Investigating Student Errors in Code Refactoring

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

ABSTRACT

Learning to develop code of good quality is challenging. One way to improve code quality is through code refactoring. Students make several mistakes when refactoring code. This research project aims to comprehend student errors in code refactoring, as well as to evaluate how the use of automated tools can help students remediate these errors.

CCS CONCEPTS

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

KEYWORDS

code refactoring; code quality; refactoring misconceptions; student refactoring errors; refactoring tools; programming education

ACM Reference Format:
Eduardo Oliveira. 2023. Investigating Student Errors in Code Refactoring. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 2 (ITiCSE 2023), July 8–12, 2023, Turku, Finland.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3587103.3594146

1 CONTEXT AND MOTIVATION

Writing code of good quality is a challenge for students. Code refactoring is a common approach to support the development of high quality code. Refactoring code may contribute in different ways to software development, such as an increase in code maintainability and comprehension. When refactoring code, students take various incorrect refactoring steps. These errors might be caused by misconceptions that students hold. Previous studies in code refactoring mostly focus on other topics, such as the use of automated tools for teaching, but little has been investigated on the errors that students make when refactoring code. The present research project aims to address this gap, as well as to evaluate how the use of automated tools can help with the remediation of student refactoring errors.

2 BACKGROUND

Stegeman et al. [9] describe **code quality** as *"an aspect of software quality that concerns directly observable properties of code"*, such as the organization of the control flow, use of expressions and code structure in terms of decomposition and modularization. Recent ITiCSE working groups have investigated the perception of code quality from the perspective of students, educators and professional developers [2], as well as the quality of example programs used in textbooks from CS1 courses [1].

One way to address code quality in programming courses is through the teaching of **code refactoring**. Fowler [3] defines refactoring as a *"change made to the internal structure of a software (...) without changing its observable behavior"*. Due to the potential benefits that refactoring code may bring to programmers, such as better code comprehension and bug detection [3], educators have taught code refactoring to novice CS students.

Professional tools have been used to support the teaching of refactoring. However, using such tools for code refactoring may be hard for students, since these tools are often too advanced for novices and not designed to support learning [5]. The recent emerge and dissemination of AI models in education, such as ChatGPT and Copilot, may help students with code refactoring. However, this still needs to be investigated.

Another approach is the development of educational tools and resources for code refactoring. For instance, Ureel II and Wallace [10] have designed a tool that assists with the detection of student programming antipatterns while attempting to salvage promising portions of their code. Izu et al. [4] have proposed a resource to help students identify and refactor code smells when writing conditional statements. Keuning et al. [7] have developed the Refactoring Programming Tutor (RPT)[1], a tutoring system that helps students improve functionally correct code.

In another study, Keuning et al. [6] performed an experiment in which 133 students improved code of six programming exercises present in RPT. Their analysis include general aspects of student refactoring behavior when using the system, including the number of students who completed each exercise or asked for hints to solve a specific code quality issue. The study does not address specific aspects of student refactoring steps, such as student errors when refactoring code.

3 PROBLEM STATEMENT

Currently, there is little known about how students approach refactoring, such as the steps to remove code smells and the errors that students make. It is also unclear how the use of supporting tools can contribute to the student learning of code refactoring and remediation of refactoring errors.

4 RESEARCH GOALS

The main goals of this research project are to explore student errors when refactoring code and develop supporting tools to remediate these errors. The research questions that guide this project are:

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

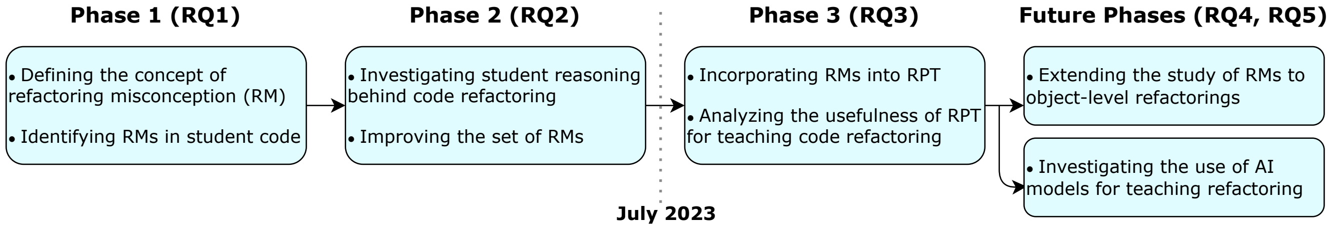
Phase 1 (RQ1)
• Defining the concept of refactoring misconception (RM)
• Identifying RMs in student code
Phase 2 (RQ2)
• Investigating student reasoning behind code refactoring
• Improving the set of RMs
Phase 3 (RQ3)
• Incorporating RMs into RPT
• Analyzing the usefulness of RPT for teaching code refactoring
Future Phases (RQ4, RQ5)
• Extending the study of RMs to object-level refactorings
• Investigating the use of AI models for teaching refactoring
July 2023

Figure 1: Research Project Plan

RQ1: What are the common refactoring misconceptions that students hold when refactoring simple programs? To which code quality issues are they connected?

RQ2: What are the student reasonings behind their refactoring misconceptions?

RQ3: How can the findings on code refactoring misconceptions be incorporated into a refactoring tool?

RQ4: What are the common refactoring misconceptions that students hold regarding object-level refactorings?

RQ5: How can the use of AI models impact the teaching and learning of code refactoring?

5 METHOD

Figure 1 summarizes the research project plan. For RQ1, we analyzed the dataset from Keuning et al.'s [6] experiment. This dataset contains program snapshots of students working on refactoring exercises from RPT. In our analysis, we used grounded theory to identify and categorize incorrect refactoring steps student took when refactoring code. We define such an error as a **refactoring misconception** [8]: *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.* As a result of the analysis, we have identified 25 code refactoring misconceptions.

The first phase of our research project focused on *what* are the students' incorrect refactoring steps. However, we still need to comprehend *why* students take particular steps. To answer this question (RQ2) and to extend our set of RMs, we have recently carried out a think-aloud experiment with 12 CS students working on five refactoring exercises. Each exercise was functionally correct, but contained a number of code quality issues. The student task was to remove these issues with code refactoring while verbalizing their ideas. Data analysis for this study involves grounded theory and is currently in progress.

From the think-aloud study, we envision to better comprehend student reasoning when taking incorrect refactoring steps. This understanding may help us with the next phases, which involves incorporating the refactoring misconceptions into RPT. Currently, the tutoring system cannot recognize these misconceptions. We plan to make RPT detect the most common misconceptions, as well as to offer adequate hints and feedback concerning these misconceptions. Thereafter for RQ3, we plan to verify the usefulness of these changes in RPT for students working on refactoring exercises.

Possible future directions for this research project include extending the study of code refactoring misconceptions to more advanced programming topics, such as object-level design. Another direction could be the investigation of the use of AI models, such as ChatGPT or Copilot, for the teaching and learning of code refactoring.

6 CONTRIBUTIONS

Our current contributions are the introduction of a formal definition for the concept of refactoring misconceptions and the development of a structured collection of such refactoring misconceptions. A full description of the RMs and code examples can be found online.[2] As future contributions, we first expect to obtain an insight into the underlying reasonings behind student RMs. Later, we foresee to improve an existing refactoring tutoring system to identify those misconceptions, as well as to investigate the effects for students using such a tool for learning code refactoring.

REFERENCES

[1] Jürgen Börstler, Mark S Hall, Marie Nordström, James H Paterson, Kate Sanders, Carsten Schulte, and Lynda Thomas. 2010. An evaluation of object oriented example programs in introductory programming textbooks. *SIGCSE Bulletin* (2010).
[2] 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.*
[3] Martin Fowler. 2018. *Refactoring: improving the design of existing code.*
[4] Cruz Izu, Paul Denny, and Sayoni Roy. 2022. A Resource to Support Novices Refactoring Conditional Statements. In *ITiCSE.*
[5] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code quality issues in student programs. In *ITiCSE.*
[6] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2020. Student refactoring behaviour in a programming tutor. In *Koli Calling.*
[7] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2021. A tutoring system to learn code refactoring. In *SIGCSE.*
[8] Eduardo Oliveira, Hieke Keuning, and Johan Jeuring. 2023. Student Code Refactoring Misconceptions. In *ITiCSE (Forthcoming).*
[9] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. 2016. Designing a rubric for feedback on code quality in programming courses. In *Koli Calling.*
[10] Leo C Ureel II and Charles Wallace. 2019. Automated critique of early programming antipatterns. In *SIGCSE.*

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