Student Code Refactoring Misconceptions

Eduardo Oliveira
Utrecht University
The Netherlands
e.carneirodeoliveira@uu.nl

Hieke Keuning
Utrecht University
The Netherlands
h.w.keuning@uu.nl

Johan Jeuring
Utrecht University
The Netherlands
j.t.jeuring@uu.nl

ABSTRACT

Teaching students to develop code of good quality is important. Refactoring – rewriting a program into a semantically equivalent program of better quality – is a common technique to improve code quality. It is therefore relevant for students to learn about refactoring, even for the smaller programs they write as beginners. However, students make mistakes when refactoring programs. Some of these mistakes appear often, and might be caused by misconceptions they have. In this paper, we investigate common student code refactoring misconceptions. We do this by analyzing log data containing program snapshots of students working on refactoring exercises in a tutoring system. We manually inspect all transitions from a correct program state to an incorrect state. We then use grounded theory to identify and categorize misconceptions students might have when refactoring programs. As a result, this work (1) defines the concept of refactoring misconception, and (2) provides an initial list of 25 such misconceptions, together with an accompanying website with full details.

CCS CONCEPTS

• **Social and professional topics → Computing education**; **Software engineering education**.

KEYWORDS

code refactoring; misconceptions; code quality; program snapshot analysis; grounded theory; student code refactoring misconceptions; programming education

ACM Reference Format:
Eduardo Oliveira, Hieke Keuning, and Johan Jeuring. 2023. Student Code Refactoring Misconceptions. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2023), July 8–12, 2023, Turku, Finland.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3587102.3588840

1 INTRODUCTION

The CS Education community studies, amongst others, how to teach programming to novice students. The interest in the question of how to teach novice students to develop software of *good quality* has increased in the last decade [22, 25, 30]. Good code quality can be obtained through code refactoring. Code refactoring is an internal change in a program structure, made by a programmer, that does not modify its observable behavior [14]. Inappropriate refactoring may lead to either semantically incorrect code or a decrease in code quality aspects, such as code readability.

Another topic of interest in programming education is understanding the difficulties novice students experience when learning programming [2, 8, 26, 29]. Qian and Lehman [31] have performed a literature review on student misconceptions in introductory programming. They emphasize that raising educators' awareness of student misconceptions can contribute to a proper selection and use of strategies and tools for teaching. Chiodini et al. [9] have collected student programming activities for more than a decade to identify their programming misconceptions. The authors introduce a framework to structure these misconceptions as well as a collection of such misconceptions.

Previous studies on programming misconceptions mostly target either syntax errors or misconceptions that typically occur when a student is following specifications to write a program from scratch. As far as we are aware, student errors while refactoring programs have not been studied. Our work addresses this gap by first introducing the concept of refactoring misconception (RM). Then, we use a dataset from an experiment in which students worked on refactoring exercises in a tutoring system [23]. From this dataset, we analyze 482 sequences of program snapshots from 133 students. We inspect all consecutive program snapshots in which a student turns a correct program state into an incorrect state, due to an incorrect refactoring step. Our goal is to identify and develop an organized collection of RMs held by students, where we focus on small-scale refactorings. The research question that guides this study is:

RQ: *What are the common refactoring misconceptions that students hold when refactoring simple programs? To which code quality issues are they connected?*

Our main contributions in this work are: (1) the definition of a new concept of refactoring misconception and (2) the development of a structured collection of refactoring misconceptions. The rest of this paper is organized as follows: in Section 2, we present background and related studies; in Section 3, we define the concept of a refactoring misconception and provide examples of RMs; we describe our processes of data collection and analysis in Section 4; in Section 5, we present a list of RMs as well as discuss a few relevant cases; Section 6 is dedicated to the concluding remarks and future work.

2 BACKGROUND AND RELATED WORK

Code Quality. Code quality has been a topic of interest in the CS Education community. An ITiCSE working group has investigated how code quality is perceived by educators, students and professional developers [5]. A common definition of code quality was not found, but a few terms were often used to describe code quality, typically associated with readability and structure. In the present work, we follow Stegeman et al.'s [35] definition of code quality: *"an aspect of software quality that concerns directly observable properties of source code"*, such as algorithmic aspects (flow, expressions) and structure (decomposition, modularization).

One way to evaluate code quality is through the presence or absence of quality issues. Beck and Fowler [3] label these quality issues as *code smells*. McConnell [28] presents examples of code smells, which range from issues that may be found in smaller programs, e.g. having a deeply nested loop or duplicated code, to issues related to more advanced topics, such as changing multiple inheritance hierarchies in parallel after code updates.

Several recent studies focus on investigating code smells found in student programs [6, 11, 12, 19, 40]. For example, Keuning et al. [22] examine code quality issues present in Java snapshots written by novice programmers over a period of time. They compose a list of 24 major issues and find that students rarely fix these issues. Similarly, Effenberger and Pelánek [13] identify code quality issues in functionally correct Python programs for introductory programming level problems. Furthermore, Luxton-Reilly et al. [27] propose a taxonomy to classify different student program solutions in terms of structure, syntax and style.

Code Refactoring. Fowler [14] describes refactoring as *"a change made to the internal structure of software without modifying its observable behavior"*. Although *software* is usually associated with large systems, code refactoring can also be used to improve the quality of small programs. Beck and Fowler [3] present several code smells that may indicate a need for refactoring and a catalog of such refactorings to address these smells. For instance, the *Extract Method* refactoring can be used to break down a long and complex method into smaller ones; another example is the *Replace Magic Number with Symbolic Constant* refactoring, which can be used to increase code readability.

Fowler suggests that refactoring code may have several benefits for programmers, including better code understanding and finding bugs [14]. Most past studies addressed the impact of these advantages on large systems, which are typically designed and maintained by experienced developers. Nonetheless, there have been recent efforts to also present this topic to novices. Izu et al. [20] developed a resource to support students with identifying and refactoring code related to conditional statements; Ureel II and Wallace [37] introduce a tool to detect antipatterns produced by novices and to help them preserve promising code fragments; Wiese et al. [39] analyze student difficulties with coding style when editing code; Keuning et al. [24] designed a tutoring system to help students improve functionally correct code; Birillo et al. [4] propose a tool to assess the code quality of student programming solutions, and compare their tool with Keuning et al.'s [24] tutoring system; and Choudhury et al. [10] developed a programming tutor supporting student code style skills.

Programming Misconceptions. The term *misconception* has long been used in the field of learning sciences. Smith III et al. [33] characterize misconception as *"a student conception that produces a systematic pattern of errors"*. Likewise, VanLehn [38] describes these systematic errors as *bugs*, whereas occasional and unintentional actions are *slips*.

There does not seem to be consensus on the definition of a programming misconception in the context of programming education. Chiodini et al. [9] present a few definitions from other studies, such as Sorva's [34], in which misconception is used to refer to *"understandings that are deficient or inadequate for many practical programming contexts"*, and Qian and Lehman [31]'s, that defines it as errors in conceptual understanding of programming topics.

Chiodini et al. introduce the concept of *programming language misconception*, which is *"a statement that can be disproved by reasoning entirely based on the syntax and/or semantics of a programming language"* [9]. They also propose a structure for organizing collections of programming misconceptions, and present an inventory of programming misconceptions following this structure.[1] Each misconception is exemplified and described in terms of its possible *origin*, typical *symptoms*, *value* for the CS community as well as a proper *correction* for the error. The listing below shows an example of a programming misconception, *AssignCompares*, in which a single equal operator (=) is incorrectly used in an attempt to compare two values.

```
if (x = 1) {
    ...
}
```

Several other studies also address the topic of programming misconceptions. Sirkiä and Sorva [32] analyze the typical mistakes that students make in the context of programming with a visual program simulation tool, compiling a total of 26 programming misconceptions. Hristova et al. [18] identify a set of 20 programming mistakes that often occur in student solutions. Based on their work, Brown and Altadmri [7] survey educators to find out how often students make such mistakes, and how much time they need to fix them. Gusukuma et al. [15] evaluate the impact of a misconception-driven feedback model on student learning.

Refactoring Behavior. Keuning et al. [23] designed and evaluated a refactoring tutoring system to help students refactor semantically correct code. They carried out an experiment in which 133 students used the tutoring system to improve code of six programming exercises. Their work focused on the analysis of student behavior when using the system, such as the number of students who requested hints to refactor code or completed each exercise.

Our work focuses on refactoring steps that students take to improve code and their errors while refactoring those exercises. We use their dataset to perform a deep analysis of student code submissions by examining the refactoring steps taken between each snapshot. Our goal is to identify and categorize incorrect refactoring steps taken by students between these states that might have been caused by a misconception.

[1] https://progmiscon.org/

3 REFACTORING MISCONCEPTIONS

We define a refactoring misconception as follows:

A refactoring misconception (RM) is an error made by a programmer when refactoring semantically correct code resulting in incorrect code. The error shows an inadequate understanding of a particular programming concept.

We give two typical examples of RMs. First, a programmer tries to shorten the arithmetic expression in the green frame below. They remove the second reference to the variable *score* and intend to use a compound operator (-=), but do not swap the order of the equal and minus sign. As a result, the refactored expression in the red frame assigns -3 to the variable *score*.

```
score = score - 3;
```

→

```
score =- 3;
```

In the next example, a programmer tries to merge the two nested conditions in the green frame. The red frame shows how the programmer uses an &&-operator to combine the boolean expressions, but does not take into account the else part of the outer-if when introducing the new boolean expression.

```
if (posOnly) {
    if (value >= 0) {
        sum += value;
    }
}
else {
    sum += value;
}
```

→

```
if (posOnly && value >= 0) {
    sum += value;
}
```

We do not consider the following errors to be refactoring misconceptions:

- Syntax errors and typos.
- Occasional errors – described elsewhere as slips [38].
- Programming misconceptions – errors when incrementally developing code.

A refactoring misconception occurs when an existing piece of semantically correct code is incorrectly changed, whereas a programming misconception occurs when a programmer follows specifications to write a program from scratch, and makes an error. A refactoring misconception can be identified through the analysis of the *transition* between two code snapshots, whilst a programming misconception can be detected in a *single* code snapshot. Thus, the concept of refactoring misconception is fundamentally different from programming misconception.

4 METHOD

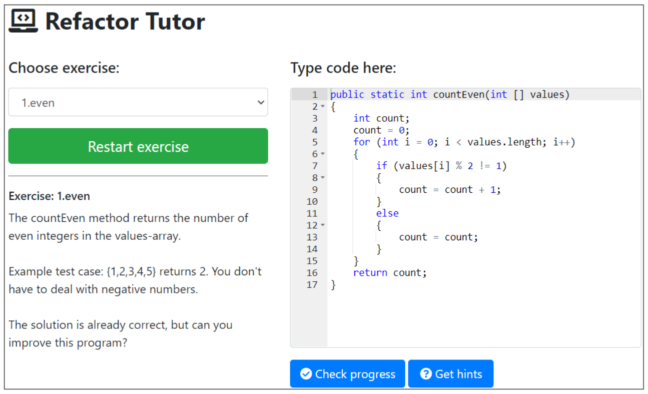
In this section, we describe the tutoring system from which we gathered data as well as the use of both transaction log analysis and a grounded theory-based approach for our data collection and analysis.

4.1 The RPT System

The Refactoring Programming Tutor (RPT)[2] is a web-based system which currently includes six Java programming exercises. In each exercise, functionally correct code containing a number of quality issues is given. The task is to remove the issues by refactoring the code. The RPT target audience are undergraduate students, who have already studied programming basics – such as loops, conditionals and functions [23].

As presented in Figure 1, the RPT system contains a regular code editor as well as two features to support code refactoring: (1) *Check progress* and (2) *Get hints*. When a student checks their progress, the system diagnoses the current and the previous state of the code mainly to determine whether the code is still correct and whether one or more code quality issues were solved. After evaluating the code, the system presents one of these diagnoses:

- *Expected*: RPT identifies an adequate refactoring step.
- *Correct*: the code is correct, but RPT cannot recognize what has been changed.
- *Similar*: the code has not changed much in comparison to its previous state.
- *Buggy*: RPT identifies an incorrect refactoring step.
- *Test case failed*: the code is not functionally correct.
- *Compiler error*: the code contains a syntax error or an unsupported language construct.



Exercise: 1.even

The countEven method returns the number of even integers in the values-array.

Example test case: {1,2,3,4,5} returns 2. You don't have to deal with negative numbers.

The solution is already correct, but can you improve this program?

```
public static int countEven(int [] values)
{
    int count;
    count = 0;
    for (int i = 0; i < values.length; i++)
    {
        if (values[i] % 2 != 1)
        {
            count = count + 1;
        }
        else
        {
            count = count;
        }
    }
    return count;
}
```

Figure 1: RPT - Home screen

4.2 Dataset

In 2019, Keuning et al. [23] carried out an experiment at Windesheim University of Applied Sciences, in the Netherlands, with mostly second-year Computing students, who were enrolled in a C# programming course. A total of 133 students gave consent to participate in the experiment in which they attempted to solve the refactoring exercises present in RPT.

This experiment resulted in a dataset with 12,254 log entries. The dataset includes, amongst others, the program snapshots from the students. A program snapshot contains the current state of the program that the student is working on. A new snapshot is logged in the database when the student clicks on the *Check progress* or *Get hints* button.

[2] http://hkeuning.nl/rpt

Table 1: Refactoring Misconceptions

Misconception	Description	Occ.
Arithmetic Expressions		
A1. IncorrectArithmeticExpressionUpdate	Update an arithmetic expression incorrectly.	10
A2. DoubleNegatedArithmeticExpression	Use an unnecessary minus sign as a compound operator in a subtraction.	3
A3. BadArithmeticExpressionShortening *	Simplify an arithmetic expression incorrectly.	23
Boolean Expressions		
B1. IncorrectNegationEvenCheck	Incorrectly change an even/odd check.	9
B2. IncorrectBooleanExpressionSimplification *	Incorrectly simplify a single boolean expression.	12
B3. IncorrectComposedExpressionSimplification *	Incorrectly simplify a composed boolean expression.	8
B4. UnnecessaryComparisonOperatorUpdate	Update a comparison operator unnecessarily.	14
B5. UnnecessaryLogicalOperatorUpdate	Update a logical operator unnecessarily.	4
Conditionals		
C1. IfsMerged *	Merge if statements incorrectly.	45
C2. RequiredIfStatementRemoved	Remove a required if statement.	38
C3. UnnecessaryElseKept	Keep an unnecessary else block after an update in the if statement.	35
C4. BadIfElseSimplification *	Update a boolean expression incorrectly in an if-else block simplification.	28
C5. ConditionalCopiedToOuterIf	Copy an unnecessary boolean expression from the inner if to the outer if statement.	10
C6. ElseRemoved	Remove an else block without updating the boolean expression in the if statement.	37
Flow		
F1. ReturnMoved	Move a flow-related command (return/break/continue) to an incorrect location.	7
F2. ElseBlockMovedToInnerIf	Unnecessarily move an else block from the outer if to the inner if statement.	4
F3. StatementReplacedByFlowCommand	Replace a statement/group of commands by an incorrect flow-related command.	6
F4. ReplacedByReturn *	Incorrectly move a return command to replace a statement.	5
F5. SelfAssignmentReplacedByReturn	Replace a self-assignment by a return command.	6
Loops		
L1. ForEachVariablesNotUpdated *	Replace a for loop by a for-each loop without updating the references to the variables.	71
L2. BadForLoopValues	Use an incorrect initial value/increment in a for loop.	9
L3. ForReplacedByForEach *	Replace a for loop by an incorrect for-each loop.	18
L4. ReversedComparisonOperatorInWhile	Reverse a comparison operator in a while loop unnecessarily.	6
L5. IncorrectForLoopBreak	Use an incorrect break condition in a for loop.	7
L6. ForReplacedByWhile	Replace a for loop by an incorrect while loop.	3

An asterisk (*) indicates that the RM is discussed in this paper.

4.3 Data Analysis

To answer our research question, we performed a transaction log analysis (TLA) [21], addressing three major stages: (i) data collection, (ii) data preparation and (iii) data analysis. For the first TLA stage, we used the dataset from Keuning et al. [23] described in 4.2.

In our analysis, we were particularly interested in examining four different scenarios, in which a student program state went from a previous *"good state"* – (1) Expected, (2) Correct, (3) Similar or (4) at the beginning of the exercise – to a failed test case, which is labeled as *Not Equivalent* in the dataset.

For the data preparation stage, we retrieved all consecutive snapshots from Exercises 1 to 5 in the dataset, in which a good state submission was followed by a submission with a Not Equivalent state. Submissions for Exercise 6 were not considered in this study, since this exercise asks to design a complete solution to a problem from scratch, and is less directly related to refactoring code.

For the final TLA stage, we adopted a grounded theory-based analysis in which three authors actively participated. Our analysis followed the first two iterative coding phases proposed by Strauss and Corbin [36], which are *open coding* and *axial coding*. We did not perform the third phase, *selective coding*, as we were not proposing a new theory or modifying a previous one. We performed the following activities:

Open coding. Initially, one researcher identified all incorrect refactoring steps between two program snapshots which transformed a correct program state to an incorrect state. Then, the researcher inspected the sequences of program snapshots surrounding (before and after) these incorrect states. For group discussion, the researcher selected a subset of those sequences that represented the most common student errors. Then, all three researchers individually analyzed this subset of program snapshots and developed an initial coding of misconceptions.

Axial coding. One researcher evaluated the initial coding and proposed a preliminary categorization of those misconceptions. Then, the researchers carried out multiple rounds of discussion to refine the collection of misconceptions and categorization. For this, each misconception was analyzed in terms of the conditions in which it occurred, such as the type of exercise and code structures involved. When fewer than three different students performed the same incorrect step, the researchers considered it to be an occasional error, not an actual RM.

5 RESULTS AND DISCUSSION

We analyzed 482 sequences of student program snapshots and identified 25 refactoring misconceptions. We classified each misconception according to the code structure related to it. The only exception is *Flow*, which derives from Stegeman et al.'s rubric [35]. Our five categories are (1) Arithmetic Expressions, (2) Boolean Expressions, (3) Conditionals, (4) Flow and (5) Loops. The list of refactoring misconceptions is shown in Table 1, which also includes a short description and the number of occurrences of each misconception. A full description of the RMs along with code examples can be found online.[3] Here we discuss eight cases of frequent refactoring misconceptions in student solutions. We give at least one example from each category, and also include some particularly illustrative examples.

Case C1. IfsMerged. In Exercise 2, combining two nested if statements into a single one is considered a desirable refactoring step, as it reduces duplication. 45 students took incorrect steps when attempting to solve this issue. In Section 3, this RM case was further discussed. In addition to these 45 cases, RPT already recognizes a specific instance of this error, of which 441 occurrences were found in the dataset.

Case A3. BadArithmeticExpressionShortening. In Exercises 1, 3 and 4, using compound operators to simplify an arithmetic expression is an appropriate refactoring step. A common student error when attempting to do this is described in Section 3. Another typical error was to assign the simplified expression to *count* itself (*count = count++*), which does not increase the value of *count*.

```
count = count + 1;
```

→

```
count = count++;
```

Case B2. IncorrectBooleanExpressionSimplification. In Exercise 3, a possible refactoring step concerns the shortening of a boolean expression, in which, for example, *stop == false* can be simplified to *!stop*. 12 students rewrote this expression omitting the logical NOT operator (!), consequently changing the semantic of the expression. This error might have been influenced by the shortening of a boolean expression in the previous exercise, in which *posOnly == true* should be shortened to *posOnly*.

```
if (stop == false) {
    total += array[i];
}
```

↓

```
if (stop) {
    total += array[i];
}
```

Case B3. IncorrectComposedExpressionSimplification. To deal with the merging of nested if statements (Case C1) in Exercise 2, students wrote a composed boolean expression in a single if statement. This new expression typically contained an unnecessary check for the value of the variable *posOnly* being true. When students attempted to simplify the expression, a common incorrect step was to remove another (required) part of the expression, not the actual redundant check.

```
if (!posOnly || (posOnly && i > 0)) {
    sum += i;
}
```

↓

```
if (posOnly && i > 0) {
    sum += i;
}
```

Case C4. BadIfElseSimplification. In Exercise 4, simplifying an if-else statement to a single if statement is part of a larger correct refactoring step, as shown in the green frame below. In most cases, students properly extracted the return statements from both if and else blocks, but could not apply De Morgan's rules [16] properly when updating the boolean expression. When attempting to move the remaining statement (*score -= 3*) to the if block and remove else, 28 students also updated the expression by replacing EQUAL operators by NOT EQUAL (!= symbol), but they did not update the OR-operator by an AND-operator.

```
if (day == 6 || day == 7) {
    return score;
} else {
    score -= 3;
    return score;
}
```

→

```
if (day != 6 || day != 7) {
    score -= 3;
}
return score;
```

Case F4. ReplacedByReturn. An expected refactoring step in Exercise 3 is to remove the *stop* variable, which handles the for loop flow, and to exit this loop as soon as a given condition is met – this is displayed in the green frame below. When attempting to code an immediate exit to the for loop, students moved the method return to the loop. A possible correct refactoring step here could be the use of a break command to exit the for-loop.

```
for (int i = 1; i < arr.length; i += 2) {
    if (!stop) {
        if (arr[i] != -1) {
            total += arr[i];
        } else {
            stop = true;
        }
    }
}
return total;
```

↓

```
for (int i = 1; i < arr.length; i += 2) {
    if (arr[i] != -1) {
        total += arr[i];
    } else {
        return total;
    }
}
```

[3]https://sites.google.com/view/refactoring-misconceptions

Case L1. ForEachVariablesNotUpdated. In Exercises 1 and 2, replacing a for loop by a for-each loop is considered a correct refactoring step – it is a safer option when iterating through the arrays, since their elements cannot be modified during the iteration. 71 students took an incorrect step in which they did not update the reference to the for-each variable created to get the current item. In a for loop, this can be done by referring to the array along with the index (*nums[i]* in the green frame), whereas in a for-each loop, it is only necessary to refer to the element (*i* in the red frame).

```
for (int i = 0; i < nums.length; i++) {
    sum += nums[i];
}
```

↓

```
for (int i : nums) {
    sum += nums[i];
}
```

Case L3. ForReplacedByForEach. In Exercises 1 and 2, replacing a for loop by a for-each loop is considered a proper refactoring step. This is not the case in Exercise 3, since the for loop in this exercise should only iterate through the elements at odd indices. Still, 18 students incorrectly attempted to replace the for loop by a for-each loop. In some cases, the *i* variable from the for-each loop, which gets an element from the array, was incorrectly used in an attempt to increment the index.

```
for (int i = 1; i < arr.length; i += 2) {
    if (arr[i] != -1) {
        total += arr[i];
    } else {
        break;
    }
}
```

↓

```
for (int i : arr) {
    if (i != -1) {
        total += i;
    } else {
        break;
    }
    i += 2;
}
```

Summary. Our results show that several student refactoring misconceptions involve rewriting boolean expressions. We find common student errors when updating conditionals (C1, C4), which is also discussed by Izu et al. [20]. These errors may indicate an insufficient understanding of formal logic, such as how to apply De Morgan's rules in boolean expressions – an issue also identified in other studies [1, 17]. We think that the typical RMs from the other categories may be related to a lack of comprehension of code flow and control structures, as a considerable number of students could not distinguish when to use and how to update a few structures. Two examples are the incorrect use of a *return* command (F1, F4, F5) and an incorrect association of a for-each loop with indices (L1). We expect that our set of RMs may also be found when students work in other imperative programming languages, such as C# or Python, but with a slightly different syntax.

5.1 Threats and Limitations

A few aspects need to be considered when interpreting our results. In this paper, we focus on code refactoring steps that lead to semantically incorrect code. We do not address refactoring steps that decrease code quality, since these are not considered refactoring misconceptions in our definition.

Also, Keuning et al.'s [24] tutoring system may not have diagnosed all student steps correctly, including both correct refactoring steps and errors. We attempted to mitigate this by manually inspecting a sample of student snapshots in our analysis. Another aspect of the tutoring system that may have influenced the data and consequently our results is the wording of the hints available for each exercise. In some cases, these hints might have influenced students to take incorrect refactoring steps.

In addition, not all students worked on all five exercises. Most students had difficulties completing Exercises 2 and 3. As a consequence, the last two exercises were mostly attempted by more experienced students, who succeeded in completing the previous exercises. We also recognize that the experiment was conducted in a specific scenario: a homogeneous group of students who worked on five programming exercises containing specific code quality issues. Therefore, further research is needed to improve the validity and extend our set of RMs.

6 CONCLUSIONS AND FUTURE WORK

We analyzed 482 sequences of program snapshots from 133 students working on refactoring exercises in a tutoring system, and identified 25 refactoring misconceptions. This work makes the following contributions: First, the introduction of the concept of a refactoring misconception. Then, the development of an initial catalogue of refactoring misconceptions held by CS students.

From this study, we advise educators to address code refactoring topics in programming courses, including both the use of correct refactoring rules and the existence of refactoring misconceptions, specifically on the use of multiple control structures and formal logic. We also hope that developers of programming tutors take refactoring misconceptions into account to provide adequate feedback to student refactoring errors.

As future work, we intend to replicate this study in other contexts, extend our set of RMs and determine their occurrences in other datasets. Also, we are conducting a think-aloud study with students working on refactoring exercises, to better comprehend their reasoning when refactoring code.

REFERENCES

[1] Vicki L Almstrum. 1996. Investigating student difficulties with mathematical logic. *Teaching and Learning Formal Methods* (1996).
[2] Jecton Tocho Anyango and Hussein Suleman. 2018. Teaching Programming in Kenya and South Africa: What is difficult and is it universal?. In *Koli Calling.*

[3] Kent Beck and Martin Fowler. 2018. Bad Smells in Code. In *Refactoring: improving the design of existing code*. Chapter 3.

[4] Anastasiia Birillo, Ilya Vlasov, Artyom Burylov, Vitalii Selishchev, Artyom Goncharov, Elena Tikhomirova, Nikolay Vyahhi, and Timofey Bryksin. 2022. Hyperstyle: A Tool for Assessing the Code Quality of Solutions to Programming Assignments. In *SIGCSE*.

[5] Jürgen Börstler, Harald Störrle, Daniel Toll, Jelle Van Assema, Rodrigo Duran, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, and Bonnie MacKellar. 2018. " I know it when I see it" Perceptions of Code Quality: ITiCSE'17 Working Group Report. In *ITiCSE*.

[6] Dennis M Breuker, Jan Derriks, and Jacob Brunekreef. 2011. Measuring static quality of student code. In *ITiCSE*.

[7] Neil CC Brown and Amjad Altadmri. 2017. Novice Java programming mistakes: Large-scale data vs. educator beliefs. *Transactions on Computing Education* (2017).

[8] Yuliya Cherenkova, Daniel Zingaro, and Andrew Petersen. 2014. Identifying challenging CS1 concepts in a large problem dataset. In *SIGCSE*.

[9] Luca Chiodini, Igor Moreno Santos, Andrea Gallidabino, Anya Tafliovich, André L Santos, and Matthias Hauswirth. 2021. A curated inventory of programming language misconceptions. In *ITiCSE*.

[10] Rohan Roy Choudhury, Hezheng Yin, and Armando Fox. 2016. Scale-driven automatic hint generation for coding style. In *ITS*.

[11] Giuseppe De Ruvo, Ewan Tempero, Andrew Luxton-Reilly, Gerard B Rowe, and Nasser Giacaman. 2018. Understanding semantic style by analysing student code. In *ACE*.

[12] Stephen H Edwards, Nischel Kandru, and Mukund BM Rajagopal. 2017. Investigating static analysis errors in student Java programs. In *ICER*.

[13] Tomáš Effenberger and Radek Pelánek. 2022. Code Quality Defects across Introductory Programming Topics. In *SIGCSE*.

[14] Martin Fowler. 2018. *Refactoring: improving the design of existing code*.

[15] Luke Gusukuma, Austin Cory Bart, Dennis Kafura, and Jeremy Ernst. 2018. Misconception-driven feedback: Results from an experimental study. In *ICER*.

[16] Paul Richard Halmos. 1960. *Naive set theory*. van Nostrand.

[17] Geoffrey L Herman, Michael C Loui, Lisa Kaczmarczyk, and Craig Zilles. 2012. Describing the what and why of students' difficulties in Boolean logic. *Transactions on Computing Education* (2012).

[18] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. Identifying and correcting Java programming errors for introductory computer science students. *SIGCSE* (2003).

[19] Christopher Hundhausen, Anukrati Agrawal, Dana Fairbrother, and Michael Trevisan. 2009. Integrating pedagogical code reviews into a CS 1 course: an empirical study. *SIGCSE* (2009).

[20] Cruz Izu, Paul Denny, and Sayoni Roy. 2022. A Resource to Support Novices Refactoring Conditional Statements. In *ITiCSE*.

[21] Bernard J Jansen. 2006. Search log analysis: What it is, what's been done, how to do it. *Library & information science research* (2006).

[22] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code quality issues in student programs. In *ITiCSE*.

[23] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2020. Student refactoring behaviour in a programming tutor. In *Koli Calling*.

[24] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2021. A tutoring system to learn code refactoring. In *SIGCSE*.

[25] Diana Kirk, Tyne Crow, Andrew Luxton-Reilly, and Ewan Tempero. 2020. On assuring learning about code quality. In *ACE*.

[26] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A study of the difficulties of novice programmers. *SIGCSE* (2005).

[27] Andrew Luxton-Reilly, Paul Denny, Diana Kirk, Ewan Tempero, and Se-Young Yu. 2013. On the differences between correct student solutions. In *ITiCSE*.

[28] Steve McConnell. 2004. *Code complete*.

[29] Ioana T Mow. 2008. Issues and difficulties in teaching novice computer programming. In *Innovative Techniques in Instruction Technology, E-learning, E-assessment and Education*.

[30] Sebastian C Müller and Thomas Fritz. 2016. Using (bio) metrics to predict code quality online. In *ICSE*.

[31] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *Transactions on Computing Education* (2017).

[32] Teemu Sirkiä and Juha Sorva. 2012. Exploring programming misconceptions: an analysis of student mistakes in visual program simulation exercises. In *Koli Calling*.

[33] John P Smith III, Andrea A DiSessa, and Jeremy Roschelle. 1994. Misconceptions reconceived: A constructivist analysis of knowledge in transition. *The journal of the learning sciences* (1994).

[34] Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *Transactions on Computing Education* (2013).

[35] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. 2016. Designing a rubric for feedback on code quality in programming courses. In *Koli Calling*.

[36] Anselm Strauss and Juliet Corbin. 1990. *Basics of qualitative research*.

[37] Leo C Ureel II and Charles Wallace. 2019. Automated critique of early programming antipatterns. In *SIGCSE*.

[38] Kurt VanLehn. 1982. Bugs are not enough: Empirical studies of bugs, impasses and repairs in procedural skills. *The Journal of Mathematical Behavior* (1982).

[39] Eliane S Wiese, Anna N Rafferty, Daniel M Kopta, and Jacqulyn M Anderson. 2019. Replicating novices' struggles with coding style. In *ICPC*.

[40] Eliane S Wiese, Michael Yen, Antares Chen, Lucas A Santos, and Armando Fox. 2017. Teaching students to recognize and implement good coding style. In *L@S*.