SSW 625 - AI for software engineering

Phase 1 Example

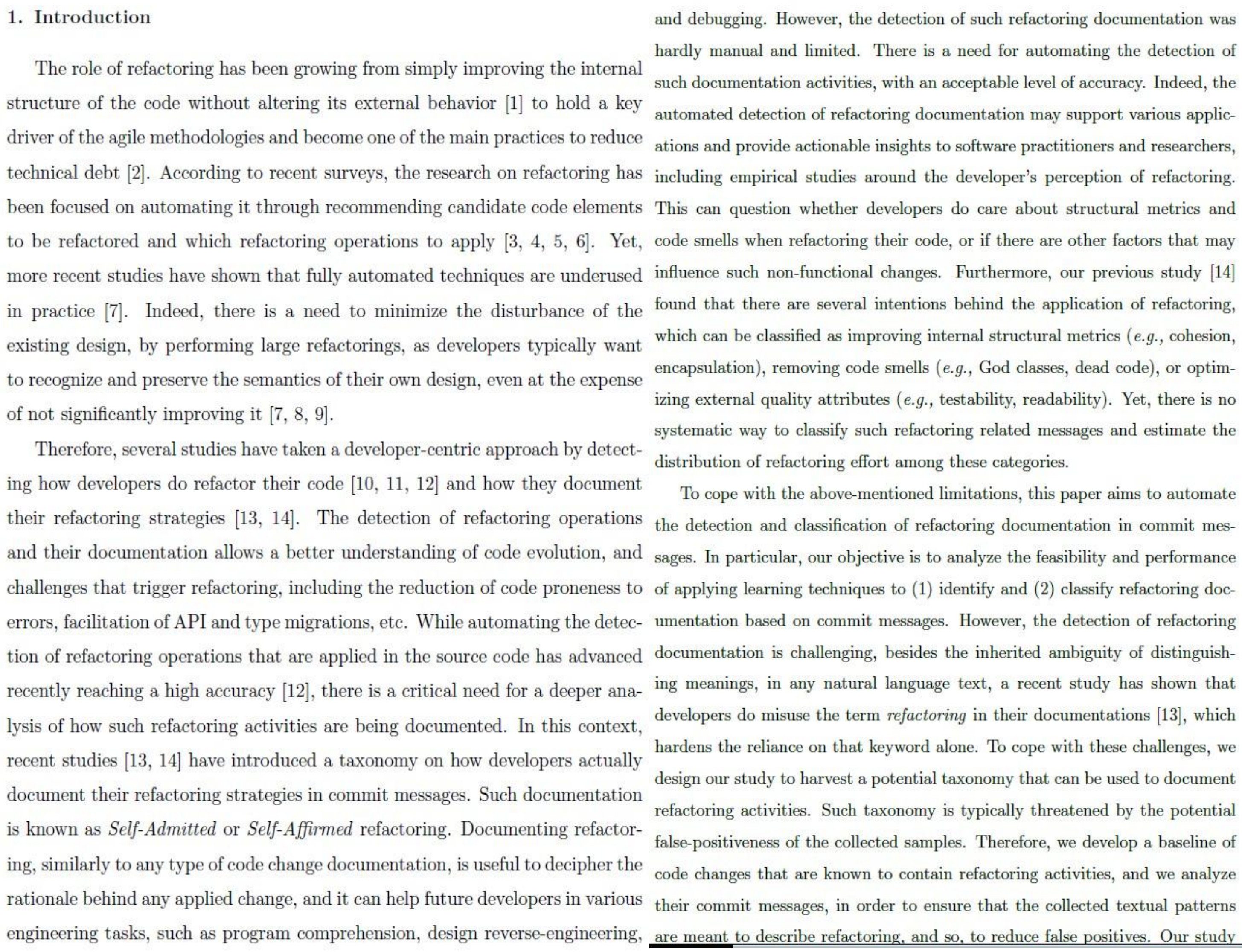
Phase 1

- **Introduction.** It introduces the topic. It contains the necessary background information for the type of technical concepts and keywords that will be later used.

- **Open Investigation.** It introduces the problem and explains how you are planning on talking it. You may want to give details about the problem, and what makes it challenging. Then you can give the rationale behind the proposed solution, without giving any details about its design and implementation, since that would be the content of Phase 2. You can split this sections into 2 subsections: Problem Statement and Proposed Solution.

Phase 1

- This looks like a typical traditional introduction.
- But, actually, this introduction follows an implicit template.
- This template can be your guide for phase 1.



1. Introduction

The role of refactoring has been growing from simply improving the internal structure of the code without altering its external behavior [1] to hold a key driver of the agile methodologies and become one of the main practices to reduce technical debt [2]. According to recent surveys, the research on refactoring has been focused on automating it through recommending candidate code elements to be refactored and which refactoring operations to apply [3, 4, 5, 6]. Yet, more recent studies have shown that fully automated techniques are underused in practice [7]. Indeed, there is a need to minimize the disturbance of the existing design, by performing large refactorings, as developers typically want to recognize and preserve the semantics of their own design, even at the expense of not significantly improving it [7, 8, 9].

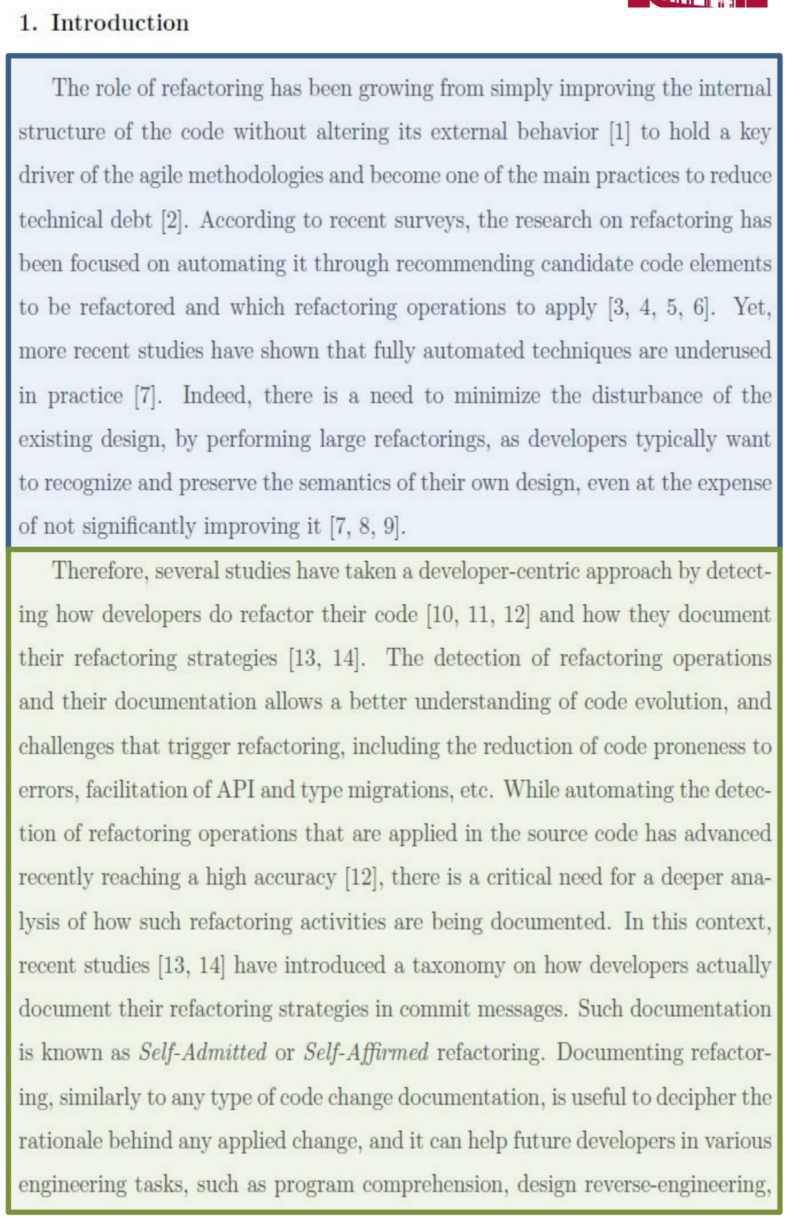
Therefore, several studies have taken a developer-centric approach by detecting how developers do refactor their code [10, 11, 12] and how they document their refactoring strategies [13, 14]. The detection of refactoring operations and their documentation allows a better understanding of code evolution, and challenges that trigger refactoring, including the reduction of code proneness to errors, facilitation of API and type migrations, etc. While automating the detection of refactoring operations that are applied in the source code has advanced recently reaching a high accuracy [12], there is a critical need for a deeper analysis of how such refactoring activities are being documented. In this context, recent studies [13, 14] have introduced a taxonomy on how developers actually document their refactoring strategies in commit messages. Such documentation is known as *Self-Admitted* or *Self-Affirmed* refactoring. Documenting refactoring, similarly to any type of code change documentation, is useful to decipher the rationale behind any applied change, and it can help future developers in various engineering tasks, such as program comprehension, design reverse-engineering, and debugging. However, the detection of such refactoring documentation was hardly manual and limited. There is a need for automating the detection of such documentation activities, with an acceptable level of accuracy. Indeed, the automated detection of refactoring documentation may support various applications and provide actionable insights to software practitioners and researchers, including empirical studies around the developer's perception of refactoring. This can question whether developers do care about structural metrics and code smells when refactoring their code, or if there are other factors that may influence such non-functional changes. Furthermore, our previous study [14] found that there are several intentions behind the application of refactoring, which can be classified as improving internal structural metrics (*e.g.*, cohesion, encapsulation), removing code smells (*e.g.*, God classes, dead code), or optimizing external quality attributes (*e.g.*, testability, readability). Yet, there is no systematic way to classify such refactoring related messages and estimate the distribution of refactoring effort among these categories.

To cope with the above-mentioned limitations, this paper aims to automate the detection and classification of refactoring documentation in commit messages. In particular, our objective is to analyze the feasibility and performance of applying learning techniques to (1) identify and (2) classify refactoring documentation based on commit messages. However, the detection of refactoring documentation is challenging, besides the inherited ambiguity of distinguishing meanings, in any natural language text, a recent study has shown that developers do misuse the term *refactoring* in their documentations [13], which hardens the reliance on that keyword alone. To cope with these challenges, we design our study to harvest a potential taxonomy that can be used to document refactoring activities. Such taxonomy is typically threatened by the potential false-positiveness of the collected samples. Therefore, we develop a baseline of code changes that are known to contain refactoring activities, and we analyze their commit messages, in order to ensure that the collected textual patterns are meant to describe refactoring, and so, to reduce false positives. Our study

Phase I Example

- This part is your introduction: contains the necessary background information for **refactoring** will be later used

- This part is your Open Investigation: It starts with talking about existing studies, then introduces their limitations, then its shows how the proposed solution will solve the detected problematic



1. Introduction

The role of refactoring has been growing from simply improving the internal structure of the code without altering its external behavior [1] to hold a key driver of the agile methodologies and become one of the main practices to reduce technical debt [2]. According to recent surveys, the research on refactoring has been focused on automating it through recommending candidate code elements to be refactored and which refactoring operations to apply [3, 4, 5, 6]. Yet, more recent studies have shown that fully automated techniques are underused in practice [7]. Indeed, there is a need to minimize the disturbance of the existing design, by performing large refactorings, as developers typically want to recognize and preserve the semantics of their own design, even at the expense of not significantly improving it [7, 8, 9].

Therefore, several studies have taken a developer-centric approach by detecting how developers do refactor their code [10, 11, 12] and how they document their refactoring strategies [13, 14]. The detection of refactoring operations and their documentation allows a better understanding of code evolution, and challenges that trigger refactoring, including the reduction of code proneness to errors, facilitation of API and type migrations, etc. While automating the detection of refactoring operations that are applied in the source code has advanced recently reaching a high accuracy [12], there is a critical need for a deeper analysis of how such refactoring activities are being documented. In this context, recent studies [13, 14] have introduced a taxonomy on how developers actually document their refactoring strategies in commit messages. Such documentation is known as *Self-Admitted* or *Self-Affirmed* refactoring. Documenting refactoring, similarly to any type of code change documentation, is useful to decipher the rationale behind any applied change, and it can help future developers in various engineering tasks, such as program comprehension, design reverse-engineering,

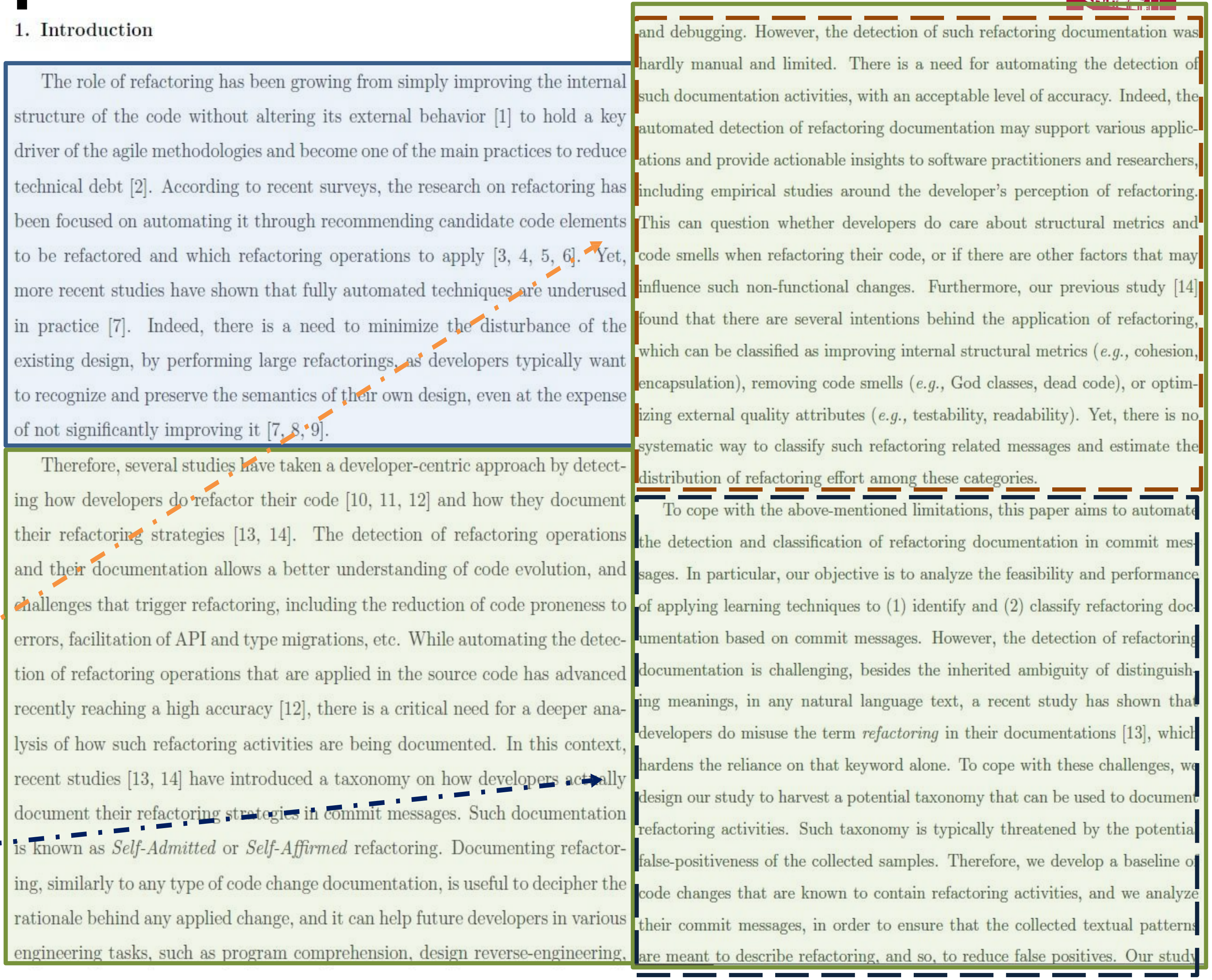
Phase I Example

Introduction and background

Existing studies

Problem Statement (optional) limitations of existing studies

Proposed solution



1. Introduction

The role of refactoring has been growing from simply improving the internal structure of the code without altering its external behavior [1] to hold a key driver of the agile methodologies and become one of the main practices to reduce technical debt [2]. According to recent surveys, the research on refactoring has been focused on automating it through recommending candidate code elements to be refactored and which refactoring operations to apply [3, 4, 5, 6]. Yet, more recent studies have shown that fully automated techniques are underused in practice [7]. Indeed, there is a need to minimize the disturbance of the existing design, by performing large refactorings, as developers typically want to recognize and preserve the semantics of their own design, even at the expense of not significantly improving it [7, 8, 9].

Therefore, several studies have taken a developer-centric approach by detecting how developers do refactor their code [10, 11, 12] and how they document their refactoring strategies [13, 14]. The detection of refactoring operations and their documentation allows a better understanding of code evolution, and challenges that trigger refactoring, including the reduction of code proneness to errors, facilitation of API and type migrations, etc. While automating the detection of refactoring operations that are applied in the source code has advanced recently reaching a high accuracy [12], there is a critical need for a deeper analysis of how such refactoring activities are being documented. In this context, recent studies [13, 14] have introduced a taxonomy on how developers actually document their refactoring strategies in commit messages. Such documentation is known as *Self-Admitted* or *Self-Affirmed* refactoring. Documenting refactoring, similarly to any type of code change documentation, is useful to decipher the rationale behind any applied change, and it can help future developers in various engineering tasks, such as program comprehension, design reverse-engineering, and debugging. However, the detection of such refactoring documentation was hardly manual and limited. There is a need for automating the detection of such documentation activities, with an acceptable level of accuracy. Indeed, the automated detection of refactoring documentation may support various applications and provide actionable insights to software practitioners and researchers, including empirical studies around the developer's perception of refactoring. This can question whether developers do care about structural metrics and code smells when refactoring their code, or if there are other factors that may influence such non-functional changes. Furthermore, our previous study [14] found that there are several intentions behind the application of refactoring, which can be classified as improving internal structural metrics (*e.g.*, cohesion, encapsulation), removing code smells (*e.g.*, God classes, dead code), or optimizing external quality attributes (*e.g.*, testability, readability). Yet, there is no systematic way to classify such refactoring related messages and estimate the distribution of refactoring effort among these categories.

To cope with the above-mentioned limitations, this paper aims to automate the detection and classification of refactoring documentation in commit messages. In particular, our objective is to analyze the feasibility and performance of applying learning techniques to (1) identify and (2) classify refactoring documentation based on commit messages. However, the detection of refactoring documentation is challenging, besides the inherited ambiguity of distinguishing meanings, in any natural language text, a recent study has shown that developers do misuse the term *refactoring* in their documentations [13], which hardens the reliance on that keyword alone. To cope with these challenges, we design our study to harvest a potential taxonomy that can be used to document refactoring activities. Such taxonomy is typically threatened by the potential false-positiveness of the collected samples. Therefore, we develop a baseline of code changes that are known to contain refactoring activities, and we analyze their commit messages, in order to ensure that the collected textual patterns are meant to describe refactoring, and so, to reduce false positives. Our study