming courses. This method supports critical thinking, encourages reflective learning, and helps students better understand programming concepts by reviewing their peers' work [31].

A substantial body of research has explored the use of code review in CS education across various academic levels, including undergraduate and graduate programs [43]. This dissertation focuses on introductory programming courses in a CS program, where peer code review is introduced as a teaching strategy to support beginners with a systematic, guided code analysis. The goal is to strengthen students' understanding of programming concepts and promote the development of essential skills such as code reading and debugging.

The primary objective of this work is to help novice programmers develop foundational skills in reading and analyzing code. Code review is a time-consuming task for instructors and could add excessive cognitive workload to beginner programmers who lack knowledge and experience in programming concepts.

As noted by Indriasari et al. [43], two common approaches are used to implement peer code review in introductory courses: *Individual* and *Collaborative*

Individual

In individual peer review activities, students work independently to review programming assignments submitted by their peers. These reviews typically begin after students have submitted their assignments [18, 103]. Code is assigned for review either randomly or by allowing students to choose their reviewers [116, 41, 100, 34].

Reviews are conducted using both digital tools and traditional paper-based methods. The review process is usually integrated into the course, taking place during class or as homework [75]. In some cases, the activity ends once feedback is given [34, 100, 75]. In others, students are encouraged to revise their code based on the feedback before submitting a final version to the instructor [116, 41].

The purpose of incorporating code review into programming courses is to enhance the learning experience and help students develop core programming skills [105, 102]. These practices are inspired by industry standards, where developers review each other's code to assess its correctness and quality and to provide constructive feedback.

However, beginner programmers often struggle with code review tasks. Due to their limited understanding of programming concepts, they may struggle to identify errors or evaluate code effectively [60]. Misconceptions and fragile knowledge can make it challenging for them to provide meaningful feedback [84, 79]. As a result, asking novice students to perform these highly cognitively demanding tasks independently, without any guidance and outside of class time, may place an unfair burden on them.

Collaborative

In collaborative peer code review, students work in pairs or small groups to review and discuss each other's code. This approach of integrating code review practices in instruction promotes shared learning, where students explain their reasoning, ask questions, and clarify misunderstandings as they analyze code together [42, 50].

Collaborative reviews are often conducted during class sessions, with instructors provid-

ing structure and support. These sessions may include guided checklists or rubrics to help students focus on important aspects of code quality, such as correctness, readability, and style [30, 50]. This method reduces the excessive cognitive load on individual students and helps them build confidence by learning from their peers' approaches to problem-solving [42]. It also reflects real-world software development practices, where teamwork and communication are essential. However, in CS education, these skills are not always clearly emphasized as some of the most important learning objectives in programming courses.

Despite its benefits, collaborative peer review presents several challenges. Group dynamics can affect the quality of interaction; some students may dominate discussions, while others may hesitate to contribute. Differences in programming ability can also lead to unequal participation, where stronger students take over the task, and weaker students remain passive. Additionally, without clear guidance, students may focus on surface-level issues rather than deeper conceptual understanding. Time constraints and coordination difficulties can further limit the effectiveness of collaborative activities, especially in large classes.

To address these challenges, instructors must carefully design collaborative review tasks, provide clear expectations, and foster an inclusive environment that encourages all students to participate. Structured prompts, role assignments, and instructor feedback can ensure that the collaboration is meaningful and beneficial for all participants.

2.1.4 Gap in Code Review Literature in CS Education

Although integrating code review into programming courses has been shown to support student reflection and revision [116, 105], it also presents notable challenges, particularly for novice programmers with limited understanding of core programming concepts. Many implementations rely on structured rubrics or predefined guidelines to help students provide feedback on peer code. While these tools are intended to scaffold the review process, research indicates that students often focus on superficial aspects, such as formatting or variable names, rather than engaging with the deeper logic or functionality of the code. This tendency

can compromise the quality of feedback and lead to inconsistent levels of engagement and low motivation towards analyzing the code in detail [99, 101].

Prior studies have identified several limitations in the effectiveness of peer feedback, especially in introductory programming contexts. Common issues include low engagement, poor-quality reviews, and reduced motivation, challenges that are often linked to students' incomplete understanding of programming principles. For example, students have been found to misinterpret rubrics or misunderstand the code they are reviewing, resulting in inaccurate or unhelpful feedback [106, 100]. These findings underscore the challenges of relying on novice reviewers to provide meaningful evaluations without a sufficient conceptual foundation and instructional support.

Collaborative approaches, where students work in teams to review each other's code, have shown promise in promoting peer support through meaningful discussion and improved learning experience [39]. However, scaling peer code review in large classrooms remains difficult due to the time-intensive nature of reviewing and assessing student feedback [105, 99]. Moreover, novice programmers often struggle with writing and debugging their own code, and the added responsibility of reviewing peers' work can increase stress and create excessive cognitive demands for students, which can potentially lead to disengagement or reduced motivation [104, 90].

Despite the growing body of research on peer code review in CS education, a significant gap remains in the development of scalable, low-overhead instructional strategies that support meaningful code analysis without overwhelming novice learners. For students already managing the cognitive demands of learning to program, traditional peer review methods can be difficult to implement effectively at scale.

This dissertation addresses this gap by introducing a structured, active learning-based intervention designed to simplify the code review process while supporting the development of essential code analysis and debugging skills. The proposed in-class activity, *Code Insight*, is intentionally designed to minimize cognitive load while helping students systematically re-

view and debug code using efficient strategies. By integrating this in-class activity into early programming instruction, the study aims to enhance students' self-efficacy and confidence in managing complex programming tasks.

2.1.5 Debugging in CS Education

Debugging is a complex and cognitively demanding activity, making it one of the most difficult and time-intensive stages in the software development process for both novices and professional programmers. Despite these challenges, it is essential for producing reliable and high-quality software [118]. Given its importance, there is a strong case for teaching debugging explicitly in programming courses in a CS program [37]. Studies show that software developers spend nearly half of their time debugging existing code to ensure software quality and functionality, making it a major contributor to the overall cost of software development projects [83, 16].

In educational contexts, students often develop debugging skills informally through trial and error while working on programming assignments. This approach can lead to increased frustration and stress, especially considering the high cognitive load required for effective debugging [74]. Although debugging is a critical skill, it is not always taught explicitly in programming-intensive courses of a CS program. This may be due to the complexity of the task and the additional difficulties faced by novice programmers because of a lack of knowledge and skills [94]. Many instructors report that debugging is one of the most complex topics to teach, and they often lack a clear understanding of structured strategies for integrating it into the curriculum [69, 56].

A recent systematic literature review on debugging instructions in CS education found that most approaches focus on helping students identify and fix errors in code. However, there is limited attention given to building students' confidence in debugging or encouraging reflective practices that help them learn from their experiences and improve over time [113]. Furthermore, a study of professional software developers revealed that only about half

had received formal instruction on debugging. Interestingly, the study found no significant difference in debugging effectiveness between those who had received instruction and those who had not. This suggests that current instructional methods may not be sufficient for developing systematic debugging skills that enhance students' learning [80].

Recognizing these challenges and gaps in the literature, particularly in the context of introductory programming courses, this dissertation introduces **Code Insight**, a tool designed to support students' self-efficacy in debugging and code review tasks. This instructional intervention encourages beginner programmers to explore multiple approaches to the same problem and to identify different strategies for correcting errors in the programs they review.

To guide the design and evaluation of this intervention, this dissertation draws on an integrated theoretical framework that combines key learning theories relevant to CS education. This theoretical framework, as discussed in the next section, provides the foundation for understanding how students engage with complex programming tasks, such as writing and debugging code, and how instructional design can better support their learning, motivation, and confidence.

2.2 Theoretical Framework

This dissertation adopts an integrated theoretical framework that combines constructivist learning theory, active learning, and situated learning theories to develop and integrate **Code Insight** intervention to improve the learning experience of novice programmers in introductory programming courses. This blended approach supports the design of an in-class code review activity by highlighting genuine, collaborative, and reflective learning experiences for beginner programmers, encouraging the development of code review and debugging skills while increasing confidence in tackling complex programming tasks. The active learning approach supports this intervention by highlighting the importance of meaningful and sustained

student engagement during in-class activities. By actively participating, students are more likely to develop a deeper understanding of programming concepts as they learn through hands-on experience and reflection on their analysis, evaluation, and interpretation of the existing code. By encouraging discussions on complex questions, students can communicate their understanding and reasoning, engage with various perspectives, and co-construct knowledge to address any confusion or existing misconceptions about the programming concepts they are learning in the introductory course. When combined, these theoretical perspectives provide a strong foundation for helping beginner programmers develop essential skills, such as code review and debugging, while also improving engagement, confidence, and overall learning experience in complex programming tasks that can be challenging to acquire through trial and error.

The design of the in-class peer code review activity in this study is grounded in an integrated theoretical framework that draws from multiple well-established learning theories: *constructivist learning theory*, *active learning*, and *situated learning theory*. Together, these perspectives provide a comprehensive foundation for understanding how students learn best in collaborative, authentic, and reflective environments, guided by the instructor.

- **Constructivist Learning Theory** emphasizes that learners actively construct knowledge through experience and reflection [73]. In the context of code review, students engage with real code, analyze logic, and reflect on their understanding, which deepens their conceptual grasp of the programming concepts [42].

- **Active Learning** supports the idea that students learn more effectively when they are actively involved in the learning process [24]. The peer code review activity requires students to read, critique, and revise code, promoting hands-on engagement and critical thinking.

- **Situated Learning Theory** highlights the importance of learning in authentic contexts [10]. By simulating real-world software development practices, the code review

activity provides a meaningful and relevant learning experience that mirrors professional environments and encourages the development of essential skills among the new programmers.

These theoretical components converge to support the **Code Insight** integration in introductory programming courses, which serves as the central pedagogical strategy to address motivation, engagement, and self-efficacy among students as they learn the new concepts. This in-class activity, in turn, leads to two key outcomes: the *development of essential programming skills by improving self-efficacy* (such as code review and debugging skills) and an *improved learning experience, engagement, and motivation* for novice programmers. By grounding the instructional design in this integrated theoretical framework, the in-class intervention aims to foster both technical competence and learner confidence in introductory programming courses.

2.3 Summary

Code review and debugging are essential practices for developing high-quality software. However, these skills are not consistently or explicitly taught in programming courses in a CS program. The literature reviewed in this chapter highlights two major concerns in CS education: first, the lack of formal instruction in code review and debugging, which affects students' skill development even at intermediate and advanced levels; and second, the challenges of implementing effective instructional strategies to teach these practices, particularly in introductory programming courses where students are still building foundational knowledge.

Research on peer code review in introductory programming courses in CS shows that while it can aid student reflection and revision, it often falls short in practice, especially for novice programmers. Students tend to focus on surface-level issues rather than the more profound logic or functionality of the program, and many struggle to provide meaningful feedback due

to limited conceptual understanding. The introduction of collaborative review strategies and structured rubrics to address these issues is a step in the right direction. However, these peer code review strategies are challenging to scale effectively in large classrooms. Low review quality and a potential increase in excessive cognitive load for beginner programmers underscore the urgent need for instructional strategies that simplify the code review practices for in-class integration and systematically implement a well-guided activity to minimize workload and additional stress on both instructors and students.

Similarly, debugging, a cognitively demanding and time-consuming task, is often overlooked in CS curricula and left for students to learn this essential skill through experience and practice from programming assignments in the class. Students typically learn debugging through informal, trial-and-error methods, which can lead to frustration and hinder skill development. Instructors have reported difficulty teaching debugging effectively due to its complexity and often a lack of understanding of systematic debugging strategies for integrating it into course instruction. Moreover, the oversight of the importance of building students' confidence and reflective abilities in debugging tasks is a significant gap that needs to be addressed in CS education.

Together, these gaps in the literature underscore the need for educational interventions that support novice programmers in developing both code review and debugging skills in a manageable and systematic manner, without placing a heavy burden on instructors or students. In response, this dissertation introduces **Code Insight**, a structured, active learning-based in-class intervention designed to reduce excessive cognitive load while enhancing students' self-efficacy and analytical abilities without adding any additional workload and stress on students. By integrating this approach into early programming instruction, the study aims to enhance students' engagement, confidence, and competence in managing complex programming tasks, as well as improve their problem-solving skills. The focus of this novel in-class integration is to encourage the development of complex skills, such as code review and debugging, among beginner programmers.

Chapter 3

Exploring How Computer Science Students Develop Debugging Skills in a Curriculum

This chapter is based on a full research paper accepted for presentation at the IEEE International Conference called Frontiers in Education in 2025.

3.1 Overview

Programming is a cognitively demanding activity that requires learners to develop strong problem-solving and analytical skills [53]. Among the most challenging aspects are code review and debugging, both of which require strategic thinking and close attention to detail. Debugging, in particular, is often difficult for novice programmers, who may struggle to identify, understand, and correct errors in their code and struggle to explore different approaches to fix errors [48].

As a core component of the software development process, debugging plays a vital role in ensuring software quality and reliability [80, 112]. Developers typically spend between 20% and 40% of their time on debugging tasks, with some cases exceeding 50% of their

work time [80]. This time investment is expected to grow as software systems become more complex.

Recent advances in artificial intelligence (AI) have introduced new tools to support debugging, and AI is increasingly handling more routine programming tasks. AI-assisted systems are increasingly used to automate error detection, suggest fixes, and provide intelligent feedback [91]. These tools aim to reduce developers' cognitive load and improve the speed and accuracy of debugging. For example, AI Debugging Assistants integrated into development environments can guide users through the debugging process by recommending breakpoints and explaining program behavior step-by-step [6]. The intersection of AI and debugging is evolving in two directions: using AI to assist with debugging (AI4SD) and developing methods to debug AI systems themselves (SD4AI), both of which are gaining attention in research and industry [91].

Despite its importance, debugging is often not explicitly taught in undergraduate computer science (CS) programs. Many courses emphasize syntax and logic but provide limited instruction on systematic debugging strategies. This lack of structured guidance can lead to low confidence, frustration, and inefficient problem-solving among students [67, 28].

This chapter explores the experiences of intermediate to advanced undergraduate CS students as they engage in debugging tasks. The aim is to identify the challenges they face and the strategies they use, with the goal of informing more effective teaching practices that support the development of debugging skills. To address this research problem, the following research questions are examined using a qualitative research approach:

- **RQ1.** What programming practices do students employ when developing software?
- **RQ2.** What motivates students to adopt specific debugging strategies, and how did they acquire these practices?
- **RQ3.** How do students address and resolve obstacles encountered during the debugging process?

3.2 Benefit of Debugging as Explicit Instruction in CS education

One of the main challenges faced by novice programmers is evaluating their code to understand its behavior and identify errors related to programming concepts [4, 56]. These difficulties can reduce students' confidence, hinder their ability to complete programming tasks, and limit opportunities to learn from mistakes [32]. When such challenges are not addressed, they can lead to frustration and stress, negatively affecting learning outcomes [110]. These issues are often linked to limited programming experience and a lack of structured exposure to essential practices such as debugging and code review [72]. Without this foundation, students may develop misconceptions about how code works, make frequent syntax and logic errors, and write poorly organized code [12, 92].

After completing a year of programming courses, students usually gain some experience with debugging, often through trial and error. However, this unstructured approach can be inefficient and frustrating, leading to increased time spent resolving issues and limiting the development of effective problem-solving skills [74]. Even when students eventually succeed in their programming tasks, the effort and frustration involved can lead to negative perceptions of their abilities, which may impact their motivation and learning experience [52].

Despite the importance of debugging, it remains neglected in many CS programming courses curricula due to its perceived complexity [65, 2]. Some studies have explored the benefits of explicitly teaching debugging strategies in programming courses. These studies suggest that incorporating debugging instruction can improve learning outcomes and help students develop essential programming skills [70, 27]. However, there are still gaps in the literature regarding how to guide students in adopting systematic debugging strategies and how to support their self-efficacy in debugging tasks [113].

Although several researchers advocate for introducing debugging instruction early in programming education, little is known about the long-term effects of its absence. Specifically,

there is limited research on how the lack of explicit debugging instruction influences students' learning and skill development as they progress through a CS program.

This study aims to explore the debugging practices students use when explicit instruction is not provided. It also investigates the challenges students face in learning and developing debugging skills independently.

3.3 What Are Debugging Strategies

Debugging is a structured, multi-step process and is often one of the most complex and challenging skills for beginner programmers to master. This study employs a set of debugging strategies proposed by Li et al. (2019) [57] as a baseline to establish a systematic approach for debugging code effectively. This study focuses on investigating whether students, after gaining at least one year of programming experience, naturally apply these debugging strategies systematically when working independently, even without receiving explicit instruction in debugging. The study also explores how students learn to identify, understand, and fix errors in their code without formal guidance in programming courses. These insights help us understand how debugging skills evolve among intermediate to advanced-level programmers throughout the CS curriculum.

Li et al. (2019) outline five key stages of debugging, which serve as the foundation for this study [57]. These stages are also supported by earlier work from Fitzgerald et al. (2008), who conducted a multi-institutional study highlighting the common struggles novice programmers face when debugging [28]. Following are the essential steps in a cognitively demanding debugging task:

- **Construct the problem space**

  In this stage, students build a mental model of the program. This involves understanding both the intended behavior and the actual behavior of the code, as well as the program's structure, logic, and control flow [117]. A clear mental model helps stu-

dents reason about what the program is supposed to do and where it might be going wrong, and to effectively implement the next steps.

- **Identify fault symptoms**

  Once students understand the control and logic of the program, they must identify where the behavior deviates from expectations. This step involves recognizing symptoms of faults, such as incorrect outputs or unexpected behavior. Novice programmers often struggle here, especially with logical errors, which are harder to detect than syntax errors [72, 25].

- **Diagnose the fault**

  After identifying the symptoms, students must locate the exact cause of the error. This requires both strategic thinking and prior experience. Beginners often focus on a single possible cause and may overlook other explanations, limiting their ability to troubleshoot effectively [44].

- **Generate and verify solutions**

  Once the error is located, students attempt to fix it. While they are often able to correct the issue, they may not test the program thoroughly afterward. In particular, they may overlook edge cases or fail to verify that the fix did not introduce new errors [46, 28]. This issue is not unique to students and has also been observed in professional software development [19].

- **Reflect and document**

  Professional developers often document their debugging process and maintain logs of common errors to support future problem-solving [80]. For students, reflecting on why a fix worked can reinforce learning and improve long-term debugging skills. However, many students skip this step, especially when they rely on trial and error, which limits their understanding and growth [28].

These stages provide a useful lens for analyzing how students approach debugging on their projects when working independently. Understanding where students struggle and face significant challenges can inform the design of instructional strategies that better support the development of debugging skills among students in a CS program.

Research Questions	**Interview Questions**
What programming practices do students use to develop a program?	How do you proceed to solve a programming problem like a class project? How do you evaluate whether your program is working correctly or not? What would you do if your approach toward debugging the program is not working as expected?
Why do students follow specific debugging practices, and how did they learn them?	Why did you follow a specific set of strategies to debug the program?
How do students overcome challenges and roadblocks encountered while debugging a program?	What are some challenges you have faced when working on programming projects? What steps do you follow to handle the challenges you encounter when working on the projects?

Table 3.1: The List Of Interview Questions Related To Research Questions

3.4 Methodology

This study investigates the knowledge and skills that undergraduate students develop throughout their CS education, with a particular focus on how they acquire debugging competencies in the absence of explicit instruction on debugging strategies in a CS program at the University of Oklahoma. The analysis is based on qualitative data collected through student

interviews, allowing the research to capture the challenges students face in their own words.

The study received approval from the Institutional Review Board (IRB) as included in the appendix chapter D. All participants were 18 years of age or older and provided informed consent prior to participation. As a token of appreciation, each participant received a $10 Starbucks gift card after completing the interview.

3.4.1 Research Context and Participant Recruitment

To contextualize the study, a review of publicly available course syllabi from the university's website was conducted. This review aimed to identify the core learning objectives and instructional practices across required CS courses. The analysis revealed that debugging strategies and processes are not explicitly included in course learning outcomes and are rarely emphasized as key instructional components. Furthermore, the syllabi do not provide direct guidance on debugging techniques or code review practices, particularly in relation to interpreting feedback from automated grading systems commonly used in large-enrollment courses.

The CS curriculum includes integrated laboratory sessions during the first two or three semesters. These courses cover two semesters of Java programming, including the use of Unified Modeling Language (UML) class diagrams for software design, followed by a course in data structures using C++. In later stages of the program, programming-intensive courses typically do not include lab components. Students are expected to work independently on programming assignments, although some courses incorporate collaborative projects involving pairs or small teams.

This curriculum review provided valuable context for interpreting students' interview responses, particularly when they described programming experiences from completed courses and the strategies they used to analyze, evaluate, and debug their code for class assignments.

To support student success, the CS program offers a range of academic resources. Large introductory courses are staffed by faculty members who hold regular office hours, along

with teaching assistants who are available during lab sessions and additional office hours. The department and college also provide tutoring services, available both in person and online throughout the week, with limited availability on weekends. These institutional supports are important to consider when analyzing participants' responses, especially regarding the resources they commonly used to overcome major challenges or roadblocks during the development process.

Participants were recruited through email invitations sent to all undergraduate CS majors and through in-class announcements. To ensure that participants had sufficient programming experience while still representing intermediate to advanced learners, eligibility was limited to students who had completed at least one year of programming coursework. This typically included an introductory course and an intermediate-level course, such as advanced Java programming. A total of eleven students participated in the study. The interviews were conducted by the researcher (KB), who did not hold any current instructional or supervisory role over the participants, although some had previously taken courses taught by the researcher.

3.4.2 Data Collection

Data were collected through semi-structured interviews conducted either in person on the university campus or via Zoom within a year between October 2023 and October 2024. Each interview lasted approximately 30 minutes and was audio-recorded with the participant's consent. To ensure participants felt comfortable sharing their experiences, interviews were conducted in locations away from the CS department offices on campus, such as the libraries, thereby minimizing any perceived pressure or influence.

The interviews were designed to gather detailed insights into students' programming and debugging practices. Participants were asked to describe how they approach writing code, identifying and resolving errors, evaluating their work, and the resources they rely on. They also discussed the challenges they face when working independently on programming

assignments.

The semi-structured format allowed for flexibility in the conversation, enabling the interviewer to ask follow-up questions and encouraging participants to elaborate on their experiences and reflect on their learning processes. All interviews were transcribed using automated transcription software to convert the audio files to text file. The transcripts were then manually reviewed to correct errors, recover any missed content, and remove verbal fillers and hesitations. This process ensured that the core content of each student's response was preserved accurately for analysis.

3.4.3 Interview Protocol

The interview protocol was carefully designed to align with the study's research questions and to explore both the cognitive and emotional aspects of students' experiences with debugging. The questions focused on how students begin programming tasks, the strategies they use to identify and fix errors, and the types of support they rely on throughout the process. Table 3.1 presents the interview questions alongside the corresponding research questions used during data analysis.

The insights gathered from these interviews provide a valuable foundation for understanding how students develop debugging skills over time. These findings also inform recommendations for incorporating structured debugging instruction into the CS education curriculum.

The following interview questions were designed to explore and address the research questions.

Question1. ***Which CS courses have you completed until now?***

This question is asked to collect the current level of the participant in the CS program and determine the amount of experience the student has in programming classes. The undergraduate CS program in this university requires intensive programming courses in

the program's first two years, which include courses like one year of Java programming followed by data structures in C++. These courses provide students with the knowledge and skills of programming languages, giving them some experience working with and writing a program. Participants in this study were either currently enrolled in or had completed the data structures course.

Question2. ***How do you proceed to solve a programming problem like a class project?***

This general question aims to understand the student's approach to solving programming problems. A follow-up question, such as *"Could you please walk through the steps you follow to develop a programming solution?"*, is sometimes used to gather more detail.

The goal is to identify the challenges students face when writing programs from scratch and to examine whether they apply any debugging strategies without having received formal instruction.

Question3. ***How do you evaluate if your program is working correctly or not?***

This question explores how students test and evaluate their programs. Understanding the program's specifications is a key part of assessing its correctness [54]. Participants were asked to describe how they analyze their code to determine whether it behaves as expected. The goal is to identify the steps students use to create a mental model of the program and how they analyze the intended and actual behavior of the program to further investigate the problems with the program.

Question4. ***What would you do if your approach toward debugging the program is not working as expected?***

This interview question explores the possible roadblocks a student may encounter when writing the program from scratch. The domain knowledge of the problem and concepts is

crucial to understand the functionality and expected output of the program. This question investigates how students respond when their initial debugging efforts are unsuccessful. It focuses on how they identify and resolve errors, especially when the source of the problem is unclear.

Question5. ***Why did you follow a specific set of strategies to debug the program?***

This question explores the reasoning behind the debugging strategies students choose. Follow-up questions were used to identify whether these strategies were taught in class or learned from external sources.

Question6. ***What are some challenges you have faced when working on programming projects?***

This question aims to understand students' learning experiences and the challenges they encounter during programming assignments. It also provides insight into the availability and use of instructional and external resources.

Question7. ***What steps do you follow to handle the challenges you encounter when working on the projects?***

This question investigates how students manage difficulties during the development process. It also sheds light on the resources they use, both from completed courses and external sources.

The expectation behind these questions is to understand a student's process in approaching the programming projects and analyze the emotions underlying the student's motivation for a specific strategy.

Because this study was conducted among students who were currently majoring or minoring in CS at a single institution, the University of Oklahoma, one of the main challenges was recruiting participants. The advanced programming courses in CS are complicated, and

recruiting students to spend 30 minutes discussing their experience in a CS program was challenging. All the participants are 18 years or older. The interview was conducted with 11 participants from the CS program, two of whom are women and nine of whom are men. The primary researcher of this study is not an instructor or supervisor of any of the participants.

The following section describes the qualitative research approach used to analyze the interview data. It outlines the thematic analysis process employed to identify recurring patterns and themes related to each research question.

3.4.4 Thematic Analysis

Interview data were analyzed using thematic analysis to identify patterns in students' responses and examine their programming strategies [20]. NVivo was used for coding and visualizing themes through multiple iterations. The six-phase framework by Braun and Clarke [15, 17] guided the analysis.

In Phase 1, transcripts were anonymized and reviewed for familiarization. Phase 2 involved generating initial codes for each interview question (excluding demographic questions), with 5 to 8 codes per interview question refined iteratively by the primary coder [initials redacted]. In Phase 3, codes were grouped into themes aligned with the research questions. These themes were collaboratively reviewed and refined in Phase 4. Phases 5 and 6 involved synthesizing and presenting the findings, supported by a thematic map illustrating key patterns.

3.5 Themes

This section presents the findings from the thematic analysis of interview data, highlighting key patterns and insights that emerged from participants' accounts of their programming experiences. Each theme is supported by illustrative quotes from participants, which provide contextual depth and help illuminate the reasoning behind participants' programming and

debugging practices. These excerpts further clarify how such practices emerge and evolve for students following this CS curriculum.

Figure 3.1 shows a thematic map that highlights the main themes related to programming practices, along with their related sub-themes, that are derived directly from the codes generated during the data analysis process. The map illustrates the programming strategies participants employ when developing code for course projects, the testing strategies they use to evaluate whether their programs function correctly, and the debugging strategies they use when program testing fails.

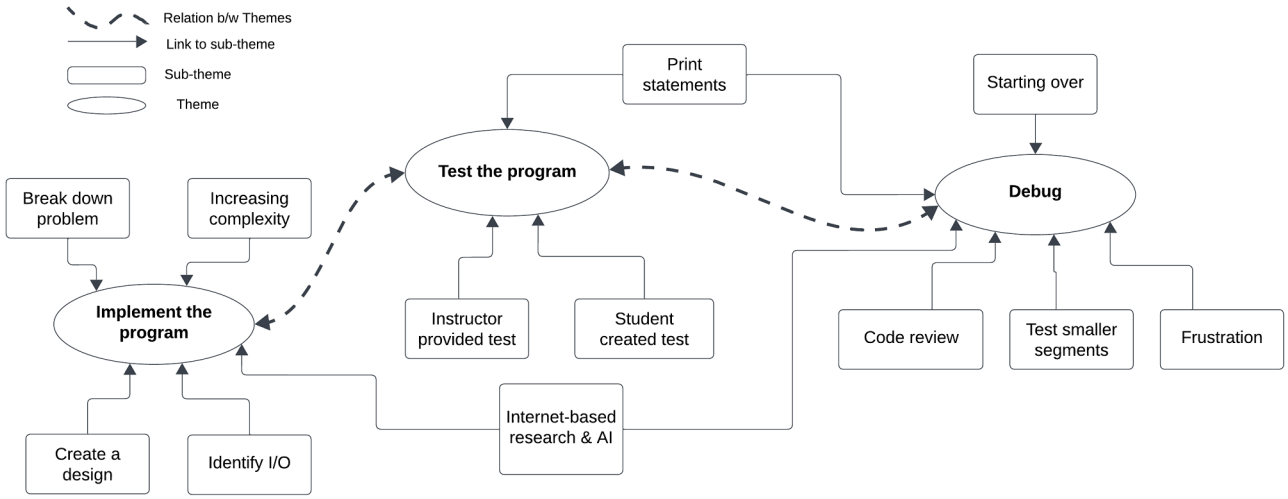


Figure 3.1: Thematic map representing ad hoc approach of the students when writing a program

3.5.1 Implement the Program

This theme captures the processes the participants report following when writing a program. The associated sub-themes reveal that students begin by attempting to understand the problem by identifying the expected input and output (I/O), creating some steps to start writing (with little systematic design), and exploring applicable programming concepts through internet-based research.

Identify I/O

Students commonly reported starting a project by identifying the expected input and output. This step serves as a foundational strategy to understand the problem requirements and clarify what is expected of them.

> ***"For a class project, I typically read the instructions, and make sure I really know what I'm supposed to do"***
>
> ***"I start with specifications of the project, I usually look at the expected output."***

The work participants describe in this section aligns with professional practice, where a careful review of the requirements is a typical early process.

Create a design

Most students reported implementing one method or component at a time, without discussing any systematic design process. Other students had developed novel approaches to design.

> ***"So to begin with, I just like brainstorming the idea like all the algorithms to solve a problem, find a solution."***
>
> ***"If there's any duplicate steps, I definitely like to identify those because that is key to identifying whether you're going to put in methods or classes."***

Only one student mentioned creating a structured design before coding, with no mention of maintaining documentation, when asked.

> ***"I start breaking down the structure of the class, kind of similar to a UML diagram, next I think about what I know and how that translates to the idea"***

The lack of design in participants' processes may stem from the reliance on automated grading in introductory courses, which emphasizes unit testing. This approach limits flexibility in method signatures and design choices, potentially preventing students from engaging in the design process.

Break down problem

Several participants reported breaking down the programming task into smaller components, which enabled them to incrementally identify key variables and prioritize the order of implementation of methods.

> ***"I just try to implement small tasks like task by task for each of the problems to solve."***
>
> ***"I'll write method bodies in the class, an outline for the method, and like we have our variables, we have our methods, and then I'll go one by one in the methods."***

This linear approach to implementation often depends on predefined method signatures or identifying a starting point from scratch. However, responses indicated a lack of structured planning as project complexity increases.

Internet-based Research and AI

Many participants describe using internet search as part of their initial development process.

> ***"I had to do a lot of googling before I figured out what we were supposed to do."***
>
> ***"I start with googling the concept and learning what that concept is so I can then implement it in code."***

Participants often rely on easily accessible resources, particularly online forums and artificial intelligence programs (AI), to search for programming concepts, logic structures, or example code that can help them begin constructing their solutions.

> ***"I can look up a template online and start there."***
>
> ***"When I'm going through a piece of code for research and documentation, honestly, ChatGPT and Copilot are really great."***

The use of internet search and AI tools is prevalent in both implementation and debugging tasks. While professional software developers routinely consult online resources, participants' use of terms like "templates" may obscure intentions to copy other people's solutions. This tendency to rely on readily accessible tools like AI raises concerns about academic integrity and plagiarism in CS education.

The participants are aware of some of the challenges of internet searches.

> ***"you have to google what each tool is, and then you kind of go down this rabbit hole where you're like oh this technology does this because of this and then this one does this because of this and it kind of spirals right like that."***

Increasing complexity

This theme highlights students' challenges in writing functional programs, particularly due to the time and effort required to fix errors when new programming languages are integrated and the need for self-directed learning across multiple programming languages.

> ***"The fact is that we code in so many different languages, I think that's the hardest part."***
>
> ***"Recently I've been using new languages, so a lot of times it's like something's not installed & I didn't know how to fix this."***

Students also reported that working with multiple languages in a single semester increased complications when implementing a program, caused problems when debugging, and required repeated testing.

3.5.2 Test the Program

Once students begin implementing their programs, a common strategy adopted is evaluating the correctness of their program by comparing the actual output with the intended output. Building on their initial understanding of the problem, participants frequently use provided test data, often via feedback from provided auto-graders.

Instructor provided test

Some responses mention that they first use the test cases provided by the instructor to test. The tests provided by the instructor are usually through auto-graders. Often, the first step in testing the program is using the input samples provided in the project description and comparing the outputs to identify any faults in the program.

> ***"We get input-output files that are like example files for our projects, and I will put in the same inputs and then see if I get the expected outputs for each method."***
>
> ***" Sometimes they give us tests that we can use. I usually just run it and look for the correct output."***

Using instructor-provided test data exclusively removes students from the process of strategic test data selection. Students who rely exclusively on instructor-generated data will not have experience with this process.

Student created tests

Some students mentioned that they test the program by identifying the edge cases and evaluating the code with different possible inputs.

> ***"I like to identify what I want to do. Then I'll identify any edge cases and test those."***
>
> ***"For complex problems, I add test outputs in the code or test if this is working first."***

When faults were identified, they proceeded to debugging, as discussed in the next theme. Although students described writing tests, they often described using different inputs to compare the output; they did not mention writing their own formal unit tests to evaluate the program.

Print statements

This sub-theme appeared consistently across programming stages, with nearly all students reporting the use of print statements to evaluate program behavior and locate logical errors (Fig. 3.1). Print statements were favored for their simplicity and ability to visualize program functionality, reflecting students' comfort with this informal testing and debugging method in the programming process.

> ***"If it's something that's not very easy to print, I usually try to make it to where it is. So, if it's a server-client, I'll print out every single message I receive from it."***

This theme highlights students' reliance on informal, error-prone, and time-consuming testing methods to iteratively evaluate and refine their code.

> ***"Using print statements because you can comment out the bottom of the method and then print out just the top part or keep going like that until you find out where it is going wrong."***

"If it's something like my program not returning the right numbers, then I would have to go into my code and add more print statements."

"I like using the print output for whichever language you're using. I will use them throughout the program"

A concerning pattern emerges in students' reliance on print statements for debugging, which often results in excessive and unstructured output that can mask additional errors when debugging.

"That was probably one of the earliest things I learned, to help with debugging, was using print statements."

The student above was explicit about his preference for the early-learned strategy. Some responses indicate that there is an awareness of the importance of using a debugger instead of print, but learning it independently is an additional effort in an already taxing CS program.

"I probably should learn how to use the debugger a little bit better, but I don't use the debugger"

"I know there's like a break point thing but I don't know how to use that, I really want to learn how to do it though"

Many students described that they don't use tools like debuggers or break points and instead rely on print statements to trace the flow of execution and examine intermediate outputs to identify any discrepancies. Print statements are crude debugging tools. They modify the code that they are testing, which is never desirable, and can be left behind as ugly relics of the debugging process.

3.5.3 Debug

As previously discussed in the theme *"Test the program"*, the most commonly reported debugging strategy among students was the use of print statements. In addition, many students

utilized external resources, including online forums and AI-based tools, to identify and resolve programming errors. Frequently employed techniques included tracing program execution through print outputs, conducting line-by-line code reviews, commenting out code segments to isolate specific functionalities, and incrementally reactivating code to detect logical faults. This stage of the development process was consistently described as the most challenging, often resulting in significant frustration and disappointment.

Frustration

A key finding of this study is the pattern of frustration and increased stress experienced during debugging. Many reported spending significant time attempting to fix errors, sometimes without success.

> ***"I was genuinely stuck, I wasn't able to finish the project and I wasn't able to get any help from the professor and I waited a bit too long to talk to any tutors, so unfortunately I just failed"***
>
> ***"I think it's very easy to get very frustrated after spending a long time on something and it is not working"***

Programming is particularly unforgiving since small errors can derail entire projects. While some frustration can contribute to learning. The degree of frustration and stress experienced by students writing programs can be excessive. It should be remembered that the participants are CS majors who have passed multiple CS classes. The students who did not pass these classes might have more to say about this topic. This frustration is likely to be a significant source of attrition in the CS program [51].

Test smaller segments

In the absence of a defined systematic approach or the use of a debugger, many students reported isolating and testing individual code segments by comparing program outputs to identify errors.

"I had to take chunks of it out and run it separately and put in different inputs to make sure this section was doing what I wanted it to do."

"if something is wrong and I have no idea, I'll take that method and comment out the rest of code, make up some test cases where it'll show what I need to get from output."

While this method can be effective for targeted testing, it is time-consuming and may introduce errors in the program when code is cut and pasted back and forth.

Review

Findings in this theme suggest that students often review code line by line. Reading code is an appropriate debugging strategy widely used by software professionals; however, it is done manually without the use of breakpoints or debugger tools.

" I just go through and look at the logic of each method and see if it's correct."

"it took me a lot of time and it was a lot of trouble to fix that, I just ended up just reviewing everything, going down from the top of the list."

"I'd trace the loop with given inputs and then figure out why it's outputting whatever it is outputting, where the error is happening"

"I will look at each line of code and think through what each line of code is doing"

In some cases, students sought help from peers, classmates, or family members with programming experience and directly mentioned reaching out to them, rather than a teaching assistant (TA) or their instructor.

"If you have friends or family or colleagues who can help test it, so having other people look at it with fresh eyes and just asking them to try to break it"

"Sometimes I've had to reach out to classmates and say, hey, I'm having trouble with this, did you have trouble with this, and can we help each other out?"

While this collaborative approach to code review can be beneficial, it also raises concerns about equity, as not all students have access to support networks with software expertise. In addition, each of these sources of support can also be a source of plagiarized code.

Starting Over

An example of a particular inefficient strategy that was discussed by more than one student was starting over from scratch.

"I would like to say sometimes when the code isn't working, I just completely restart."

"But if nothing else is working and it's just a huge mess, usually I'll just restart and try to trace my steps back on where it went wrong."

This is a concerning finding, as reliance on such an approach may hinder deeper learning by obscuring the root cause of initial errors. Without understanding what caused the error or how it was resolved, students risk developing superficial debugging habits. Consequently, this may lead to repeated mistakes in future programming tasks.

Internet based research and AI

This is a repeated and common sub-theme for *"Implementing program"* and *"Debug"*. Many responses reported using internet searches, internet forums for programmers, or AI tools to resolve logical errors, and sometimes finding fixes to syntax errors.

"I'll do a lot of googling. I don't know if that's debugging, but I'm getting this error. How do I fix this? And look at Stack Overflow and other forums."

"I'll turn to google, see if anyone else has experienced these same problems as me, and see if there's another way that I can implement the code."

"If there's an error I don't understand, I'll usually google it or recently use ChatGPT, that I've been using a lot recently"

While internet forums can offer valuable learning opportunities, they are also a significant source of plagiarism, particularly when students copy solutions in an attempt to fix their code.

Analysis of emergent themes indicates that students often fail to develop systematic debugging strategies when relying on trial and error. The resulting frustration frequently leads to ad hoc approaches to coding and testing. The following section addresses the research questions and highlights key gaps in students' programming competencies, reinforcing the need for structured debugging and code review instruction, particularly in light of the growing use of internet-based and AI-driven tools.

3.6 Discussion

Interview data revealed that most participants did not use design plans, unit testing, or debuggers, despite prior exposure to automated grading tools. Only one participant reported using an integrated debugger; most relied on print statements, a novice-level strategy. In the absence of explicit instruction [57], participants defaulted to inefficient debugging habits, often compounding errors and prolonging the process. These findings emphasize the need for explicit debugging instruction in CS curricula to better equip students for modern programming environments.

No participants reported seeking help from tutors, teaching assistants, or faculty, opting instead to work independently. The rationale for this choice was not given. Finding ways to help students make better use of the many people hired to support course learning should be done. A future investigation of the rationale behind these student choices is future work.

Research Questions

This study encourages researchers in CS education to investigate additional ways to provide strategic guidance to students for developing both systematic debugging strategies and non-cognitive skills, aiming to reduce student frustration and improve learning outcomes.

RQ1: What programming practices do students use to develop a program?
Analysis of student responses indicates that, in the absence of explicit instruction or strategic guidance, students often adopt ad hoc methods when writing programs and tackling complex debugging tasks. During fault identification and diagnosis, they frequently rely on unsystematic strategies, such as excessive use of print statements or internet searches, which can inadvertently introduce new errors. This sometimes leads to abandoning the current attempt and restarting the project.

Students also show discomfort with systematic debugging tools, such as debuggers, which are designed to help efficiently locate and understand errors. Without formal instruction in debugging strategies and code review practices, students in CS programs face challenges similar to those encountered by individuals entirely new to programming.

RQ2: Why do students follow specific debugging practices, and how did they learn them?
Thematic analysis indicates that students tend to rely heavily on the methods they were first introduced to during the early stages of their learning. Even when exposed to more advanced techniques later in the curriculum, such as unit testing or the use of debuggers, students frequently default to their initial strategies. This suggests a strong anchoring effect, where early learning experiences shape long-term programming habits, and transitioning to more structured or sophisticated methods, which are new to the students, is often resisted.

RQ3: How do students overcome challenges and roadblocks encountered while debugging a program?

The findings indicate that students frequently experience frustration and invest significant time when diagnosing logical errors in code. This qualitative analysis highlights that many lack a systematic approach to debugging, which contributes to persistent difficulties. The patterns identified in the responses suggest a continuous struggle to produce functioning programs, relying on ad hoc strategies during development. When challenges persist, they commonly turn to internet searches or AI-based tools for support. One of the concerning findings is that no participant mentioned reaching out to the tutors, course TAs, or the class instructor to address the roadblocks when trying to fix the program. This study was conducted at a CS program where tutors and TAs are available during regular help sessions.

These findings underscore the need to integrate structured code review and debugging instruction into introductory CS courses. Such practices enhance students' analytical skills and support code reading [9, 78]. With the growing use of AI tools for code generation, it is crucial to teach students how to critically evaluate and debug code, promoting deep learning and upholding academic integrity in the CS program [26]. The findings of this study reveal consistent patterns in how students develop programming and debugging skills, highlighting the lasting influence of early learning experiences. These insights provide a foundation for future research on instructional strategies and underscore the importance of integrating structured code review and debugging practices early in the CS curriculum. By identifying the practices students tend to retain over time, this work supports the design of scalable, evidence-based approaches that can enhance learning outcomes across all levels of programming education.

Limitations

This study has several limitations that should be considered when interpreting the findings. First, the research was conducted within a single CS program at one university, which may

limit the generalizability of the results to other institutional contexts. The relatively small sample size, constrained by Institutional Review Board (IRB) recruitment guidelines, further narrows the scope of the study. Although the consistency of participant responses suggests that data saturation may have been approached, this cannot be formally confirmed.

Second, the authors' familiarity with the curriculum may have influenced the interpretation of findings. To mitigate this, the analysis was grounded in participants' own words and supported by direct quotations. Additionally, many participants were former students of the researchers involved in this study, which may have introduced social desirability bias or inhibited critical feedback about their early programming experiences. However, such critiques were generally absent from the data.

Finally, interview coding was conducted by a single researcher. While this ensured consistency in the analysis, it also limited the opportunity for inter-rater validation and alternative interpretations that might have emerged through collaborative coding.

Despite these limitations, the study offers valuable insights into student learning patterns and instructional design in early programming education. Future research should aim to replicate and extend these findings across diverse institutional settings and with larger, more varied participant samples.

3.7 Summary

This study explored how undergraduate students from CS program at University of Oklahoma develop and debug programs without structured guidance. Despite completing several programming courses, many intermediate and advanced students continued to use informal and often ineffective strategies, such as print statements, online searches, or restarting their code, instead of more reliable methods like unit testing or using debuggers. These habits often led to inefficient debugging and increased frustration, as students spent more time fixing problems.

A key finding from the interviews is that students tend to stick with the first strategies they learn, even after being introduced to more advanced tools later in their studies. This anchoring effect highlights the lasting impact of early instruction on students' programming habits and their confidence in handling complex tasks. Without explicit teaching of essential skills like code review and debugging, students often fall back on trial-and-error methods, which can limit their ability to solve difficult problems effectively. It is also concerning that students rarely sought help from teaching assistants or faculty, preferring to work alone using less reliable sources such as online forums or AI tools. The reasons behind this behavior are not yet clear and should be studied further. Additionally, unequal access to informal support, such as help from peers or family, raises concerns about fairness and the potential for unintentional academic misconduct.

These findings underscore the importance of incorporating structured instruction on debugging and code review into the CS curriculum early on. Doing so can help students build systematic problem-solving skills, think critically, and rely less on ad hoc methods. As AI tools become more common in programming education, it is especially important to teach students how to evaluate and debug both their own code and AI-generated code.

Future research could include long-term studies that follow individual students as they develop their debugging and code review skills, as well as studies across multiple institutions to improve generalizability. Including the perspectives of instructors could also provide useful insights into effective teaching strategies and the challenges of integrating these skills into coursework. Finally, understanding why students seek or do not seek help from teaching staff, such as tutors, teaching assistants, and faculty, could inform the development of more effective support systems. Improving students' confidence in their programming abilities may also reduce over-reliance on AI tools and promote deeper learning and academic integrity in CS education.

Chapter 4

Code Insight: Combining Code Reading and Debugging Practices for Active Learning

This chapter was presented as a work-in-progress paper [9] in an IEEE international conference, Frontiers in Education (FIE), in 2024. The full research paper for this chapter has been accepted for presentation at the IEEE International Conference of FIE in 2025.

4.1 Overview

This chapter presents an innovative in-class activity designed to strengthen code reading and debugging skills among novice programmers in a computer science (CS) program. The primary aim is to enhance the learning experience for students who are new to programming by supporting the early development of essential skills such as code review, error detection, and debugging.

Over the past two decades, research has consistently shown that students in introductory programming courses often struggle to produce high-quality code, even after a full semester of instruction. These challenges are commonly linked to persistent misconceptions, difficulty

in evaluating code, and limited ability to identify and correct errors [66, 60, 84]. As discussed in Chapter 3, these difficulties persist beyond the introductory level. Students continue to find debugging tasks challenging, particularly when reviewing code for correctness and identifying appropriate fixes. Many rely on unsystematic, trial-and-error, and ad hoc approaches to find and fix errors rather than structured debugging strategies. This pattern mirrors earlier findings among students who were just beginning to learn programming [66], raising concerns about the long-term development of debugging competence among programmers in CS education.

These challenges are often rooted in a fragile understanding of core programming concepts [84, 79]. Research suggests that students who can effectively read and explain code tend to perform better in writing their own programs [62, 61]. However, traditional programming instruction typically emphasizes writing code from scratch, offering limited opportunities for students to analyze or evaluate existing code. This approach conflicts with industry practices, where developers frequently work with and maintain existing software.

One such industry practice, code review, is well-researched in CS education but remains underutilized in classroom settings. Despite its potential benefits, code review activities are often avoided due to concerns about increased instructor workload and the excessive cognitive demands placed on novice learners [90, 93]. According to Bloom's revised taxonomy [55], introductory students typically operate at the application level, while code review requires higher-order thinking skills such as evaluation and creation [3].

To address these challenges, this study proposes a structured code review in-class activity tailored for introductory programming courses. The intervention is designed for easy integration into existing curricula, without adding grading burdens for instructors. It emphasizes code analysis over code creation, helping students identify discrepancies between expected and actual behavior, detect logical errors, and evaluate program functionality. This approach aims to promote deeper conceptual understanding and improve students' confidence in code review (analysis and evaluation of the program) and debugging tasks.

Code Insight is the in-class active learning activity developed for an introductory programming course in a CS program to encourage the development of essential skills among beginner programmers. It presents students with varied programming examples and uses multiple-choice questions to guide them in analyzing specific code segments. Through frequent, low-stakes practice, which includes a set of multiple-choice questions, **Code Insight** reinforces conceptual understanding and builds students' confidence in reading, analyzing, and debugging code. The questions developed for this activity can easily be reused from one semester to another, as they are part of the class lectures.

This chapter examines students' experiences with Code Insight across two semesters of an introductory programming course in a CS program at one university. It addresses the following research questions:

- **RQ1:** How do students describe the role of Code Insight in enhancing their learning experience?
- **RQ2:** In what ways does Code Insight support students when debugging their own programs?
- **RQ3:** In what ways does Code Insight influence students' understanding of multiple approaches to solving programming problems?

To evaluate the impact of Code Insight, anonymous feedback surveys were administered during the Spring and Fall 2024 semesters. In Spring, surveys were conducted both mid-semester and after the final activity to assess changes in student perceptions. In Fall, a single survey was administered at the end of the semester. All surveys were completed immediately following the in-class activity. The responses were analyzed qualitatively to explore students' learning experiences and to address the research questions outlined above.

4.2 Code Insight

The implementation of the *Code Insight* activity in this study builds on the version introduced in our earlier work-in-progress publication, which was shaped by preliminary feedback from online learners [9]. This section describes the design, development, and classroom implementation of the activity. *Code Insight* is an in-class active learning strategy aimed at helping novice programmers develop essential skills in code review and debugging [9]. It addresses common challenges in introductory programming courses, such as low motivation, limited engagement, and difficulty applying problem-solving strategies [43]. The activity provides structured opportunities for students to evaluate peer code, practice identifying and fixing errors, and build confidence in debugging, all with instructor support and minimal added workload.

The design of this activity was informed by classroom observations, which revealed that beginner students rarely encounter diverse, valid solutions from their peers. This lack of exposure limits their ability to recognize alternative approaches and can hinder their debugging skills when faced with logical errors.

The introductory programming course in the CS program at the University of Oklahoma is taught in two different sections, one designed for students with some prior programming experience and another for students with no programming experience. This design was inspired by Cohoon's separation of students without prior programming experience into a dedicated section [21]. The courses have identical topical coverage, with the course for non-programmers offering additional laboratory time to allow students to engage in pair programming for projects. Either course fulfills the prerequisite for the second programming course in the CS program.

Code Insight emphasizes presenting multiple solutions to a single programming problem. These examples may include correct implementations, logical errors, or redundant code. By analyzing these variations, students develop cognitive skills needed to compare intended and actual program behavior [81]. Research shows that reviewing multiple worked examples

improves learning outcomes, especially for complex programming concepts [82].

To address common misconceptions, some multiple-choice questions in the activity include distractors—plausible but incorrect answers. Discussing these misconceptions in class and providing immediate feedback through active learning techniques has been shown to improve conceptual understanding and student engagement [115, 95].

4.2.1 Preparing Questions for In-Class Implementation

This section outlines the development of Code Insight questions, building on a preliminary study [9]. Questions were derived from student-generated code on prior introductory programming exams, capturing diverse problem-solving strategies.

Rubric-based analysis of past exams identified common correct solutions and misconceptions. One exam question per topic was selected, and student responses were reviewed, typically during grading, to extract varied correct approaches and typical errors. These misconceptions were embedded in multiple-choice options to promote critical evaluation.

Code samples were revised for clarity, originality, and appropriate difficulty, avoiding direct reuse. Problems were rewritten to be concise, syntactically correct, and discussion-ready, with clear assumptions to reduce cognitive load. Multiple-choice questions guided students step by step through code analysis. Some options reflected common misconceptions, encouraging students to reason through program behavior and assess correctness, reinforcing conceptual understanding and debugging skills.

Two approaches were used to construct multiple-choice options for the Code Insight code review questions [9]:

- Evaluating whether the code functions correctly *always*, *never*, or *sometimes*.
- Directing students' attention to specific lines of code that can contain a logical error, redundant code, or an unnecessary step.

For each problem, two distinct code examples are typically developed. All code is designed

to compile successfully when run as instructed, emphasizing logical reasoning over syntax. The programs are adequately documented, and some different approaches to the problem can have portions that could be generated using publicly available AI tools based on large language models (LLMs). In this study, the programs are not developed using AI tools, and utilize the approaches used by students from previous semesters, however, Code Insight can be expanded to include programs generated by AI. Instructors may request access to the question sets to replicate the activities.

Since the concepts are new to most students, the primary goal is to help them evaluate existing code. Qualitative analysis of survey responses provides valuable insights into student experiences, confirming that the activity effectively supports learning without adding excessive stress due to the additional workload on students.

An example of a partial code segment used in the activity is shown below, where students are asked to analyze and evaluate the correctness of the code:

```
1  public static void main(String[] args) {
2      Scanner keyboard = new Scanner(System.in);
3      final int MAX_DIMENSION = 36;
4      final int MAX_WEIGHT = 75;
5      System.out.println("Enter the width, height, depth, and weight separated by
           spaces");
6      double width = keyboard.nextDouble();
7      double height = keyboard.nextDouble();
8      double depth = keyboard.nextDouble();
9      double weight = keyboard.nextDouble();
10     keyboard.nextLine();
11
12     if (width <= MAX_DIMENSION) {
13
```

```
14          System.out.println("Luggage is checked in  successfully !");
15      } else if (height <= MAX_DIMENSION) {
16          System.out.println("Luggage is checked in  successfully !");
17      } else if (depth <= MAX_DIMENSION) {
18      System.out.println("Luggage is checked in  successfully !");
19      } else if (weight <= MAX_WEIGHT) {
20      System.out.println("Luggage is checked in  successfully !");
21      } else {
22      System.out.println("Not valid  for  check-in");
23      }
24      //Closing the Scanner object
25      keyboard.close();
```

The code presented above is adapted from a midterm exam question that required students to determine whether luggage would be accepted by an airline. The problem scenario was designed to be meaningful and relatable, simulating an airline check-in system where luggage must meet specific size and weight constraints. Specifically, no dimension (width, height, or depth) may exceed 36 inches, and the weight must be under 75 pounds. The question was designed to test conditional statements and logical operators.

To help students understand the intended program behavior, the problem description specifies that the program should read four space-separated inputs, width, height, depth, and weight, and output a single message indicating whether the luggage is allowed on the plane. Assumptions, such as valid input format and a single piece of luggage, are clearly stated and discussed during class.

The code shown represents one approach to solving the problem. Following its review and discussion, a second version of the problem is introduced to illustrate an alternative solution. Each version is accompanied by a multiple-choice question. These questions reference specific line numbers, which correspond to the downloadable code provided to students. Which one

of the following is true about the given section of code?

a. Code will run correctly to print the expected output for all valid input data

b. The last else statement (lines 20 to 22) should be removed

c. The program will show multiple responses sometimes

d. In lines 14 to 22, the else-if statements should be if statements

e. In some cases when the weight is more than 75, the luggage will still be checked in

For this question, option **(e)** is the correct answer, as the code contains a logical error. The multiple-choice questions reflect common misconceptions. A frequent issue among novice programmers is misunderstanding the behavior of cascading conditional statements, particularly the importance of the order in which conditions are evaluated. While the code may produce correct output in some cases, a closer analysis reveals that if an earlier condition is satisfied, subsequent checks, such as the weight constraint are skipped, potentially leading to incorrect results. For example, given the input **"32 35 34 78"**, the program incorrectly outputs **"Luggage is checked in successfully!"**, despite the weight exceeding the limit. The goal of this study is to provide strategic guidance to students when they are reviewing peers' code and making sure that the program works as per the requirements for all valid scenarios.

Since the program contains a logical flaw, the class discussion focuses on identifying and correcting the error. Each code segment is reviewed collaboratively to demonstrate strategies for locating and resolving such issues. This process is designed to help students develop code review skills, where they analyze and evaluate written code, thereby improving their ability to reason about program behavior.

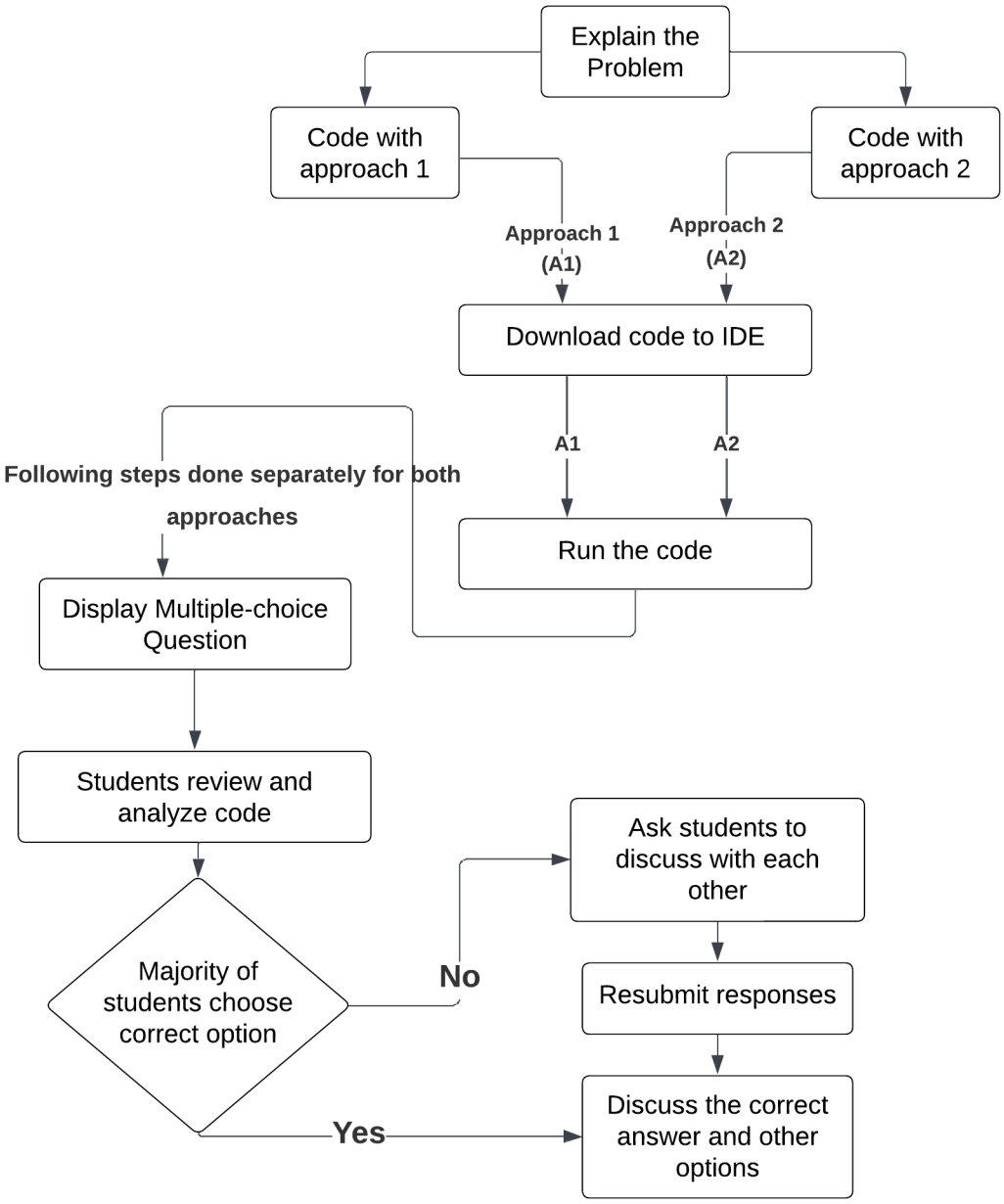


Figure 4.1: Implementation Process

4.2.2 In-class Implementation

The Code Insight activity is implemented after introducing the relevant programming concept, in this case, conditional statements. Its step-by-step structure is shown in Fig. 4.1 [9]. The instructor begins with an in-class explanation of the problem, outlining expected program behavior and compilation assumptions. Students access the initial code and a multiple-choice question via the course management system, and the instructor demonstrates execution for a sample input.

Multiple-choice questions are presented on lecture slides, with responses collected through a classroom response system. The instructor reviews the response distribution before revealing the correct answer. If most students answer correctly, a brief explanation and, if needed, a demonstration of logical corrections follow. When responses are widely distributed, indicating misconceptions, polling is paused for peer discussion. These discussions often lead to improved understanding and revised answers. Observations suggest this process helps students confront misconceptions in real time, reducing repeated errors.

4.3 Methodology

In Fall 2023, we designed and implemented three pilot code review activities in an introductory programming course within a CS program. Following positive initial feedback from students and instructors, we developed a semester-long version of the intervention, referred to as **Code Insight**. This extended version was subsequently integrated into the same introductory programming course in the CS program, during the Spring and Fall 2024 semesters [9].

The list of all the **Code Insight** activities are listed in the appendix under chapter B, with the description of the problem and different approaches showed to the students towards a single problem. To ensure that the course schedule and coverage of core programming topics remained unchanged, we replaced selected in-class participation questions with Code Insight activities. This adjustment allowed us to incorporate the intervention without alter-

ing the instructional timeline or compromising the course's intended learning outcomes and deliverables.

4.3.1 Data Collection

The **Code Insight** intervention was implemented across the Spring and Fall 2024 semesters in introductory programming courses. Enrollment varied between semesters, with the Fall semester providing a larger dataset due to increased student participation. This study focuses on understanding students' experiences with the **Code Insight** activity implemented in an introductory course. To gather student feedback, Institutional Review Board (IRB) approval was obtained to administer anonymous surveys. Initially, feedback was collected online during Fall 2023; however, response rates were low, as presented in the preliminary results [9], during the implementation of the pilot study. To improve participation and capture more immediate and authentic reflections, we transitioned to administering paper-based surveys during class sessions, immediately following the activity. To minimize potential bias and reduce the risk of perceived coercion, neither the course instructors nor the researchers were present during the feedback collection process. This approach was intended to create a more neutral environment and encourage candid responses from students. The survey responses were returned to the researchers after the final grades of the courses were posted, as required by the IRB.

To evaluate the effectiveness and student perception of the **Code Insight** activity implementation, the surveys were collected at two different times during the Spring 2024 semester. The first round of formative feedback was collected shortly after the midpoint of the semester, following the completion of five **Code Insight** sessions. This initial survey aimed to assess the perceived complexity of the activity and to ensure that students were not experiencing undue frustration with reviewing and analyzing code they had not written, and were confused about the structure of the questions.

The second summative survey was conducted during the final two weeks of the semester,

after the last **Code Insight** activity was implemented in the class, prior to the final exam. This survey focused on capturing students' overall impressions of the activity and soliciting suggestions for improvement. Students were asked open-ended questions such as, "How did you feel when working on these activities?" and "Do you have any suggestions for improving them?"

During the second full implementation of the in-class activity in Fall 2024, a single summative survey was administered at the end of the semester, following the completion of all **Code Insight** activities integrated in the class lectures. This final survey also included open-ended questions, enabling a qualitative, in-depth analysis of student responses.

4.3.2 Qualitative analysis

To understand the impact of the **Code Insight** activity, an in-class intervention, and explore students' learning experiences, a qualitative analysis was conducted of their written feedback from the surveys collected after **Code Insight** activity was implemented in the class. This feedback included reflections on the activity as well as suggestions for improvement. The surveys were collected on paper and manually transcribed.

To analyze the comments provided by the students as part of the survey response, we employed Braun and Clarke's [15] six-phase approach to thematic analysis, which aims to identify common patterns in the responses. This method helped us uncover how students perceived, engaged with, and were influenced by the **Code Insight** activities.

The analysis was carried out using NVivo software. Student comments were first coded, and the codes were refined through multiple rounds of review. These codes were then grouped into broader themes to better understand students' experiences. In total, 97 student responses are analyzed, and after reviewing the codes, it is found that each comment is assigned to just one code. When reviewing the codes, it was observed that the students' remarks were generally concise and focused on specific skill development, expressing how the activity contributed to their learning experience. The number of references to the code

represents the number of students mentioning the code in their comments. To ensure a meaningful interpretation of the data, a threshold of 10 references per theme was applied to identify patterns that were both relevant and representative. The threshold is decided after the coding is finalized. With 97 participants, 10 is approximately closer to or more than 10% of the total number of students. This approach enabled us to focus on the most prominent themes and effectively address the research questions to gain a deeper understanding of the students' experiences with this intervention.

The following section presents the findings from the survey, including both the Likert-scale responses and the open-ended feedback. The results highlight key patterns in students' perceptions and experiences with the **Code Insight**, offering insights into its impact on their learning.

The primary researcher has served as both a teaching assistant and instructor in the introductory programming courses for over five years. The researcher was also the instructor for the class where **Code Insight** is integrated into the instruction. The advisor to the primary researcher has also integrated the **Code Insight** activity into the introductory course. With over fifteen years of experience teaching beginning programming classes and more than thirty years as a faculty member, they are well-positioned to guide the effective integration of the **Code Insight** activity into instruction without disrupting the course's critical deliverables. Their experiences with the beginning programming classes could both contextualize and influence their evaluation of the student comments and learning experience.

4.4 Results

The data collected over two semesters was analyzed separately to address the research questions and evaluate the impact of the proposed active learning intervention on students' learning experiences. The **Code Insight** activity was implemented in two entry-level programming courses, designed for students with little to no prior programming experience. The

surveys are placed to ensure that students are not overwhelmed by the complexity of these activities and feel supported in the guided classroom environment, where they can develop code review skills to analyze and evaluate code and gain critical thinking, problem-solving, and debugging skills.

To gain a comprehensive understanding of the intervention's effectiveness on the students learning outcome, this study examined students' experiences through their written feedback distributed across the two semesters. In the following section, the data from the Spring and Fall semesters are presented and discussed separately to highlight any differences or trends across the two cohorts.

4.4.1 Survey from Spring 2024

Two in-class paper-based surveys were administered during this semester to gather student feedback on the **Code Insight** activity. The first survey included Likert-scale items, where students rated their agreement with various statements on a scale from 1 (Strongly Agree) to 5 (Strongly Disagree).

The study examines whether the in-class intervention facilitates the development of code analysis and evaluation skills and enhances students' ability to write and debug programs.

Table 4.1: Student feedback during mid-semester of Spring 2024

Survey Questions	**Student response**		
	Positive	***Neutral***	***Negative***
Found it confusing	12	16	15
only one possible solution	1	1	41
Feel confident in debugging	24	12	7
Feel confident in writing program	26	12	5

Students new to programming often struggle with identifying and correcting logical errors and analyzing program behavior [28]. The survey inquired whether **Code Insight** intervention during the class lectures aided students in identifying errors in their own code

and visualizing multiple approaches to solving a problem using different programming constructs.

Table 4.1 summarizes the survey results from 43 students across two sections of introductory programming courses. The original five-point Likert scale was collapsed into three categories: positive (agree/strongly agree), neutral, and negative (disagree/strongly disagree). As shown in the table, the majority of the students responded with positive or neutral scale, indicating that they feel confident when writing a program and debugging it.

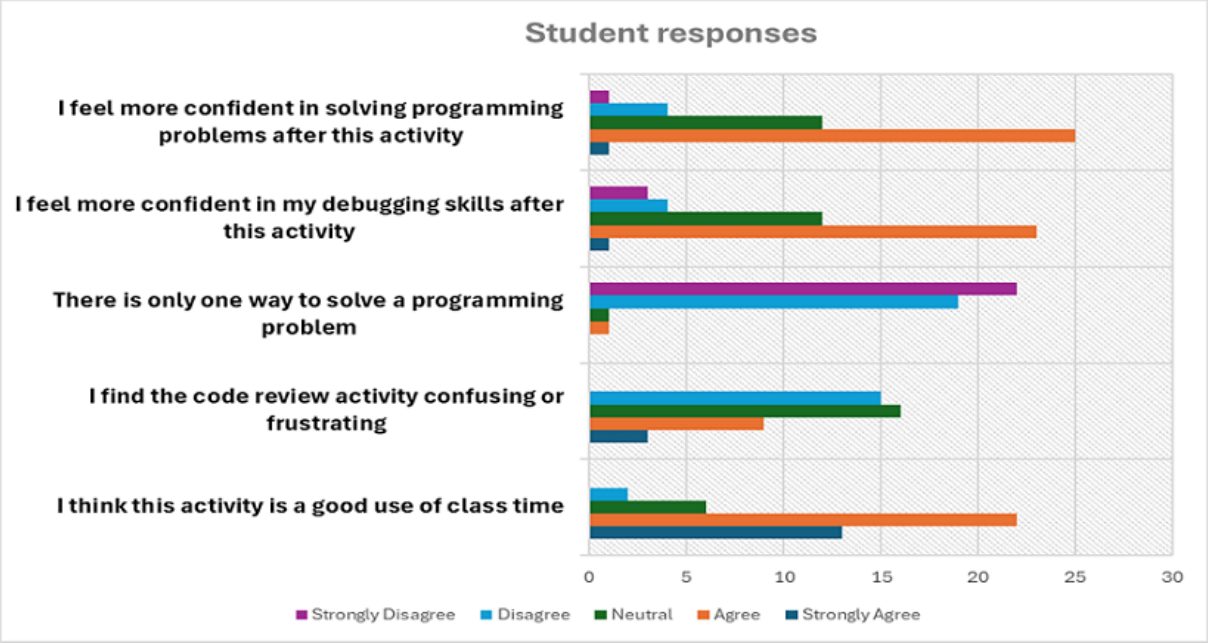


Figure 4.2: The graph representing students' responses during the Mid-semester of Spring'24

One of the main focuses of the **Code Insight** intervention is to demonstrate different approaches to solving a single problem, encouraging students to practice and explore various solutions when the program encounters logical errors, thereby guiding them to identify and resolve these errors. When asked in the survey about if there is only one way to solve a programming problem, almost all the students disagree with that, hence understanding and learning different programming concepts to solve a give problem.

The results, with the survey questions, are explained in the Figure 4.2. As shown in Figure 4.2, overall, 82% of students reported that the **Code Insight** activity positively

impacted their learning and provided appropriate challenges to strengthen their problem-solving skills. More than 50% expressed increased confidence in debugging, while around 30% remained neutral.

Similarly, over half of the students reported an improvement in their confidence in their programming abilities. Nearly all students disagreed with the notion that programming problems have only one solution, aligning with the activity's design, which presented multiple approaches to each problem. Although prior research notes that code review can be difficult for novices [43], over 70% of students did not find the activity confusing or frustrating. However, 27% did report some level of difficulty, consistent with challenges typically faced by beginners.

The sample size for the Spring semester was relatively small, with only 15 students participating in the final survey. The survey was administered during the final weeks of the semester, when class attendance typically declines. Responses to the first survey question were mostly positive. Most participating students indicated that the code review activity, implemented within an active learning approach, had a beneficial impact on their understanding of complex programming concepts. They also reported that the activity provided an appropriate level of challenge to support their skill development. The sample quotes for this part of the data analysis are selected based on the codes with the highest number of references among this semester's data. There are only 13 students who provided some comments; the other two have no comments. The themes represented by the following quotes are *"Productive challenge"*, which consists of codes such as "learning debugging" and "learning concepts", similar to the coding for the fall semester.

> ***"Sufficiently challenged. The problems were not insanely hard. Developed my problem-solving skills."***

Several students emphasized the development of critical thinking and problem-solving skills, as well as the opportunity to explore multiple approaches to a single problem.

"I felt pushed to think critically about the problem. I learn better by solving problems, so it was helpful to do them in class."

Students also noted that conducting these activities during class enabled real-time clarification of misconceptions through peer or instructor interaction.

"I feel fine working on them. I do not think code review is essential, but it does help when later identifying bugs in my own code, so it has a positive impact."

The activity was perceived as engaging and helpful for identifying and correcting errors in students' own programming projects. In response to the second open-ended question for comments, students provided constructive suggestions for improving the implementation of the **Code Insight** intervention. Some expressed a desire for more frequent integration of code review activities into the course structure.

"I would like to have a code review portion of the class to be a regular everyday thing."

"Honestly, just do more of them. Other than that, I would not change them."

Others suggested incorporating collaborative elements to enhance peer discussion and engagement.

"Maybe put us in a group and force the conversation. I know most CS students are shy, but I think this will help a lot."

By the end of the semester, we observed a decline in peer discussion. To address this, future implementations will include pre-assigned groups to facilitate structured collaboration and collective responses. This should go under future work. We also plan to adopt a team-based learning strategy as proposed by Michaelsen and Fink [71].

Some students recommended breaking down complex code into smaller segments to improve focus and comprehension.

"Review smaller portions of the code at a time."

This feedback aligns with our observation that the complexity of code increased later in the semester, particularly with topics such as *ArrayLists*.

"Nothing, explaining the code after is very helpful though."

Post-activity code walkthroughs were reported as beneficial. Students appreciated seeing how to correct errors or verify correct behavior, which supported their understanding of the relationship between intended and actual program behavior.

In the second survey, which included Likert-scale questions (1 to 5), students expressed a preference for in-class code review activities over traditional lectures. Over 70% reported that the **Code Insight** exercises enhanced their ability to solve programming problems, consistent with the qualitative feedback and the first survey from Table 4.1.

4.4.2 Survey from Fall 2024

During this semester, only one feedback survey was conducted after the final **code insight** activity is implemented. The students response was collected using the Likert-scale questions and two open-ended question for describing their learning experience and constructive feedback. The Figure 4.3 included the response to the four questions asked to understand the students' learning experience through the **Code Insight** activity duirng class lectures.

The Figure 4.3 describes the survey responses for the Likert-scale based questions from Fall 2024. The surveys are conducted across two sections of the introductory class, each of which has about 80 students enrolled.

The survey findings indicate that the implementation of the **Code Insight** activity has positively influenced students' learning outcomes, particularly by enhancing their ability to

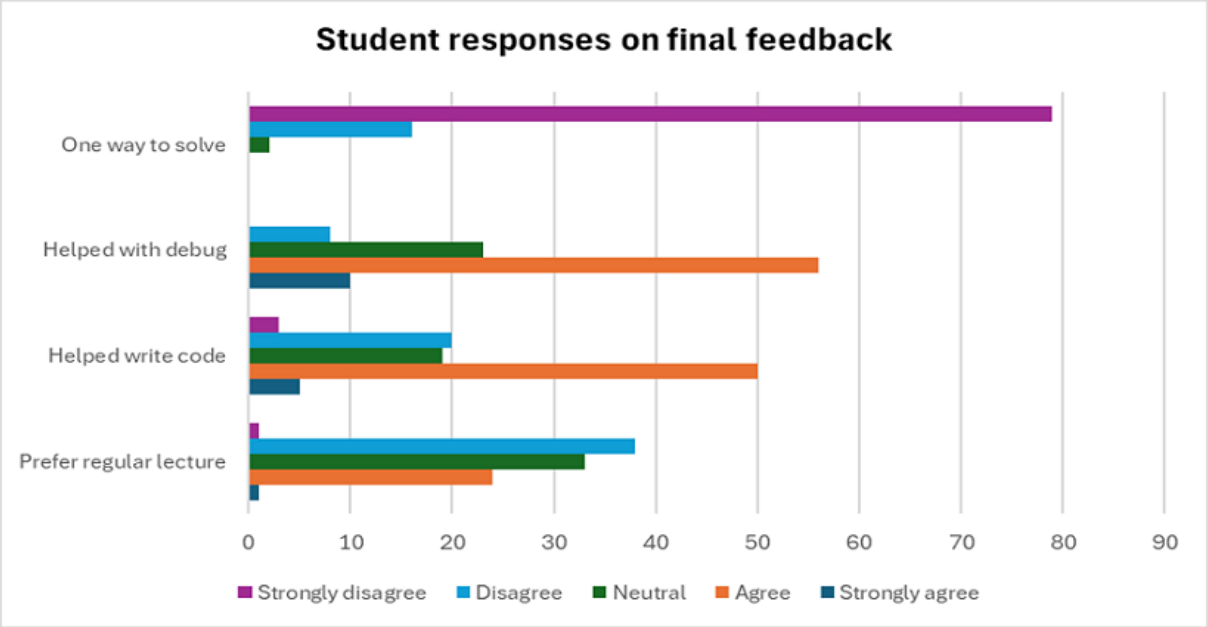


Figure 4.3: Graph representing students' response during Fall semester survey

write code independently from the ground up. A key result from this semester's data, illustrated in Figure 4.3, highlights that approximately 70% of respondents reported feeling supported during the debugging process. Many students mentioned that the **Code Insight** helped them when writing their program. This finding highlights the importance of providing consistent and structured guidance when reviewing and analyzing code. Such support appears to play a critical role in developing students' debugging skills, including persistence, problem-solving, and self-efficacy in identifying and correcting errors.

The findings from the Likert scale are consistent with the analysis of the students comments. The identified themes are presented using a threshold of 10 or more responses. The following themes are identified in the students' responses:

Learning concepts

The feedback responses shows that the **Code Insight** activities were sufficiently challenging for the students and helped them with learning programming concepts.

"I felt that the code review better helps us to understand newly intro-

duced coding techniques by using them in fully functioning programs instead."

"I believe code reviews help newer programmers catch the most common mistakes as they happen, reducing frustration at confusing error messages."

Findings from this theme suggest that the **Code Insight** offers a structured approach to analyzing and evaluating code, enhancing students' ability to understand code written by others and reinforcing recently learned concepts through practical examples. Student feedback indicated increased confidence in both writing and analyzing code after engaging with the activity.

"I am better at my tasks. I look at the code more deeply."

"I felt more engaged than in a usual lecture and did not mind looking for the problem in the code."

"I feel like I am challenged to see things I haven't seen before when reading code. It helps me get an idea of how to go about my coding."

"Code reviews are informative and I felt that I was grasping the material and understanding the code more."

The analysis of the students' comments revealed patterns indicating that they feel challenged during activities when reviewing the written code, and that the guided steps in the questions motivate them to examine the code in more detail. Overall, 30 percent of all the participants indicated that these activities helped them learn the concepts and analyze the code.

Learning debugging

A key objective of this study was to assess whether the **Code Insight** activity supports students in performing debugging tasks and in developing confidence in addressing errors in their own code. The findings suggest that repeated practice with error identification through the activity helped students feel more comfortable and capable when debugging their programs.

> ***"Good, debugging is not my strong suit, but I find it very helpful to know how to do it when writing my own programs"***
>
> ***"I felt as if I got a better grasp on the concept of debugging."***
>
> ***"It allows me to practice a methodical way of thinking to analyze and debug code."***
>
> ***"It helped analyze the code and find out what was or wasn't wrong with the code."***
>
> ***" I feel that code reviews benefit me b/c I can actually run code and get feedback and attempt to fix the problems."***
>
> ***"It's kind of fun when you did not make the mistake."***
>
> ***"I had to think about what a code needs to work as well as how to fix what's broken."***

The findings indicate that the **Code Insight** activity integrated in the instruction supports the development of students' debugging skills by guiding them through the process of identifying faults in a program and applying conceptual knowledge to resolve them. Notably, 20% of the student responses mentioned that these activities helped them debug their code independently outside of class, suggesting that the skills practiced during the **Code Insight** activity extended beyond the classroom and contributed to their self-directed learning.

Critical-thinking and Problem-solving

Some students reported that the activity encouraged them to critically evaluate code and consider how to resolve issues when the code did not function as expected.

> ***"It feels like it demands more critical analysis on thinking in the moment, which I believe is helpful."***

Several students indicated that the activity provided an appropriate level of challenge, prompting them to think critically and develop problem-solving skills while reviewing written code.

Code analysis

The analysis showed that although students found the **Code Insight** activities challenging, they recognized their value in deepening code analysis and enhancing learning. While reviewing code was initially difficult, structured in-class guidance helped transform this challenge into a productive learning experience.

> ***"I feel intrigued and ready to learn when doing code reviews. It helps to see actual code and attempt to actually understand what is happening."***

> ***"it feels a little overwhelming as we are just given a bunch of code and at first it is difficult to start. Once it's broken down, it's easier."***

> ***"Stressed, but in a productive way if that makes sense"***

> ***"I feel lost sometimes because they are very difficult. However, when I do figure it out myself, I feel lots of satisfaction."***

> ***"It can be frustrating at times but it is worth it."***

Approximately 20% of students reported that they found it challenging to read and interpret code written by others. Despite this initial difficulty, many of these students acknowledged that the activity ultimately supported their understanding of programming concepts and improved their ability to analyze written code.

These findings reinforce the notion that code review is a cognitively demanding task for novice programmers. However, when strategically implemented in the classroom with appropriate scaffolding, it can reduce cognitive load and foster meaningful learning. The structured support provided during the **Code Insight** activity appears to play a critical role in helping students engage with complex programming tasks, transforming a challenging experience into an opportunity for conceptual growth.

Improvements

This theme highlights students' interest in more frequent in-class code review activities to reinforce programming concepts and improve code analysis skills. Based on a threshold of 10 or more responses per code, several key preferences emerged.

Students preferred reviewing shorter code segments, noting that longer, project-level code, especially near the end of the semester, felt overwhelming. They also expressed a desire for more in-class explanation, particularly when logical errors were involved, valuing guided discussions that demonstrated how such errors could be identified and corrected.

> ***"Have more interactive questions and shorter code segments (more short examples rather than long)."***
>
> ***"I would add more code reviews into class."***
>
> ***"Make them even more interactive like we edit the code to fix it instead of just reading it and finding the issue."***
>
> ***"Make us add code to fix it if it doesn't work as intended."***

"Do more of them and go more in depth with what something does exactly rather than going in so deep on how it works."

Some students suggested that receiving the code in advance would improve their understanding and help them come to class better prepared. These findings indicate that scaffolding code review activities, by shortening code length, providing pre-class materials, and including in-class walkthroughs, should be explored.

Negative feedback

A small number of students reported difficulties that hindered their learning experience. While these responses fell below the reporting threshold, they are included to inform future implementation. Some students found the questions unclear and struggled with the step-by-step code review process, citing insufficient problem explanations.

"I felt lost at some point as I did not understand the purpose of some of the code."

"Sometimes, it was overwhelming to do these activities and I honestly felt like it, because it was for a grade."

These challenges may be attributed to the inherent difficulty of code review for novices and the added pressure of limited time. Notably, the survey was conducted during the final two weeks of the semester, a period typically associated with elevated stress levels.

These findings are consistent across both semesters, where students feel more confident in their code-writing skills and can debug the code when working on the program independently. The activities are engaging for the students, and findings from the survey suggest that students would like more practice and time spent on the **Code Insight** activity during the class lecture. To address the research questions, a qualitative analysis is performed on students' written responses on their experience of this intervention in the instruction and

any constructive feedback that help them improve the code review task of analyzing and evaluating the code.

4.5 Discussion

This study examined the integration of code review activities into introductory CS courses, focusing on students' learning experiences. As noted by Indriasari et al. [43], there is a need for more empirical research on developing code review skills among novice programmers. Prior studies have highlighted challenges such as low participation [59], engagement [40], and motivation [100].

The *Code Insight* intervention was designed to address these issues by offering structured, low-overhead activities that promote participation and reduce workload for both students and instructors. While this study did not directly assess coding proficiency, survey responses suggest that the activity improved students' debugging confidence and conceptual understanding.

4.5.1 Research Questions

RQ1: How do students describe the role of Code Insight in enhancing their learning experience?

Survey responses indicate that the **Code Insight** activity helped students deepen their understanding of programming concepts by reviewing code written by others. While many found the tasks challenging, they reported improved conceptual learning. Beginner programmers often struggle with reading and analyzing unfamiliar code, but when guided in a structured classroom setting, these activities enhance their overall learning experience.

Several students mentioned that reading someone else's code can sometimes feel overwhelming. However, when the code is explained and discussed in class, it helps them become more comfortable with analyzing unfamiliar code and understanding complex programming

concepts. The **Code Insight** intervention has contributed positively to students' learning experiences by fostering deeper engagement and collaborative exploration of programming tasks by analyzing code.

RQ2: In what ways does Code Insight support students when debugging their own programs?

Several students noted that following the structured steps in the **Code Insight** activity helped them identify faults in their own projects, thereby improving their debugging skills. Both Likert-scale responses and open-ended comments from the student feedback support this finding. By offering a guided strategy, the activity effectively supports the development of debugging skills among novice programmers.

Through the process of comparing actual and intended program behavior, **Code Insight** helps students locate problem areas in their code and address misconceptions related to new programming concepts.

Importantly, students reported gaining confidence in approaching debugging tasks, as the structured intervention provided clarity and reduced discomfort often associated with complex debugging tasks. Additionally, by exposing students to different coding approaches for the same given problem, the activity encouraged them to explore alternative solutions when their initial logic did not produce the expected results, fostering a more flexible and resilient problem-solving mindset to effectively debug the code.

RQ3: How does Code Insight influence students' understanding of multiple approaches to solving programming problems?

In both semesters, almost all the students disagreed with the idea that a programming problem has only one solution. This consistent finding suggests that students value exploring multiple approaches to a single problem. The **Code Insight** activity supports this by presenting alternative solutions to the same problem, encouraging flexible thinking among beginner programmers. This experience helps students approach debugging more effectively by considering different strategies for identifying and fixing errors.

Findings were consistent across the two semesters, reinforcing the value of guided code review in early programming education. Embedding common misconceptions in multiple-choice questions and discussing them in class helped clarify misunderstandings. Peer discussions and opportunities to revise answers further supported learning. Although the analysis of current implementation is limited to a single institution, the **Code Insight** intervention shows promise for broader application, with implementation in classes of different sizes. Its integration with in-class response systems and participation-based grading reduces stress and grading overhead, making it adaptable to other contexts, including more advanced courses.

4.5.2 Limitations

As the **Code Insight** activity is a new intervention, its implementation was limited to a single institution and two semesters. While the students in the classes were informed that their responses were anonymous and would not be reviewed until after the semester was over, it is possible that students provided positive feedback in an effort to please the instructors. In addition, the qualitative analysis was done by course instructors who designed the activity. While the instructors made every effort to analyze the data fairly, they may be biased towards the success of the intervention.

4.6 Summary

This study explored the integration of structured code review activities into introductory CS courses through the **Code Insight** intervention. The goal was to support novice programmers in developing debugging skills, enhancing their conceptual understanding, and fostering flexible thinking. The findings address a gap in the literature regarding how beginners learn to review and reason about code, as previously noted by Indriasari et al. [43], and align with broader concerns about the lack of systematic debugging instruction in CS education [65, 113].

Survey responses and student feedback suggest that the **Code Insight** activity helped students better understand programming concepts by analyzing code written by others. Although many students found the tasks challenging, they reported improved confidence in debugging and a deeper grasp of core ideas. The structured nature of the activity provided a helpful guide for identifying and fixing errors in their code.

Students also showed an increased appreciation for the idea that programming problems can have multiple valid solutions. By reviewing alternative approaches, they developed more flexible problem-solving strategies, which are essential for effective debugging.

The activity was implemented across two semesters and showed consistent results, reinforcing its value in early programming education and presenting improved learning experience for the beginner programmers. Embedding common misconceptions in classroom discussions and allowing peer interaction further supported learning. While the study was limited to one institution and course level, the low-overhead design of the **Code Insight** as an in-class activity makes it adaptable to other contexts, including more advanced courses.

Overall, the findings highlight the importance of incorporating structured code review and debugging instruction early in the CS curriculum. Doing so can help students develop essential skills, reduce their reliance on trial-and-error methods, and prepare them to critically evaluate both self-written and AI-generated code, thereby enhancing their confidence in tackling complex programming tasks.

Chapter 5

Evaluation of Code Insight

5.1 Overview

To evaluate the impact of the *Code Insight* intervention and its in-class implementation on student learning, this chapter analyzes student performance across three exams administered during the semester. The study includes two sections of an introductory programming course: one for students with prior programming experience and another for complete beginners. Despite being taught by different instructors, both sections followed an identical course structure to ensure consistency in instruction and assessment.

The exams were modeled after those used in the previous two years of the course. Questions for the *Code Insight* activities were developed using student responses from earlier exams, allowing instructors to incorporate relevant examples into instruction and provide students with targeted practice on key programming concepts.

This chapter investigates the research question: *How does the* ***Code Insight*** *in-class intervention influence the code-writing skills of novice programmers in an introductory programming course?* To answer this, we use quantitative methods to examine the relationship between students' performance in **Code Insight** activities and their scores on programming questions in exams. Correlation analysis is employed to assess whether participation in struc-

tured code review supports the development of programming logic and writing skills [96, 65].

5.2 Data Collection

The **Code Insight** intervention was implemented in introductory programming courses during Spring and Fall 2024. These in-class activities were distributed throughout the semester, following the completion of each major programming concept. Some instructional examples from previous semesters were replaced with the Code Insight activities. However, no core programming concepts were omitted or compromised during the semester-long implementation.

The introductory programming course is divided into two sections: one for students with no prior programming experience and another for those with a limited background in programming languages. Both sections use the same exams, which include identical programming questions across semesters. To ensure consistency in grading, a common rubric is applied to evaluate programming questions in both sections.

Student learning was supported through three main sources:

- Pre-class reading assignments, where students reviewed concepts before the concepts were covered in the lectures;
- Interactive lectures, where understanding was assessed using in-class response systems; and
- Independent practice through programming assignments.
- Homeworks and interactive tutors to practice the programming concepts

In 2024, the *Code Insight* in-class intervention was fully implemented throughout the Spring and Fall semesters, covering all essential programming concepts. During the Fall 2023 semester, a pilot version of the activity was introduced in only one section of the course.

Student performance in the **Code Insight** activities was recorded using an in-class response system, which captured individual responses to each question. These responses contributed to the participation grade, accounting for 5% of the overall course grade. The grading criteria for the introductory course are distributed between assignments, exams, and class participation. The **Code Insight** consist of 25% of the total grade of participation in the class for the entire semester.

For exam grading, the Gradescope platform was used, which supports rubric-based evaluation of programming questions [88]. Only the scores from the programming problems from the exam were extracted for analysis. The programming question is the last problem in the exam, which is worth 30 points; other questions focus on lower-level learning objectives, such as tracing the output of an existing program. A sample of the final programming problem from the Fall 2024 semester is presented in the appendix chapter C, which includes all three parts of the question. As described in the appendix, students were provided with appropriate method descriptions in the format of Java documentation to support their understanding and assist them in writing their code.

5.3 Methodology

This chapter uses correlation analysis to examine the relationship between students' performance in structured code review activities and their scores on programming questions in exams. The purpose is to determine whether participating in the **Code Insight** activities helps students improve their coding skills and gain a deeper understanding of complex programming concepts. To measure this relationship, Pearson's correlation coefficient is used. This statistical method is commonly applied in educational research to evaluate the strength and direction of relationships between learning activities and student outcomes [36].

The two sections of the introductory programming course, each comprising students with distinct programming backgrounds, are the focus of our analysis. The correlation between

writing skills and in-class activities is analyzed, examining the impact on students with and without prior programming experience. This analysis is of interest as it sheds light on the learning dynamics in the diverse student body. The following section presents the analysis of student performance, focusing on the correlation between their scores in the **Code Insight** activities and their performance on the programming question in the final exam, representing the code-writing skills. The exams are conducted on paper, where students write programs by hand in the absence of a development environment, which is typically used for project assignments and in-class examples. The results are reported separately for each semester and each section of the introductory programming course to provide insight into students' code-writing performance during exams. This approach also enables a comparison between student outcomes in the **Code Insight** activity and program writing tasks.

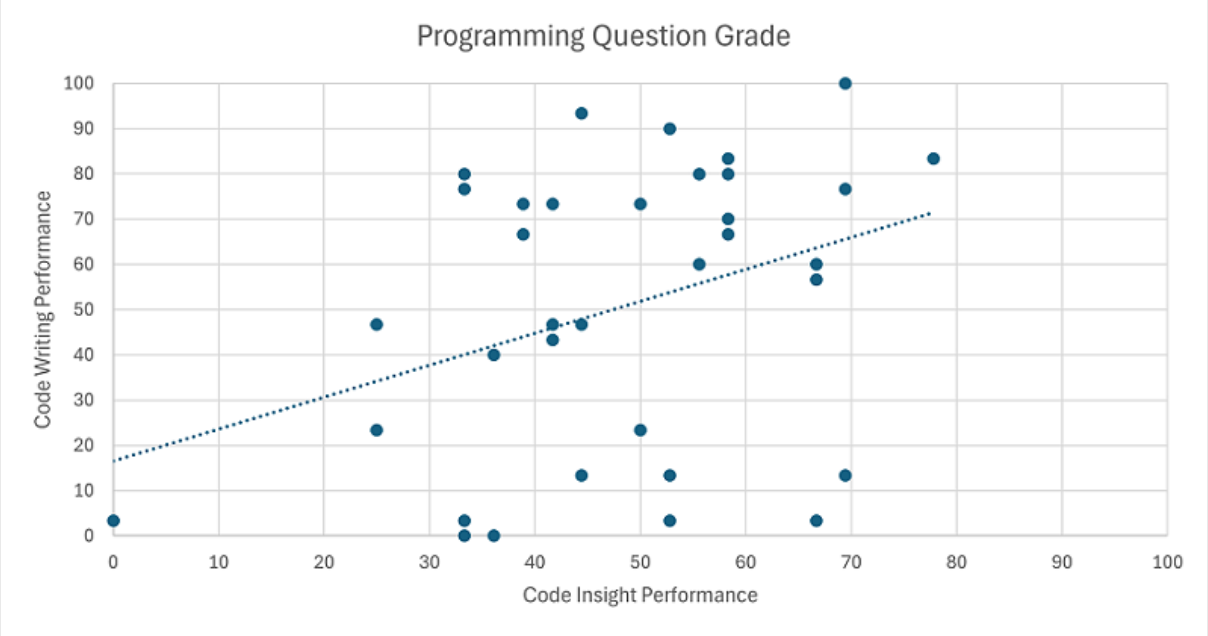


Figure 5.1: Performance in Programming Question in Spring 2024 with no Programming Experience

5.3.1 Spring 2024, with No Prior Programming Experience

The number of students enrolled in the introductory programming course during the spring semester is lower than during the fall semester. The spring semester sometimes consists of the students retaking the course for the second time.

The Pearson correlation coefficient of 0.356 indicates a low to moderate positive relationship between students' performance in the **Code Insight** activity and their performance on the programming question in the final exam.

As shown in Figure 5.1, the findings indicate a pattern among students' performance and suggest that students who performed better in the **Code Insight** activity were somewhat more likely to perform better on the programming question. However, the relationship is not particularly strong. These results provide some evidence that the **Code Insight** activity may support students' development of programming skills when writing a program and highlight the potential value of the **Code Insight** as a supplementary learning tool for students' learning of complex programming concepts.

5.3.2 Spring 2024, with Programming Experience

A similar correlation coefficient is calculated between the students' performances. The structure of the exam questions is consistent with that of previous semesters for each section of the introductory programming course.

The Pearson correlation coefficient of 0.167 indicates a weak positive relationship between students' performance in the **Code Insight** activity and their performance on the programming question in the final exam. As shown in Figure 5.2, while the direction of the relation is positive, the strength of the correlation is low. This implies that the **Code Insight** activity may have limited predictive value for performance on the programming question, which represents program writing skills in the final exam, and that other factors likely play a more significant role in students' performance on the programming question. In this section of the course, students who have some prior programming experience may

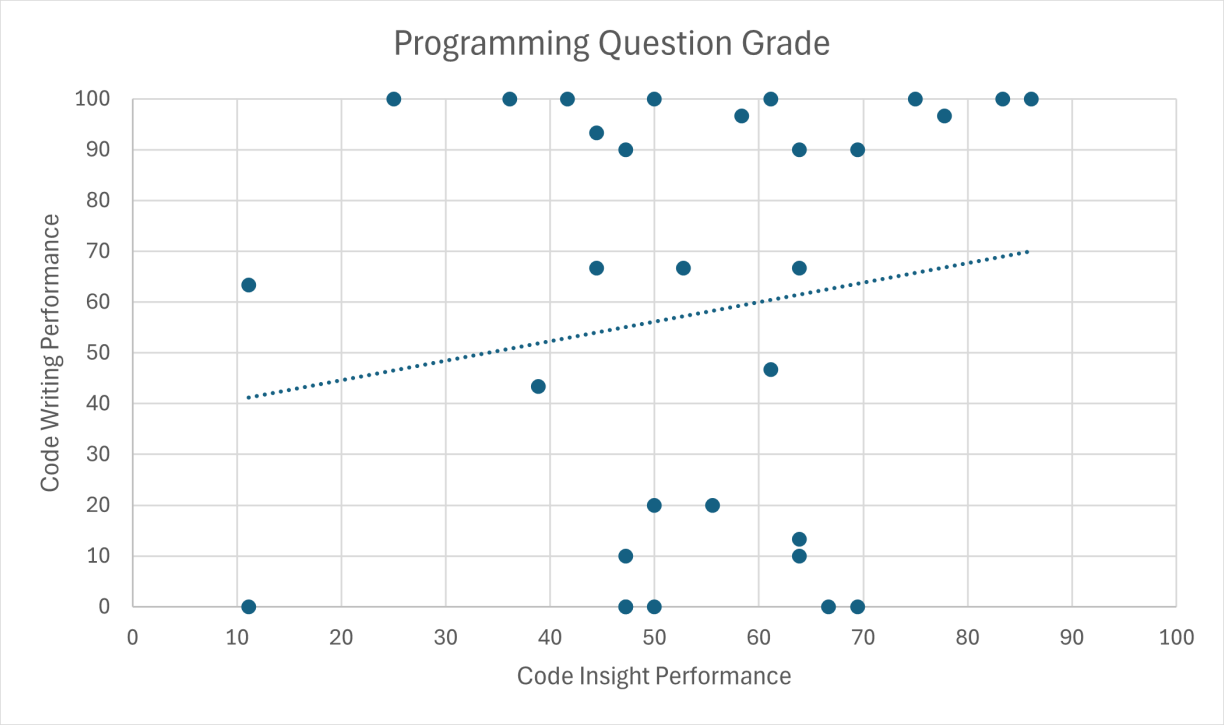


Figure 5.2: Performance in Programming Question in Spring 2024 with Programming Experience

skip class and are confident in their programming skills and they can't get any points for the **Code Insight** activity.

5.3.3 Fall 2024, with No Prior Programming Experience

Students' performance on the programming question in the final exam is compared to their performance on the **Code Insight** activity conducted during lectures [77]. This comparison is motivated by the fact that the **Code Insight** activities are designed using programming questions from the mid-term and final exams of previous semesters. These activities are intended to match the exams in both difficulty level and the programming concepts they assess, providing a consistent basis for evaluating students' learning.

The code for the review during these activities was written by drawing inspiration from the programming logic that the students used in their answers during the exam in previous years of the course, and it included the different approaches that they employed. The code

has been rewritten to avoid copying student code without permissionand only uses inspiration to avoid any integrity issues.

The grades are collected from the questions of the final exam that requires students to write a full program, usually in the format and length of a class project.

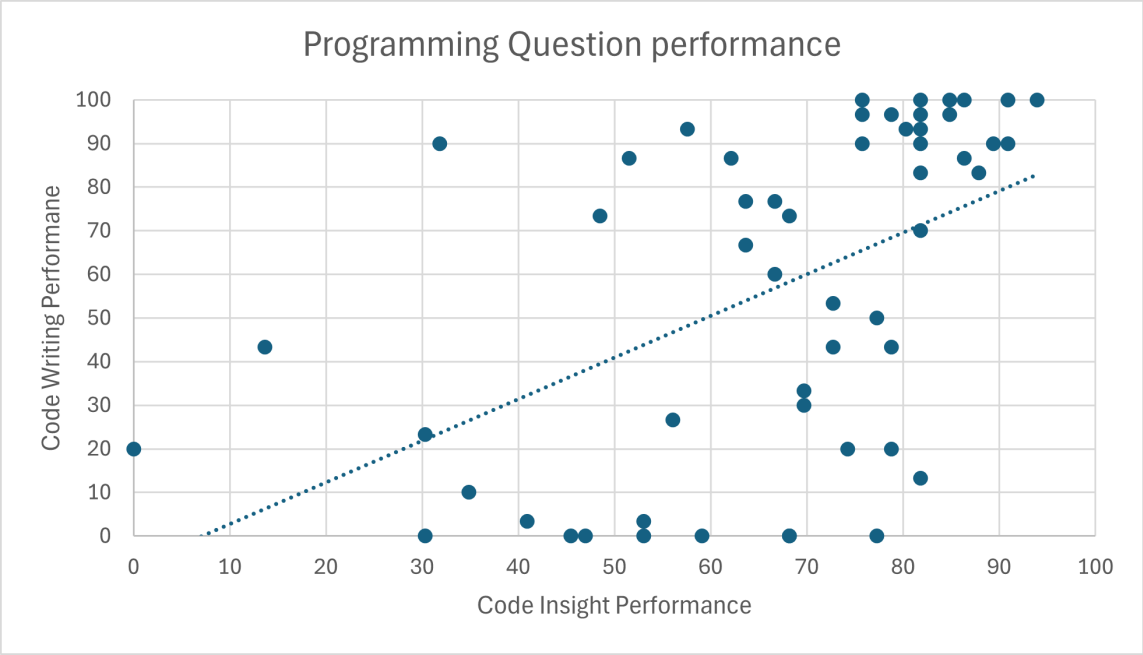


Figure 5.3: Performance in Programming Question in Fall 2024 with no Programming Experience

The calculated Pearson correlation coefficient between students' performance in the **Code Insight** activity and the programming question on the final exam is 0.52. This indicates a moderate positive relationship, suggesting that students who performed well in the **Code Insight** activity also tended to do well on the programming question.

As shown in Figure 5.3, the findings suggest that the **Code Insight** activity may have contributed to students' improved understanding of complex programming concepts and enhanced code-writing skills. However, it is important to acknowledge that this observed positive correlation does not confirm causation, and other factors may have influenced the outcomes. Furthermore, the results are consistent with the observations discussed in chapter 4, where students reported increased confidence in their programming abilities and de-

scribed the activity as a positive and supportive learning experience.

5.3.4 Fall 2024, with Programming Experience

In this section of the introductory programming course, students have some programming experience when enrolled in the class. The programming question being evaluated is the same exam question used in the other section of the course.

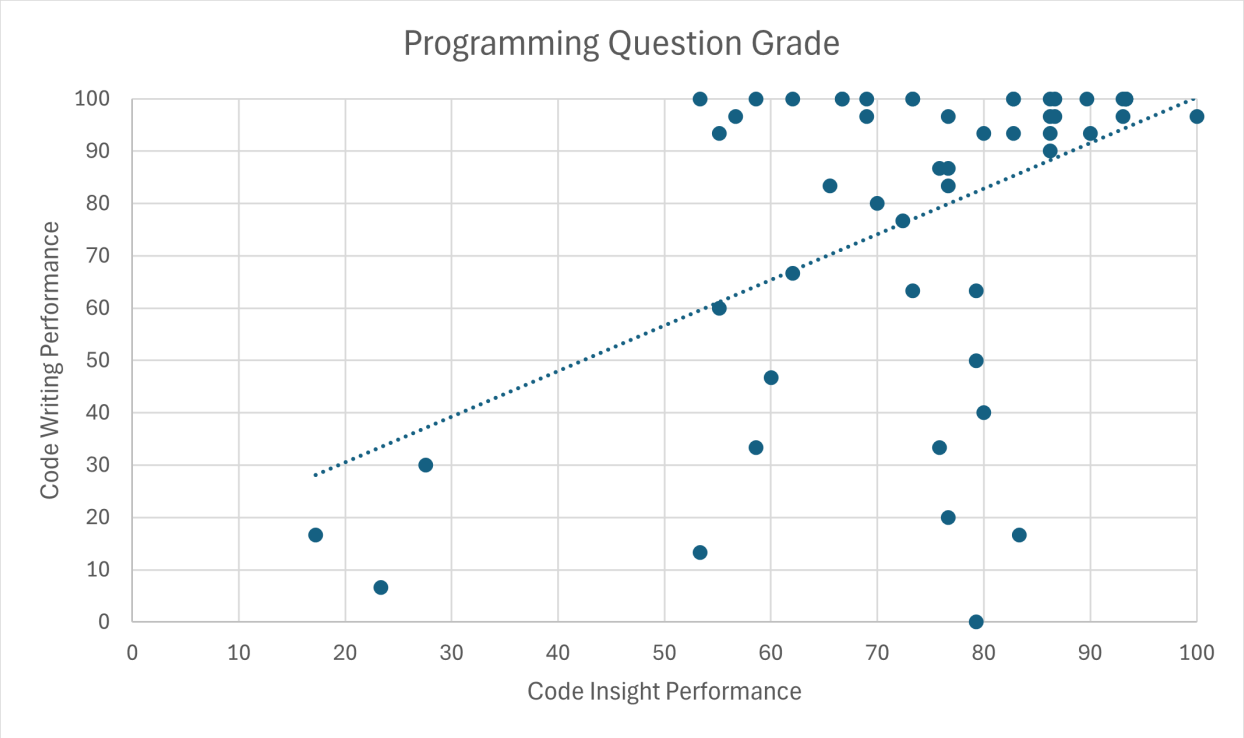


Figure 5.4: Performance in Programming Question in Fall 2024 with Programming Experience

The calculated Pearson correlation coefficient between students' performance in the **Code Insight** activity and the programming question on the final exam is 0.49, indicating a moderate positive relationship of similar strength to that observed in the other section of the introductory programming course in a CS program. This suggests that students who performed better in the **Code Insight** activity generally performed better on the exam question of the full programming problem as well.

As shown in Figure 5.4, the performance on the question shows a positive trend with

respect to the performance on the **Code Insight**. Although the correlation is not strong, it still provides evidence that the **Code Insight** activity may support students in developing the programming skills assessed in the final exam. These findings reinforce the potential value of integrating such activities into instruction to enhance students' understanding of complex programming concepts.

5.3.5 Comparing Final exam grade

The **Code Insight** activity was implemented over three consecutive semesters and were integrated into class lectures, beginning with Fall 2023, which served as a pilot phase to explore how the activity could be integrated into lecture sessions. During this initial semester, at least five **Code Insight** activities were conducted. Each activity was documented with details such as the time required for implementation and suggestions for improving its integration into the instructional flow. These insights informed refinements in subsequent semesters to ensure smoother and more effective use of the activity in classroom teaching.

Exam Grades	**W/O Programming Experience**			**With Programming Exp**	
	Fall 2023	Spr 2024	Fall 2024	Spr 2024	Fall 2024
Mean	76.53	65.2	67.25	75.8	78.94
Median	82	75	73.5	84	87.5

Table 5.1: Final Exam Grades Comparison Based on Programming Experience

As shown in the Table 5.1, the mean and median final exam scores remained relatively consistent across all three semesters and both course sections. The final exams were administered on paper and included programming questions focused on advanced topics such as ArrayLists. This consistency in exam performance suggests that the integration of **Code Insight** activities did not negatively impact students' overall outcomes and may have contributed to maintaining a stable level of performance on complex programming tasks. The activity appears to have provided targeted support for developing essential programming skills, such as code review, debugging, and enhancing confidence among students when performing these complex tasks.

5.4 Limitations

Collecting students' grades to evaluate the impact of the **Code Insight** activity presented several challenges. Student consent was obtained in class on a specific day early in the semester, and attendance on that day directly affected the number of consents collected. The original plan for data analysis involved comparing grades from previous semesters when **Code Insight** was not implemented. However, according to IRB guidelines, access to student grades requires individual consent. Due to limited time before graduation and the insufficient number of consents, a direct comparison between student performance with and without **Code Insight** could not be conducted.

5.4.1 Factors Affecting Data Collection

The *Code Insight* activities were conducted during lectures to provide structured guidance on reviewing and analyzing peer-written code. Student consent was obtained during the first month of each semester to use their grades for research purposes. Some students were absent during the consent collection process, while others declined to participate. A few students who initially consented later withdrew from the course, resulting in a reduced final sample size.

5.4.2 Limitations in Using Historical Grades for Comparison

According to IRB guidelines, anonymized grade data from previous semesters cannot be used without individual consent. Consent was collected across three semesters: Fall 2023, Spring 2024, and Fall 2024. However, due to different instructors assigned during Fall 2023, only one section from Fall 2023 from the introductory programming course was included for direct comparison.

5.4.3 Data from One Institution

The activities were integrated into introductory programming courses at a single institution. The underlying theoretical framework used in this implementation is designed to be scalable across institutions and adaptable to classes of any size. This scalability is possible because the intervention does not require manual grading and is conducted during regular lecture sessions.

5.4.4 Correlation vs. Causation

The findings from the two semesters in 2024 indicate a moderate to weak positive correlation. However, due to the absence of a comparison group, specifically, students from previous semesters who did not participate in the activity, it is not possible to conclude that participation in **Code Insight** directly improves students' code writing skills. Other factors may have influenced students' performance during both the activity and the exam.

It is important to emphasize that correlation does not imply causation. The data supports the hypothesis that students with strong code writing skills tend to perform well in **Code Insight**. However, both outcomes may be influenced by external variables not accounted for in this study. Future research involving **Code Insight** will aim to address these limitations by investigating its direct impact on students' code writing abilities when implemented in classroom settings.

5.5 Summary

This chapter examined the relationship between students' performance in the **Code Insight** activities and their scores on the final exam programming question across multiple semesters and student backgrounds. The analysis revealed varying degrees of positive correlation, with the strongest relationships observed in sections where students had no prior programming experience.

- In **Fall 2024**, students without prior programming experience showed a *moderate positive correlation* ($r = 0.52$), suggesting that the **Code Insight** activities were particularly effective in supporting their learning of complex programming concepts.
- Students with prior experience in the same semester also showed a *moderate correlation* ($r = 0.49$), indicating that the activity benefits a broad range of learners.
- In **Spring 2024**, the correlation was *lower* for both groups ($r = 0.356$ for students without experience and $r = 0.167$ for those with experience), suggesting that other factors may have influenced performance during this term.

Overall, these findings support the use of **Code Insight** as a valuable instructional intervention in introductory programming courses. The activity not only aligns well with assessment goals but also enhances students' confidence and engagement with complex programming tasks and essential skills such as code review and debugging. Future implementations of the activity could be further improved by adapting its design to better support students with different levels of prior programming experience.

Chapter 6

Conclusion

The growing popularity of AI-based tools, particularly large language models (LLMs), is transforming how students approach programming tasks in computer science (CS) courses. At the same time, the rapid expansion of CS programs has led to significantly larger class sizes. To manage this growth, many institutions have adopted automated grading systems to provide students with timely feedback on their work. While both AI tools and autograders offer potential benefits for supporting student learning, they also introduce new challenges for CS education.

One of the key challenges that students face is the need to comprehend and act on the feedback provided by autograders and AI tools. Whether it is understanding the feedback messages from autograders, identifying the errors in their code, and finding the fix for these errors, or critically evaluating and integrating AI-generated code, students need to be fully engaged and be aware of the systematic code review and debugging practices to efficiently perform these tasks.

Without proper guidance, these tasks can be daunting and cause additional stress and frustration, especially for beginners. Such an impact can have negative consequences for the development of essential skills among students in their future programming courses, where they face low self-efficacy with programming tasks, which can affect their learning outcomes

towards complex programming tasks.

Although educational technologies are advancing rapidly, explicit instruction on essential practices such as code review and debugging remains limited in many CS courses. As a result, students are often left to develop these skills through trial and error, which can increase cognitive load and reduce learning effectiveness. Overreliance on AI tools without a deep understanding of the code can also raise concerns about academic integrity and hinder the development of core programming competencies.

These challenges are not limited to the education sector. In the software industry, AI is also transforming how software is developed and how it can be integrated into the software development process to enhance the productivity of code development among developers. However, code review and debugging remain critical components of professional programming practice. Code review ensures consistency, quality, and knowledge sharing within teams, while debugging directly impacts software reliability and user satisfaction.

Given the changing nature of software development, this dissertation emphasizes the urgent need for teaching strategies that support the development of students' code review and debugging skills. It argues for structured guidance from instructors to help reduce the cognitive burden on students. This structured guidance is not just a teaching strategy, but a way to foster a sense of confidence and better prepare students for both academic achievement and the challenges of real-world software development.

6.1 Challenges Faced by Intermediate to Advanced Level Programmers

As Kernighan and Plauger noted in 1978, "Debugging is twice as hard as writing the program in the first place" [47]. This insight remains highly relevant today, even with the rise of AI-assisted programming tools. Debugging remains a complex task for both beginners and experienced programmers [80]. Although research has explored ways to integrate debugging

instruction into programming courses, it is often overlooked due to its complexity and time-intensive nature.

In chapter 3, a qualitative study explores how students develop debugging skills without formal instruction. It addresses the research question: *"What challenges do intermediate to advanced CS students face throughout their undergraduate program, and how do they navigate these difficulties?"* This study was conducted at the University of Oklahoma and focuses on the challenges faced by students enrolled in the undergraduate CS program at this institution. Learning a complex and mentally demanding skill like debugging without guidance can lead to stress and frustration. The study found that students often rely on ad hoc methods to write code and use ineffective strategies to debug it, sometimes even starting over from scratch. Many students turn to online forums and AI tools for help when they face obstacles.

Even at advanced levels, students rarely use a structured programming approach. Instead, they continue to apply habits formed during their early learning experiences. This is concerning because these early strategies may work for simple programs in introductory courses, but they become less effective as the complexity of assignments increases. As a result, students spend more time debugging and may struggle to progress.

The findings suggest that structured instruction in debugging should begin at the introductory level. This is a crucial point to reiterate, as it can help students develop effective strategies early on and avoid frustration in later stages of their CS program. The standard practices identified in this research can help CS educators understand which strategies students retain and how these affect their confidence in programming tasks. Since frustration with debugging persists across all levels of a CS program, early and consistent support is essential.

6.2 Code Insight

This dissertation investigates the research question: *"How can an in-class active learning activity be designed and implemented to effectively support the development of code reading and debugging skills among novice programmers?"*. To address this question, an instructional intervention called **Code Insight** was designed and implemented. This intervention aims to promote code review and debugging skills in beginner programmers by providing structured guidance that supports critical thinking and problem-solving skills, along with crucial programming skills such as code review and debugging, while minimizing additional workload for both students and instructors.

Chapter 4 presents the integration of the **Code Insight** activity into classroom lectures, outlining an implementation process that follows the introduction of each programming concept. The activity was first introduced in the Fall 2023 semester as a pilot, which provided initial feedback from small number of students on their learning experience and provided us with the details on the time required to conduct the activity in class and highlighted potential issues to address before full integration in subsequent semesters. The activity was implemented over two semesters, Spring and Fall 2024, and student experiences were evaluated through anonymous feedback collected after the final activity was implemented in class.

The findings indicate that students felt more confident when working on programming projects. Many students reported increased confidence in debugging tasks, particularly when applying what they learned during the in-class activities. Although students initially found it difficult to review and analyze code written by others, the structured guidance provided during the **Code Insight** sessions helped them feel more capable and engaged. The use of guided multiple-choice questions during lectures encouraged active participation and deeper analysis of code.

The **Code Insight** intervention addresses several gaps identified in the literature. These include low student engagement and motivation in code review activities, as well as the ten-

dency for novice programmers to produce low-quality reviews due to limited experience. By embedding the activity into regular lectures and offering a clear structure for analyzing code, the in-class activity helps students develop more effective review strategies. Additionally, by presenting multiple approaches to solving a problem, the activity encourages students to explore alternative solutions and improve their debugging skills.

Several students expressed a strong interest in having these in-class activities conducted more frequently throughout the semester, highlighting the perceived value and effectiveness of the **Code Insight** intervention in supporting their learning of the complex programming concepts.

6.3 Evaluate Code Insight Using Student Performance

The chapter 5 addresses the research question: *"How does the **Code Insight** intervention influence the code writing skills of novice programmers in an introductory programming course?"* To investigate this, student performance on midterm and final exams was analyzed and compared with their performance during the in-class **Code Insight** activities.

The **Code Insight** intervention was implemented in two sections of an introductory programming course within a CS program at University of Oklahoma. Both sections followed the same course structure and completed an identical final exam, ensuring consistency in evaluation.

The results showed a positive correlation between students' performance in the **Code Insight** activities and their ability to solve programming problems on the final exam. These findings align with patterns observed in student feedback, where many reported feeling more confident in writing and debugging code. This suggests that regular engagement with structured code review and debugging tasks during class may have contributed to the development of stronger programming skills.

Overall, the results highlight the potential of the **Code Insight** intervention to improve

learning outcomes by reinforcing key programming concepts through active, guided practice integrated into lecture sessions.

6.4 Future Work

This study presents several promising directions for future research. One important avenue is the implementation of longitudinal studies that track individual students over time to better understand the development of their debugging and code review skills within a CS program. Such studies could offer deeper insights into how these competencies evolve throughout the CS curriculum. Additionally, expanding the research across multiple institutions would help assess the generalizability of the findings and account for differences in instructional practices and student demographics.

Another valuable direction is to conduct a qualitative study aimed at identifying the essential deliverables that instructors provide in programming courses at various institutions. This could involve interviewing instructors to explore the challenges they face when integrating code review and debugging practices into their teaching.

Further enhancement of the **Code Insight** tool is also recommended, particularly in ways that support more meaningful peer discussions and collaborative learning. Finally, a comparative analysis of student performance in courses with and without **Code Insight** integration would provide valuable evidence regarding the tool's effectiveness in supporting learning outcomes.

Bibliography

[1] A. Frank Ackerman, Lynne S. Buchwald, and Frank H. Lewski. Software inspections: an effective verification process. *IEEE Software*, 6(3):31–36, 1989.

[2] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. An analysis of patterns of debugging among novice computer science students. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 84–88, 2005.

[3] Fernando Almeida. Framework for software code reviews and inspections in a classroom environment. *International Journal of Modern Education and Computer Science*, 10(10):31–39, 2018.

[4] Basma S Alqadi and Jonathan I Maletic. An empirical study of debugging patterns among novices programmers. In *Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education*, pages 15–20, 2017.

[5] B.S. Alqadi and J.I. Maletic. An empirical study of debugging patterns among novice programmers. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 15–20. ACM, March 2017.

[6] Elizaveta Artser, Daniil Karol, Anna Potriasaeva, Aleksey Rostovskiy, and Anastasiia Birillo. Ai debugging assistant: Enhancing debugging skills with intelligent guidance. In *Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 2*, pages 779–779. ACM, 2025.

[7] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE, 2013.

[8] Deepika Badampudi, Ricardo Britto, and Michael Unterkalmsteiner. Modern code reviews – preliminary results of a systematic mapping study. In *Proceedings of the 23rd International Conference on Evaluation and Assessment in Software Engineering (EASE '19)*, pages 1–6, Copenhagen, Denmark, 2019. ACM.

[9] Keerti Banweer and Deborah A Trytten. Wip: Code insight: Combining code reading and debugging practices for active learning in entry-level computer science courses. In *2024 IEEE Frontiers in Education Conference (FIE)*, pages 1–5. IEEE, 2024.

[10] Mordechai Ben-Ari. Situated learning in computer science education. *Computer Science Education*, 14(2):85–100, 2004.

[11] Senad Bećirović. Challenges and barriers for effective integration of technologies into teaching and learning. In *Digital Pedagogy*, pages 123–133. Springer, 2023.

[12] Jeffrey Bonar and Elliot Soloway. Preprogramming knowledge: A major source of misconceptions in novice programmers. In *Studying the novice programmer*, pages 325–353. Psychology Press, 2013.

[13] Amiangshu Bosu and Jeffrey C. Carver. Impact of peer code review on peer impression formation: A survey. In *Proceedings of the 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 133–142. IEEE, 2013.

[14] Amiangshu Bosu, Michaela Greiler, and Christian Bird. Characteristics of useful code reviews: An empirical study at microsoft. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR)*, pages 146–156. IEEE, 2015.

[15] Virginia Braun and Victoria Clarke. One size fits all? what counts as quality practice in (reflexive) thematic analysis? *Qualitative research in psychology*, 18(3):328–352, 2021.

[16] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 229:2013, 2013.

[17] David Byrne. A worked example of braun and clarke's approach to reflexive thematic analysis. *Quality & quantity*, 56(3):1391–1412, 2022.

[18] Antonella Carbonaro and Mirko Ravaioli. Peer assessment to promote deep learning and to reduce a gender gap in the traditional introductory programming course. *Journal of e-Learning and Knowledge Society*, 13(3), 2017.

[19] Robert N Charette. Why software fails [software failure]. *IEEE spectrum*, 42(9):42–49, 2005.

[20] Victoria Clarke and Virginia Braun. Thematic analysis. In *Encyclopedia of critical psychology*, pages 1947–1952. Springer, 2014.

[21] James P Cohoon. An introductory course format for promoting diversity and retention. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 395–399, 2007.

[22] Computing Research Association. Generation cs: Computer science undergraduate enrollments surge since 2006. Technical report, Computing Research Association, 2017. Accessed: 2025-06-26.

[23] Zheyuan Kevin Cui, Mert Demirer, Sonia Jaffe, Leon Musolff, Sida Peng, and Tobias Salz. The effects of generative ai on high skilled work: Evidence from three field experiments with software developers. *Available at SSRN 4945566*, 2024.

[24] Diana-Margarita Córdova-Esparza, Julio-Alejandro Romero-González, Karen-Edith Córdova-Esparza, Juan Terven, and Rocio-Edith López-Martínez. Active learning strategies in computer science education: A systematic review. *Multimodal Technologies and Interaction*, 8(6):50, 2024.

[25] Paul Denny, James Prather, and Brett A Becker. Error message readability and novice debugging performance. In *Proceedings of the 2020 ACM conference on innovation and technology in computer science education*, pages 480–486, New York, NY, USA, 2020. Association for Computing Machinery.

[26] Breanna Devore-McDonald and Emery D. Berger. Mossad: Defeating software plagiarism detection. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

[27] Kiran LN Eranki and Kannan M Moudgalya. Program slicing technique: a novel approach to improve programming skills in novice learners. In *Proceedings of the 17th Annual Conference on Information Technology Education*, pages 160–165, 2016.

[28] Sue Fitzgerald, 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(2):93–116, 2008.

[29] Paul Fleckney, James Thompson, and Paulo Vaz-Serra. Designing effective peer assessment processes in higher education: a systematic review. *Higher Education Research & Development*, 44(2):386–401, 2025.

[30] Edward F Gehringer. Building resources for teaching computer architecture through electronic peer review. In *Proceedings of the 2003 workshop on Computer architecture education: Held in conjunction with the 30th International Symposium on Computer Architecture*, pages 9–es, 2003.

[31] Edward F Gehringer, Donald D Chinn, Manuel A Pérez-Quiñones, and Mark A Ardis. Using peer review in teaching computing. In *Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 321–322, 2005.

[32] Jamie Gorson and Eleanor O'Rourke. Why do cs1 students think they're bad at programming? investigating self-efficacy and self-assessments at three universities. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, pages 170–181, New York, NY, USA, 2020. Association for Computing Machinery.

[33] John D. Gould and Paul Drongowski. An exploratory study of computer program debugging. *Human Factors*, 16(3):258–277, 1974.

[34] Harri Hämäläinen, Ville Hyyrynen, Jouni Ikonen, and Jari Porras. Applying peer-review for programming assignments. *Int. J. Inf. Technol. Secur*, 1:3–17, 2011.

[35] John Hamer, Quintin Cutts, Jana Jackova, Andrew Luxton-Reilly, Robert McCartney, Helen Purchase, Charles Riedesel, Mara Saeli, Kate Sanders, and Judithe Sheard. Contributing student pedagogy. *ACM SIGCSE Bulletin*, 40(4):194–212, 2008.

[36] Sakib Haque, Zachary Eberhart, Aakash Bansal, and Collin McMillan. Semantic similarity metrics for evaluating source code summarization. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 36–47, 2022.

[37] Robert L Heilman and Gordon P Ashby. Re-evaluation of debugging in the computer science curriculum. *ACM SIGCSE Bulletin*, 3(4):15–18, 1971.

[38] Bart Huisman, Nadira Saab, Paul Van Den Broek, and Jan Van Driel. The impact of formative peer feedback on higher education students' academic writing: a meta-analysis. *Assessment & Evaluation in Higher Education*, 44(6):863–880, 2019.

[39] Christopher Hundhausen, Anukrati Agrawal, Dana Fairbrother, and Michael Trevisan. Integrating pedagogical code reviews into a cs 1 course: an empirical study. *ACM SIGCSE Bulletin*, 41(1):291–295, 2009.

[40] Christopher Hundhausen, Anukrati Agrawal, and Kyle Ryan. The design of an online environment to support pedagogical code reviews. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 182–186, 2010.

[41] Christopher D Hundhausen, Pawan Agarwal, and Michael Trevisan. Online vs. face-to-face pedagogical code reviews: an empirical comparison. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 117–122, 2011.

[42] Christopher D. Hundhausen, Ali M. Brown, and Sarah A. Agrawal. Understanding the design of pedagogical code reviews: A synthesis of the literature. *ACM Transactions on Computing Education (TOCE)*, 13(3):1–28, 2013.

[43] Theresia Devi Indriasari, Andrew Luxton-Reilly, and Paul Denny. A review of peer code review in higher education. *ACM Transactions on Computing Education (TOCE)*, 20(3):1–25, 2020.

[44] David H Jonassen and Woei Hung. Learning to troubleshoot: A new theory-based design architecture. *Educational Psychology Review*, 18(1):77–114, 2006.

[45] Marian Jureczko, Łukasz Kajda, and Paweł Górecki. Code review effectiveness: an empirical study on selected factors influence. *IET Software*, 14(7):794–805, 2020.

[46] Irvin R Katz and John R Anderson. Debugging: An analysis of bug-location strategies. *Human-Computer Interaction*, 3(4):351–399, 1987.

[47] Brian W. Kernighan and P. J. Plauger. *The Elements of Programming Style*, volume 1. McGraw-Hill, New York, 1978.

[48] Claudius M Kessler and John R Anderson. A model of novice debugging in lisp. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 198–212, 1986.

[49] Hans Keuning, Johan Jeuring, and Bastiaan Heeren. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)*, 19(1):1–43, 2018.

[50] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)*, 19(1):1–43, 2019.

[51] Päivi Kinnunen and Lauri Malmi. Why students drop out cs1 course? In *Proceedings of the second international workshop on Computing education research*, pages 97–108, 2006.

[52] Päivi Kinnunen and Beth Simon. My program is ok–am i? computing freshmen's experiences of doing programming assignments. *Computer Science Education*, 22(1):1–28, 2012.

[53] Amy J Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, et al. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)*, 43(3):1–44, 2011.

[54] Amy J Ko, Thomas D LaToza, Stephen Hull, Ellen A Ko, William Kwok, Jane Quichocho, Harshitha Akkaraju, and Rishin Pandit. Teaching explicit programming strategies to adolescents. In *Proceedings of the 50th ACM technical symposium on computer science education*, pages 469–475, 2019.

[55] David R Krathwohl. A revision of bloom's taxonomy: An overview. *Theory into practice*, 41(4):212–218, 2002.

[56] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. *Acm sigcse bulletin*, 37(3):14–18, 2005.

[57] Chen Li, Emily Chan, Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. Towards a framework for teaching debugging. In *Proceedings of the Twenty-First Australasian Computing Education Conference*, pages 79–86, New York, NY, USA, 2019. Association for Computing Machinery.

[58] Xiaosong Li. Using peer review to assess coding standards-a case study. In *Proceedings. Frontiers in Education. 36th Annual Conference*, pages 9–14. IEEE, 2006.

[59] Xiaosong Li and Donald Joyce. Towards a framework for a code review process. *Journal of Applied Computing and Information Technology*, 12(1), 2008.

[60] Raymond Lister, Elizabeth S Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, et al. A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, 36(4):119–150, 2004.

[61] Raymond Lister, Colin Fidge, and Donna Teague. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *Acm sigcse bulletin*, 41(3):161–165, 2009.

[62] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the fourth international workshop on computing education research*, pages 101–112, 2008.

[63] Andrew Luxton-Reilly, Arthur Lewis, and Beryl Plimmer. Comparing sequential and parallel code review techniques for formative feedback. In *Proceedings of the 20th Australasian Computing Education Conference*, pages 45–52, 2018.

[64] Nestor Maslej, Loredana Fattorini, Raymond Perrault, Yolanda Gil, Vanessa Parli, Njenga Kariuki, Emily Capstick, Anka Reuel, Erik Brynjolfsson, John Etchemendy, et al. Artificial intelligence index report 2025. *arXiv preprint arXiv:2504.07139*, 2025.

[65] Renee McCauley, 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(2):67–92, 2008.

[66] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 125–180. 2001.

[67] C. Mcdowell, L. Werner, H.E. Bullock, and J. Fernald. The impact of pair programming on student performance, perception and persistence. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 602–607, 2003.

[68] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189, 2016.

[69] Tilman Michaeli and Ralf Romeike. Current status and perspectives of debugging in the k12 classroom: A qualitative study. In *2019 IEEE Global Engineering Education Conference (EDUCON)*, pages 1030–1038, 2019.

[70] Tilman Michaeli and Ralf Romeike. Improving debugging skills in the classroom: The effects of teaching a systematic debugging process. In *Proceedings of the 14th workshop in primary and secondary computing education*, pages 1–7, 2019.

[71] Larry K Michaelsen, Arletta Bauman Knight, and L Dee Fink. *Team-based learning: A transformative use of small groups in college teaching*. Taylor & Francis, 2023.

[72] Laurie Murphy, 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*, pages 163–167. ACM, 2008.

[73] R. Narayan, C. Rodriguez, J. Araujo, A. Shaqlaih, and G. Moss. Constructivism—constructivist learning theory. In B. J. Irby, G. Brown, R. Lara-Alecio, and S. Jackson, editors, *The Handbook of Educational Theories*, pages 169–183. IAP Information Age Publishing, 2013.

[74] Devon H O'Dell. The debugging mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills. *Queue*, 15(1):71–90, 2017.

[75] Noel O'Hara and Daire O'Broin. Incorporating game elements into programming practical classes to encourage collaboration and knowledge sharing. In *European Conference on Games Based Learning*, page 515. Academic Conferences International Limited, 2016.

[76] Ernesto Panadero and Maryam Alqassab. An empirical review of anonymity effects in peer assessment, peer feedback, peer review, peer evaluation and peer grading. *Assessment & Evaluation in Higher Education*, 2019.

[77] Karl Pearson. Vii. note on regression and inheritance in the case of two parents. *proceedings of the royal society of London*, 58(347-352):240–242, 1895.

[78] Yulia Pechorina, Keith Anderson, and Paul Denny. Metacodenition: Scaffolding the problem-solving process for novice programmers. In *Proceedings of the 25th Australasian Computing Education Conference*, pages 59–68, New York, NY, USA, 2023. Association for Computing Machinery.

[79] David N Perkins and Fay Martin. Fragile knowledge and neglected strategies in novice programmers. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 213–229, 1986.

[80] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal*, 25:83–110, 2017.

[81] Peter Pirolli. Effects of examples and their explanations in a lesson n recursion: A production system analysis. *Cognition and Instruction*, 8(3):207–259, 1991.

[82] Peter Pirolli and Margaret Recker. Learning strategies and transfer in the domain of programming. *Cognition and instruction*, 12(3):235–275, 1994.

[83] Strategic Planning. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, 1(2002), 2002.

[84] Yizhou Qian and James Lehman. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)*, 18(1):1–24, 2017.

[85] Peter C. Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 202–212, New York, NY, USA, 2013. ACM.

[86] Guoping Rong, Jingyi Li, Mingjuan Xie, and Tao Zheng. The effect of checklist in code review for inexperienced students: An empirical study. In *2012 IEEE 25th Conference on Software Engineering Education and Training*, pages 120–124, 2012.

[87] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: A case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 181–190. ACM, 2018.

[88] Arjun Singh, Sergey Karayev, Kevin Gutowski, and Pieter Abbeel. Gradescope: a fast, flexible, and fair system for scalable assessment of handwritten work. In *Proceedings of the fourth (2017) acm conference on learning@ scale*, pages 81–88, 2017.

[89] Harald Søndergaard and Raoul A Mulder. Collaborative learning through formative peer review: Pedagogy, programs and potential. *Computer Science Education*, 22(4):343–367, 2012.

[90] Xiangyu Song, Seth Copen Goldstein, and Majd Sakr. Using peer code review as an educational tool. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, pages 173–179, New York, NY, USA, 2020. Association for Computing Machinery.

[91] Yi Song, Xiaoyuan Xie, and Baowen Xu. When debugging encounters artificial intelligence: state of the art and open challenges. *Science China Information Sciences*, 67(2):141101, 2024.

[92] James G Spohrer and Elliot Soloway. Analyzing the high frequency bugs in novice programs. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 230–251, 1986.

[93] Tor Stalhane, Cat Kutay, Hiyam Al-Kilidar, and Ross Jeffery. Teaching the process of code review. In *2004 Australian Software Engineering Conference. Proceedings.*, pages 271–278. IEEE, 2004.

[94] Randy Stein and Susan E Brennan. Another person's eye gaze as a cue in solving programming problems. In *Proceedings of the 6th international conference on Multimodal interfaces*, pages 9–15, 2004.

[95] Leigh Ann Sudol-DeLyser, Mark Stehlik, and Sharon Carver. Code comprehension problems as learning events. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, pages 81–86, 2012.

[96] Chen Sun, Stephanie Yang, and Betsy Becker. Debugging in computational thinking: A meta-analysis on the effects of interventions on debugging skills. *Journal of Educational Computing Research*, 62(4):867–901, 2024.

[97] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hiroshi Iida. Review participation in modern code review: An empirical study of the android, qt, and openstack projects. *Empirical Software Engineering*, 22(2):768–817, 2016.

[98] Keith Topping. Peer assessment between students in colleges and universities. *Review of educational Research*, 68(3):249–276, 1998.

[99] Deborah A Trytten. A design for team peer code review. *ACM SIGCSE Bulletin*, 37(1):455–459, 2005.

[100] Scott Turner, Manuel A. Pérez-Quiñones, Stephen Edwards, and Joseph Chase. Student attitudes and motivation for peer review in cs2. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, page 347–352, New York, NY, USA, 2011. Association for Computing Machinery.

[101] Scott Turner, Manuel A Pérez-Quiñones, Stephen Edwards, and Joseph Chase. Peer review in cs2: conceptual learning. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 331–335, 2010.

[102] Scott A Turner, Ricardo Quintana-Castillo, Manuel A Pérez-Quiñones, and Stephen H Edwards. Misunderstandings about object-oriented design: Experiences using code reviews. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 97–101, 2008.

[103] Scott Alexander Turner, Manuel A Pérez-Quiñones, and Stephen H Edwards. Peer review in cs2: Conceptual learning and high-level thinking. *ACM Transactions on Computing Education (TOCE)*, 18(3):1–37, 2018.

[104] Iris Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459–494, 1985.

[105] Y. Wang, H. Li, Y. Feng, Y. Jiang, and Y. Liu. Assessment of programming language learning based on peer code review model: Implementation and experience report. *Computers & Education*, 59(2):412–422, 2012.

[106] Yanqing Wang, Yaowen Liang, Luning Liu, and Ying Liu. A multi-peer assessment platform for programming language learning: Considering group non-consensus and personal radicalness. *Interactive Learning Environments*, 24(8):2011–2031, 2016.

[107] Yanqing Wang, LI Yijun, Michael Collins, and Peijie Liu. Process improvement of peer code review and behavior analysis of its participants. *ACM SIGCSE Bulletin*, 40(1):107–111, 2008.

[108] Yuzhu Wei and Donghong Liu. Incorporating peer feedback in academic writing: A systematic review of benefits and challenges. *Frontiers in Psychology*, 15, 2024.

[109] Karen Weise and Cade Metz. At amazon's biggest data center, everything is supersized for a.i. *The New York Times*, May 2025. Accessed: 2025-06-26.

[110] Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. Novice reflections on debugging. In *Proceedings of the 52nd ACM technical symposium on computer science education*, pages 73–79, New York, NY, USA, 2021. Association for Computing Machinery.

[111] Andrew Williams. Delivering effective student feedback in higher education: An evaluation of the challenges and best practice. *International Journal of Research in Education and Science (IJRES)*, 10(2):473–501, 2024.

[112] Min Xie and Bo Yang. A study of the effect of imperfect debugging on software development cost. *IEEE Transactions on Software Engineering*, 29(5):471–473, 2003.

[113] Stephanie Yang, Miles Baird, Eleanor O'Rourke, Karen Brennan, and Bertrand Schneider. Decoding debugging instruction: A systematic literature review of debugging interventions. *ACM Transactions on Computing Education*, 24(4):1–44, 2025.

[114] Zezhou Yang, Cuiyun Gao, Zhaoqiang Guo, Zhenhao Li, Kui Liu, Xin Xia, and Yuming Zhou. A survey on modern code review: Progresses, challenges and opportunities. *arXiv preprint arXiv:2405.18216*, 2024.

[115] Nesra Yannier, Scott E Hudson, Kenneth R Koedinger, Kathy Hirsh-Pasek, Roberta Michnick Golinkoff, Yuko Munakata, Sabine Doebel, Daniel L Schwartz, Louis Deslauriers, Logan McCarty, et al. Active learning:"hands-on" meets "minds-on". *Science*, 374(6563):26–30, 2021.

[116] Wang Yanqing, Li Hang, Sun Yanan, Jiang Yu, and Yu Jie. Learning outcomes of programming language courses based on peer code review model. In *2011 6th International Conference on Computer Science & Education (ICCSE)*, pages 751–754. IEEE, 2011.

[117] Byung-do Yoon and Oscar N Garcia. Cognitive activities and support in debugging. In *Proceedings fourth annual symposium on human interaction with complex systems*, pages 160–169. IEEE, 1998.

[118] Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Morgan Kaufmann, 2009.

Appendix A

Publications

Chapter 3 is a full research paper with title ***"A Qualitative Study of How Computer Science Students Develop Debugging Skills in a Curriculum"*** and is currently accepted at an International IEEE conference, Frontiers in Education 2025.

Chapter 4 was originally published at an international conference of engineering education, Frontiers in Education 2024, with the following citation: *"Banweer, K., & Trytten, D. A. (2024, October). WIP: Code Insight: Combining Code Reading and Debugging Practices for Active Learning in Entry-Level Computer Science Courses. In 2024 IEEE Frontiers in Education Conference (FIE) (pp. 1-5). IEEE."* This paper includes the preliminary results from students' feedback on the Code Insight implementation during the Fall 2023 semester.

Chapter 4 also includes the results presented in the full research paper with the data analysis from Spring and Fall 2024 is accepted to present at an IEEE international conference, Frontiers in Education 2025, with the title ***"Code Insight: Evaluating an Active Learning Framework for Novice Programmers to Promote the Development of Code Review Skills"***.

Appendix B

Code Review Problems

B.1 Arithmetic Operators

This is the first Code Insight activity introduced in class after covering arithmetic operators. At this stage, the purpose of Code Insight activities is explained to students, along with how similar exercises will be integrated throughout the semester. Since the code is relatively small and students are still becoming familiar with the Eclipse IDE, both the code and the related question are displayed using a projector during class.

The problem is explained in detail so that students can analyze the code and compare the expected output with the actual output produced by the program.

B.1.1 Problem 1

```
1 public static void main(String[] args) {
2     //This program calculates the average with decimal points
3     int gradeOne = 89;
4     int gradeTwo = 92;
5     int gradeThree = 95;
6     //Calculate the average grades of three midterms by dividing the sum with 3,
          and stored the result into a variable
7     int averageOne = (gradeOne + gradeTwo + gradeThree) / 3;
8 }
```

Question: True or False: The code calculates the average of three numbers correctly

a) True

b) False

Answer: The correct answer is **B**, the calculation is truncating the decimal values because of the integer division in java.

B.1.2 Problem 2

```
1 public static void main(String[] args) {
2     //This program calculates the average with decimal points
3     int gradeOne = 89;
4     int gradeTwo = 92;
5     int gradeThree = 95;
6     //Calculate the average by dividing the sum with 3, and stored the result into a
          double variable
7     double averageTwo = (gradeOne + gradeTwo + gradeThree) / 3;
8 }
```

Question: Which one of the following is true about the given section of code?

a) averageTwo is calculated correctly

b) averageTwo is not calculated with decimal points

c) Code will give error because an int value cannot be assigned to a double identifier

Answer: The correct answer is **B**, the calculation is truncating the decimal values because of the integer division in java, even the final value is assigned to double variable.

B.1.3 Problem 3

```
1 public static void main(String[] args) {
2     //This program calculates the average with decimal points
3     int gradeOne = 82;
4     int gradeTwo = 92;
5     int gradeThree = 95;
6     //Calculate the average by dividing the sum with 3.0, and stored the result into
          a double variable
7     double averageThree = (gradeOne + gradeTwo + gradeThree) / 3.0;
8 }
```

Question: True or False: The code calculates the average of three numbers correctly

a) True

b) False

Answer: The correct answer is **A**. The calculation retains the decimal value because Java automatically promotes the numerator to a `double`, resulting in a `double` type output. This behavior aligns with the expected outcome of the program.

B.1.4 Problem 4

```
1 public static void main(String[] args) {
2     //This program calculates the average with decimal points
3     int gradeOne = 82;
4     int gradeTwo = 92;
5     int gradeThree = 95;
6     //Calculate the average by dividing the sum (using double casting) with 3, and
          stored the result into a double variable
7     double averageFour = (double)((gradeOne + gradeTwo + gradeThree) / 3);
8 }
```

Question: Which one of the following is true about the given section of code?

a) averageFour is calculated correctly with decimal points

b) The double cast is in the wrong place

c) gradeOne, gradeTwo and gradeThree should be declared as double

Answer: In this program, students should carefully examine the placement of parentheses. Although the division is explicitly cast to `double`, the use of parentheses causes integer division, which truncates the decimal portion and leads to an actual output that differs from the expected result. The correct answer is **B**.

B.2 User Interaction

This activity is conducted after the in-class discussion on the topic of User Interaction. Since the program is relatively short, both the code and the corresponding question are presented together on a single slide using a PowerPoint presentation. The code is intentionally written without any compilation errors to allow students to focus on analyzing and evaluating its logic and structure.

Before beginning the analysis, the problem statement is clearly explained to the students. They are then given sufficient time to review and understand the task in order to effectively assess the provided code. Following is the explanation of the problem, as presented in the class:

- The program will read data from console in the format below and print out the number of reams of paper that should be charged to that person. A ream is 50o pieces of paper and so number of reams should be rounded up.
- Input Format: <Number-of-pages>pages printed by <name-of-person>
- For example: When user enters "650 pages printed by Sonic the Hedgehog", this should be printed to the console: "Sonic the Hedgehog: 2 ream(s)"
- Read the code written and answer the following questions by selecting the best option.

B.2.1 Problem 1

```
1 public static void main(String[] args) {
2     Scanner keyboard = new Scanner(System.in);
3     int numberOfReams = 0;
4     final int PAGES_PER_REAM = 500;
5     int numberOfpages = keyboard.nextInt();
6     String personsName = keyboard.nextLine();
7     numOfReams = (int)Math.ceil((double)numberOfpages / PAGES_PER_REAM
          );
8     System.out.println(personsName + ": " + numOfReams + " ream(s)");
9     keyboard.close();
10 }
```

Question: Which one of the following is true about the given section of code?

a) Code will correctly print the expected output

b) double cast of numberOfPages is not needed

c) keyboard.nextLine() is not reading the person's name correctly

d) Math.ceil() should be Math.round()

Answer: This problem assesses students' understanding of key concepts such as explicit type casting and the use of methods from Java's `Scanner` class. The correct answer is **C** because the `nextLine` method reads the entire line of input, including the string `"pages printed by"` along with the person's name. This occurs because `nextLine` reads input until it encounters a newline or tab character.

B.2.2 Problem 2

```
1 public static void main(String[] args) {
2       Scanner keyboard = new Scanner(System.in);
3       final int PAGES_PER_REAM = 500;
4       int numOfpages = keyboard.nextInt();
5       keyboard.next();
6       String personsName = keyboard.nextLine();
7       int numOfReams = (int)Math.ceil((double)numOfpages /
            PAGES_PER_REAM);
8       System.out.println(personsName + ": " + numOfReams + " ream(s)");
9 }
```

Question: Which one of the following is true about the given section of code?

a) Code will correctly print the expected output

b) Person's name will be printed incorrectly

c) numberOf Reams is off by one

d) keyboard.next() result must be assigned to a variable

Answer: This code attempts to resolve the issue by using the `next` method from Java's `Scanner` class to read the word `"pages"`. However, the `next` method only reads a single word, stopping at a space, newline, or tab character. As a result, the actual output does not match the expected output, because the person's name in the output incorrectly includes the string `"printed by"`. The correct answer is option **B**, because person's name is incorrectly printed when the code is run.

B.2.3 Problem 3

```
1 public static void main(String[] args) {
2       Scanner keyboard = new Scanner(System.in);
3       final int PAGES_PER_REAM = 500;
4       int numOfpages = keyboard.nextInt();
5       keyboard.next(); keyboard.next(); keyboard.next();
6       String personsName = keyboard.nextLine();
7       int numOfReams = (int)Math.ceil(numOfpages / (double)
            PAGES_PER_REAM);
8       System.out.println(personsName + ": " + numOfReams + " ream(s)");
9 }
```

Question: Which one of the following is true about the given section of code?

a) Code will correctly print the expected output

b) double cast of PAGES_PER_REAM is not needed

c) int cast for the Math.ceil method is not needed

d) keyboard.next() should not be called 3 times

Answer: In this code, some statements use slightly different approaches. For example, the denominator in a division operation is explicitly cast to the `double` type. A common misconception among students is that type casting changes the value of a constant variable, which is not the case.

The correct answer is **A**. In this option, the `next` method from Java's `Scanner` class is used to skip the words `"pages printed by"`. Then, the `nextLine` method correctly reads the person's name from the remaining input. In this case, the expected output matches the actual output of the program.

B.3 Conditional Structures

This activity is conducted after completing the topic of conditional statements. The question used for this activity is taken from previous semester examinations. The problem is first explained to the students to ensure they understand the context and what are expected output of the program.

At this stage in the course, the length of the code increases, and students are already familiar with the Eclipse IDE, which is used for writing Java programs. Therefore, students are instructed to download the code from Canvas and import it into Eclipse for review and analysis. The programs are well-documented to support students in understanding the code in depth.

Any necessary assumptions are clearly explained to simplify the problem and help students focus on the programming concepts covered during the semester.

- Airlines require that suitcases be of limited size and weight to go on planes. No dimension (width, height, and depth) can be more than 36 inches. In addition, luggage must weigh less than 75 pounds.
- The program will read-in the data from the user and print out if the suitcase is allowed to go on a plane or not on console.
- Assuming the code will read width, height, depth, and weight (in the same order followed by spaces) from the user and print single message whether the luggage is allowed to go on a plane or not.
- Assuming, only one luggage is allowed to check-in and user will input valid data, that the user will follow expected format to input the data
- Read the code for each question and answer the best possible choice

B.3.1 Problem 1

```
1 import java.util.Scanner;
2 /**
3  * Generating the boarding pass.
4  * Airlines require that suitcases be of limited size and weight to go on planes.
5  * No dimension (width, height, depth) can be more than 36 inches.
6  * In addition, the luggage must weigh less than 75 pounds.
7  * The Program reads width, height, depth, and weight (in the same order followed by
       spaces)
8  * from user and print single message whether the luggage is allowed to go on a
       plane or not.
9  * Assuming, only one luggage is allowed to check-in and user will input valid data
10 * Question by: Dr. Deborah Trytten
11 * @author Keerti Banweer
12 */
13 public class GeneratePassOne {
14     public static void main(String[] args) {
15
16         // Declare Scanner to read the input from user
17         Scanner keyboard = new Scanner(System.in);
18
19         final int MAX_DIMENSION = 36;
20         final int MAX_WEIGHT = 75;
21         //Reading the luggage dimensions and weight for checkin process
22         System.out.println("Enter the width, height, depth, and weight separated by
               spaces");
23
```

```
24          double width = keyboard.nextDouble();
25          double height = keyboard.nextDouble();
26          double depth = keyboard.nextDouble();
27          double weight = keyboard.nextDouble();
28          keyboard.nextLine();
29
30          if (width <= MAX_DIMENSION) {
31              System.out.println("Luggage is checked in successfully !");
32          }
33          else if (height <= MAX_DIMENSION) {
34              System.out.println("Luggage is checked in successfully !");
35          }
36          else if (depth <= MAX_DIMENSION) {
37              System.out.println("Luggage is checked in successfully !");
38          }
39          else if (weight <= MAX_WEIGHT) {
40              System.out.println("Luggage is checked in successfully !");
41          }
42          else {
43              System.out.println("Not valid for checkin");
44          }
45          //Closing the Scanner object
46          keyboard.close();
47      }
48  }
```

Question: Which one of the following is true about the given section of code (use GeneratePassOne.java)?

a) Code will run correctly to print the expected output for all valid input data

b) The last else statement (42 to 44) should be removed

c) The program will show multiple responses sometimes

d) In the lines 32 to 46, the else-if statements should be if statements

e) In some cases when the weight is more than 75, the luggage will still be checked in

Answer: This code evaluates students' understanding of conditional statements. Concepts such as cascading and nested conditionals can often lead to misconceptions among beginner programmers. The multiple-choice options are designed to address and clarify these misunderstandings.

The correct answer is **E**, because the conditional statement produces the expected output in some cases, but in other cases, it may result in an output that differs from what is expected. This highlights the importance of carefully analyzing the logic and structure of conditional statements.

B.3.2 Problem 2

```
1 import java.util.Scanner;
2 /** Generating the boarding pass.
3  * Airlines require that suitcases be of limited size and weight to go on planes.
4  * No dimension (width, height, depth) can be more than 36 inches.
5  * In addition, the luggage must weigh less than 75 pounds.
6  * The Program reads width, height, depth, and weight (in the same order followed by
       spaces)
7  * from user and print single message whether the luggage is allowed to go on a
       plane or not.
8  * Assuming, only one luggage is allowed to check-in and user will input valid data
9  * Question by: Dr. Deborah Trytten and @author Keerti Banweer
10 */
11 public class GeneratePassTwo {
12     public static void main(String[] args) {
13         // Declare Scanner to read the input from user
14         Scanner keyboard = new Scanner(System.in);
15
16         final int MAX_DIMENSION = 36;
17         final int MAX_WEIGHT = 75;
18         //Reading the luggage dimensions and weight for check-in process
19         System.out.println("Enter the width, height, depth, and weight separated by
               spaces");
20
21         double width = keyboard.nextDouble();
22         double height = keyboard.nextDouble();
23         double depth = keyboard.nextDouble();
```

```
24          double weight = keyboard.nextDouble();
25          keyboard.nextLine();
26
27          if (width <= MAX_DIMENSION) {
28              if (height <= MAX_DIMENSION) {
29                  if (depth <= MAX_DIMENSION) {
30                      if (weight <= MAX_WEIGHT) {
31                          System.out.println("Luggage is checked in successfully !");
32                      }
33                  }
34              }
35          }
36          else {
37              System.out.println("Not valid for checkin");
38          }
39          //Closing the Scanner object
40          keyboard.close();
41      }
42  }
```

Question: Which one of the following is true about the given section of code (use GeneratePassTwo.java)?

a) Code will run correctly to print the expected output for all valid input data

b) Sometimes the code will not show a response

c) Lines 36 and 38 should be removed

d) For lines 27 to 30, the <= should be >

e) Each of the cascading if statements on lines 27 to 30, should be a separate if statement containing line 31

Answer: The correct answer is **B**, because when the height exceeds the maximum allowed value, the program does not produce any output. This indicates that the condition is not handled explicitly, resulting in no response being printed.

B.4 Logical Operators

This activity is conducted after students have learned about logical operators in class. The procedure follows the same steps as the previous topic: students download the provided code and import it into Eclipse to closely examine and evaluate the program. All necessary assumptions are clearly explained to the students beforehand. The following problem is explained to the students and questions are asked on the two different approaches:

- The program will printout either "We're Online" or "We're not online" in response to the current weather conditions. OU goes online for three possible reasons:
  - There is expected to be more than 0.10 inches of ice
  - There is expected to be more than 4 inches of snow when the temperature is below 10 degrees
  - The temperature is expected to be below 5 degrees when the wind is more than 20 miles per hour
- User will input the data for snow, ice, temperature, and wind separated by spaces.
- If anyone of the three conditions is true, OU will be online.
- Assuming user will input valid data
- Review the given code and answer the best possible answer choice.

B.4.1 Problem 1

```
1 import java.util.Scanner;
2
3 /*
4  * The program will printout either  "We're Online" or "We're not online" in
       response to the current weather conditions. OU goes online for three possible
       reasons:
5  * There is expected to be more than 0.10 inches of ice
6  * There is expected to be more than 4 inches of snow when the temperature is below
        10 degrees
7  * The temperature is expected to be below 5 degrees when the wind is more than 20
       miles per hour
8  * User will input the data for snow, ice, temperature, and wind separated by
       spaces.
9  * If anyone of the three conditions is true, OU will be online.
10 *
11 */
12
13 public class OUOnlineOrNot1 {
14     public static void main(String[] args) {
15
16         //Declare and construct Scanner
17         Scanner input = new Scanner(System.in);
18
19         double snow = 0.0;
20         double ice = 0.0;
21         int temp = 0;
```

```
22          int wind = 0;
23
24          System.out.println("Enter the weather data: ");
25          snow = input.nextDouble();
26          ice  = input.nextDouble();
27          temp = input.nextInt();
28          wind = input.nextInt();
29          input.nextLine();
30
31          if (temp < 5 || wind > 20)
32          {
33              System.out.println("We're Online");
34          } else if (snow > 4.0 || temp < 10)
35          {
36              System.out.println("We're Online");
37          } else if (ice >= 0.10)
38          {
39              System.out.println("We're Online");
40          } else
41          {
42              System.out.println("We're not Online");
43          }
44          input.close();
45      }
46  }
```

Question: Which one of the following is true about the given section of code (use OUOnlineOrNot1.java)?

a) Code will correctly print the expected output

b) "We're Online" will be printed when it shouldn't be

c) One || operator should be an && operator

d) Two || operators should be two && operators

e) The else-if statements should be if-statements

Answer: This problem has multiple correct answers due to the incorrect use of the logical **"OR"** operator. Option **B** is correct in identifying the issue, while option **D** provides the correct solution to fix the code. The goal of this exercise is to help students understand logical operators and how to apply them correctly in a programming context. Therefore, if a student selects either of the two correct options, they will receive full credit. Responses are collected using an in-class response system, which allows students to submit only one answer per question.

B.4.2 Problem 2

```
1 import java.util.Scanner;
2
3 /*
4  * The program will printout either "We're Online" or "We're not online" in response
       to the current weather conditions. OU goes online for three possible reasons:
5  * There is expected to be more than 0.10 inches of ice
6  * There is expected to be more than 4 inches of snow when the temperature is below
       10 degrees
7  * The temperature is expected to be below 5 degrees when the wind is more than 20
       miles per hour
8  * User will input the data for snow, ice, temperature, and wind separated by
       spaces.
9  * If anyone of the three conditions is true, OU will be online.
10 *
11 */
12
13 public class OUOnlineOrNot2 {
14     public static void main(String[] args) {
15
16         //Declare and construct Scanner
17         Scanner input = new Scanner(System.in);
18
19         double snow = 0.0;
20         double ice = 0.0;
21         int temp = 0;
22         int wind = 0;
```

```
23
24          System.out.println("Enter the weather data: ");
25          snow = input.nextDouble();
26          ice  = input.nextDouble();
27          temp = input.nextInt();
28          wind = input.nextInt();
29          input.nextLine();
30
31          if  ((temp < 5 && wind > 20)
32               ||  (ice  > 0.10)
33               ||  (snow > 4 && temp < 10))
34          {
35               System.out.println("We're Online");
36          }
37          else
38          {
39               System.out.println("We're not Online");
40          }
41          input.close();
42      }
43 }
```

Question: Which one of the following is true about the given section of code (use OUOnlineOrNot2.java)?

a) Code will correctly print the expected output

b) The || operators should be &&

c) The && operators should be ||

d) From Line 30 to 32, the order of the logical expressions is not correct

Answer: Option **A** is correct answer in this case, as the conditional statement are written correctly with proper use of logical operators.

B.5 While Loops

This activity is implemented after the while loops are covered in class. The following problem is explained to the students and one sample run is shown by running the code on Eclispe, to display the expected input and output for the problem:

- The program reads a list of words separated by spaces from the console, ending with a sentinel QUIT.
- The program will print out the number of times a given target value was read in, ignoring case.
- If User enters "Fiona Shrek Donkey SHrek fiona QUIT" and the target is "shrek", the program will return 2
- The assumptions for valid data
  - Each word must be separated by spaces
  - Each list of words must end with the word QUIT (in Uppercase)

Sample run: The values in bold are the user input on console. This code is run for the students to explain the expected input and output for the problem and that there are no compilation errors in the two different approaches used to write the programs.

Enter the word for which you want to calculate the frequency.

shrek

Enter the list of words ending with QUIT to exit

Fiona Shrek Donkey SHrek fiona QUIT

The word shrek appeared 2 times

B.5.1 Problem 1

```
1 import java.util.Scanner;
2 /**
3  * Find frequency of a given word in user input
4  *
5  * Description: Below program reads words from the console, ending with a sentinel
       QUIT.
6  * The program should print out the number of times a given target value was read in,
       ignoring case.
7  * @author Dr. Deborah Trytten
8  * Updated by: Keerti Banweer
9  */
10 public class SentinelControlledLoop1 {
11     public static void main(String[] args) {
12
13         //Declare and construct a Scanner object
14         Scanner input = new Scanner(System.in);
15         final String SENTINEL = "QUIT";
16
17         System.out.println("Enter the word for which you want to calculate the
               frequency.");
18         String target = input.nextLine();
19
20         System.out.println("Enter the list of words ending with QUIT to exit");
21         String word = input.next(); //priming read
22
23         int count = 0; // to calculate the frequency of the target
```

```
24
25          while (!word.equals(SENTINEL)) {
26
27              word = input.next();//reading the next word
28
29              if  (word.equalsIgnoreCase(target))
30              {
31                  count = count + 1;
32              }
33
34          }
35
36          System.out.println("The word " + target + " appeared " + count + " times")
                ;
37          //Closing the Scanner
38          input.close();
39      }
40  }
```

Question: Use file SentinelControlledLoop1.java (File name on Canvas)

In the given code, which one of the following statements is true?

a) The program runs correctly as its written for all valid inputs

b) The frequency count calculated will always be off by one

c) The frequency count calculated will sometimes be off by one

d) In line 27, input should be read using nextLine()

e) If the target is quit, the answer will be correct

Answer: In this program, students are expected to test the output using different valid inputs. In some cases, the program may return the correct frequency for the entered target. The correct answer is **C**, as the frequency calculation is sometimes off by one due to the loop skipping the first input value in the list. Option **E** is included to highlight the use of a sentinel value, which is not part of the actual data and should not be counted as one. Therefore, if the target value is entered as `quit`, the output should always be 0.

B.5.2 Problem 2

```
1 import java.util.Scanner;
2 /**
3  * Find frequency of a given word in user input
4  * Description: Below program reads a line of words from the console, ending with a
       sentinel QUIT.
5  * The program should print out the number of times a given target value was read in,
       ignoring case.
6  * @author Dr. Deborah Trytten
7  * Updated by: Keerti Banweer
8  *
9  */
10 public class SentinelControlledLoop2 {
11     public static void main(String[] args) {
12
13         //Declare and construct a Scanner object
14         Scanner input = new Scanner(System.in);
15         final String SENTINEL = "QUIT";
16
17         System.out.println("Enter the word for which you want to calculate the
               frequency.");
18         String target = input.nextLine();
19
20         System.out.println("Enter the list of words ending with QUIT to exit");
21         String word = input.next(); //priming read
22
23         int count = 0; // to calculate the frequency of the target
```

```
24
25          while (!word.equals(SENTINEL)) {
26              word = word.toLowerCase();
27              target  = target.toLowerCase();
28
29              if  (word.equals(target))
30              {
31                  count = count + 1;
32              }
33              word = input.next(); //reading the next word
34          }
35
36          System.out.println("The word " + target + " appeared " + count + " times")
                ;
37          //Closing the Scanner
38          input.close();
39      }
40  }
```

Question: Use file SentinelControlledLoop2.java

In the given code, which one of the following statements is true?

a) The program runs correctly as its written for all valid inputs

b) The frequency count calculated will always be off by one

c) The frequency count calculated will sometimes be off by one

d) If the target is quit, the answer will be incorrect

e) Line 26 and 27, should be moved after line 33, as the last two statements of the while loop

Answer: In this question, the loop is rewritten to address the issue from the previous version and demonstrates the use of various methods from the `String` class. The correct answer is **A**, as the code executes properly, does not skip any inputs, and correctly terminates when the user enters the sentinel value.

B.6 For Loops

This Code Insight activity is introduced after the topic of `for` loops is covered in class. The following question is explained to students, accompanied by a sample run that displays the expected input and output of the program. This helps students understand the program's behavior and prepares them to analyze the code effectively.

- What the program does:
  - It allows you to pick a range of dates in a single month for exercise challenge and
  - Earn money from your friends who have pledged a donation using the number of steps you take beyond 5000 steps each day.
- For example:
  - If you took 6000 steps one day, your friends would pay for 1000 of them.
  - If you only took 3000 steps, they would not pay for any since the 5000 steps base was not met.
- The assumptions for valid data
  - User will enter the data as expected.
  - The days will be between 1 and 31
  - The start day will always be smaller than the end day
  - The number of steps will be greater than or equal to 0

B.6.1 Problem 1

```
1 import java.util.Scanner;
2
3 /**
4  * Question: Does the code run correctly for all the inputs?
5  *
6  * Assuming user will enter valid data
7  * Write a complete program that keeps track of the amount of money you earned for
       WildCare Wildlife Rehabilitation
8  * by participating in an exercise challenge.
9  * Wildcare takes in wild animals that are injured or ill, provides medical care,
       houses the animals until they are
10 * ready to return to the wild, and releases them. This challenge allows you to pick
       of range of dates in a single
11 * month and earn money from your friends who have pledged to provide a donation
       based on the number of
12 * steps you take beyond 5000 steps each day. For example, if you took 6000 steps
       one day, your friends
13 * would pay for 1000 of them. If you only took 3000 steps, they would not pay for
       any since the 5000
14 * step base was not met.
15 * Problem by Dr. Deborah A. Trytten from Midterm 1 Spring 2023
16 *
17 * @author Keerti Banweer
18 *
19 */
20 public class RaisingMoney1 {
```

```
21
22      public static void main(String[] args) {
23
24          //Declare and construct Scanner
25          Scanner input = new Scanner(System.in);
26
27          //Declare variables to store the start day and end day
28          int startDay = 0;
29          int endDay = 0;
30
31          //Declare variables to store the number of steps in a day
32          int stepsPerDay = 0;
33          int stepsOverLimit = 0;
34
35          //Declaring a threshold as a constant
36          final int STEPS_THRESHOLD = 5000;
37
38          //Assuming user will only enter the days between 1 and 31;
39          // and start day number will always be less than end day number
40          System.out.println("What day did you start?");
41          startDay = input.nextInt();
42
43          System.out.println("What day did you end?");
44          endDay = input.nextInt();
45
46          //Calculating number of days
47          int numberOfDays = (endDay - startDay) + 1;
```

```
48
49          //Asking user to enter the pledge amount for the number of steps over the
                threshold
50          System.out.println("How much have people pledged per step beyond 5000
                steps per day?");
51          double amountPledged = input.nextDouble();
52
53          //iterate through each day and calculate the total steps over the limit
54          for (int i = 0; i < numberOfDays; ++i) {
55
56              // This condition will check for extra steps and add them to steps over
                    limit calculation
57              if (stepsPerDay >= STEPS_THRESHOLD) {
58                  stepsOverLimit = stepsOverLimit + (stepsPerDay -
                        STEPS_THRESHOLD);
59              }
60
61              System.out.println("Enter your steps from day " + (i + startDay));
62              stepsPerDay = input.nextInt();
63          }
64
65          //printing the raised amount
66          System.out.println("You raised $" + amountPledged * stepsOverLimit + " for
                charity.");
67          input.close();
68      }
69  }
```

Question: Use file RaisingMoney1.java. In the given code, which one of the following statements is true, assuming the input is valid?

a) The program always runs correctly

b) The program sometimes run incorrectly

Answer: In this code, students are expected to analyze and evaluate both the loop and the conditional statement. The program skips input for one day, which can cause it to produce incorrect results where the actual output differs from the expected output. The correct answer is **B**.

B.6.2 Problem 2

```
1 import java.util.Scanner;
2
3 /**
4  * Question: Does the code run correctly for all the inputs?
5  *
6  * Assuming user will enter valid data
7  * Write a complete program that keeps track of the amount of money you earned for
       WildCare Wildlife Rehabilitation
8  * by participating in an exercise challenge.
9  * Wildcare takes in wild animals that are injured or ill, provides medical care,
       houses the animals until they are
10 * ready to return to the wild, and releases them. This challenge allows you to pick
       of range of dates in a single
11 * month and earn money from your friends who have pledged to provide a donation
       based on the number of
12 * steps you take beyond 5000 steps each day. For example, if you took 6000 steps
       one day, your friends
13 * would pay for 1000 of them. If you only took 3000 steps, they would not pay for
       any since the 5000
14 * step base was not met.
15 * Problem by Dr. Deborah A. Trytten from Midterm 1 Spring 2023
16 *
17 * @author Keerti Banweer
18 *
19 */
20 public class RaisingMoney2 {
```

```
21
22      public static void main(String[] args) {
23
24          //Declare and construct Scanner
25          Scanner input = new Scanner(System.in);
26
27          //Declare variables to store the start day and end day
28          int startDay = 0;
29          int endDay = 0;
30
31          //Declare variables to store the number of steps in a day
32          int stepsOverLimit = 0;
33
34          //Declaring a threshold as a constant
35          final int STEPS_THRESHOLD = 5000;
36
37          //Assuming user will only enter the days between 1 and 31;
38          // and start day number will always be less than end day number
39          System.out.println("What day did you start?");
40          startDay = input.nextInt();
41
42          System.out.println("What day did you end?");
43          endDay = input.nextInt();
44
45          //Asking user to enter the pledge amount for the number of steps over the
                threshold
46          System.out.println("How much have people pledged per step beyond 5000
```

```
            steps per day?");
47          double amountPledged = input.nextDouble();
48
49          // Call the totalStepsOverlimit method
50          stepsOverLimit = totalStepsOverlimit(startDay, endDay,
                STEPS_THRESHOLD, stepsOverLimit, input);
51
52          //printing the raised amount
53          System.out.println("You raised $" + amountPledged * stepsOverLimit + " for
                charity.");
54
55          input.close();
56      }
57
58      public static int totalStepsOverlimit (int start, int end, int threshold, int
            stepsOverLimit, Scanner keyboard) {
59
60          //declare local variables
61          int stepsPerDay = 0;
62
63          //iterate through each day and calculate the total steps over the limit
64          for (int i = start; i <= end; ++i) {
65              System.out.println("Enter your steps from day " + i);
66              stepsPerDay = keyboard.nextInt();
67
68              if (stepsPerDay > threshold) {
69                  stepsOverLimit = stepsOverLimit + (stepsPerDay - threshold);
```

```
70                  }
71
72              }
73              return stepsOverLimit;
74          }
75  }
```

Question: Use file RaisingMoney2.java. In the given code, which one of the following statements is true?

a) The program always runs correctly

b) The program sometimes run incorrectly

Answer: The correct answer is **A**, here the code will run correctly for different valid inputs.

B.7 Arrays of Primitive Data Types

This Code Insight activity is introduced after the topic of arrays is covered in class. The problem is explained to students, along with one or two sample runs to demonstrate that the program compiles without errors. It is clarified that array values can vary, but to simplify the code for instructional purposes, the values are hard-coded in the written program and students must evaluate the code with different possible data of the array variable.

- Review the given code and answer the question after review
- This program will find out if the given array is in sorted order or not
  - If the array is sorted, program will print the message: "The array is sorted",
  - Otherwise, the program will print "The array is not sorted"

B.7.1 Problem 1

```
1 import java.util.Arrays;
2 /*
3  * This program will find out if the given array is in sorted order or not
4  * if the array is sorted, program will print the message: "The array is sorted",
5  * otherwise, the program will print "The array is not sorted"
6  */
7 public class IsSortedOne {
8     public static void main(String[] args) {
9         //declare and construct an array of integers
10        int[] array = {1, 3, 5, 4, 7, 2}; //reading input from a file
11        boolean flag = true;
12        int index = 0;
13
14        //This loop will iterate through the elements of the array
15        for (index = 0; index < array.length - 1; ++index)
16        {
17            if (array[index] > array[index+1])
18            {
19                flag = false;
20            }
21            else
22            {
23                flag = true;
24            }
25        }
26
```

```
27            if (flag) {
28                System.out.println("The array is sorted");
29            }
30            else {
31                System.out.println("The array is not sorted");
32            }
33        }
34 }
```

Question: Use file IsSortedOne.java. In the given code, which one of the following statements is true, assuming the input is valid?

a) The program always runs correctly

b) The program says sorted data is unsorted

c) The program says unsorted data is sorted

d) In line 15, the for loop is off by one iteration

Answer: The correct option is **C**. In this program, the output incorrectly indicates that the array is sorted even when it is not. This issue arises due to the presence of the `else` statement within the conditional logic, which causes the program to produce an incorrect result under certain conditions.

B.7.2 Problem 2

```
1 /*
2  * This program will find out if the given array is in sorted order or not
3  * if the array is sorted, program will print the message: "The array is sorted",
4  * otherwise, the program will print "The array is not sorted"
5  */
6 public class IsSortedTwo {
7     public static void main(String[] args) {
8         //declare and construct an array of integers
9         int [] array = {3, 5, 7, 9}; //reading input from a file
10
11        int index = 0;
12
13        //This loop will iterate through the elements of the array
14        for (index = array.length - 2; index >= 1 && array[index] > array[index -
              1]; --index);
15
16        if (index > 0) {
17            System.out.println("The array is not sorted");
18        }
19        else {
20            System.out.println("The array is sorted");
21        }
22    }
23 }
```

Question: Use file IsSortedTwo.java. In the given code, which one of the following statements is true, assuming the input is valid?

a) The program always runs correctly

b) The program says sorted data is unsorted

c) The program says unsorted data is sorted

d) In line 14, the for loop is off by one iteration

Answer: The correct answers are **C** and **D**. In this program, the `for` loop is off by one, which means that if only the last element of the array is unsorted, the program will incorrectly print that the array is sorted. This behavior leads to an actual output that differs from the expected result.

B.8 Perfect Size and Oversize Arrays

This topic extends the discussion on arrays previously covered in class and also introduces the concept of methods. The problem is presented to students with a detailed explanation. Since methods can be challenging for beginner programmers, this activity is designed to help students understand how methods are used within a program. For this problem, the array data is entered by the user and stored in a oversize array. The program will then calculate if the entered array is in sorted order or not.

- Review the given code and answer the question after review
- This program will find out if the given array is in sorted order or not
  - If the array is sorted, program will print the message: "The array is sorted",
  - Otherwise, the program will print "The array is not sorted"

B.8.1 Problem 1

```
1 import java.util.Arrays;
2 import java.util.Scanner;
3 /*
4  * This program will find out if the given array is in sorted order or not
5  * if the array is sorted, program will print the message: "The array is sorted",
6  * otherwise, the program will print "The array is not sorted"
7  */
8 public class IsSortedOversize1 {
9
10     public static void main(String[] args) {
11
12         Scanner keyboard = new Scanner(System.in);
13         //Max size for oversize
14         final int MAX_LENGTH = 100;
15
16         //declare and construct an oversize array
17         int[] array = new int[MAX_LENGTH];
18         Arrays.fill(array, MAX_LENGTH);
19
20         //Prompt user for input
21         System.out.println("Enter the list of numbers");
22         String source = keyboard.nextLine();
23
24         //read data from user input and store it into an oversize array
25         int arraySize = readNumbers(source, array);
26
```

```
27          int index = findFirstUnsorted(array, arraySize);
28
29          if (index >= 0) {
30              System.out.println("The array is not sorted");
31          } else {
32              System.out.println("The array is sorted");
33          }
34
35          keyboard.close();
36      }
37
38      /**
39       * This method returns the index of the first element in the array
40       * that is not sorted. If the array is sorted return -1
41       * @param array is an Oversize array
42       * @param arraySize is the size of the array that represent
43       * the number of elements in the array
44       * @return the index of the first element in the array that is not sorted.
45       * If the array is sorted return -1
46       */
47      public static int findFirstUnsorted(int[] array, int arraySize) {
48          int index = -1;
49
50          for (int i = 0; i < array.length - 1; ++i) {
51              if(array[i] > array[i + 1])
52              {
53                  index = i;
```

```
54                  return index;
55              }
56          }
57          return index;
58      }
59
60
61      /**
62       * This method will read the user input and store it in an oversize array
63       * @param input from the user
64       * @param the oversize array
65       * @return the size of the array that represent the number of elements
66       * in the array
67       */
68      public static int readNumbers(String source, int[] array) {
69          Scanner input = new Scanner(source);
70
71          int size = 0;
72          while(input.hasNextInt()) {
73              array[size] = input.nextInt();
74              size++;
75          }
76
77          input.close();
78          return size;
79      }
80  }
```

Question: Use file IsSortedOversize1.java. In the given code, which one of the following statements is true, assuming the input is valid?

a) The program always runs correctly

b) The program always says that the data is sorted

c) The program always says that the data is unsorted

d) The program sometimes says unsorted when the array data is in sorted order

e) The program sometimes says sorted when the array data is in unsorted order

Answer: The correct answer is **D**. In this program, the array is an oversize array, meaning it contains additional elements that are not part of the actual user input. The loop iterates through the entire array, including these unused elements, which can lead to incorrect results.

For example, if the user inputs: `105 109 110 500`, the program will output "The array is not sorted" because the remaining elements in the array are initialized to `100`. The loop compares `500` (the last user input) with `100` (a default value), resulting in an incorrect conclusion.

This activity helps students understand the importance of using the correct range when iterating through arrays, especially when working with oversize arrays.

B.8.2 Problem 2

```
1 import java.util.Arrays;
2 import java.util.Scanner;
3 /*
4  * This program will find out if the given array is in sorted order or not
5  * if the array is sorted, program will print the message: "The array is sorted",
6  * otherwise, the program will print "The array is not sorted"
7  */
8 public class IsSortedOversize2 {
9
10     public static void main(String[] args) {
11
12         Scanner keyboard = new Scanner(System.in);
13         final int MAX_LENGTH = 100; //Max size for oversize
14         final String prompt = "Enter the list of numbers"; //user prompt
15
16         //declare and construct an oversize array
17         int[] array = new int[MAX_LENGTH];
18         //total elements stored in the array
19         int arraySize = 0;
20
21         //read data from user input
22         readNumbers(keyboard, prompt, array, arraySize);
23
24         int index = findFirstUnsorted(array, arraySize);
25
26         if (index >= 0) {
```

```
27              System.out.println("The array is not sorted");
28          } else {
29              System.out.println("The array is sorted");
30          }
31
32          keyboard.close();
33      }
34
35      /**
36       * This method returns the index of the first element in
37       * the array that is not sorted. If the array is sorted return -1
38       * @param array is an Oversize array
39       * @param arraySize is the size of the array that represent
40       * the number of elements in the array
41       * @return the index of the first element in the array that is not sorted.
42       * If the array is sorted return -1
43       */
44      public static int findFirstUnsorted(int[] array, int arraySize) {
45          int index = -1;
46
47          for (int i = 1; i < arraySize; ++i) {
48              if(array[i - 1] > array[i])
49              {
50                  index = i - 1;
51                  return index;
52              }
53          }
```

```
54          return index;
55      }
56
57
58      /**
59       * This method will read the user input and store it in an oversize array
60       * @param input from the user
61       * @param the oversize array
62       * @return the size of the array that represent the number of elements
63       * in the array
64       */
65      public static int readNumbers(Scanner keyboard, String prompt, int[] array, int
            size) {
66
67          //Prompt user for input
68          System.out.println(prompt);
69          String source = keyboard.nextLine();
70
71          //Declare and construct scanner to access and read the user input
72          Scanner input = new Scanner(source);
73
74          while(input.hasNextInt()) {
75              array[size] = input.nextInt();
76              size++;
77          }
78
79          input.close();
```

```
80            return size;
81        }
82
83 }
```

Question: Use file IsSortedOversize2.java. In the given code, which one of the following statements is true, assuming the input is valid?

a) The program always runs correctly

b) The program always says that the data is sorted

c) The program always says that the data is unsorted

d) In line 22, the method call is incorrect

e) In line 24, the method call is incorrect

Answer: The correct answers are **B** and **D**. Due to an incorrect method call, the value of the `arraysize` variable is never updated, and the index remains at `-1`. As a result, the program always returns "The array is sorted" in all sample runs, regardless of the actual data. This activity helps students recognize the importance of correctly updating variables and validating method calls when working with arrays.

B.9 Classes with Generics

This was the final Code Insight activity implemented in class, following the coverage of ArrayLists. The problem is explained in detail to the students. For this activity, students are required to use an additional text file. Instructions are provided on how to download and import the text file into Eclipse to ensure proper setup and execution of the program.

Problem explanation:

- Store product reviews from a website
  - Three parallel ArrayList objects contain the product names, comments and ratings
- Read the product reviews from a file and stores the data into ArrayLists. File format:
  - One review per line
  - First word is name of the product
  - Second is user rating
  - Remainder of the line is comment
- Remove the product information, comments, and ratings if the comment contains a forbidden word.

Input assumptions: The assumptions for valid data

- The input file is in the valid format
- First word is the product (String), followed by an integer, which is a rating and third is the review by the buyer
- The product will have one word
- Reviews can be empty and are in mixed case

- Reviews can be a single word or sentence(s)
- Each line ends with a new line character

Example file data:

Toy 2 My kid barely played with this piece of CRAP

Blender 2 This darn thing was so noisy I couldn't hear myself think

Makeup 2 This made me look darn like Frankenstein

Makeup 5 I felt so sexy in this color

B.9.1 Problem 1

```
1 /**
2  * This program will read the product reviews from a file and stores the data into
       ArrayLists.
3  * The file contains the feedback data in the following format:
4  * The first word on each line is the name of the product. The second item on the
       line is the %user's rating.
5  * The remainder of the line contains the review.
6  * Using a given list of words that are considered profane (in an perfect size array)
       , the program will remove any post that
7  * contains a list of banned words.
8  * This program will remove the product information as well as comments and ratings
       from the ArrayLists
9  * if the comment contains a forbidden word.
10 *
11 * @author keerti banweer and Dr. Deborah Trytten
12 */
13 import java.util.ArrayList;
14 import java.util.Scanner;
15 import java.io.File;
16 import java.io.FileNotFoundException;
17
18 public class Exercise1 {
19
20     public static void main(String[] args) throws FileNotFoundException {
21
22         Scanner keyboard = new Scanner(System.in);
```

```
23          //declaring filename for the reviews file and reading data from console
24          System.out.println("Enter the filename");
25          String fileName = keyboard.nextLine();
26
27          // Declare and initialize an array with forbidden words
28          String[] forbidden = {"sex", "darn", "crap", "cheap", "ugly", "yuck", "
                smelly"};
29
30          // Declare and construct ArrayLists to store the information of products,
                ratings and comments
31          ArrayList<String> product = new ArrayList<String>();
32          ArrayList<Integer> ratings = new ArrayList<Integer>();
33          ArrayList<String> comments = new ArrayList<String>();
34
35          //reading the data on the product reviews from a file
36          readFile(product, ratings, comments, fileName);
37
38          //Call the removeForbidden method to remove the comments that contains
                forbidden words
39          removeForbidden(product, ratings, comments, forbidden);
40
41          System.out.println("The comments with the forbidden words are removed.");
42
43          keyboard.close();
44      }
45
46      /**
```

```
47      * This method takes three parallel ArrayList objects that contain a product
            name, a rating, and user comments.
48      * The method will also be given an array that contains words that are forbidden.
49      * Any comment that contains these words (either in whole or part) should be
            removed from all the three ArrayLists.
50      * @param product is an ArrayList of String object and stores the products type
51      * @param ratings is an ArrayList of Integer object that stores the rating on a
            product
52      * @param comments is an ArrayList of String object that stores the comments
            on the product feedback
53      * @param forbidden is a perfect size array that contains a list of forbidden
            words
54      */
55      public static void removeForbidden(ArrayList<String> product, ArrayList<
            Integer> ratings,
56              ArrayList<String> comments, String[] forbidden)
57      {
58          //The loop will iterate through the array of forbidden words
59          for (int f = 0; f < forbidden.length; ++f)
60          {
61              String forbiddenWord = forbidden[f];
62
63              //if the word exists in the Arraylist of reference name comments,
64              // remove the object from the three lists
65              if(comments.contains(forbiddenWord)) {
66                  int i = comments.indexOf(forbiddenWord);
67                  product.remove(i);
```

```
68                  ratings.remove(i);
69                  comments.remove(i);
70              }
71          }
72      }
73
74      /**
75       * The method reads the review data from a file and store the data into three
             separate ArrayLists
76       * @param product is an ArrayList of String object and stores the products type
77       * @param ratings is an ArrayList of Integer object that stores the rating on a
             product
78       * @param comments is an ArrayList of String object that stores the comments
             on the product feedback
79       * @param fileName is the name of the file that contains the user feedback data
80       * @throws FileNotFoundException
81       */
82      public static void readFile(ArrayList<String> product, ArrayList<Integer>
             ratings,
83              ArrayList<String> comments, String fileName) throws
                  FileNotFoundException
84      {
85          //Scanner declared and constructed to access the file
86          Scanner file = new Scanner(new File(fileName));
87
88          //Loop will go through each line in the file and stop when the end of file is
                reached
```

```
89          while(file.hasNextLine()) {
90              String line = file.nextLine();
91              // Scanner is constructed to read the data from a line in the feedback
                    file
92              Scanner data = new Scanner(line);
93              //Using the methods in the Scanner class to store the data into the
                    ArrayList
94              product.add(data.next());
95              ratings.add(data.nextInt());
96              comments.add(data.nextLine());
97              data.close();
98          }
99          //closing the scanner
100         file.close();
101     }
102 }
```

Question: Use file Exercise1.java In the given code, which one of the following statements is true?

a) The program runs as required to give the correct result with all valid input files

b) In line 31 to 33, the ArrayList objects should have been constructed inside the readFile method

c) In line 65, the contains method from ArrayList class won't work correctly

d) In line 66, the indexOf method from ArrayList class won't work correctly

e) In lines 94 to 96, the readFile method works correctly if the product has multiple words

Answer: The correct answers are **C** and **D**. To evaluate this code correctly, students must understand how methods are used with the `ArrayList` class. A single loop is insufficient for solving this problem; instead, it requires a different approach using nested loops. This activity reinforces the importance of selecting appropriate control structures when working with collections such as `ArrayList`.

B.9.2 Problem 2

```
1 /**
2  * This program will read the product reviews from a file and stores the data into
      ArrayLists.
3  * The file contains the feedback data in the following format:
4  * The first word on each line is the name of the product. The second item on the
      line is the user's rating.
5  * The remainder of the line contains the review.
6  * Using a given list of words that are considered profane (in an perfect size array)
      , the program will remove any post that
7  * contains a list of banned words.
8  * This program will remove the product information as well as comments and ratings
      from the ArrayLists
9  * if the comment contains a forbidden word.
10 *
11 * @author keerti banweer and Dr. Deborah Trytten
12 *
13 */
14 import java.util.ArrayList;
15 import java.util.Scanner;
16 import java.io.File;
17 import java.io.FileNotFoundException;
18
19 public class Exercise2 {
20
21     public static void main(String[] args) throws FileNotFoundException {
22
```

```
23          Scanner keyboard = new Scanner(System.in);
24          //declaring filename for the reviews file and reading data from console
25          System.out.println("Enter the filename");
26          String fileName = keyboard.nextLine();
27
28          // Declare and initialize an array with forbidden words
29          String[] forbidden = {"sex", "darn", "crap", "cheap", "ugly", "yuck", "
                smelly"};
30
31          // Declare and construct ArrayLists to store the information of products,
                ratings and comments
32          ArrayList<String> product = new ArrayList<String>();
33          ArrayList<Integer> ratings = new ArrayList<Integer>();
34          ArrayList<String> comments = new ArrayList<String>();
35
36          //reading the data on the product reviews from a file
37          readFile(fileName, product, ratings, comments);
38
39          //Call the removeForbidden method to remove the comments that contains
                forbidden words
40          removeForbidden(product, ratings, comments, forbidden);
41
42          System.out.println("The comments with the forbidden words are removed.");
43
44          keyboard.close();
45      }
46
```

```
47      /**
48       * This method takes three parallel ArrayList objects that contain a product
             name, a rating, and user comments.
49       * The method will also be given an array that contains words that are forbidden.
50       * Any comment that contains these words (either in whole or part) should be
             removed from all the three ArrayLists.
51       * @param product is an ArrayList of String object and stores the products type
52       * @param ratings is an ArrayList of Integer object that stores the rating on a
             product
53       * @param comments is an ArrayList of String object that stores the comments
             on the product feedback
54       * @param forbidden is a perfect size array that contains a list of forbidden
             words
55       */
56      public static void removeForbidden(ArrayList<String> product, ArrayList<
            Integer> ratings,
57              ArrayList<String> comments, String[] forbidden)
58      {
59          // This loop will access each word in the forbidden array reference
60          for (int f = 0; f < forbidden.length; ++f)
61          {
62              String word = forbidden[f].toLowerCase();
63              // this inner loop will access each comment in the Arrays
64              for (int c = 0; c < comments.size(); ++c)
65              {
66                  String comment = comments.get(c).toLowerCase();
67
```

```
68                  // if the forbidden word exist in the specific comment,
69                  // it will be removed from all the three ArrayLists
70                  if (comment.contains(word))
71                  {
72                      //remove the object at index c
73                      product.remove(c);
74                      ratings.remove(c);
75                      comments.remove(c);
76                      --c;
77                  }
78              }
79          }
80      }
81
82      /**
83       * The method reads the review data from a file and store the data into three
             separate ArrayLists
84       * @param product is an ArrayList of String object and stores the products type
85       * @param ratings is an ArrayList of Integer object that stores the rating on a
             product
86       * @param comments is an ArrayList of String object that stores the comments
             on the product feedback
87       * @param fileName is the name of the file that contains the user feedback data
88       * @throws FileNotFoundException
89       */
90      public static void readFile(String fileName, ArrayList<String> product,
            ArrayList<Integer> ratings,
```

```
91              ArrayList<String> comments) throws FileNotFoundException
92      {
93          //Scanner declared and constructed to access the  file
94          Scanner  file  = new Scanner(new File(fileName));
95
96          //Loop will go through each line  in  the  file  and stop when the end of file  is
                reached
97          while(file.hasNextLine()) {
98              String  line  =  file.nextLine();
99              // Scanner is constructed to read the data from a line  in  the feedback
                    file
100             Scanner data = new Scanner(line);
101             //Using the methods in the Scanner class to store  the data into  the
                    ArrayList
102             product.add(data.next());
103             ratings.add(data.nextInt());
104             comments.add(data.nextLine());
105             data.close();
106         }
107         //closing  the scanner
108          file .close();
109     }
110 }
```

Question: Use file Exercise2.java In the given code, which one of the following statements is true?

a) The program runs as required to give the correct result with all valid input files

b) In line 62, toLowerCase() method is not necessary

c) In line 64, c should be initialized to f instead of 0

d) In line 70, the contains method from String class won't work correctly

e) In line 76, decrementing the counter c is not needed

Answer: The correct option is **A**; the code functions as intended. Some of the other options include common misconceptions to address potential challenges students may face when working with more complex programming concepts. In the initial attempt, several students selected option **E** as the correct answer. However, this option represents an essential part of the logic when removing values from an `ArrayList`, and misunderstanding it can lead to incorrect conclusions.

Appendix C

Programming Question from Final Exam of Fall 2024

The following example is taken from the final exam of the introductory programming course conducted during the Fall 2024 semester. Both sections of the course administered the same exam and used a consistent grading rubric to evaluate student responses. The final programming question consists of three parts and is worth 30 points out of a total exam score of 100. Descriptions of all relevant methods were provided in the format of Java documentation to support students during the exam.

The exams were conducted on paper, and students were required to write their solutions manually for each part of the question.

The programming questions are similar to class project assignment that student work on everyweek to practice concept and program writing skills.

Figure C.1 presents the description of the final programming question, including a sample run and an example of valid input data. The students were instructed to assume that all user input would be valid. The primary objective of this question was to assess students' understanding of the programming concepts, code-writing proficiency, and problem-solving abilities.

8. (30 points, 10 points each part) Write three methods to help solve the problem below.

There is usually a ton of food left over from holidays, but often a few ingredients for any given recipe are missing. This program is designed to help a cook find a recipe while purchasing a minimum number of additional ingredients.

Suppose that the cook has the ingredients in the pantry.txt file on hand.
The recipes are stored in files by their name. Three examples are shown below (on the right)

pantry.txt
flour
sugar
salt
pepper
turkey
bread
broccoli
milk
cheese
onion
mayo
raisin
vegetable oil

broccoliSalad.txt
broccoli
vinegar
sugar
vegetable oil
raisins
red onion

gardenSalad.txt
lettuce
radish
carrot
cucumber
vinegar
vegetable oil

turkeySalad.txt
turkey
mayo
onion
grapes

The program will allow the user to set a maximum number of ingredients that they wish to purchase and then select recipes until they find one that requires fewer than the maximum number of ingredients. A sample run of the program is below. User input is bold, italicized and underlined. You may assume that the user does not make any mistakes.

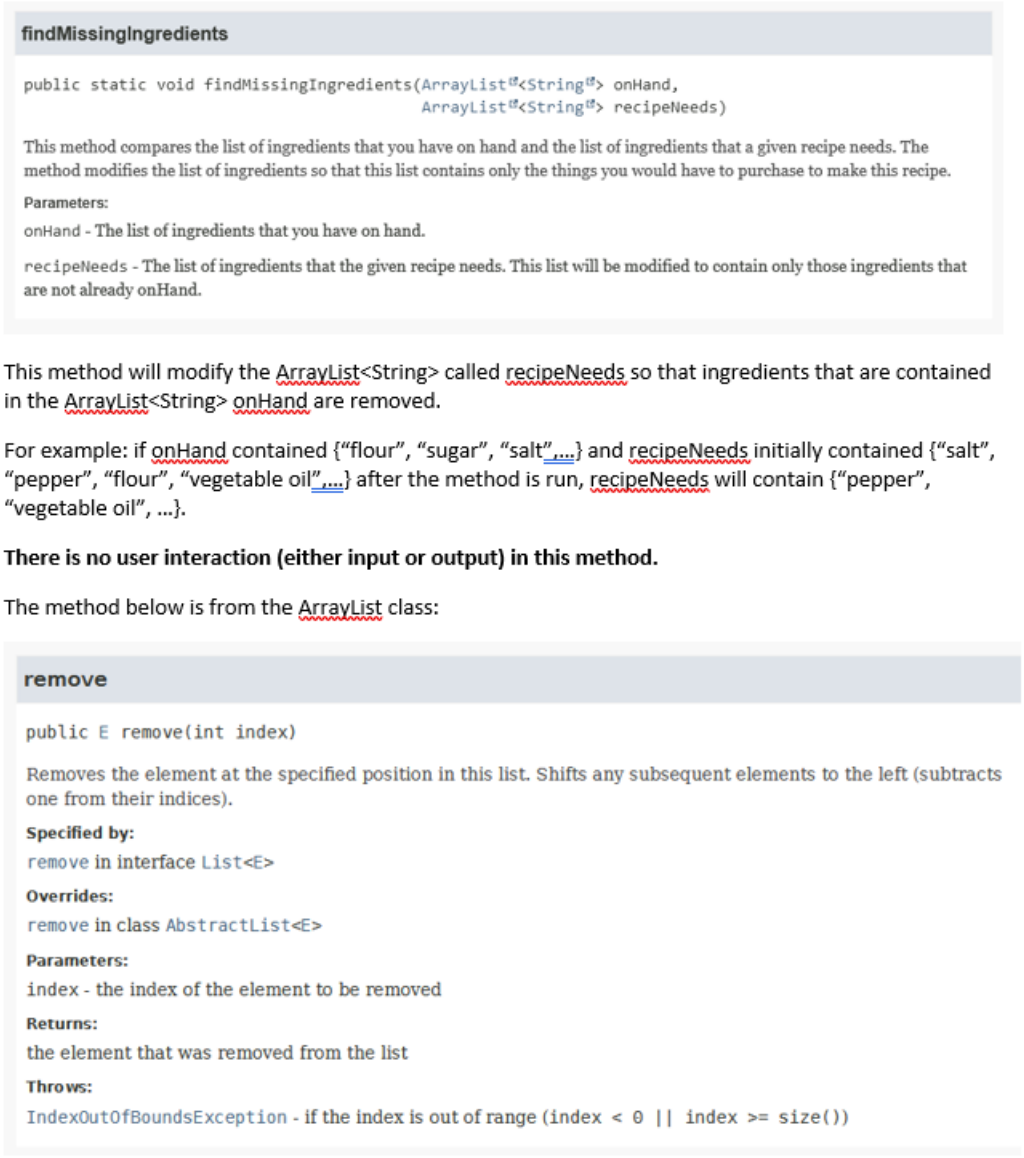
How many ingredients are you willing to purchase?
2
Which recipe would you like to try next?
broccoliSalad
You need 3 additional ingredients
Which recipe would you like to try next?
gardenSalad
You need 5 additional ingredients
Which recipe would you like to try next?
turkeySalad
You need 1 additional ingredient(s)
Purchase the following ingredient(s): [grapes]

The methods signatures are below. Methods are described on the back of the examination pages.

- The main program
- **void** findMissingIngredients(ArrayList<String> onHand, ArrayList<String> recipeNeeds)
- ArrayList<String> findAcceptableRecipe(ArrayList<String> stuffIHave, Scanner keyboard, **int** maximum)
- ArrayList<String> readFile(String fileName) // implemented by someone else

Figure C.1: The Description of the Programming Question

The supporting methods for the first part of the question is provided as below in the Figure C.2:



findMissingIngredients

```
public static void findMissingIngredients(ArrayList<String> onHand,
                                          ArrayList<String> recipeNeeds)
```

This method compares the list of ingredients that you have on hand and the list of ingredients that a given recipe needs. The method modifies the list of ingredients so that this list contains only the things you would have to purchase to make this recipe.

Parameters:

`onHand` - The list of ingredients that you have on hand.

`recipeNeeds` - The list of ingredients that the given recipe needs. This list will be modified to contain only those ingredients that are not already onHand.

This method will modify the ArrayList<String> called recipeNeeds so that ingredients that are contained in the ArrayList<String> onHand are removed.

For example: if onHand contained {"flour", "sugar", "salt",...} and recipeNeeds initially contained {"salt", "pepper", "flour", "vegetable oil",...} after the method is run, recipeNeeds will contain {"pepper", "vegetable oil", ...}.

There is no user interaction (either input or output) in this method.

The method below is from the ArrayList class:

remove

```
public E remove(int index)
```

Removes the element at the specified position in this list. Shifts any subsequent elements to the left (subtracts one from their indices).

Specified by:

`remove` in interface `List<E>`

Overrides:

`remove` in class `AbstractList<E>`

Parameters:

`index` - the index of the element to be removed

Returns:

the element that was removed from the list

Throws:

`IndexOutOfBoundsException` - if the index is out of range (`index < 0 || index >= size()`)

The API for the get() and size() methods from the ArrayList class is on the back of the last page of the exam.

The API for the sort() and binarySearch methods from the Collections class is on the back of problem #5, if you choose to use these methods. They are not required.

Figure C.2: Useful Methods Descriptions

The Figure C.3 represent the first part of the problem. The method signature is provided to the students are expected to use the parameters from the method signature. The students would review the method description from the Figure C.2 and write their program.

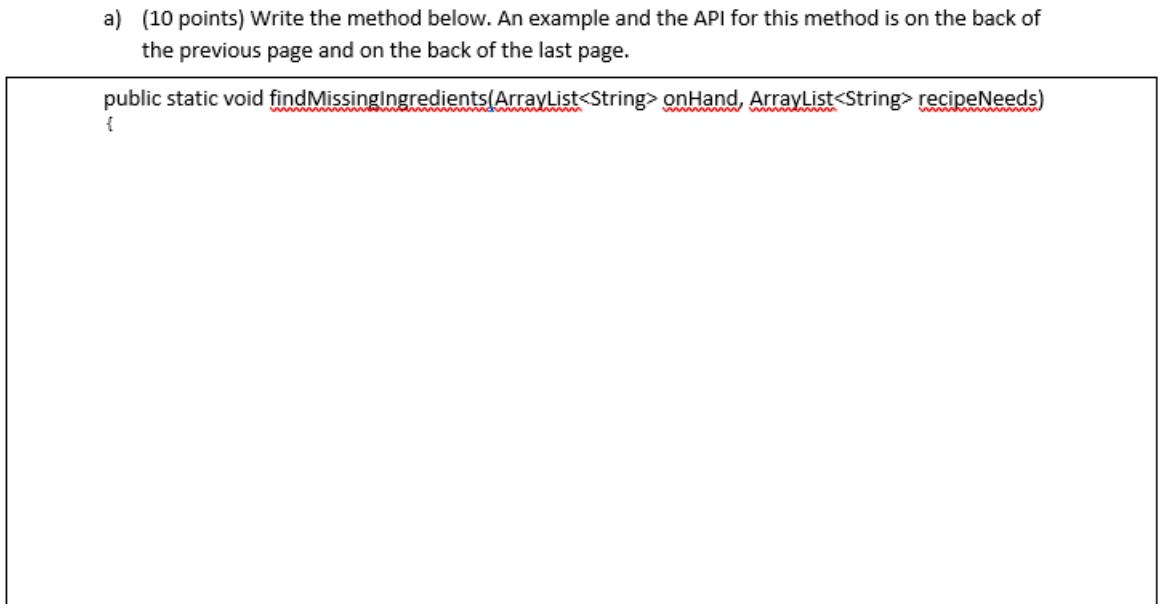
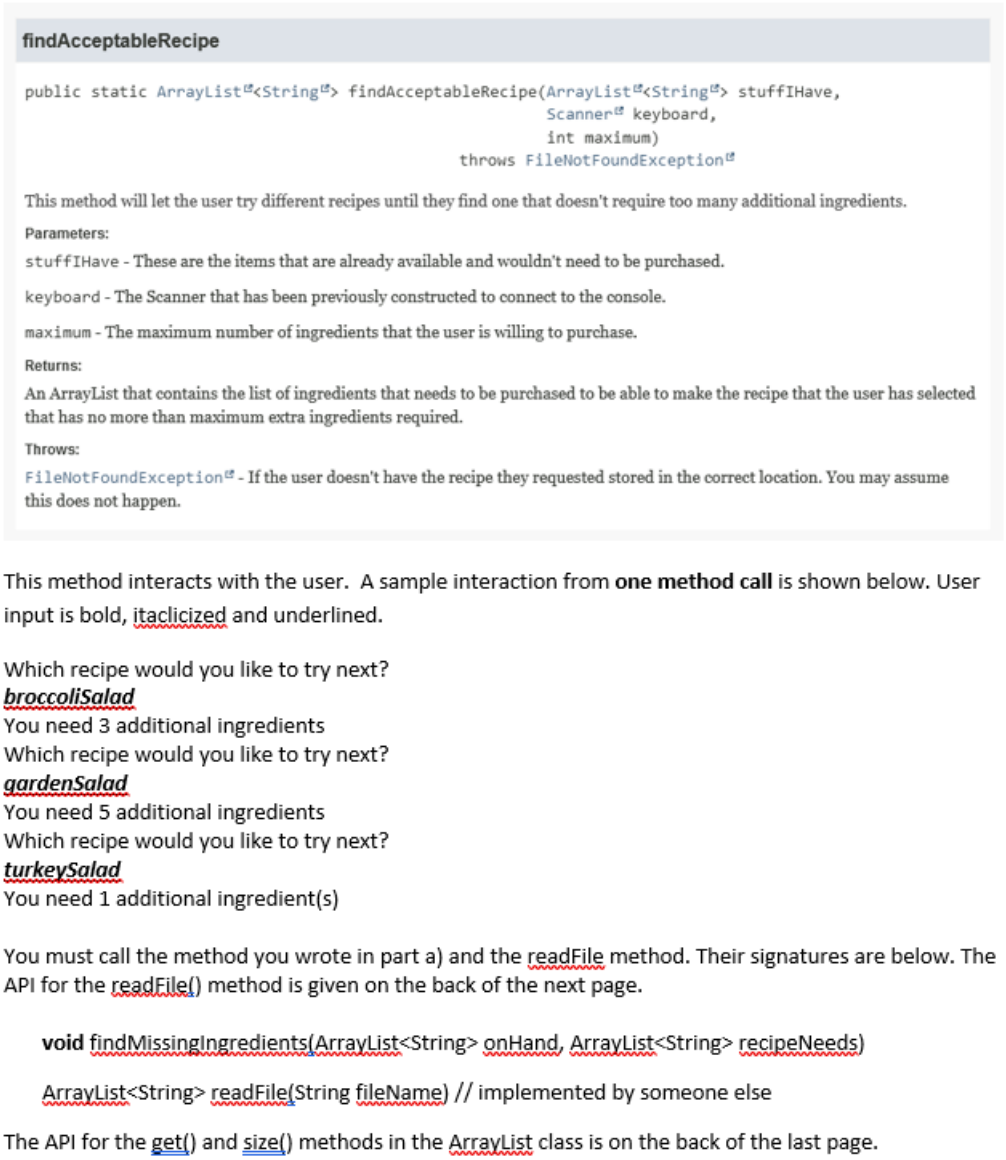


Figure C.3: First Part of the Programming Problem

Figure C.4 represent the description of additional methods that will be used to answer the second part of the programming problem. Each part asks student to write the code segment for the given method.



findAcceptableRecipe

```
public static ArrayList<String> findAcceptableRecipe(ArrayList<String> stuffIHave,
                                                     Scanner keyboard,
                                                     int maximum)
                                              throws FileNotFoundException
```

This method will let the user try different recipes until they find one that doesn't require too many additional ingredients.

Parameters:

`stuffIHave` - These are the items that are already available and wouldn't need to be purchased.

`keyboard` - The Scanner that has been previously constructed to connect to the console.

`maximum` - The maximum number of ingredients that the user is willing to purchase.

Returns:

An ArrayList that contains the list of ingredients that needs to be purchased to be able to make the recipe that the user has selected that has no more than maximum extra ingredients required.

Throws:

`FileNotFoundException` - If the user doesn't have the recipe they requested stored in the correct location. You may assume this does not happen.

This method interacts with the user. A sample interaction from **one method call** is shown below. User input is bold, itaclicized and underlined.

Which recipe would you like to try next?
broccoliSalad
You need 3 additional ingredients
Which recipe would you like to try next?
gardenSalad
You need 5 additional ingredients
Which recipe would you like to try next?
turkeySalad
You need 1 additional ingredient(s)

You must call the method you wrote in part a) and the readFile method. Their signatures are below. The API for the readFile() method is given on the back of the next page.

```
void findMissingIngredients(ArrayList<String> onHand, ArrayList<String> recipeNeeds)

ArrayList<String> readFile(String fileName) // implemented by someone else
```

The API for the get() and size() methods in the ArrayList class is on the back of the last page.

Figure C.4: Additional Method Description for the Second Part of the Programming Problem

The Figure C.5 represent the second part of the problem. The method signature is provided to the students are expected to use the parameters from the method signature.

b) (10 points) Implement the method below. The API for this method and an example are on the back of the previous page. This method must call the method you wrote in part a) and the readFile() methods. The signature of these methods are on the back of the previous page.

```
public static ArrayList<String> findAcceptableRecipe(ArrayList<String> stuffIHave, Scanner keyboard,
        int maximum)

{
```

Figure C.5: Second Part of the Programming Problem

The next Figure C.7 represent the detail description of the programming question for the third part of the programming question.

The next Figure **??** represent the third part of the programming problem, where students are asked to write the main method, and can call any of the methods from the previous part.

readFile

```
public static ArrayList<String> readFile(String fileName)
                                 throws FileNotFoundException
```

Read in a file with the given fileName. Each line of the file will be stored at consecutive indices in an ArrayList. The text in the file will be converted to all lower case.

Parameters:

`fileName` - The name of the file.

Returns:

The contents of the file, stored as an ArrayList with each element of the ArrayList containing one line of the file converted to lower case (in order).

Throws:

`FileNotFoundException` - If the requested file is not found. You may assume this does not happen.

Here is a sample run of the program. User input is bold, italicized and underlined. **The input that is inside the box is written by the findAcceptableRecipe()** method and should not appear in the main program.

How many ingredients are you willing to purchase?
1
Which recipe would you like to try next?
broccoliSalad
You need 3 additional ingredients
Which recipe would you like to try next?
gardenSalad
You need 5 additional ingredients
Which recipe would you like to try next?
turkeySalad
You need 1 additional ingredient(s)
Purchase the following ingredient(s): [grapes]

You must call the readFile() method and the findAcceptableRecipe() method. The API for the findAcceptableRecipe() method is on the back of the previous page.

Figure C.6: Additional Method Description

c) (10 points) Write the main program. Both methods below should be called. **To get credit for calling methods, all arguments must be declared and initialized.** A sample run or the program, and the API for the readFile method are given on the back of the previous page.

public static ArrayList<String> findAcceptableRecipe(ArrayList<String> stuffIHave, Scanner keyboard, **int** maximum)

public static ArrayList<String> readFile(String fileName)

Figure C.7: Third Part of the Porgramming Problem

Appendix D

IRB Approvals

The research presented in this dissertation, which addresses the central research problem and outlines its contributions, was approved by the Institutional Review Board (IRB) under the following three applications:

- The study presented in chapter 3 is associated with IRB application number: 16326. This approval allowed the recruitment and interviewing of students enrolled in the CS program at the University of Oklahoma.
- The study presented in chapter 4 is associated with IRB application number: 16436. This application permitted the collection of anonymous feedback from students enrolled in the introductory programming course, focusing on their experiences with the **Code Insight** activity.
- The findings presented in chapter 5 are associated with IRB application number: 16407. This approval enabled the collection of student consent to anonymously use their course grades in order to examine the impact of the **Code Insight** activity on their code-writing skills.