Defining, Measuring, and Investigating Creative Elaboration in Programming

by

Vidhi Mittal

2025

Defining, Measuring, and Investigating Creative Elaboration in Programming

By

Vidhi Mittal

Davide Fossati
Adviser

An abstract of
a thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Science with Honors

Computer Science

2025

Abstract

Defining, Measuring, and Investigating Creative Elaboration in Programming
By Vidhi Mittal

While programming assignments often yield functionally equivalent solutions, they differ dramatically in creative expression. Creativity in programming has traditionally been theorized along four dimensions: fluency (quantity of ideas), flexibility (diversity of ideas), originality (novelty of ideas), and elaboration (depth and refinement of ideas). While prior work has explored the first three, elaboration remains underexamined in code.

This thesis investigates creative elaboration in source code: the use of modular decomposition, naming clarity, documentation, and whitespace as forms of creative structure. We formally define a computational metric of elaboration based on five syntactic features and use it to examine how elaborative expression relates to code comprehension, correctness, and perceived quality.

In Phase 1, we conduct a controlled user study (n = 15) comparing programmer performance across elaborated and minimal versions of the same Java codebase. Participants interacting with elaborated code achieved significantly higher comprehension accuracy and reported greater perceived clarity. Debugging performance and efficiency were also higher in the elaborated group, though not always statistically significant. Feature addition outcomes trended favorably but showed greater variability.

In Phase 2, we apply the elaboration metric to 6,132 real-world student programs from the PROGpedia dataset. Elaboration scores were significantly higher in accepted submissions than in incorrect ones, predictive of correctness in supervised models, and stable across Java and Python. We further observed systematic variation in elaboration by problem domain, with algorithmic concepts such as MST and Graph Traversal eliciting higher elaboration than string processing or backtracking tasks.

Together, these findings demonstrate that elaborative design features might influence both human and automated interpretations of code.

Defining, Measuring, and Investigating Creative Elaboration in Programming

By

Vidhi Mittal

Davide Fossati
Adviser

A thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Science with Honors

Computer Science

2025

Acknowledgements

This thesis reflects many hours of learning, thinking, and revising. More importantly, it reflects the support, insight, and encouragement I've received from so many people during my time at Emory.

I am deeply grateful to my advisor, Dr. Davide Fossati, for his steady mentorship, generous feedback, and faith in this project. I also sincerely thank my committee members, Dr. Andreas Züfle and Dr. Brajesh Samarth, whose perspectives and thoughtful questions helped sharpen this work. A special thanks to Dr. Arnon Hershkovitz from Tel Aviv University for his kind guidance and expertise in the field.

To my family—thank you for your love, patience, and unwavering support. And to my friends, thank you for the laughter, late-night pep talks, and belief in me that never wavered.

I couldn't have done this without you.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Creativity has long been a hallmark of human problem-solving, driving innovation across disciplines from the arts and literature to engineering and the sciences. In the field of computer science, however, creativity has often been marginalized in favor of correctness, efficiency, and rigor. As the discipline evolves to encompass not only algorithmic precision but also design thinking, user-centered development, and computational media, the recognition of creativity as a core competency in programming has gained momentum [10]. Yet despite growing interest, there remains a significant gap in our understanding of how to define, measure, and cultivate creativity within the context of code.

Existing models of creativity, particularly those drawn from psychology and education, offer a useful starting point. Torrance [12] and Guilford [5] conceptualize creativity as a multidimensional construct comprising *fluency* (the number of ideas generated), *flexibility* (diversity of ideas), *originality* (novelty of ideas), and *elaboration* (the depth, detail, and development of ideas). While these dimensions have been widely applied in assessments of general creative thinking, their translation into programming remains inconsistent. Most empirical research in computing education that addresses creativity focuses disproportionately on originality while largely neglecting elaboration, which may be critical in determining the completeness, coherence, and expressive depth of a program.

Elaboration, as a construct, captures the degree to which an idea is extended, refined, or implemented with care and attention to detail. In the context of programming, elaboration may manifest through additional functionality, user interface polish, documentation, modular structure, or thoughtful narrative and aesthetic elements, especially in open-ended or creative computing tasks. However, measuring elaboration presents significant challenges. Traditional software engineering metrics, such as lines of code or cyclomatic complexity, offer limited insight into the *quality* or *intentionality* of elaborative features. At the same time, rubric-based assessments and expert judgments, while contextually rich, are labor-intensive, subjective, and difficult to scale.

The absence of robust and scalable methods for evaluating creative elaboration in code creates a critical bottleneck in both research and practice. Without reliable measures, educators cannot assess or support elaborative thinking; platforms cannot provide actionable feedback; and researchers cannot compare creativity across contexts or interventions. Moreover, neglecting elaboration distorts the broader picture of creativity in programming by privileging sparse novelty over richly developed expression.

Chapter 2

Background

2.0.1 Creativity in Programming Education

The integration of creativity into computer science education has emerged as a core pedagogical aim, especially in efforts to broaden participation and cultivate design thinking [10]. Programming, traditionally associated with logic and precision, is increasingly framed as a medium for *personal expression*, *exploration*, and *creative problem-solving* [2]. Educational standards reflect this priority: the K–12 Computer Science Framework emphasizes creativity as a key theme in computing practices [7], and the College Board's AP Computer Science Principles course lists creative development as one of its core "Big Ideas" [3]. However, these documents stop short of defining how creativity should be assessed in the context of programming.

2.0.2 Theoretical Models of Creativity

Creativity research across psychology and education offers a foundational structure for understanding and measuring creativity. A widely adopted framework comes from Torrance [12], who extended Guilford's [5] theory of *divergent thinking* into four core dimensions: *fluency*, *flexibility*, *originality*, and *elaboration*.

In design and creativity studies, elaboration is seen as essential for transforming raw ideas into fully realized products. O'Quin and Besemer [8] argue that *purposeful elaboration*—detail that enhances clarity, completeness, or aesthetic value—is a hallmark of high-quality creative output. This perspective resonates with programming, where a creative solution is not only unexpected but also *well-developed*, robust, and expressive.

2.0.3 Defining Elaboration in Programming

In programming, elaboration may take many forms: expanding minimal solutions with additional functionality, modularizing and documenting code, adding user interface elements or storytelling, or refining performance through abstraction. However, elaboration is frequently conflated with *code complexity* or verbosity, which can lead to misclassification. For example, automated grading systems may favor concise code, discouraging elaborative additions that do not directly impact correctness or runtime efficiency.

Existing literature offers multiple interpretations of elaboration. In block-based environments like Scratch, elaboration may involve the number of sprites, scenes, and interactions used [4]. In textual languages, it might involve use of advanced features, architectural decisions, or stylistic enhancements. Yet these forms are not consistently acknowledged or rewarded in instructional or evaluative settings.

2.0.4 Methods for Assessing Elaboration

Rubric-Based Evaluation

Rubric-based assessment is a widely used method to evaluate student programming artifacts, particularly in project-based learning environments. Grover et al. [4] incorporated "engagement" and "creativity" as rubric categories in their curriculum to capture elaborative aspects of student-authored Scratch projects, such as interactive features, narrative development, and aesthetic enhancements. These dimensions encouraged students to go beyond functional correctness, fostering deeper, more personalized expressions of computational thinking. However, while effective in capturing rich, context-specific elaboration, such rubrics are time-intensive to apply, challenging to scale, and often lack generalizability or validation across different programming environments and learner populations.

Expert Judgment (CAT)

The Consensual Assessment Technique (CAT) [1] has been employed to assess creativity holistically. Studies using CAT in programming (e.g., [11]) suggest that elaboration is rewarded implicitly—judges often prefer feature-rich, polished, or engaging programs. However, CAT is inherently subjective and suffers from limited inter-rater reliability without rigorous calibration.

Quantitative and Automated Metrics

Automated systems attempt to proxy elaboration through measurable features. Israel-Fishelson and Hershkovitz [6] define elaboration as the number of "baskets" (grouped command types) in a correct solution string, reflecting the diversity of computational steps. While this enables scalable measurement, the authors note that elaboration may conflate meaningful complexity with task-driven verbosity, underscoring the need for refined, context-aware modeling.

2.0.5 Process-Based Insights and Behavioral Data

Beyond static code, elaboration can be inferred from how students engage with programming over time. Practices such as iterative development, exploratory debugging, or enhancement beyond functional correctness may signal deeper engagement with computational ideas. Brennan and Resnick [2] emphasize the importance of studying students' design processes—not just final artifacts—through methods like artifact-based interviews and real-time observation. While they suggest that richer insights could be gained from capturing process data, such as development histories or in-project documentation, most platforms do not systematically collect or interpret this data as evidence of creative or elaborative thinking.

2.0.6 Gaps in the Literature

Despite recent progress, several significant gaps persist in the literature on creative elaboration in programming. First, there is a clear **lack of standardization**: definitions of elaboration vary widely across studies, with no agreed-upon framework to guide assessment or comparison. Second, there is an **very little emphasis on elaboration**, as most creativity assessments focus primarily on novel solutions while neglecting elaboration as a distinct and meaningful dimension. Third, current approaches exhibit **limited generalizability**, with most research centered on block-based, K–12 environments rather than textual programming in higher education or professional contexts. Finally, there is a widespread **neglect of process**: few frameworks capture elaboration as a dynamic behavior unfolding over time through iterative coding, revision, or refinement.

These gaps motivate the development of scalable, valid, and multidimensional approaches to measuring elaboration in programming creativity, which this research aims to address.

Chapter 3

Approach

3.1 Phase 1: Controlled User Study

To examine the influence of elaboration on software comprehension and maintainability, we conducted a *between-subjects controlled experiment* with 15 participants. The goal was to determine whether elaborative code features—such as modular decomposition, naming clarity, spacing, and inline documentation—enhance programmers' task performance and subjective experience when interacting with functionally equivalent source code.

3.1.1 Study Protocol and Participants

The study followed a fixed-sequence protocol comprising five stages:

1. Pre-Study Programming Assessment

2. Code Comprehension Task

3. Debugging Task

4. Feature Addition Task

5. Post-Study Perception Survey

Participants (n = 15) were undergraduate students with programming experience, recruited from computing-related majors. They were randomly assigned to one of two treatment conditions: **Minimal** or **Elaborated**, corresponding to the structure and elaboration level of the code they were asked to work with.

All tasks were completed on researcher-provided laptops configured with *Visual Studio Code (VS Code)* as the development environment. The editor was preloaded with the assigned codebase, and participants were instructed not to use external resources or execute the code. Task timing and interaction flow were monitored by a researcher present throughout the session.

3.1.2 Stimulus Materials

The code stimulus was a Java-based command-line library management system supporting book addition, borrowing, returning, and listing. All functional behavior was preserved across conditions; the only manipulated variable was the presence or absence of elaborative features. Full code listings for all four versions—minimal, elaborated, and their bugged counterparts—are provided in Appendix C.

Minimal Condition Participants assigned to the **Minimal** group interacted with `MinimalLibrary.java` (45 LOC), a single-method Java class that exhibited:

- No modular decomposition (all logic in `main`)
- Opaque variable names (`b`, `t`, `c`)
- Absence of inline or block comments
- Dense and repetitive branching logic
- No user-facing documentation

Elaborated Condition Participants in the **Elaborated** group were provided `ElaboratedLibrary.java` (~150 LOC), which included:

- Modular design (`addBook`, `borrowBook`, `returnBook`, `listBooks`, `run`)
- Descriptive identifiers (`books`, `title`, `isAvailable`)
- JavaDoc comments for each method
- Inline commentary for control logic
- Consistent structural formatting and whitespace

Bugged Variants For the debugging task, each group received a syntactically valid but semantically incorrect variant of their assigned program. In both cases, a new `removeBook()` feature had been introduced, which failed to remove the book from the internal data structure (`books.put(title, false)` instead of `books.remove(title)`). This bug preserved compilation correctness but resulted in silent logic errors, requiring participants to trace and reason about program state changes.

3.1.3 Task Design

Pre-Study Programming Assessment Participants completed a Java proficiency diagnostic via Qualtrics (see Appendix B). The assessment included:

- Five multiple-choice questions targeting language-specific behavior (e.g., integer division, method declarations, loop control structures)
- Self-reported experience level (coursework, years of use)
- Confidence in code reading and debugging tasks (4-point Likert scale)

Code Comprehension Task Participants answered 7 timed multiple-choice questions referencing their assigned program. Items covered:

- Data structure semantics (`HashMap<String, Boolean>`)
- Behavioral response to user input
- Return values, state mutation, and output predictions
- Edge-case tracing and flow control

Debugging Task Participants were presented with the bugged variant of their code and instructed:

> *"This version of the program contains an implementation bug in the removeBook feature. Please identify the incorrect behavior and propose a fix."*

They were asked to:

- Describe the observed behavior and its deviation from expected functionality
- Identify the buggy line(s)
- Write corrected replacement code

Feature Addition Task Participants were instructed to add a **Search for a Book** feature (new menu option). Functional requirements included:

- Prompting the user for a title
- Checking existence and availability
- Displaying one of three possible status messages

Submissions were scored based on:

- Functional correctness

- Integration with the existing codebase
- Task completion time

Post-Study Perception Survey Participants completed a 4-item post-task survey assessing:

- Ease of debugging
- Ease of modification
- Overall clarity
- Confidence in correctness

All items used a 5-point Likert scale (1 = Strongly Disagree, 5 = Strongly Agree).

3.1.4 Hypotheses

We wanted to test the following hypotheses:

- **H1**: Participants in the Elaborated condition will achieve higher accuracy and lower time on the comprehension task.
- **H2**: Participants in the Elaborated condition will be more likely to correctly and quickly identify and fix the seeded bug.
- **H3**: Participants in the Elaborated condition will complete the feature addition task more accurately and in less time.
- **H4**: Participants in the Elaborated condition will report significantly higher subjective ratings of code clarity, structure, and perceived creativity.

3.2 Phase 2: Formalizing and Scaling the Elaboration Metric

In Phase 2, we developed and validated a computational metric to estimate *creative elaboration* in source code. The metric was explicitly designed to enable scalable measurement of elaboration across large corpora of functionally equivalent programs.

We then applied this metric to the `PROGpedia` dataset [9] to characterize elaborative practices in real-world student code, across both Python and Java.

3.2.1 Metric Definition

Let a program P be represented as a tuple of measurable syntactic features:

$$P = (f_1, f_2, f_3, f_4, f_5)$$

Where:

- f_1: Comment density
- f_2: Number of methods
- f_3: Average identifier length
- f_4: Doc comment density
- f_5: Blank line ratio

Each feature f_i is normalized to a bounded domain $[0, 1]$ via domain-specific capping functions $\phi_i : \mathbb{R} \to [0, 1]$ as follows:

$$\phi_1(f_1) = \min(f_1, 1.0)$$

$$\phi_2(f_2) = \min(f_2, 10)/10$$

$$\phi_3(f_3) = \min(f_3, 15)/15$$

$$\phi_4(f_4) = \min(f_4, 1.0)$$

$$\phi_5(f_5) = \min(f_5, 0.3)/0.3$$

The elaboration score is then computed as a combination:

$$E(P) = \sum_{i=1}^{5} w_i \cdot \phi_i(f_i)$$

With weights $\vec{w} = (0.25, 0.20, 0.15, 0.25, 0.15)$ chosen via heuristic calibration based on theoretical emphasis on communicative clarity (e.g., documentation, structure). The function $E : \mathcal{P} \to [0, 1]$ thus provides a bounded elaboration score interpretable across languages and problem types.

3.2.2 Feature Extraction via Static Analysis

To compute the features f_1 through f_5, we implemented a dual-mode static analysis pipeline tailored to the syntactic characteristics of Python and Java. For both languages, feature vectors were computed using a combination of AST-based analysis and regular expression–based parsing.

Python: AST-Based Semantic Feature Extraction

Python programs were parsed using the `ast` module from the standard library. AST traversal was used to extract:

- f_2: Number of `FunctionDef` and `ClassDef` nodes
- f_4: Number of docstring-bearing definitions via `ast.get_docstring()`
- f_3: All identifier names from `ast.Name`, excluding built-ins

Line-level analysis computed:

- f_1: Ratio of lines starting with `#`
- f_5: Ratio of blank lines

Doc comment density was defined as:

$$f_4 = \frac{n_{\text{doc}}}{n_{\text{methods}} + 1}$$

Java: Structural Parsing via Regular Expressions

In the absence of a native Java AST interface, we used regular expressions to extract semantic structure:

- **Method detection**: matched typical Java method signatures with access modifiers, return types, and parameters.
- **JavaDoc detection**: recognized `/**` ... `*/` blocks and mapped them to methods.
- **Identifier extraction**: all non-keyword tokens were matched using the pattern:

`[a-zA-Z_][a-zA-Z0-9_]*`

filtered to exclude Java reserved words and standard library types.

- **Line statistics**: `//` and `/* */` comments were counted for f_1, and blank lines for f_5.

3.2.3 Application to the PROGpedia Dataset

We applied the elaboration metric to student-submitted code from the `PROGpedia` dataset [9], a curated repository of programming solutions collected between 2003 and 2020 from undergraduate students enrolled in introductory programming courses. The dataset contains submissions to 16 programming problems, primarily in `Java` and `Python`, captured via the *Mooshak* online judge platform.

Each solution in PROGpedia is annotated with its corresponding execution outcome, that is, *Accepted*, *Runtime Error*, or *Wrong Answer*.

Each program was statically analyzed and transformed into a 5-dimensional elaboration feature vector $(f_1, \ldots, f_5)$, from which the elaboration score $E(P)$ was computed.

This large-scale application enabled us to conduct comparative analyses across:

- Functionally equivalent solutions with varying elaboration levels
- Correct versus incorrect submissions
- Language-specific elaboration tendencies in Python and Java

By applying the elaboration metric to a pedagogically grounded, multilingual dataset, we aimed to observe stylistic and structural patterns in student code.

For implementation details, source code, and data processing scripts, please see Appendix A.

Chapter 4

Experiments & Analysis

4.1 Phase 1 Results

4.1.1 Overview

We conducted a between-subjects experimental study ($n = 15$) to study whether our self-defined elaborative features measurably enhance code comprehension, debugging performance, and functional extension tasks among the group. Participants were randomly assigned to interact with either a **Minimal** or **Elaborated** version of a Java-based command-line library management system. Quantitative outcomes were analyzed across three axes: task *accuracy*, *efficiency* (accuracy per unit time), and *subjective perception*. Statistical comparisons employed both parametric (t-test, linear regression) and non-parametric (Mann–Whitney U) methods.

4.1.2 Descriptive Statistics

Before presenting inferential analyses, we report descriptive statistics for task performance across conditions. Table 4.1 summarizes the mean and standard deviation of accuracy, time-on-task, and efficiency for each of the three programming tasks (comprehension, debugging, and feature addition), separately for the **Minimal** and

Elaborated code conditions. These descriptive patterns suggest consistent performance advantages associated with elaborated code, particularly in terms of efficiency, within the group.

Table 4.1: Descriptive statistics for each programming task by code condition.

Task	**Metric**	**Minimal (M)**	**Minimal (SD)**	**Elaborated (M)**	**Elaborated (SD)**
Comprehension	Score	7.75	1.67	9.00	0.00
	Time (min)	3.99	0.54	3.68	1.83
	Efficiency	2.01	0.58	3.13	1.72
Debugging	Score	6.13	3.87	9.00	2.24
	Time (min)	4.49	1.13	3.06	1.70
	Efficiency	1.51	1.33	4.52	3.60
Feature Addition	Score	7.00	3.66	8.71	2.21
	Time (min)	6.96	1.91	7.11	2.72
	Efficiency	1.08	0.63	1.57	1.12

4.1.3 Comprehension Task Performance

Participants assigned to the elaborated condition achieved perfect comprehension accuracy ($M = 9.00$, $SD = 0.00$), while those in the minimal condition demonstrated lower and more variable performance ($M = 7.75$, $SD = 1.67$). A Welch's t-test indicated marginal significance ($t(13) = -2.12$, $p = 0.072$), while a non-parametric Mann–Whitney U test identified a statistically significant difference ($U = 10.5$, $p = 0.019$).

To assess whether task time or prior Java proficiency mediated this effect, we fit a multiple linear regression model predicting comprehension accuracy from code condition, Java pre-quiz score, and comprehension task time. The model accounted for 37.4% of the variance ($R^2 = 0.374$). Code condition was as a marginally significant predictor ($\beta = 1.24$, $p = 0.074$), while neither pre-quiz score nor time-on-task reached statistical significance. This suggests elaboration had an independent effect on comprehension beyond baseline skill or time spent.

Figure 4.1 shows a boxplot of comprehension scores across the two conditions.

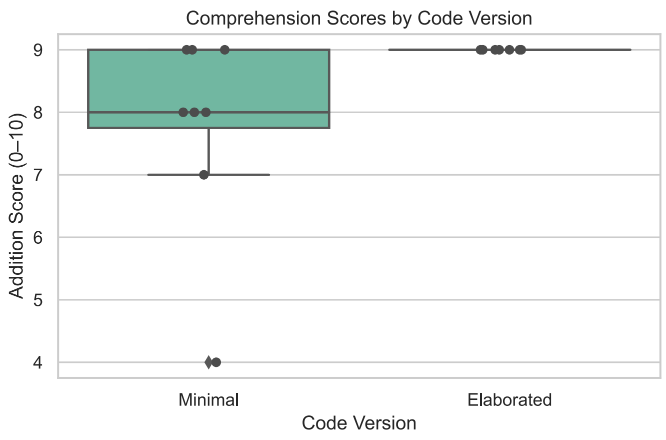

Figure 4.1: Comprehension scores by code condition. Participants working with elaborated code achieved uniformly high accuracy, whereas the minimal group exhibited lower and more variable performance.

4.1.4 Debugging Task Performance

Mean debugging scores were higher in the elaborated group ($M = 9.00$, $SD = 2.24$) than in the minimal group ($M = 6.13$, $SD = 3.87$). While the group difference did not achieve statistical significance under a parametric test ($t(13) = -1.79$, $p = 0.100$), a Mann–Whitney U test suggested marginal significance ($U = 13.5$, $p = 0.087$).

We constructed an OLS regression model incorporating code condition, Java pre-quiz, and debugging task time. The model explained 30.3% of the variance ($R^2 = 0.303$). No individual predictor was statistically significant ($p > 0.1$), though code condition ($\beta = 2.47$) showed a positive trend. Notably, task duration did not predict outcome, implying that higher scores in the elaborated group were not simply a function of more time. Figure 4.2 shows a boxplot of debugging scores across the two conditions.

4.1.5 Feature Addition Performance

Participants asked to implement a novel feature (`searchBook()`) scored higher in the elaborated group ($M = 8.71$, $SD = 2.21$) relative to the minimal group ($M = 7.00$,

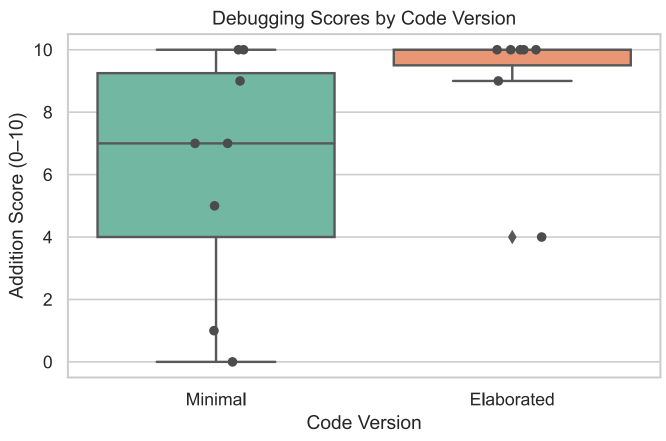

Figure 4.2: Debugging scores by code condition. Elaborated code was associated with higher mean scores and reduced variability across participants.

$SD = 3.66$). The difference was not statistically significant ($t(13) = -1.11$, $p = 0.289$). The non-parametric Mann–Whitney test similarly failed to detect significance ($U = 21.5$, $p = 0.45$).

An OLS model including code version, Java pre-quiz, and addition time explained 20.3% of the variance in addition scores ($R^2 = 0.203$), though no predictors achieved significance ($p > 0.2$). We note that feature addition, as an open-ended task, perhaps introduced greater variance and coding style variability.

Figure 4.3 shows a boxplot of feature addition scores across the two conditions.

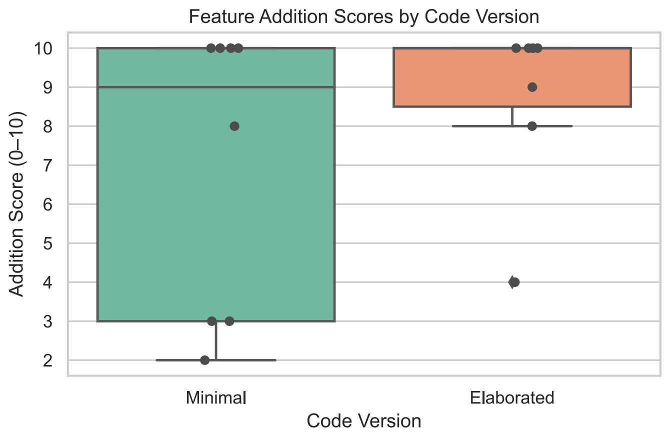

Figure 4.3: Feature addition scores by code condition. While both groups spanned a wide range of outcomes, the elaborated condition skewed toward higher correctness.

4.1.6 Efficiency Metrics

We computed task-specific efficiency as the ratio of accuracy to time (in minutes). Results indicated higher efficiency in the elaborated group across all tasks:

- **Comprehension Efficiency:** $M = 3.13$ (elaborated) vs. 2.01 (minimal)
- **Debugging Efficiency:** $M = 4.52$ vs. 1.51
- **Addition Efficiency:** $M = 1.57$ vs. 1.08

The difference in debugging efficiency approached statistical significance ($t(13) = -2.09$, $p = 0.073$), reinforcing the idea that elaboration aids not only correctness but also time-effective performance. Comprehension and addition efficiency did not show statisticially significant results.

4.1.7 Subjective Perceptions

Participants rated their experience on four 5-point Likert items assessing perceived difficulty (debugging and addition), code clarity, and confidence in correctness. Ratings ranged from 1 (extremely difficult/unclear/unsure) to 5 (extremely easy/clear/confident).

The elaborated group rated the code as significantly clearer and more understandable than the minimal group ($M = 4.57$, $SD = 1.13$ vs. $M = 3.50$, $SD = 0.76$), with a statistically significant difference ($t(13) = -3.20$, $p = 0.0073$). This aligns with our hypothesis that elaborative structure improved perceived readability within the group.

Participants in the elaborated condition also reported greater confidence in the correctness of their solutions ($M = 3.86$, $SD = 1.07$) compared to the minimal group ($M = 3.13$, $SD = 1.13$), though the difference was not statistically significant ($t(13) = -1.29$, $p = 0.2195$).

Perceived difficulty ratings for the debugging and addition tasks similarly favored the elaborated condition:

- **Debugging Difficulty:** $M = 4.14$ vs. 3.38, $t(13) = -1.34$, $p = 0.2038$
- **Addition Difficulty:** $M = 3.43$ vs. 3.13, $t(13) = -0.47$, $p = 0.6447$

While not statistically significant, these trends consistently indicate a more favorable experience with the elaborated code within the group.

4.1.8 Summary of Findings

The results of our controlled user study provide directional support for all four hypotheses, with varying degrees of statistical strength. Below, we revisit each hypothesis:

- **H1 (Comprehension)** — *Directionally supported with strong statistical and practical evidence.* Participants working with elaborated code achieved higher comprehension scores ($M = 9.00$, $SD = 0.00$) compared to those in the minimal condition ($M = 7.75$, $SD = 1.67$). A non-parametric Mann–Whitney U test found a significant difference ($p = 0.019$). While the parametric t-test result was marginal ($p = 0.072$), both accuracy and efficiency were consistently higher in the elaborated condition.

- **H2 (Debugging)** — *Directionally supported with moderate statistical evidence.* The elaborated group outperformed the minimal group on debugging accuracy ($M = 9.00$ vs. 6.13) and efficiency. Group differences approached significance in both t-test and Mann–Whitney U test ($p = 0.100$ and $p = 0.087$, respectively). Regression analysis confirmed that this performance was not attributable to longer task duration.

- **H3 (Feature Addition)** — *Directionally supported but not statistically significant.* Participants in the elaborated condition exhibited higher mean accuracy ($M = 8.71$ vs. 7.00) and efficiency, but neither t-test nor regression analyses found statistically significant differences. Greater task variability and open-endedness may have contributed to this outcome.

- **H4 (Perception)** — *Partially supported.* Participants rated elaborated code as significantly clearer ($p = 0.0073$), consistent with the hypothesis. However, other perception metrics, including confidence and perceived difficulty of debugging and addition, showed non-significant but directionally favorable differences. These trends suggest perceived benefits of elaboration within the group.

In sum, we find converging evidence that elaborated software design improved both objective and subjective programming outcomes within the group. Statistically significant gains were observed for comprehension accuracy and perceived clarity, with consistent directional trends for debugging and addition performance. These results justify a broader investigation into elaboration practices in real-world programming contexts, as undertaken in Phase 2.

4.2 Phase 2 Results

4.2.1 Overview

We applied our elaboration metric to 6,132 student-submitted programs from the PROGpedia dataset [9], spanning 16 algorithmically distinct problems written in Java and Python. Each program was statically analyzed to extract a five-dimensional elaboration feature vector $(f_1, \dots, f_5)$, from which a bounded elaboration score $E(P)$ was computed. We conducted comprehensive statistical and predictive analyses to examine the relationship between elaboration, functional correctness, programming language, and problem domain.

Table 4.2 summarizes the distribution of submissions across correctness labels and programming languages, as well as summary statistics for the five elaboration features and the computed elaboration score.

Table 4.2: Descriptive overview of the PROGpedia dataset (N = 6132)

Elaboration Feature Distributions						
Feature	**Mean**	**SD**	**Min**	**25%**	**75%**	**Max**
Elaboration Score	0.206	0.075	0.000	0.153	0.248	0.578
Comment Density	0.036	0.058	0.000	0.000	0.051	0.401
Number of Methods	3.958	4.237	0.000	2.000	4.000	30.000
Avg. Identifier Length	3.612	1.184	0.000	2.873	4.279	10.857
Doc Comment Density	0.023	0.226	0.000	0.000	0.000	6.000
Blank Line Ratio	0.177	0.092	0.000	0.109	0.232	1.000
Submission Outcome Distribution						
Accepted	2264 submissions (36.9%)					
Wrong Answer	1982 submissions (32.3%)					
Runtime Error	1886 submissions (30.8%)					
Programming Language Distribution						
Java	4362 submissions (71.1%)					
Python	1770 submissions (28.9%)					

4.2.2 Elaboration and Submission Correctness

Each solution in the dataset was labeled as `Accepted`, `Wrong Answer`, or `Runtime Error`. A Kruskal–Wallis H test revealed a statistically significant difference in elaboration scores across these three outcome categories ($H = 12.97$, $p = 0.0015$), which was corroborated by a one-way ANOVA ($F(2, 6129) = 5.95$, $p = 0.0026$).

Pairwise Mann–Whitney U tests showed that:

- `Accepted` submissions had significantly higher elaboration than `Wrong Answer` submissions ($p < 0.001$),

- `Accepted` and `Runtime Error` submissions were not significantly different ($p = 0.13$),
- `Wrong Answer` and `Runtime Error` submissions were marginally distinguishable ($p = 0.094$).

Tukey's HSD test further confirmed that the only statistically robust pairwise difference was between `Accepted` and `Wrong Answer` programs ($p = 0.0021$). These findings suggest that higher elaboration is meaningfully associated with functional correctness in the dataset, particularly in avoiding incorrect but syntactically valid solutions.

4.2.3 Language-Agnostic Robustness

Despite syntactic and stylistic differences between Java and Python, elaboration scores were consistent across languages. A two-sample t-test comparing elaboration scores by language showed no statistically significant difference ($t = -0.18$, $p = 0.86$). This indicates that the metric might maintain cross-linguistic stability and can be interpreted meaningfully in multilingual settings.

4.2.4 Predictive Modeling of Correctness

To assess whether elaboration features are predictive of functional correctness, we trained a logistic regression model to classify submissions as either `Accepted` or `Wrong Answer` using the five normalized elaboration features.

Initial models showed severe class imbalance, leading to poor recall on the minority class. Applying class balancing improved performance substantially:

- Accuracy: 71%
- Macro F_1 score: 0.67

- Confusion Matrix:
$$\begin{bmatrix} 655 & 119 \\ 235 & 218 \end{bmatrix}$$

While the model does not reach high precision on incorrect submissions, it nonetheless demonstrates that elaborative structure might contribute to correctness prediction beyond chance.

4.2.5 Feature-Level Discriminability

We conducted Kruskal–Wallis tests to examine whether each individual elaboration feature significantly differed across correctness labels. All five dimensions showed significant variation:

- **Comment density:** $H = 28.69$, $p < 0.0001$
- **Number of methods:** $H = 51.27$, $p < 0.0001$
- **Avg. identifier length:** $H = 51.09$, $p < 0.0001$
- **Doc comment density:** $H = 29.71$, $p < 0.0001$
- **Blank line ratio:** $H = 16.24$, $p = 0.0003$

These results validate that each component feature contributes a discriminative signal to the elaboration score, and that they jointly reflect meaningful variation in code elaboration within the dataset.

4.2.6 Per-Problem and Per-Concept Analysis

We examined elaboration variation across the 16 problem IDs in PROGpedia, finding substantial heterogeneity. A Kruskal–Wallis test confirmed significant differences in elaboration scores by problem ($H = 841.87$, $p < 0.0001$). To interpret these trends, we aggregated problems into broader *conceptual categories* (e.g., Greedy, MST, Graph Traversal) and re-analyzed elaboration scores by concept.

Mean elaboration scores by concept revealed that:

- **Highest elaboration:** MST (0.235), Graph Traversal (0.232), Shortest Path (0.230)
- **Lowest elaboration:** String Comparison (0.166), Backtracking (0.167), Constraint Satisfaction (0.176)

Kruskal–Wallis tests across concepts yielded $H = 690.71$, $p < 0.0001$, and post-hoc pairwise comparisons showed significant differences between many concept pairs (e.g., MST vs. Backtracking, $p < 10^{-40}$). This suggests that elaboration is shaped not just by author preference but by underlying algorithmic domain and problem framing within the dataset.

4.2.7 Summary of Findings

Through our large-scale application of the elaboration metric, we find that elaboration within the dataset is:

- **Statistically associated with correctness**, particularly in distinguishing accepted from incorrect solutions.
- **Predictive of correctness** when used as input to supervised models.
- **Stable across languages**, validating its use in multilingual corpora.
- **Driven by problem semantics**, with significant variation by algorithmic concept.

Chapter 5

Conclusion

This thesis set out to define, measure, and investigate elaboration as a measurable dimension of creativity in programming—specifically, how non-functional aspects of code such as documentation, structure, and stylistic clarity may enhance a program's comprehensibility and communicative value. While elaboration has long been theorized as a component of creative expression, it has received relatively limited attention in programming education and assessment contexts. This work offers a two-phase investigation aimed at bridging that gap.

In **Phase 1**, we designed a controlled user study to assess whether elaborative features in source code—such as modular decomposition, naming clarity, and in-code documentation—support tasks like comprehension, debugging, and feature addition. The results, though based on a small sample, suggest that elaboration may meaningfully improve both objective performance and perceived clarity. The study also provided early empirical support for treating elaboration as a distinct and pedagogically relevant dimension of code quality.

In **Phase 2**, we introduced a computational metric to approximate elaboration in code through five structural and stylistic features. Applying this metric to over 6,000 student programs from the `PROGpedia` dataset revealed several important patterns:

elaboration scores varied significantly across correctness labels and problem domains; were robust across Java and Python submissions; and contributed useful signal in classifying accepted versus incorrect solutions. These findings suggest that elaboration may be more than stylistic preference—it may hold functional and pedagogical significance.

5.0.1 Limitations

As with any exploratory research, this work has important limitations. First, the user study in Phase 1 involved only 15 participants, limiting statistical power and generalizability. Second, the elaboration metric is based entirely on static features and does not capture process-oriented elaboration (e.g., iterative refinement over time). Third, the metric's feature weights were selected heuristically based on theoretical reasoning; alternative weightings may yield different results. Finally, the dataset used in Phase 2 consists entirely of student-written code from introductory programming courses. Patterns observed here may not generalize to more advanced or professional programming contexts.

It is also worth noting that the operational definition of elaboration used in this thesis—focused on visibility and communicative clarity—represents just one possible interpretation. Other valid framings may prioritize different qualities, such as expressiveness, novelty, or functionality. This work offers an initial framing that we hope will support continued refinement in future research.

5.0.2 Future Work

Several extensions of this work are possible. On the technical side, future iterations of the metric could incorporate additional features, such as exception handling, user interaction, or code reuse patterns. Incorporating version control data could also enable process-aware elaboration metrics that reflect how code evolves over time.

Expanding the analysis to multilingual, professional, or open-source datasets would allow for broader validation across domains.

On the pedagogical and research side, future work could involve human-centered evaluations, such as benchmarking the elaboration score against expert or peer judgment. Experimental studies could also assess whether explicitly teaching elaborative practices improves code quality and learner outcomes. Longitudinal studies might further examine how elaboration evolves across a student's academic trajectory—for instance, comparing elaboration in freshman-level coursework versus senior capstone projects.

Finally, embedding elaboration-aware feedback tools into instructional platforms may offer new opportunities to support expressiveness and clarity in student code at scale.

Appendix A

Source Code

All source code for this thesis, including the elaboration metric implementation, feature extraction pipeline, user study materials, and statistical analysis, is available at:

GitHub Repository:

`https://github.com/vidhimittal13/creative-elaboration`

Appendix B

Full Survey Instrument

This appendix contains the complete survey instrument administered to participants. The survey was implemented using Qualtrics and delivered in five structured sections corresponding to participant background, Java knowledge, comprehension, debugging, feature implementation, and post-study reflections.

B.1 Anonymous Participant ID

Participants created an anonymous ID by concatenating the last four digits of their phone number with a randomly selected word from the following list: *Sky, Tree, Book, Moon, Fire, Wave, Rock, Wind, Cloud, Echo.*

Example: If the digits were 2764 and the word was Tree, the resulting ID would be `2764Tree`.

B.2 Background Information

- Academic major: (CS, Data Science, Engineering, Math/Stats, Business, Other)
- Year of study: (First-Year Undergraduate to PhD Student)

B.3 Java Experience & Background

- Prior experience with Java: (Yes/No)
- Highest level of Java experience:
  - Introductory Java course (e.g., CS 170, AP CS)
  - Advanced coursework (e.g., CS 253, CS 255)
  - Personal/open-source projects
  - Professional experience
  - Self-taught, no formal coursework
- Years of Java experience: (≤ 1, 1–2, 3–5, ≥ 5)

B.4 Java Knowledge Check

Participants answered the following multiple-choice questions:

1. **What is the output of the following code?**

   ```
   int x = 5;
   int y = 2;
   double z = x / y;
   System.out.println(z);
   ```

   - A) 2
   - B) 2.5
   - C) 2.0
   - D) 3.0

2. **Which Java data structure is best suited for key-value pairs?**

- A) Array
- B) ArrayList
- C) HashMap
- D) TreeSet

3. **What is the correct entry point for a Java application?**

   - A) `public void main(String[] args)`
   - B) `public static int main(String[] args)`
   - C) `public static void main(String[] args)`
   - D) `public static main(String args)`

4. **Which of the following loops prints numbers 1 to 5 (inclusive)?**

   - A) `for (int i = 1; i <= 5; i++) { System.out.println(i); }`
   - B) `for (int i = 1; i < 5; i++) { System.out.println(i); }`
   - C) `while (true) { System.out.println(i); }`
   - D) `for (int i = 1; i >= 5; i++) { System.out.println(i); }`

5. **What will `System.out.println(RecordStore.get("Short n Sweet"))` output if the key does not exist?**

   - A) true
   - B) false
   - C) null
   - D) an error

B.5 Self-Assessment

Participants rated their confidence on a 4-point Likert scale (Not at all confident – Very confident):

- Reading and understanding Java code
- Debugging Java code

Participants also answered:

- Have you worked with code that required modifying existing functions instead of writing from scratch? (Yes, frequently / Yes, occasionally / No)

B.6 Code Comprehension

After reviewing their assigned code for 5 minutes, participants answered the following:

1. What data structure is used to store the books?
2. What does the Boolean value represent?
3. What happens when a book is added that already exists?
4. What message is displayed when trying to borrow a checked-out book?
5. What message is shown when attempting to borrow/return a book not in the system?
6. What best describes how the program handles user input?
7. Trace exercise:
    - Choose 2, type "1984" → What is printed?
    - Choose 2 again, type "1984" → What is printed?
    - Choose 3, type "Gone Girl" → What is printed?

B.7 Debugging Task

Participants received a bugged program version and completed the following under a 7-minute time limit:

1. Explain what happens when a book is removed and why it is incorrect.
2. Copy/paste the exact buggy line(s).
3. Provide corrected code.

B.8 Feature Addition Task

Task: Add a new feature to allow users to search for a book.

Requirements:

- Add a new menu option: Option 6 – "Search for a Book"
- Prompt user for title
- Output:
  - "Book found: [title] – Available"
  - "Book found: [title] – Checked Out"
  - "Book not found in catalog."

Participants submitted the full modified code within a 10-minute time limit.

B.9 Post-Study Perception Survey

Participants rated the following using 5-point Likert scales:

- How difficult was the debugging task?

- How difficult was the feature addition task?
- How clearly were you able to understand the code before making changes?
- How confident are you that your solutions were correct?

Appendix C

Code Stimuli

C.1 Minimal Version (MinimalLibrary.java)

```java
import java.util.*;

public class MinimalLibrary {
    private HashMap<String, Boolean> b = new HashMap<>();

    public static void main(String[] args) {
        MinimalLibrary m = new MinimalLibrary();
        m.b.put("The Kite Runner", true);
        m.b.put("1984", true);
        m.b.put("The Hunger Games", false);

        Scanner sc = new Scanner(System.in);
        while(true){
            System.out.println("1-Add 2-Borrow 3-Return 4-List
                5-Exit");
            int c = sc.nextInt();
            sc.nextLine();
            if(c==1){
                String t = sc.nextLine();
                m.b.put(t,true);
            } else if(c==2){
                String t = sc.nextLine();
                if(m.b.containsKey(t) && m.b.get(t)){
                    m.b.put(t,false);
                    System.out.println("Borrowed.");
                } else {
                    System.out.println("Not available.");
                }
            } else if(c==3){
                String t = sc.nextLine();
                if(m.b.containsKey(t)){
                    m.b.put(t,true);
                    System.out.println("Returned.");
                } else {
                    System.out.println("Not found.");
                }
            } else if(c==4){
                for(String t : m.b.keySet()){
                    System.out.println(t + " - " + (m.b.get(t)
                        ? "Available" : "Checked Out"));
                }
            } else if(c==5){
                break;
            }
        }
        sc.close();
    }
}
```

C.2 Elaborated Version (ElaboratedLibrary.java)

```
import java.util.*;
/**
 * A Library Management System
 *
 * This class allows the user to:
 *
 *   - Add a new book to the system
 *   - Borrow an available book
 *   - Return a previously borrowed book
 *   - List all books with their current status
 *
 *
 * Implementation Details:
 *
 *   We use a HashMap to store book titles (String) mapped to
    a Boolean
 *   indicating availability (true = available, false =
    checked out).
 *
 */
public class ElaboratedLibrary
{
    /**
     * A HashMap storing each book's title as the key
     * and a Boolean indicating availability as the value.
     * True means available; false means checked out.
     */
    private HashMap<String, Boolean> books;

    /**
     * Constructs a new ElaboratedLibrary object with a few
        default books.
     */
    public ElaboratedLibrary()
    {
        books = new HashMap<>();
        // Adding initial sample books
        books.put("The Kite Runner", true);
        books.put("1984", true);
        books.put("The Hunger Games", false);
    }

    /**
     * Adds a book to the system, making it immediately
        available.
     *
     * @param title The exact title of the book to add.
     *              If this title already exists, it will
        simply be marked as available.
     */
    public void addBook(String title)
    {
        books.put(title, true);
        System.out.println("Book added (or updated): " + title);
    }

    /**
     * Attempts to borrow a book if it is currently available.
     * If the book does not exist or is already checked out,
     * an appropriate message is displayed.
     *
     * @param title The title of the book the user wishes to borrow.
     */
    public void borrowBook(String title)
    {
        // First, check if the book is in the system
        if (books.containsKey(title))
        {
            // Retrieve availability
            boolean isAvailable = books.get(title);
            if (isAvailable)
            {
                books.put(title, false);
                System.out.println("You borrowed: " + title);
            }
            else
            {
                System.out.println("Not available. The book is already borrowed.");
            }
        }
        else
        {
            System.out.println("Book not found in the catalog.");
        }
    }

    /**
     * Returns a previously borrowed book, marking it as available again.
     * If the book doesn't exist in the system, a message is displayed.
     *
     * @param title The title of the book being returned.
     */
    public void returnBook(String title)
    {
        if (books.containsKey(title))
        {
            books.put(title, true);
            System.out.println("You returned: " + title);
        }
        else
        {
```

```java
            System.out.println("Book not found in the catalog.");
        }
    }

    /**
     * Prints out the entire book collection with each title's availability status.
     * The listing includes every known title in the system.
     */
    public void listBooks()
    {
        System.out.println("\n========== Library Catalog ==========");
        for (String title : books.keySet())
        {
            boolean isAvailable = books.get(title);
            String status = isAvailable ? "Available" : "Checked Out";
            System.out.println("- " + title + " - " + status);
        }
        System.out.println("=====================================");
    }

    /**
     * Runs a loop that presents menu options to the user for
     * adding, borrowing, returning, or listing books.
     * Entering '5' exits the program.
     */
    public void run()
    {
        Scanner sc = new Scanner(System.in);
        while (true)
        {
            System.out.println(
                "\nPlease select an action: \n" +
                "1 - Add a Book\n" +
                "2 - Borrow a Book\n" +
                "3 - Return a Book\n" +
                "4 - List All Books\n" +
                "5 - Exit the Program"
            );
            int choice = sc.nextInt();
            sc.nextLine();

            switch (choice)
            {
                case 1:
                    System.out.println("Enter the title of the
                        book to add:");
                    String addTitle = sc.nextLine();
                    addBook(addTitle);
                    break;
                case 2:
                    System.out.println("Enter the title of the
                        book to borrow:");
                    String borrowTitle = sc.nextLine();
                    borrowBook(borrowTitle);
                    break;
                case 3:
                    System.out.println("Enter the title of the book to return:");
                    String returnTitle = sc.nextLine();
                    returnBook(returnTitle);
                    break;
                case 4:
                    listBooks();
                    break;
                case 5:
                    System.out.println("Exiting the Library System. Goodbye!");
                    sc.close();
                    return;
                default:
                    System.out.println("Invalid choice. Please try again.");
            }
        }
    }

    /**
     * The main entry point. Creates an instance of the ElaboratedLibrary
     * class and starts the interactive menu.
     *
     * @param args Command-line arguments (not used in this
        application).
     */
    public static void main(String[] args)
    {
        ElaboratedLibrary library = new ElaboratedLibrary();
        library.run();
    }
}
```

C.3 Minimal Bugged Version (MinimalLibraryBugged.java)

```
import java.util.*;

public class MinimalLibraryBugged {
    private HashMap<String, Boolean> b = new HashMap<>();

    public static void main(String[] args) {
        MinimalLibraryBugged m = new MinimalLibraryBugged();
        m.b.put("The Kite Runner", true);
        m.b.put("1984", true);
        m.b.put("The Hunger Games", false);

        Scanner sc = new Scanner(System.in);
        while(true){
            System.out.println("1-Add 2-Borrow 3-Return 4-List
                5-Exit 6-Remove");
            int c = sc.nextInt();
            sc.nextLine();

            if(c == 1){
                String t = sc.nextLine();
                m.b.put(t, true);
            } else if(c == 2){
                String t = sc.nextLine();
                if(m.b.containsKey(t) && m.b.get(t)){
                    m.b.put(t, false);
                    System.out.println("Borrowed.");
                } else {
                    System.out.println("Not available.");
                }
            } else if(c == 3){
                String t = sc.nextLine();
                if(m.b.containsKey(t)){
                    m.b.put(t, true);
                    System.out.println("Returned.");
                } else {
                    System.out.println("Not found.");
                }
            } else if(c == 4){
                for(String t : m.b.keySet()){
                    System.out.println(t + " - "
                        + (m.b.get(t) ? "Available" : "Checked
                            Out"));
                }
            } else if(c == 5){
                break;
            } else if(c == 6){
                String t = sc.nextLine();
                if(m.b.containsKey(t)){
                    m.b.put(t, false);
                    System.out.println("Removed book.");
                } else {
                    System.out.println("Not found.");
                }
            }
        }
        sc.close();
    }
}
```

C.4 Elaborated Bugged Version (ElaboratedLibraryBugged.java)

```
import java.util.*;

/**
 * A Library Management System
 *
 * This class allows the user to:
 *
 *   - Add a new book to the system
 *   - Borrow an available book
 *   - Return a previously borrowed book
 *   - List all books with their current status
 *   - Remove a book (Buggy)
 *
 * Implementation Details:
 *
 *   We use a HashMap to store book titles (String) mapped to
    a Boolean
 *   indicating availability (true = available, false =
    checked out).
 *
 */
public class ElaboratedLibraryBugged
{
    /**
     * A HashMap storing each book's title as the key
     * and a Boolean indicating availability as the value.
     * True means available; false means checked out.
     */
    private HashMap<String, Boolean> books;

    /**
     * Constructs a new ElaboratedLibraryBugged object
     * with a few default books added.
     */
    public ElaboratedLibraryBugged()
    {
        books = new HashMap<>();
        // Adding initial sample books
        books.put("The Kite Runner", true);
        books.put("1984", true);
        books.put("The Hunger Games", false);
    }

    /**
     * Adds a book to the system, making it immediately
        available.
     *
     * @param title The exact title of the book to add.
     *              If this title already exists, it will
        simply be marked as available.
     */
    public void addBook(String title)
    {
        books.put(title, true);
        System.out.println("Book added (or updated): " + title
           );
    }

    /**
     * Attempts to borrow a book if it is currently available.
     * If the book does not exist or is already checked out,
     * an appropriate message is displayed.
     *
     * @param title The title of the book the user wishes to
        borrow.
     */
    public void borrowBook(String title)
    {
        // First, check if the book is in the system
```

```java
        if (books.containsKey(title))
        {
            // Retrieve availability
            boolean isAvailable = books.get(title);
            if (isAvailable)
            {
                books.put(title, false);
                System.out.println("You borrowed: " + title);
            }
            else
            {
                System.out.println("Not available. The book is
                    already borrowed.");
            }
        }
        else
        {
            System.out.println("Book not found in the catalog.
               ");
        }
    }

    /**
     * Returns a previously borrowed book, marking it as
        available again.
     * If the book doesn't exist in the system, a message is
        displayed.
     *
     * @param title The title of the book being returned.
     */
    public void returnBook(String title)
    {
        if (books.containsKey(title))
        {
            books.put(title, true);
            System.out.println("You returned: " + title);
        }
        else
        {
            System.out.println("Book not found in the catalog.
               ");
        }
    }

    /**
     * Prints out the entire book collection with each title's
         availability status.
     * The listing includes every known title in the system.
     */
    public void listBooks()
    {
        System.out.println("\n========== Library Catalog
           ==========");
        for (String title : books.keySet())
        {
            boolean isAvailable = books.get(title);
            String status = isAvailable ? "Available" : "
               Checked Out";
            System.out.println("- " + title + " - " + status);
        }
        System.out.println("
            ====================================");
    }

    /**
     * (Buggy) Removes a book from the system.
     * @param title The exact title of the book to remove.
     */
    public void removeBook(String title)
    {
        if (books.containsKey(title))
        {
            books.put(title, false);
            System.out.println("Removed book: " + title);
        }
        else
        {
            System.out.println("Book not found in the catalog.
                ");
        }
    }

    /**
     * Runs a loop that presents menu options to the user for
     * adding, borrowing, returning, listing, or removing
        books.
     * Entering '5' exits the program.
     */
    public void run()
    {
        Scanner sc = new Scanner(System.in);
        while (true)
        {
            System.out.println(
                "\nPlease select an action: \n" +
                "1 - Add a Book\n" +
                "2 - Borrow a Book\n" +
                "3 - Return a Book\n" +
                "4 - List All Books\n" +
                "5 - Exit the Program\n" +
                "6 - Remove a Book (Buggy)"
            );
            int choice = sc.nextInt();
            sc.nextLine();

            switch (choice)
            {
                case 1:
                    System.out.println("Enter the title of the
                        book to add:");
                    String addTitle = sc.nextLine();
                    addBook(addTitle);
                    break;
                case 2:
                    System.out.println("Enter the title of the
                        book to borrow:");
```

```java
                    String borrowTitle = sc.nextLine();
                    borrowBook(borrowTitle);
                    break;
                case 3:
                    System.out.println("Enter the title of the book to return:");
                    String returnTitle = sc.nextLine();
                    returnBook(returnTitle);
                    break;
                case 4:
                    listBooks();
                    break;
                case 5:
                    System.out.println("Exiting the Library System. Goodbye!");
                    sc.close();
                    return;
                case 6:
                    System.out.println("Enter the title of the book to remove:");
                    String removeTitle = sc.nextLine();
                    removeBook(removeTitle);
                    break;
                default:
                    System.out.println("Invalid choice. Please try again.");
            }
        }
    }

    /**
     * The main entry point. Creates an instance of the
        ElaboratedLibraryBugged
     * class and starts the interactive menu.
     *
     * @param args Command-line arguments (not used in this
        application).
     */
    public static void main(String[] args)
    {
        ElaboratedLibraryBugged library = new
           ElaboratedLibraryBugged();
        library.run();
    }
}
```

Bibliography

[1] Teresa M. Amabile. The social psychology of creativity: A componential conceptualization. *Journal of Personality and Social Psychology*, 45(2):357–376, 1983. doi: 10.1037/0022-3514.45.2.357.

[2] Karen Brennan and Mitchel Resnick. New frameworks for studying and assessing the development of computational thinking. *MIT Media Lab*, pages 1–25, 2012.

[3] College Board. Ap computer science principles, 2025. URL `https://apstudents.collegeboard.org/courses/ap-computer-science-principles`. Accessed: 2025-04-02.

[4] Shuchi Grover, Roy Pea, and Stephen Cooper. Designing for deeper learning in a blended computer science course for middle school students. *Computer Science Education*, 25, 04 2015. doi: 10.1080/08993408.2015.1033142.

[5] J. P. Guilford. The structure of intellect. *Psychological Bulletin*, 53(4):267–293, 1956. doi: 10.1037/h0040755.

[6] Rotem Israel-Fishelson and Arnon Hershkovitz. Log-based analysis of creativity in the context of computational thinking. *Education Sciences*, 15(1), 2025. ISSN 2227-7102. doi: 10.3390/educsci15010003. URL `https://www.mdpi.com/2227-7102/15/1/3`.

[7] K–12 Computer Science Framework Steering Committee. K–12 computer science framework, 2016. URL `https://k12cs.org/`. Accessed: 2025-04-02.

[8] K. O'Quin and S. P. Besemer. The development, reliability, and validity of the revised creative product semantic scale. *Creativity Research Journal*, 2(4): 267–278, 1989. doi: 10.1080/10400418909534323.

[9] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. Progpedia: Collection of source-code submitted to introductory programming assignments. *Data in Brief*, 46:108887, 2023. ISSN 2352-3409. doi: https://doi.org/10.1016/j.dib.2023.108887. URL `https://www.sciencedirect.com/science/article/pii/S2352340923000057`.

[10] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: programming for all. *Commun. ACM*, 52(11):60–67, November 2009. ISSN 0001-0782. doi: 10.1145/1592761.1592779. URL `https://doi.org/10.1145/1592761.1592779`.

[11] Margarida Romero, Mireia Usart, and Michela Ott. Can serious games contribute to developing and sustaining 21st century skills? *Games and Culture*, 10(2): 148–177, 2015. doi: 10.1177/1555412014548919. URL `https://doi.org/10.1177/1555412014548919`.

[12] Ellis Paul Torrance. *Torrance Tests of Creative Thinking*. Personnel Press, 1974.