INSTITUTO
UNIVERSITÁRIO
DE LISBOA

Questions About Learners' Code Focusing on Semantic Style

Gonçalo Miguel Costa Serrano

Master in Computer Engineering

Supervisor:
PhD André Leal Santos, Assistant Professor,
Iscte - Instituto Universitário de Lisboa

October, 2025

TECHNOLOGY
AND ARCHITECTURE

Department of Information Science and Technology

Questions About Learners' Code Focusing on Semantic Style

Gonçalo Miguel Costa Serrano

Master in Computer Engineering

Supervisor:
PhD André Leal Santos, Assistant Professor,
Iscte - Instituto Universitário de Lisboa

October, 2025

Acknowledgments

I would like to express my gratitude to my supervisor, PhD André Leal Santos, for his guidance and availability throughout the development of this dissertation. I am also grateful to Afonso Caniço, PhD student at Iscte, for his support and assistance, which contributed greatly to the progress of this work.

As English is not my first language, OpenAI's ChatGPT[1] was used as a tool to improve clarity and grammar in the writing of this dissertation. All ideas, analyses, and conclusions presented herein remain entirely my own.

[1] https://openai.com/index/chatgpt/

Resumo

Aprender a programar continua a ser uma dificuldade central no ensino de informática. Embora programadores principiantes sejam capazes de criar código funcionalmente correto, as suas soluções expõem falta de conhecimento. Os sistemas automáticos de avaliação existentes avaliam principalmente a exatidão e a eficiência, oferecendo pouco ou nenhum suporte à reflexão sobre qualidade de código.

Esta dissertação apresenta uma extensão da biblioteca Jask através de um catálogo de Questões sobre código desenvolvido (QLC) focadas em problemas de semântica em Java. O catálogo define modelos predefinidos para problemas comuns como condições redundantes, variáveis não utilizadas, parâmetros desnecessários, e duplicações. Cada modelo gera questões de escolha múltipla fazendo refactoring do código dos alunos e criando alternativas incorretas, permitindo a geração automática de questões diretamente de submissões.

Para avaliar a aplicabilidade, foram analisadas submissões de alunos da unidade curricular de Introdução à Programação. Das submissões consideradas, 11,5% continham métodos com problemas de qualidade identificáveis. Os dois problemas mais comuns foram a comparação desnecessária a `true` ou `false` e o retorno indireto da condição do if, cada um afetando aproximadamente metade dos alunos.

Os resultados demonstram a viabilidade de transformar problemas de qualidade de código em questões estruturadas e fornecem uma visão geral da sua frequência na programação de principiantes. Embora as questões geradas não tenham sido aplicadas a estudantes, os resultados estabelecem uma base para a sua futura integração em sistemas de avaliação e para investigações adicionais sobre o apoio à reflexão no ensino da programação.

Palavras-Chave: *qualidade de código; questões sobre codigo desenvolvido; refactoring; educação de programação*

Abstract

Learning to program remains a central difficulty in computer science education. Although novice programmers can often produce functionally correct code, their solutions frequently expose misconceptions. Existing automated grading systems primarily assess correctness and efficiency, offering limited or no support for reflection on code quality.

This dissertation presents an extension of the Jask library through a catalog of Questions about Learners' Code (QLC) focused on semantic style issues in Java. The catalog defines templates for common problems such as redundant conditionals, unused variables, unnecessary parameters, and duplications. Each template generates multiple-choice questions by refactoring student code and producing distractors, enabling automated question generation directly from submissions.

To evaluate the approach's applicability, student submissions from an introductory programming course were analyzed. Of the relevant cases, 11.5% contained methods with identifiable quality issues. The two most common issues were the unnecessary comparison to `true` or `false` and the indirect return of the if condition, each affecting nearly half of the students.

The results demonstrate the feasibility of transforming code quality issues into structured questions and provide an overview of their frequency in novice programming. Although the generated questions were not administered to students, the findings establish a basis for their future integration into assessment systems and for further research on supporting reflection in programming education.

Keywords: *code quality; questions about learner's code; refactoring; programming education*

Contents

List of Figures

List of Tables

List of Acronyms

QLC: Question about Learners' Code

AST: Abstract Syntax Tree

CHAPTER 1

Introduction

1.1. Context and Motivation

Learning to program is a central challenge in computer science education, and research has consistently shown that novice programmers struggle not only with problem-solving but also with mastering the basic programming primitives required to construct reliable code. While many students are able to eventually produce functional solutions, these are often developed in ways that reveal fragile understanding, trial-and-error strategies, and persistent misconceptions [**1**–**8**].

Some programming courses rely on automated grading systems that assess submissions in terms of functionality or, in some cases, efficiency [**2**, **5**]. These systems provide almost immediate feedback, allowing students to iteratively correct their solutions [**4**]. In principle, this feedback should help learners to identify and resolve their mistakes. However, studies have shown that many novices engage with these systems superficially, either by repeatedly tweaking fragments of code until the tests pass or by relying on external resources without developing their own reasoning skills [**2**,**4**,**5**]. In recent years, this has been further amplified by the growing availability of AI-based assistants such as ChatGPT, which can generate working solutions on demand [**9**]. While such tools are increasingly present in educational contexts, their use without reflection risks reinforcing shallow learning and preventing students from developing essential debugging and problem-solving abilities [**9**–**11**].

Novice programmers tend to struggle with both the underlying concepts of programming and the syntactic and structural skills required to express them effectively [**6**, **12**]. This combination frequently leads to issues such as unused variables, unnecessary conditional statements, or similar structural problems [**3**, **13**–**15**]. Such patterns are not merely superficial mistakes, they are symptomatic of deeper misconceptions that have been widely documented in programming education. Addressing them therefore requires assessment approaches that make underlying reasoning visible, rather than focusing exclusively on whether the program's output is correct [**3**, **13**, **14**].

To address these limitations, this work explores the use of QLCs [**5**]. The central idea is to challenge students not only to write code but also to reflect on it by answering targeted questions generated directly from their own submissions. By focusing on learners' own code, this approach emphasizes understanding the reasoning behind program structure rather than simply checking for correct output [**2**, **4**, **5**, **16**].

1.2. Research Questions

The primary goal of this project is to investigate the viability of the automatic generation of multiple-choice questions related to the quality of the programming students' code. We will achieve this by developing a library capable of analyzing code to create questions about both the structure and behavior of the analyzed code.

In addition to demonstrating feasibility, this work also seeks to characterize the kinds of quality issues that can be effectively identified through automatically generated QLCs, as well as their prevalence across a large dataset of novice programmers' submissions. By examining these patterns, the study provides insights into the typical issues made by beginners and highlights which types of code quality problems are most relevant for supporting automated question generation.

To this effect, this dissertation aims to answer the following research questions:

RQ1 How can we automatically generate multiple-choice questions that target semantic style issues in novices' Java code?

RQ2 How frequently do the semantic style issues defined in (**RQ1**) appear in a large-scale introductory programming course in Java?

1.3. Document Structure

This dissertation is organized into six chapters, in addition to the references. Chapter 1 introduced the context and motivation of the work, defined the research questions, and provides an overview of the dissertation's organization. Chapter 2 reviews related work, focusing on common misconceptions in programming, automated feedback and refactoring tools, and previous research on Questions about Learners' Code (QLCs). Chapter 3 presents the created catalog of QLCs on semantic style, describing the question templates developed in this study and illustrating with examples. Chapter 4 describes the implementation of the QLC library, explaining its architecture, how QLCs are generated from student code, and how the library can be integrated. Chapter 5 details the study carried out to evaluate the applicability of the proposed approach, presenting the methodology used, the results obtained, and their implications. Finally, Chapter 6 concludes the dissertation by summarizing its contributions, highlighting limitations, and suggesting directions for future work.

CHAPTER 2

Related Work

2.1. Student Misconceptions

Introductory programming can be challenging for students learning to code for the first time. Frequently, novice programmers acquire misunderstandings related to how programming works [**1**, **6**, **7**]. Even though students can create functional code, they aren't always able to explain why it works, because they don't fully understand what they did [**1**, **7**].

Some students can only reach a correct solution by using less adequate options, such as copying from other colleagues or the internet, or even by guessing by trial-and-error [**5**]. This process ensures that students don't reflect on what they did, opening space for misconceptions [**5**]. Many factors can contribute to these misconceptions and other difficulties, such as task complexity, their native language, and understanding of logic [**6**].

A symptom of these misconceptions is semantic style issues. We can define *semantic style* as the style programmers use when following syntax guidelines designed to organize code better and prevent future problems [**1**, **13**]. An example of a Semantic Style Issue is given in Figure 2.1, where the instructions can be easily replaced with "`return condition;`" (Taken from [**13**]).

```
if(condition){
     return true;
} else {
    return false;
}
```

FIGURE 2.1. Example of Semantic Style Issue

Some of these semantic style issues appear more often in novice programmers, and these are more easily fixed using code refactoring. Martin Fowler described refactoring as "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" [**17**]. For example [**1**, **18**]:

- **IF Return Condition**: As mentioned before, when you can replace that code snippet with "`return condition;`" or "`return !condition;`".
- **Empty IF Body**: An IF statement, with a given condition, contains instructions only in its ELSE statement. One can easily replace it with an IF statement that uses the negated condition and the same body as the ELSE.

- **Unnecessary Nesting IF**: When an IF is directly inside another IF, both without an ELSE statement. An IF statement with a conjunction of the two conditions can replace this.
- **Iterated Useless Duplication in IF/ELSE**: When inside the bodies of both the IF and the ELSE, there are equal segments of code. We can move the duplicated code outside of the IF/ELSE.
- **Unnecessary Cast**: In an instruction, when there is a Cast that is not needed, for example: "`int product = (int)(items*10);`".
- **Lonely Variable**: A variable that is initialized but never used. One can delete it.

2.2. Automated Feedback and Refactoring

Several tools have been developed with the aim of addressing misconceptions among novice programmers through automated feedback and guided refactoring. "Sprinter: A Didactic Linter for Structured Programming" [**19**] illustrates how linting can be adapted for educational purposes. Unlike conventional linters that only report violations, Sprinter provides structured explanations of why certain constructs are undesirable, helping students relate issues in their code to underlying programming principles. Its focus on explanation rather than correction makes it suitable for introductory programming contexts, where misconceptions often stem from misunderstandings of fundamental structural and stylistic conventions.

Other approaches adopt interactive tutoring mechanisms. For example, "A Tutoring System to Learn Code Refactoring" [**20**] engages learners in the stepwise improvement of programs by providing hints that make the benefits of refactoring explicit. Similarly, "RefacTutor" [**21**] guides students through the resolution of common code smells, offering refactored alternatives that can be compared against their original solutions. Both systems encourage learners to reflect on the relationship between code structure, readability, and maintainability, which helps mitigate the risk of treating refactoring as a purely mechanical process.

In addition to standalone systems, static analyzers, linters and IDE-integrated refactoring tools, such as those available in Eclipse [**22**] or IntelliJ [**23**], also support the development of cleaner code [**24**–**26**]. However, these tools are generally designed for professional use and assume a level of prior knowledge that novices may lack . When integrated into instruction with sufficient scaffolding, such tools—whether didactic linters or tutoring systems—can contribute to the reduction of misconceptions and the development of programming best practices [**24**, **25**, **27**–**30**]. While there is a risk that students may rely excessively on automated suggestions, prior work indicates that structured engagement with refactoring tools has predominantly positive effects on learning outcomes [**31**].

2.3. Questions about Learners' Code (QLCs)

There are several ways to address the discrepancy between the correct execution of code and the student's deep understanding of it, such as asking students to explain their code line-by-line in plain language, challenging them to modify working code to achieve a similar goal, and asking them to fix broken code, among others.

One way to address it is by implementing QLCs as a strategy to foster self-reflection and students' understanding of their work. QLCs can be defined as specific questions referencing elements of the student's code. This approach confronts students with questions that challenge them to think critically about the decisions they made while developing their solutions [**5**].

Integrating QLCs into automated assessment systems could result in an advancement in education. It shifts the focus from evaluating code based only on functionality to promoting conceptual comprehension [**5**]. Ideally, after successful unit testing of the code, the system would prompt students with questions designed to encourage self-reflection of their solution. This updated format would complement the traditional assessment methodology and would result in the students becoming more aware of their difficulties.

It is important to recognize that the successful implementation of QLCs can vary depending on the student's level of comprehension and the relevance of the generated questions [**2**,**16**]. Therefore, it is essential that the automatic generation of QLCs considers the student's contexts, ensuring that the questions are pertinent and capable of stimulating critical thinking.

QLCs were first proposed in 2021 to "provide a window into students' program comprehension" [**5**]. Subsequently, in 2022, the feasibility of this concept was tested through the development of a system capable of generating questions about students' Java code, both through static analysis (example of a QLC in Figure 2.2, where we see a statement asking if a function is recursive, followed by the given function, submitted by the student, and two possible answers: "Yes." and "No.") and dynamic analysis (example in Figure 2.3, where we see another QLC, this time asking which return statement is executed when calling a function with the given arguments, followed by the submitted code and two possible answers, each indicating one of the two return statements present in the function), named Jask [**4**]. Jask demonstrated the technical viability of such a system by integrating two complementary libraries: Strudel[1] [**32**] and JavaParser [**33**]. Strudel supports dynamic code analysis by inspecting runtime behavior, including variable states during execution and the outputs produced for given inputs. JavaParser [**33**] enables the analysis, parsing, and structured transformation of Java source code. A follow-up questionnaire revealed that the students responded positively to the approach and expressed interest in its potential integration into the course [**4**].

[1] `https://github.com/andre-santos-pt/strudel`

Static Question

Is the function [factorial] recursive?

```
static int factorial(int n) {
    if (n == 0) return 1;
    else return n * factorial(n - 1);
}
```

[x] Yes.
[ ] No.

FIGURE 2.2. Example of a Static Question

Dynamic Question

Which return statement (line) executed when calling [abs(-2.0)]?

```
static double abs(double n) {
    if (n < 0)
        return -n;
    else
        return n;
}
```

[x] Line 3
[ ] Line 5

FIGURE 2.3. Example of a Dynamic Question

A related study tasked students with answering questions about their code. The results indicated that students who struggled with this task ultimately achieved a lower median course score compared to those who answered correctly. This finding suggests that such a method could serve as a potential tool for student evaluation [**2**].

The potential of QLCs was tested in other studies [**2**, **4**, **5**, **16**] by developing a set of preliminary questions with the aim of evaluating their relevance within the learning context of introductory programming students. Among these, examples included:

- "Which is the name of the function [declared on line n]?" [**2**];
- "A program loop starts on line n. Which is the last line inside it?" [**2**, **5**];
- "Which best describes property-name on line n?" [**2**, **5**];
- "Which of the following are variable names in the program?" [**16**];
- "Which of the following best describes the purpose of line n?" [**16**];

- "Does function [f] depend on other functions?" [**4**];
- "Which are the fixed value variables of function [f]?" [**4**];
- "How many times is variable [v] assigned when calling [f(arg1, arg2, ...)]?" [**4**];
- "During the program execution, how many iterations are performed by the loop starting in line N]?" [**5**].

From these initial explorations [**2**, **4**, **5**, **16**], it became clear that QLCs can indeed be applied effectively. Nevertheless, in the scope of this work, the focus is restricted to QLCs related specifically to code quality, as these are particularly valuable for uncovering misconceptions and encouraging reflection on good programming practices, some examples of quality issues are:

- "Assignment of a Value that is Not Used": A variable in assigned to a value but it's not used after [**19**](the example in Figure 2.4 could be replaced to a simple "`int sum = a + b;`");

```
[...]
int sum = 0;
sum = a + b;
[...]
```

FIGURE 2.4. Example of an Assignment of a Value that is Not Used

- "Identity Return": An if-else where both branches return a boolean can be simplified to a single return of the condition itself [**19**](the example in Figure 2.5 can be refactored to "`return condition;`");

```
[...]
if(condition){
    return true;
}else{
    return false;
}
[...]
```

FIGURE 2.5. Example of an Identity Return

- "Comparing a boolean expression to a boolean literal": A boolean expression that is explicitly compared to a boolean literal (true or false) [**19**, **20**](the if condition in Figure 2.6 can be replaced with "`if(condition){...`");

```
[...]
if(condition == true){
    //body
}
[...]
```

FIGURE 2.6. Example of Comparing a boolean expression to a boolean literal

- "Common Behavior in Alternative Branches": An if–else construct whose branches contain identical statements that could be factored out **[19, 20]**(the statement "`return a;`" in Figure 2.7 can be factored out of the branches);

```
[...]
if(condition){
    a = 2;
    return a;
}else{
    a = 3;
    return a;
}
[...]
```

FIGURE 2.7. Example of Common Behavior in Alternative Branches

- "Empty Block in If with Desired Behavior in Else": An if–else construct where the if branch is empty and the behavior is in the else branch, equivalent to an if with the negated condition **[19]**(the if-else construct in Figure 2.8 can be changed to an if with the negated condition containing the desired behavior "`if(!condition){//desired behavior}`");

```
[...]
if(condition){
}else{
    //desired behavior
}
[...]
```

FIGURE 2.8. Example of an Empty Block in If with Desired Behavior in Else

- "Collapsible Nested If": An if nested inside another if whose conditions can be combined into a single if using logical AND [**34**](the ifs in Figure 2.9 can be merged into one if with the condition "`if(firstCondition && secondCondition)`).

```
[...]
if(firstCondition){
    if(secondCondition){
        //body
    }
}
[...]
```

FIGURE 2.9. Example of a Collapsible Nested If

CHAPTER 3

Catalog of QLCs on Semantic Style

The QLC library [**4**] is designed to enable the creation of QLCs based on predefined templates. The templates created in this dissertation represent structured question patterns that are directly related to the quality of the submitted code. For each type of quality issue identified, a corresponding template was developed, similar to the approach proposed by Klinger et al [**34**].

By using these templates, it becomes possible to generate multiple different questions depending on the specific characteristics of the code being analyzed. In this way, the library ensures both consistency (through predefined structures) and variability (through code-dependent instantiations) in the generation of QLCs.

In this chapter, we begin by examining the structure of the developed templates. We then move on to identifying the types of templates created, discussing their appropriate applications, and illustrating their use through examples.

3.1. Template Structure

All questions are generated from templates and follow a consistent structure. The templates in the QLCs library are implemented as multiple-choice questions with one correct answer and conform to one of the following formats:

- **True or False**: The question will present two code snippets: one representing the student's solution and another that is similar in structure or logic. The student will be asked to determine whether the two snippets are functionally equivalent, meaning that the flow and variable behavior are preserved, and that both programs produce the same output when given the same input, with the possible answers being 'True' or 'False.'
- **Multiple Choices**: The question will present a snippet of the student's code and then ask the student to select the options that contain code segments with equivalent functionality.

All QLCs are designed to include at least one distractor, meaning that at least one of the provided options is intentionally incorrect. For example, in the case of True or False questions, the distractor corresponds to the opposite of the correct answer. In Multiple Choice questions, at least one of the code snippets is refactored in such a way that it is no longer equivalent to the snippet presented in the question statement. The distractors are carefully constructed to be misleading, often resembling plausible alternatives, and in several cases they are derived from common mistakes made by novice programmers [**15**, **18**].

3.2. Template Types

In this section, we present the QLCs developed in the context of this study directly inspired by the research mentioned in the chapter 2 (Sections 2.1 and 2.3).

3.2.1. If Return Condition

This template generates questions when it finds an instance of an if statement with an else branch, where both branches contain either a `return true` or `return false` statement [**18**–**20**, **35**]. The QLCs will have the question "**Which of the following methods has/have the same behavior?**" and will show the code submitted by the student (Figure 3.1a). The statement will be followed by two possible answers: the correct solution (Figure 3.1b), which is obtained by removing the if–else structure and replacing it with a single return that directly depends on the condition, and one distractor (Figure 3.1c), obtained by applying the direct negation of the condition in the return statement.

```java
boolean isPositive(int a) {
    if (a > 0) {
        return true;
    } else {
        return false;
    }
}
```

(A) Submitted Code

```java
boolean isPositive(int a) {
    return a > 0;
}
```

(B) Correct Answer

```java
boolean isPositive(int a, int b) {
    return a <= 0;
}
```

(C) Wrong Answer

FIGURE 3.1. "If Return Condition" QLCs Examples

3.2.2. Lonely Variable

This template generates questions when it finds a variable that is declared but not used throughout the code [**18**, **35**]. The QLCs will have the question "**Which of the following methods has/have the same behavior?**" and will show the code submitted by the student (Figure 3.2a). The statement will be followed by two possible answers: the correct solution (Figure 3.2b), which is obtained by removing the unused variable, and one distractor (Figure 3.2c), which is created by modifying the method so that it returns the unused variable. When necessary, the return type of the method is adjusted to accommodate the distractor.

```java
static int doubled(int a) {
    String s = "";
    int doubled = a*2;
    return doubled;
}
```

(A) Submitted Code

```java
static int doubled(int a) {
    int doubled = a*2;
    return doubled;
}
```

(B) Correct Answer

```java
static String doubled(int a) {
    String s = "";
    int doubled = a*2;
    return s;
}
```

(C) Wrong Answer

FIGURE 3.2. "Lonely Variable" QLCs Examples

3.2.3. Empty If or Empty Else

This template generates questions when it finds an if or else statement without a body, and removes such statements when found. Additionally, if it encounters an empty if statement followed by an else with a body, it replaces the construct with an if statement containing the negated condition and the body of the original else [**18**–**20**, **35**]. The QLCs will have the question "**Which of the following methods has/have the same behavior?**" and will show the code submitted by the student (Figure 3.3a). The statement will be followed by two possible answers: the correct solution (Figure 3.3b), obtained by simplifying the empty conditional construct, and one distractor (Figure 3.3c), created by introducing a semantic change, such as inverting the logic of the simplified condition or forcing the body to return a fixed value that breaks equivalence.

```java
static int[] rightSide(int[] array, boolean includeMiddle) {
    int middle = array.length / 2;
    if (includeMiddle && array.length % 2 == 1) {
    } else {
        middle++;
    }
    return subArray(array, middle, array.length - 1);
}
```

(A) Submitted Code

```java
static int[] rightSide(int[] array, boolean includeMiddle) {
    int middle = array.length / 2;
    if (!includeMiddle || array.length % 2 != 1) {
        middle++;
    }
    return subArray(array, middle, array.length - 1);
}
```

(B) Correct Answer

```java
static int[] rightSide(int[] array, boolean includeMiddle) {
    int middle = array.length / 2;
    if (includeMiddle && array.length % 2 == 1) {
        middle++;
    }
    return subArray(array, middle, array.length - 1);
}
```

(C) Wrong Answer

FIGURE 3.3. "Empty If or Else" QLCs Examples

3.2.4. Ifs That Could be One If/Else

This template generates questions when it finds two consecutive If statements with opposite conditions, transforming them into a single If-Else construct based on the original If statements. Additionally, if it encounters two consecutive If statements with identical conditions, it merges them into a single If statement **[18, 20, 35]**. The QLCs will have the question "**Which of the following methods has/have the same behavior?**" and will show the code submitted by the student (Figure 3.4a). The statement will be followed by two possible answers: the correct solution (Figure 3.4b) and one distractor (Figure 3.4c).

- In the case of two consecutive if statements with identical conditions, the correct transformation merges both bodies into a single if, ensuring that, when the condition is true, both code blocks are executed in sequence. The distractor,

however, rewrites the construct as an if-else, which changes the behavior because only one of the two blocks is executed, rather than both.

- In the case of two consecutive if statements with opposite conditions, the correct transformation is to rewrite them as an if-else, making explicit that the two branches are mutually exclusive. The distractor incorrectly merges both bodies into a single if, which causes both blocks to execute together whenever the condition is true, thereby breaking the original exclusivity.

```java
static boolean isPositive(int a) {
    boolean b = false;
    if (a > 0) {
        b = true;
    }
    if (a <= 0) {
        b = false;
    }
    return b;
}
```

(A) Submitted Code

```java
static boolean isPositive(int a) {
    boolean b = false;
    if (a > 0) {
        b = true;
    } else {
        b = false;
    }
    return b;
}
```

(B) Correct Answer

```java
static boolean isPositive(int a) {
    boolean b = false;
    if (a > 0) {
        b = true;
        b = false;
    }
    return b;
}
```

(C) Wrong Answer

FIGURE 3.4. "Ifs That Could be One If/Else" QLCs Examples

3.2.5. Unnecessary Equals to True or False

This template generates questions when it finds an If statement with a condition that explicitly compares a value to True or False, and transforms it by removing the equality check [**19**, **20**, **35**]. The QLCs will have the question "**Which of the following methods has/have the same behavior?**" and will show the code submitted by the student (Figure 3.5a). The statement will be followed by three possible answers: the correct solution (Figure 3.5b), obtained by removing the redundant comparisons (e.g., transforming "if(a

== true)" into "if(a)" and "if(b == false)" into "if(!b)") and two distractor (Figures 3.5c and 3.5d). The first distractor incorrectly negates the expression while attempting to remove the equality check (e.g., "if(a == true)" becomes "if(!a)"), which inverts the original logic and causes the if block to execute under opposite conditions. The second distractor removes the condition entirely, executing the if block unconditionally (e.g., "if(a == true) return 1;" becomes "return 1;"), thereby fundamentally altering the program's behavior.

```java
static int returnOne(boolean b) {
    int n = 0;
    if (b == true) {
        n = 1;
    }
    return n;
}
```

(A) Submitted Code

```java
static int returnOne(boolean b) {
    int n = 0;
    if (b) {
        n = 1;
    }
    return n;
}
```

(B) Correct Answer

```java
static int returnOne(boolean b) {
    int n = 0;
    if (!b) {
        n = 1;
    }
    return n;
}
```

(C) Wrong Answer (Version 1)

```java
static int returnOne(boolean b) {
    int n = 0;
    n = 1;
    return n;
}
```

(D) Wrong Answer (Version 2)

FIGURE 3.5. "Unnecessary Equals to True or False" QLCs Examples

3.2.6. Unnecessary If Nesting

This template generates questions when it finds nested If statements that can be simplified into a single If statement with a combined condition **[18, 20, 35]**. The QLCs will have the question "**Which of the following methods has/have the same behavior?**" and will show the code submitted by the student (Figure 3.6a). The statement will be followed by three possible answers: the correct solution (Figure 3.6b), obtained by merging the nested conditions into a single if using the logical AND operator ("&&") so that the original behavior is preserved, and two distractors (Figures 3.6c and 3.6d). The first distractor erroneously combines the conditions with the logical OR operator ("||"), which changes the semantics because the block will execute when either condition is true rather than only when both are true. The second distractor rewrites the nested if as an if-else, incorrectly treating the blocks as mutually exclusive and thereby altering the original control flow.

```java
static boolean natural(int n){
    if(n > 0){
        if(n%2==0){
            return true;
        }
    }
    return false;
}
```

(A) Submitted Code

```java
static boolean natural(int n){
    if(n > 0 && n%2==0){
        return true;
    }
    return false;
}
```

(B) Correct Answer

```java
static boolean natural(int n){
    if(n > 0 || n%2==0){
        return true;
    }
    return false;
}
```

(C) Wrong Answer (Version 1)

```java
static boolean natural(int n){
    if(n > 0){
    }else{
        return true;
    }
    return false;
}
```

(D) Wrong Answer (Version 2)

FIGURE 3.6. "Unnecessary If Nesting" QLCs Examples

3.2.7. Unnecessary Parameter

This template generates questions when it finds one or more parameters that are declared but not used within the code. The QLCs will have the question "**Which of the following methods has/have the same behavior?**" and will show the code submitted by the student (Figure 3.7a). The statement will be followed by three possible answers: the correct solution (Figure 3.7b), obtained by removing the unused parameters from the method signature while leaving the body unchanged, and two distractors (Figures 3.7c and 3.7d). The first distractor modifies the code so that the method returns the unused parameter instead of its intended result, thereby altering the original logic by introducing a dependency on a value that previously had no role in execution. The second distractor removes parameters that are actually necessary and used in the method, leading to compilation errors or incorrect program behavior.

```
static int square(int a, int b) {
    return a*a;
}
```

(A) Submitted Code

```
static int square(int a) {
    return a*a;
}
```

(B) Correct Answer

```
static int square(int b) {
    return a*a;
}
```

(C) Wrong Answer (Version 1)

```
static int square(int b) {
    return b;
}
```

(D) Wrong Answer (Version 2)

FIGURE 3.7. "Unnecessary Parameter" QLCs Examples

3.2.8. Useless Duplication of If's and Else's bodies

This template generates questions when it finds an If statement with an Else branch where both branches contain exactly the same body [**19**, **20**, **35**]. The QLCs will have the question "**Which of the following methods has/have the same behavior?**" and will show the code submitted by the student (Figure 3.8a). The statement will be followed by three possible answers: one correct solution (Figure 3.8b) and two distractors (Figures 3.8c and 3.8d). The Correct solution removes the duplicated if-else construct and replaces it with the single shared block given that the condition is redundant, preserving the original behavior. The first distractor removes the else but leaves the original if and its condition intact, which causes the block to execute only when the condition is true and therefore changes the program's behavior. The second distractor likewise removes the else but inverts the condition, so the block executes only when the original condition is false; this also alters the semantics and is therefore incorrect.

```
String wrap(String t, char c) {
    String s = "";
    if (c == '&') {
        s = c + t + c;
    } else {
        s = c + t + c;
    }
    return s;
}
```

(A) Submitted Code

```
String wrap(String t, char c) {
    String s = "";
    s = c + t + c;
    return s;
}
```

(B) Correct Answer

```
String wrap(String t, char c) {
    String s = "";
    if (c == '&') {
        s = c + t + c;
    }
    return s;
}
```

(C) Wrong Answer (Version 1)

```
String wrap(String t, char c) {
    String s = "";
    if (c != '&') {
        s = c + t + c;
    }
    return s;
}
```

(D) Wrong Answer (Version 2)

FIGURE 3.8. "Useless Duplication of If's and Else's bodies" QLCs Examples

3.2.9. Useless Duplication inside If's and Else's bodies

This template generates questions when it finds an If statement with an Else branch where both branches share common segments of code in their bodies [**19**, **20**, **35**]. The QLCs will have the question "**Which of the following methods has/have the same behavior?**" and will show the code submitted by the student (Figure 3.9a). The statement will be followed by three possible answers: the correct solution (Figure 3.9b), obtained by extracting the code common to both branches and placing it outside the if-else so that only the differing instructions remain inside each branch (thereby preserving original behavior while eliminating duplication), and two distractors (Figures 3.9c and 3.9d). The first distractor removes the duplicated code from inside the branches but also eliminates the conditional entirely, which destroys the original distinction between branches and thus alters program logic. The second distractor extracts the common code but deletes the else branch, leaving only the if branch, changing the behavior because the instructions that originally executed under the else are now ignored.

```
String add(String t, char c) {
    String s = "";
    if (c == '&') {
        s += t;
    } else {
        s += t;
        s += c;
    }
    return s;
}
```

(A) Submitted Code

```
String add(String t, char c) {
    String s = "";
    s += t;
    if (c == '&') {
    } else {
        s += c;
    }
    return s;
}
```

(B) Correct Answer

```
String add(String t, char c) {
    String s = "";
    s += t;
    return s;
}
```

(C) Wrong Answer (Version 1)

```
String add(String t, char c) {
    String s = "";
    s += t;
    if (c == '&') {
    }
    return s;
}
```

(D) Wrong Answer (Version 2)

FIGURE 3.9. "Useless Duplication inside If's and Else's bodies" QLCs Examples

3.2.10. Useless Variable Assignment

This template generates questions when it finds an unnecessary assignment of variables: a variable that is assigned a value but not used before being reassigned [**19**]. For this template it was decided to use a different question format, instead of being multiple-choice it is a "true or false" question. The QLCs will have the question "**Do the following methods have the same behavior?**" and display the student's code together with an alternative version, produced by simplifying the original code (Figure 3.10b) or removing the reassignment, producing a different behavior (Figure 3.10c). The statement will be followed by two possible answers: "True" or "False".

```java
static int add(int a, int b) {
    int sum = 0;
    sum = a + b;
    return sum;
}
```

(A) Submitted Code

```java
static int add(int a, int b) {
    int sum = a + b;
    return sum;
}
```

(B) Generated Code with Equal Behavior

```java
static int add(int a, int b) {
    int sum = 0;
    return sum;
}
```

(C) Generated Code with Different Behavior

FIGURE 3.10. "Useless Variable Assignment" QLCs Examples

3.2.11. Useless Self Assign

This template generates questions when it finds a self-assignment: where a variable is assigned to itself [**18**, **20**]. The QLCs will have the question "**Which of the following methods has/have the same behavior?**" and will show the code submitted by the student (Figure 3.11a). The statement will be followed by three possible answers: the correct solution (Figure 3.11b), obtained by removing the unnecessary self-assignments and leaving only statements that actually modify the flow and one distractor (Figure 3.11c), in which the self-assignment is altered to a meaningless assignment (e.g., y = null), thus changing the behavior of the program.

```
double abs(double n) {
    if (n < 0)
        n = -n;
    else
        n = n;
    return n;
}
```

(A) Submitted Code

```
double abs(double n) {
    if (n < 0)
        n = -n;
    return n;
}
```

(B) Correct Answer

```
double abs(double n) {
    if (n < 0)
        n = -n;
    else
        n = null;
    return n;
}
```

(C) Wrong Answer

FIGURE 3.11. "Useless Self Assign" QLCs Examples

CHAPTER 4

QLC Library: Structure and Integration

In this chapter, we explain how the library works and how to use it. We start by describing its internal workings, the programming libraries it relies on, and how QLCs are generated. Then, we show how to apply the library, illustrating the steps needed to transform a submitted piece of code into a fully formed question.

4.1. Usage and Integration

The library developed for generating QLCs was implemented in Kotlin. To use it, one simply needs to import the public library **Jask** [**4**][1]. The process of generating questions relies on the selection of a specific QLC template and its application to a given code element.

In this section, we will provide a follow-through example, showing step by step how a submitted piece of code can be transformed into a fully formed question using the library.

For this example, we use the code shown in Figure 4.1 and generate a QLCs using the "Useless Variable Assignment" template, as introduced in Chapter 3, Section 3.2.10.

```
double average(int a, int b) {
    double average=0;
    average=(double)(a+b)/2;
    return average;
}
```

FIGURE 4.1. Example of a Code Submission

To generate the question, the method must be placed inside a class, which can be provided either as a "String" or as a "CompilationUnit" [**36**] from the JavaParser library [**33**]. The desired language can also be specified, with English set as the default. Next, simply initialize the class corresponding to the chosen template (in this case, "UselessVariableDeclaration") and call the "generate" method, passing the code as an argument, as illustrated in Figure 4.2.

[1] `https://github.com/ambco-iscte/jask`

```
val submittedCode = """
    class ClassName{
        double average(int a, int b) {
            double average=0;
            average=(double)(a+b)/2;
            return average;
        }
    }
""".trimIndent()
val questionTemplate = UselessVariableDeclaration()
val generatedQuestion = questionTemplate.generate(submittedCode)
```

FIGURE 4.2. Example of QLCs' generation

If the submitted code does not contain any quality issues corresponding to the selected template, in this case "Useless Variable Assignment" (Chapter 3, Section 3.2.10), no question is generated. Instead, the library throws an exception ("QuestionGenerationException: Could not find a valid source."). On the other hand, if the code contains at least one instance of the relevant quality issue, the generate method returns a "Question" (from the Jask library [**37**]), which can also be converted to a "String". This produces a fully formed question, as illustrated in Figure 4.3.

Question

Do the following methods have the same behavior?

```
double average(int a, int b) {
    double average = 0;
    average = (double) (a + b) / 2;
    return average;
}

double average(int a, int b) {
    double average = (double) (a + b) / 2;
    return average;
}
```

[x] Yes.
[ ] No.

FIGURE 4.3. Example of QLCs generated

In addition to generating questions from a single selected template, the library can also analyze a code fragment to determine which quality-issue templates are applicable. This functionality is provided by the commands `applicableQualityTemplates` and `generateQuestions`, which first identify the relevant quality issues and then generate the corresponding QLCs. Figure 4.4 illustrates the process, using a submitted code that contains two semantic styles issues, while Figure 4.5 shows the resulting list of the questions generated from the applicable templates ("If Return Conditions" and "Useless Variable Assignment", 3.2.1 and 3.2.10).

```
val source = """
    class abc {
        public boolean test() {
            boolean a = true;
            a = false;
            if (a) return false;
            else return true;
        }
    }
""".trimIndent()

val templates = applicableQualityTemplates(source)
val qlcs = generateQuestions(templates, source)
qlcs.forEach { println(it)}
```

FIGURE 4.4. Example of Generating Applicable QLCs

```
"Which of the following methods has/have the same behavior?"
[
    public boolean test() {
        boolean a = true;
        a = false;
        if (a) return false;
        else return true;
    }
]
[x] public boolean test() {
    boolean a = true;
    a = false;
    return !a;
}
[ ] public boolean test() {
    boolean a = true;
    a = false;
    return a;
}


"Do the following methods have the same behavior?"
[
    public boolean test() {
        boolean a = true;
        a = false;
        if (a) return false;
        else return true;
    },
    public boolean test() {
        boolean a = false;
        if (a) return false;
        else return true;
    }
]
[x] Yes.
[ ] No.
```

FIGURE 4.5. Example of Generated Applicable QLCs

4.2. Library Architecture

As previously mentioned, the Jask library [4][2] is an existing Kotlin library capable of analyzing Java code and generating questions. It supports the generation of both static questions, based on syntactic analysis, and dynamic questions, based on program flow and execution. In this dissertation, the focus is on the static dimension. Specifically, we extend Jask by implementing a set of static QLCs templates that target code quality issues identified through static analysis, addressing them through refactoring.

The primary objective of the library is to generate multiple-choice questions. For this purpose, the main abstraction is the `Question` object, which represents the output: a QLC composed of a prompt and a set of possible answers, with at least one correct option. To generate these questions, it is necessary to define templates, one for each type of question, such as the ones described in Chapter 3, Section 3.2. Each template is implemented as a subclass of the abstract class `QuestionTemplate`, where the logic for the generation of questions is defined. The overall structure of the library, including these relationships, is illustrated in Figure 4.6.

Each class provides the following fundamental methods:

- `isApplicable`: This method evaluates whether the template is relevant for a given Java method. For example, in the case of the "Unnecessary Parameter" template (mentioned in 3.2.7), `isApplicable` checks whether any declared parameter is not used within the submitted method. The method returns a boolean value indicating whether the template applies to the analyzed code.
- `build`: This method constructs the question itself, defining both the prompt and the answer options. It is within this method that the analysis of the submitted code takes place. For the templates developed in this dissertation, `build` is also responsible for applying refactorings to improve the code and for generating distractors.
- `generate`: This method is intended for integration and is implemented in the abstract class `QuestionTemplate`, rather than being specific to each template. Invokes the previous method to create a QLC. If the template is not applicable, an exception is thrown. Otherwise, it returns the final `Question` object. An example of how this method is called is shown in Figure 4.2.

[2] `https://github.com/ambco-iscte/jask`

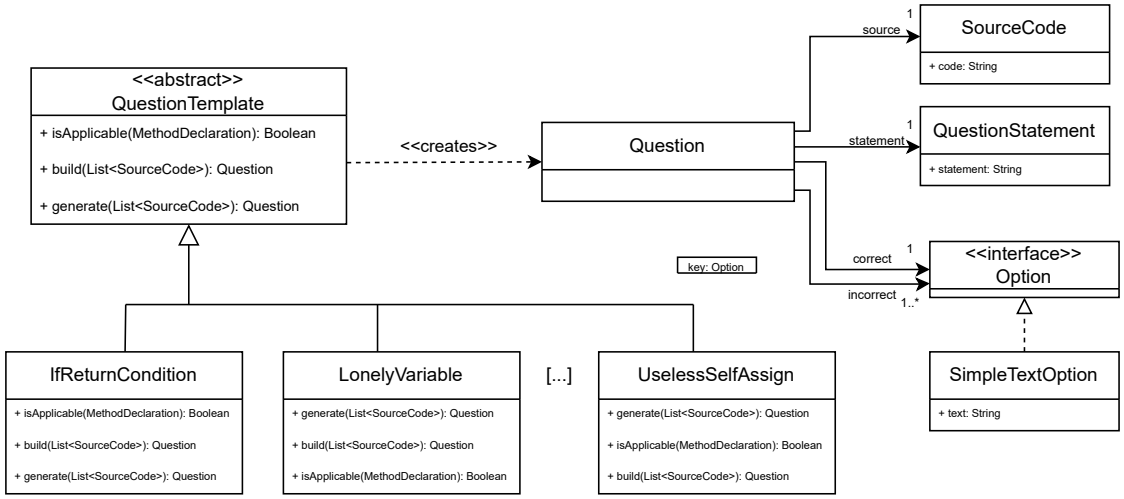

FIGURE 4.6. Class Diagram of the Developed QLCs Templates

To support the analysis and the refactoring of Java code, the Jask [**4**] library relies on external libraries that provide the necessary capabilities. The one used in this dissertation is JavaParser [**33**], a widely used open source library to analyze, parse, and transform Java source code. JavaParser converts the source code into an Abstract Syntax Tree (AST), a structured representation of the program where each element such as methods, variables, and expressions can be inspected and modified. This makes it particularly suitable for tasks that involve static analysis and code refactoring.

Within the Jask library, JavaParser plays a crucial role in both the `isApplicable` and `generate` methods of each template. In the first case, it is used to analyze the AST of a submitted method and determine whether the conditions for a given template are satisfied. In the second case, JavaParser is used not only to reanalyze the code but also to apply transformations when necessary. At this stage, the library uses JavaParser's ability to refactor or rewrite parts of the AST to produce both the correct solution and distractors. For example, unused variables may be removed or redundant constructs may be simplified. This combination of analysis and structured modification enables the generation of QLCs.

CHAPTER 5

Study

In this chapter, we demonstrate that the types of QLCs presented in this work are applicable within the context of an introductory programming course. To this end, we generated QLCs for all students' code submissions throughout the academic year 2024/2025. This process allows us to investigate both the quantity and nature of the QLCs produced, given that their generation directly depends on the quality of learners' code. Consequently, this analysis also provides insight into the most common misconceptions encountered by novice programmers.

We examine the total number of QLCs generated, their distribution throughout the course, and the average number of QLCs produced per student. In doing so, our objective is to capture not only the overall scale of QLC generation but also the frequency of each template during the learning process.

5.1. Data Extraction Methodology

The data used for this study consists of 47057 code submissions developed by students enrolled in the Introduction to Programming course during the first semester of the academic year. Each submission was automatically analyzed to determine whether any of the predefined QLCs' templates were applicable. A QLC was considered applicable whenever the corresponding code contained one or more quality issues targeted by the template. For every submission that matched one or more QLCs' templates, the respective QLCs were generated and stored.

By applying this extraction process to all student submissions, we ensured that the resulting dataset captured the full extent of the questions that could arise from the programming practices of the learners. From the total of 47057 student code submissions, we extracted all methods in order to evaluate their applicability for QLCs generation. In practice, this means that if a submission contained more than one method, each method was isolated and examined individually to determine whether it had quality issues that could be transformed into QLCs.

5.2. Results

Through this process mentioned in the previous section, we obtained 3721 applicable methods, corresponding to 2830 submissions. In other words, only 6.03% of all submissions contained detectable quality issues.

This relatively low percentage can be explained by two main factors. First, a total of 2645 submissions consisted only of code fragments without any method definitions and were excluded from the analysis, as these exercises were too simple to allow meaningful quality

issues to be identified. Similarly, many exercises consisted of extremely short methods (example: Figure 5.1), which also provided little opportunity for style or structural issues to emerge. Methods of this type rarely generate QLCs, as their simplicity leaves almost no space for meaningful quality issues to be identified. If we focus only on exercises in which at least one student submission exhibited a quality issue, the dataset narrows to 24566 submissions, and 2830 of them, now 11.52%, contained at least one quality issue.

```
int next(int n) {
    return n + 1;
}
```

FIGURE 5.1. Example of a Short Submitted Method

In Figure 5.2, we observe that most students generated a relatively small number of QLCs, with the largest group concentrated between 0 (the minimum, with 37 students) and 14. In contrast, only a few students exceeded 20 QLCs, and isolated cases reached as high as 61 (the maximum, with a single student). This distribution highlights a clear long-tail pattern, with an average of 8.61 QLCs per student. Given that each student submitted on average 96 solutions, this corresponds to nearly one in every eleven submissions generating a QLC, which represents a considerable proportion and suggests that quality issues were a recurring element in students' coding activity.

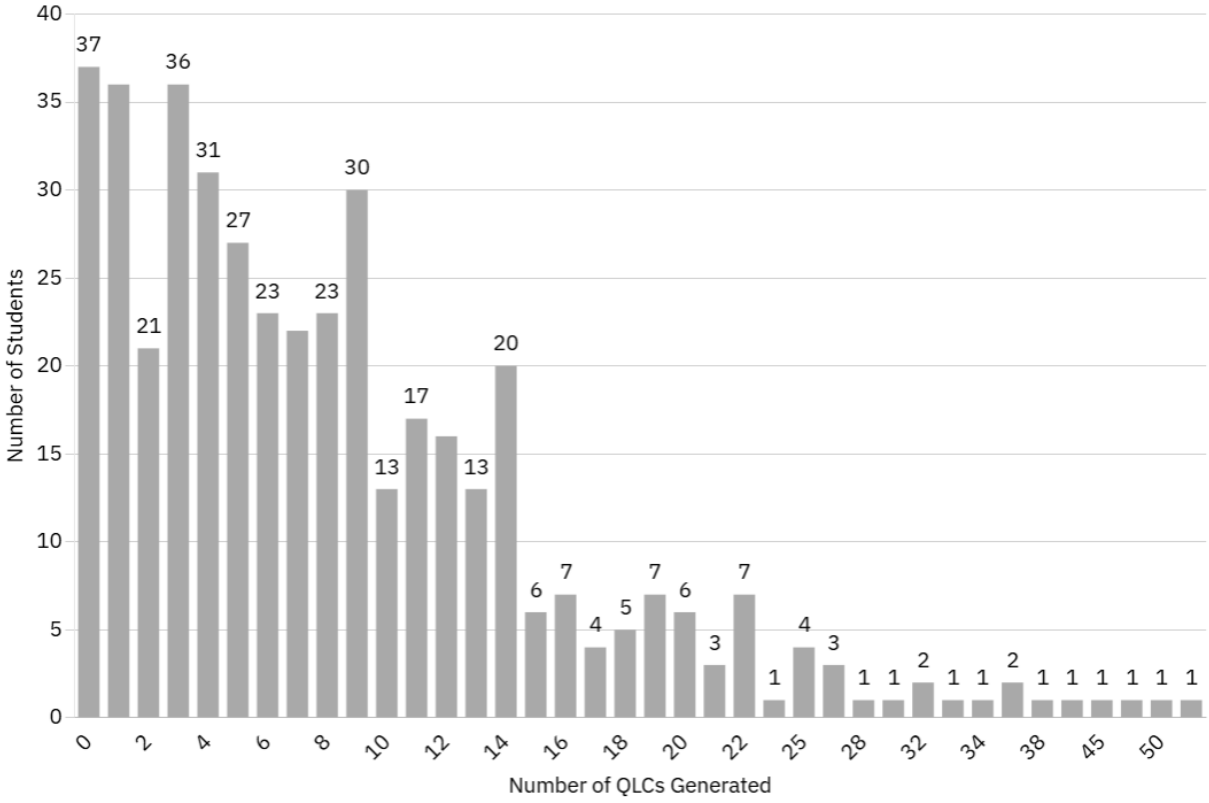

FIGURE 5.2. Number of Students by QLC Count

The table 5.1 presents the number of occurrences of each QLC template, their relative frequency, the percentage of students affected, and the average occurrences per student, based on the submitted code. The most frequently observed quality issue in the dataset was "Unnecessary Equals to True or False", with 1002 occurrences and affecting more than half of the students. The second most common issue was "If Return Condition", with 713 occurrences and impacting almost half of the students. Overall, the issues "If Return Condition", "Lonely Variable", "Unnecessary Equals to True or False", and "Useless Variable Assignment" were particularly widespread, each affecting nearly half of the students (between 47.22% and 56.02%). By contrast, the least common issues were "Ifs That Could be One If/Else", "Useless Duplication of If's and Else's Bodies", and "Useless Self Assign", which each affected fewer than 10% of students and together accounted for only 4.76% of all quality issues identified.

Quality Issue	**Occurrences (%)**	**% of Students Affected**	**Avg. per Student**
If Return Condition	713 (19.16 %)	48.84 %	1.65
Lonely Variable	489 (13.14 %)	52.55 %	1.13
Empty If or Empty Else	116 (3.12 %)	14.12 %	0.27
Ifs That Could be One If/Else	52 (1.40 %)	7.64 %	0.12
Unnecessary Equals to True or False	1002 (26.93 %)	56.02 %	2.32
Unnecessary If Nesting	184 (4.94 %)	22.22 %	0.43
Unnecessary Parameter	338 (9.08 %)	34.72 %	0.78
Useless Duplication of If's and Else's bodies	43 (1.16 %)	7.64 %	0.1
Useless Duplication inside If's and Else's bodies	305 (8.20 %)	18.52 %	0.71
Useless Variable Assignment	397 (10.67 %)	47.22 %	0.92
Useless Self Assign	82 (2.20 %)	6.02 %	0.19

TABLE 5.1. Occurrences and distribution of each QLC template across student submissions.

5.3. Discussion

The analysis of the obtained results allows for a deeper interpretation of the occurrences of QLCs identified. First, the data show that about 11.5% of the analyzed submissions contained at least one applicable question. Whenever tasks allowed for greater complexity, code quality issues emerged consistently. Among these, "Unnecessary Equals to True or False" and "If Return Condition" were particularly frequent, together representing almost half of all detected issues and affecting about 50% of all the students. This pattern suggests that specific conceptual difficulties, especially those related to the handling of boolean expressions, are recurrent in introductory programming contexts.

From a pedagogical perspective, these findings reinforce the potential of QLCs as a learning support tool. The fact that the most common issues are linked to fundamental concepts demonstrates that QLCs can serve as an early indicator of persistent difficulties. By presenting students with questions generated directly from their own code, they are encouraged to reflect not only on the correctness of program output but also on the underlying structure and reasoning. In this way, QLCs can complement existing automated

assessment systems by fostering awareness of programming practices and contributing to the consolidation of structural knowledge from the earliest stages of learning.

Nevertheless, some limitations of the study must be acknowledged. The dataset analyzed corresponds to a single course and a single programming language, which may restrict the generalization of the findings. Moreover, a significant proportion of the submissions were composed of simple exercises, limiting the opportunities for quality issues to arise.

CHAPTER 6

Conclusions

The work carried out in this dissertation was motivated by the challenges faced by students in introductory programming courses. Many students are able to submit functional code without fully understanding how it works. Since most courses rely on automated submission and grading systems, this dissertation adopted the approach of generating questions directly from students' own code, thus encouraging them to reflect on their solutions. To achieve this, we focused on code quality, as novice programmers frequently introduce semantic issues when they do not fully grasp the concepts involved, such as leaving unused variables.

The main objectives of this project were twofold: first, to explore how multiple-choice questions could be generated automatically from submitted Java code; and second, to investigate how frequently quality issues occurred in submissions, and which types were most common, by analyzing over 47000 pieces of code submitted by students.

From this study, we conclude that it is possible to generate questions directly from code by using tools that support the manipulation of code fragments, in this case JavaParser. With these tools we were able to systematically identify quality issues, transform them into corrected code, and construct distractors. This directly addresses Research Question **RQ1**. In addition, after filtering out shorter submissions and analyzing the relevant ones with the developed library, we found that 11.5% of all submissions contained at least one quality issue. Among these, "Unnecessary Equals to True or False" and "If Return Condition" were the most frequent, each affecting nearly half of the students. This provides an answer to Research Question **RQ2**.

QLCs have the potential to support student reflection, and our analysis showed they can serve as a valuable tool to measure the prevalence of quality issues. In this way, QLCs can provide an overview of the most common errors made by novice programmers, which can be particularly useful for instructors in identifying areas that may require further attention in teaching.

6.1. Limitations and Future Work

This work has shown not only that it is possible to generate questions automatically from student code but also that the developed library can be used to identify the most common quality issues among novice programmers. Nevertheless, the study is subject to several limitations. The dataset was composed of submissions collected during a single semester of an introductory programming course at ISCTE. As a result, the data is relatively restricted in scope and includes many simple submissions, which limits the range of quality issues

that could be observed. This constraint may have influenced the results, as the findings cannot be easily generalized beyond this specific context. The developed library also has technical limitations, since it is restricted to Java code. Consequently, the issues detected may not reflect the difficulties that arise in other languages which differ significantly in syntax and style. Furthermore, the set of templates was derived from quality issues identified in the literature, meaning that the coverage is incomplete and other relevant issues may not have been considered.

For future work it would be valuable to test the pedagogical impact of the library by integrating them into course assessments, allowing students to answer them after submitting their code. This would make it possible to investigate whether exposure to the created QLCs throughout a semester reduces the frequency of quality issues and improves overall performance in the course. The library itself could also be expanded by developing additional templates to cover a wider range of quality issues, providing deeper insights into the misconceptions of novice programmers. Moreover, developing similar libraries for other programming languages would allow comparative studies to determine whether the quality issues most frequently observed in Java also occur in languages, and whether the pedagogical impact of QLCs is consistent across different programming languages.

References

[1] C. Izu and C. Mirolo, "Asking Students to Refactor their Code: A Simple and Valuable Exercise," in *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*, (Milan Italy), pp. 73–79, ACM, July 2024.

[2] T. Lehtinen, L. Haaranen, and J. Leinonen, "Automated questionnaires about students' JavaScript programs: Towards gauging novice programming processes," in *Proceedings of the 25th Australasian Computing Education Conference*, ACE '23, (New York, NY, USA), p. 49–58, Association for Computing Machinery, 2023.

[3] A. Ettles, A. Luxton-Reilly, and P. Denny, "Common logic errors made by novice programmers," in *Proceedings of the 20th Australasian Computing Education Conference*, (Brisbane Queensland Australia), pp. 83–89, ACM, Jan. 2018.

[4] A. Santos, T. Soares, N. Garrido, and T. Lehtinen, "Jask: Generation of Questions About Learners' Code in Java," in *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1*, (Dublin Ireland), pp. 117–123, ACM, July 2022.

[5] T. Lehtinen, A. L. Santos, and J. Sorva, "Let's ask students about their programs, automatically," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, p. 467–475, IEEE, May 2021.

[6] Y. Qian and J. Lehman, "Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review," *ACM Transactions on Computing Education*, vol. 18, pp. 1–24, Mar. 2018.

[7] T. Lehtinen, A. Lukkarinen, and L. Haaranen, "Students Struggle to Explain Their Own Program Code," in *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, (Virtual Event Germany), pp. 206–212, ACM, June 2021.

[8] J. Bevilacqua, L. Chiodini, I. Moreno Santos, and M. Hauswirth, "Using Notional Machines to Automatically Assess Students' Comprehension of Their Own Code," in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2*, (Portland OR USA), pp. 1572–1573, ACM, Mar. 2024.

[9] G. Fan, D. Liu, R. Zhang, and L. Pan, "The impact of AI-assisted pair programming on student motivation, programming anxiety, collaborative learning, and programming performance: a comparative study with traditional pair programming and individual approaches," *International Journal of STEM Education*, vol. 12, 03 2025.

[10] M. Alanazi, B. Soh, H. Samra, and A. Li, "The influence of artificial intelligence tools on learning outcomes in computer programming: A systematic review and meta-analysis," *Computers*, vol. 14, no. 5, 2025.

[11] R. Zviel-Girshin, "The good and bad of AI tools in novice programming education," *Education Sciences*, vol. 14, no. 10, 2024.

[12] M. Neuwinger and D. Riehle, "A systematic review of common beginner programming mistakes in data engineering," in *Accepted to the 2025 IEEE/ACM 37th International Conference on Software Engineering Education and Training (CSEET)*, 2025.

[13] C. Izu and S. Chandra, "Are We There Yet? Novices' Code Smells linked to Loop Constructs," in *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 2*, (Providence RI USA), pp. 1151–1151, ACM, Mar. 2022.

[14] N. C. C. Brown and A. Altadmri, "Novice Java Programming Mistakes: Large-Scale Data vs. Educator Beliefs," *ACM Transactions on Computing Education*, vol. 17, pp. 1–21, June 2017.

[15] E. Oliveira, H. Keuning, and J. Jeuring, "Student Code Refactoring Misconceptions," in *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, (Turku Finland), pp. 19–25, ACM, June 2023.

[16] T. Lehtinen, O. Seppälä, and A. Korhonen, "Automated Questions About Learners' Own Code Help to Detect Fragile Prerequisite Knowledge," in *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, (Turku Finland), pp. 505–511, ACM, June 2023.

[17] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, June 1999.

[18] G. De Ruvo, E. Tempero, A. Luxton-Reilly, G. B. Rowe, and N. Giacaman, "Understanding semantic style by analysing student code," in *Proceedings of the 20th Australasian Computing Education Conference*, (Brisbane Queensland Australia), pp. 73–82, ACM, Jan. 2018.

[19] F. Alfredo, A. L. Santos, and N. Garrido, "Sprinter: A Didactic Linter for Structured Programming," in *Third International Computer Programming Education Conference (ICPEC 2022)* (A. Simões and J. a. C. Silva, eds.), vol. 102 of *Open Access Series in Informatics (OASIcs)*, (Dagstuhl, Germany), pp. 2:1–2:8, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

[20] H. Keuning, B. Heeren, and J. Jeuring, "A tutoring system to learn code refactoring," in *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, (New York, NY, USA), p. 562–568, Association for Computing Machinery, 2021.

[21] T. Haendler, G. Neumann, and F. Smirnov, "Refactutor: An interactive tutoring system for software refactoring," in *Computer Supported Education* (H. C. Lane, S. Zvacek, and J. Uhomoibhi, eds.), (Cham), pp. 236–261, Springer International Publishing, 2020.

[22] Z. Xing and E. Stroulia, "Refactoring practice: How it is and how it should be supported - an Eclipse case study," in *2006 22nd IEEE International Conference on Software Maintenance*, pp. 458–468, 2006.

[23] D. Jemerov, "Implementing refactorings in IntelliJ IDEA," in *Proceedings of the 2nd Workshop on Refactoring Tools*, WRT '08, (New York, NY, USA), Association for Computing Machinery, 2008.

[24] A. Kumar Das, "A linting tool without a compiler in it," in *2021 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, pp. 1–4, 2021.

[25] K. Vayadande, K. Mukhopadhyay, V. Chaudhari, S. Manwadkar, T. Mutalik, and I. Gawali, "Let us lint: A tool for code formatting and code enhancing," in *2023 14th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pp. 1–8, 2023.

[26] A. Vihavainen, J. Helminen, and P. Ihantola, "How novices tackle their first lines of code in an IDE: analysis of programming session traces," in *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, Koli Calling '14, (New York, NY, USA), p. 109–116, Association for Computing Machinery, 2014.

[27] M. Benjamin, L. Brown, J. Sticklen, L. Ureel, and M. Jarvie-Eggart, "Engaging novice programmers: A literature review of the effect of code critiquers on programming self-efficacy," in *2023 IEEE Frontiers in Education Conference (FIE)*, pp. 1–9, 10 2023.

[28] R. Plösch and C. Neumüller, "Does static analysis help software engineering students?," in *Proceedings of the 2020 9th International Conference on Educational and Information Technology*, ICEIT 2020, (New York, NY, USA), p. 247–253, Association for Computing Machinery, 2020.

[29] E. Alomar, S. AlOmar, and M. W. Mkaouer, "On the use of static analysis to engage students with software quality improvement: An experience with PMD," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pp. 179–191, 05 2023.

[30] D. Liu, J. Calver, and M. Craig, "A static analysis tool in CS1: Student usage and perceptions of PythonTA," in *Proceedings of the 26th Australasian Computing Education Conference*, ACE '24, (New York, NY, USA), p. 172–181, Association for Computing Machinery, 2024.

[31] E. A. AlOmar, M. W. Mkaouer, and A. Ouni, "Automating source code refactoring in the classroom," in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2024, (New York, NY, USA), p. 60–66, Association for Computing Machinery, 2024.

[32] A. L. Santos and A. B. Caniço, "Modeling structured programming with Strudel: a simulation runtime for programming education tools," *Software and Systems Modeling*, Nov 2025.

[33] J. Team, "Javaparser." Accessed on 2025.

[34] S. L. Klinger, P. Weber, S. Strickroth, and M. Striewe, "Stimulating reflection on code quality: An adaptive and automated question generation approach for student code," in *23. Fachtagung Bildungstechnologien (DELFI 2025)*, pp. 301–306, Bonn: Gesellschaft für Informatik e.V., 2025.

[35] T. Effenberger and R. Pelánek, "Code quality defects across introductory programming topics," in *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1*, SIGCSE 2022, (New York, NY, USA), p. 941–947, Association for Computing Machinery, 2022.

[36] J. Team, "Class CompilationUnit." `https://www.javadoc.io/doc/com.github.javaparser/javaparser-core/3.5.0/com/github/javaparser/ast/CompilationUnit.html`. Accessed on 2025.

[37] A. L. Santos, A. B. Caniço, and G. M. C. Serrano, "Jask," 2025. GitHub repository, commit 643e63d, `https://github.com/ambco-iscte/jask`.