FACULTY OF ENGINEERING OF UNIVERSITY OF PORTO

Towards a Smart Recommender for Code Refactoring

João Barbosa

Integrated Master in Informatics and Computing Engineering

Supervisor: Ademar Aguiar

Co-Supervisor: Sara Fernandes

July 16, 2020

Towards a Smart Recommender for Code Refactoring

João Barbosa

Integrated Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: Prof. João Carlos Viegas Martins Bispo
External Examiner: Prof. Eduardo Martins Guerra
Supervisor: Prof. Ademar Manuel Teixeira de Aguiar

July 16, 2020

Abstract

In a world where software products are more complex and software teams bigger, reusable and maintainable design is increasingly more important to produce high quality and valuable applications. Software's integrity and structure according to a design tends to decay over time as code is modified and new features are introduced, especially when developed by different people. One of the main techniques used to improve software maintainability is refactoring, which can be defined as the process of changing the system's internal structure without affecting its external behavior. Although the application of refactoring techniques can be done automatically in many development environments, identifying the fragments of code to which the refactoring should be applied is still a difficult task and must usually be picked by the programmer. More often than not, this is the longest and most cumbersome step in the process.

In this dissertation, we aim to understand how refactoring recommenders can be improved to mitigate current issues and needs. To do so, we explore the concept of Live Refactoring. In Live Programming, the development environment is synchronized with the programmer while he is coding, allowing shorter feedback times. In advanced levels of live refactoring, the environment should be able to predict and suggest code changes that are more likely to fit the system under development. We will focus our study on the Long Method code smell and corresponding Extract Method refactoring technique.

We explained a solution that identifies multiple candidates using clustering techniques, taking into account how tightly coupled statements inside a method are. The solution was then validated using two distinct methods, which include a controlled experiment and a survey. Our research supports the claim that live feedback on refactoring suggestions is a good solution to some of the issues identified. Namely, it not only shows that developers are able to better understand their refactoring needs, but also provides benefits in terms of code quality.

Resumo

Num mundo em que os produtos de software são mais complexos e as equipas de software maiores, o design reutilizável e sustentável é cada vez mais importante para produzir aplicações com valor e qualidade. A integridade e a estrutura do software, de acordo com um certo design, tende a decair com o tempo, à medida que o código é modificado e novas funcionalidades são introduzidas, especialmente quando desenvolvidos por pessoas diferentes. Uma das principais técnicas usadas para melhorar a manutenção do software é refactoring, que pode ser definido como o processo de alteração da estrutura interna do sistema sem afetar o seu comportamento externo. Apesar de ser possível aplicar automaticamente técnicas de refactoring em muitos ambientes de desenvolvimento, a identificação dos fragmentos de código aos quais as operações de refactoring devem ser aplicadas ainda é uma tarefa difícil e devem, geralmente, ser escolhidos pelo programador. Frequentemente, esse é o passo mais longo e complicado do processo.

Nesta dissertação, pretendemos compreender como é que recomendadores de refactoring podem ser melhorados para mitigar problemas e necessidades atuais. Para tal, exploramos o conceito de Live Refactoring. Em live programming, o ambiente de desenvolvimento está sincronizado com o programador enquanto ele está a desenvolver código, permitindo tempos de feedback mais curtos. Em níveis avançados de live refactoring, o ambiente deve ser capaz de prever e sugerir modificações de código que são mais prováveis de ser adequadas ao sistema em desenvolvimento. Vamos focar o nosso estudo no code smell Long Method e na técnica de refactoring Extract Method correspondente.

Explicamos uma solução que identifica vários candidatos usando técnicas de clustering, levando em consideração o quão fortemente relacionadas são as instruções de código dentro de um método. A solução foi posteriormente validada usando dois métodos distintos, uma experiência controlada e um survey. A nossa investigação suporta a ideia que live feedback de sugestões de refactoring é uma boa solução para alguns dos problemas identificados. Nomeadamente, não só demonstra que developers têm mais facilidade em perceber as suas necessidades de refactoring, mas também traz vantagens em termos de qualidade de código.

Acknowledgements

My deep gratitude goes to the Faculty of Engineering of the University of Porto, for providing the environment and necessary conditions for my education and growth as a person. The lessons I've learned throughout these 5 years are invaluable and will accompany me for the rest of my life. I'll miss you all.

A special thanks to my supervisor, Ademar Aguiar, for his ability to guide me - by not only providing the right answers, but also the right questions. Thank you for your great analogies, relaxing mood and constant inspiration. A big thanks also to my co-supervisor, Sara Fernandes, who was very supportive and helpful throughout this entire investigation. Thank you for your patience and invaluable contributions to this work. Finally, my gratitude goes also to my dissertation partners, Sérgio Salgado and Daniel Pinho, with whom I had the opportunity to share ideas, experiences and knowledge.

I also wish to express my sincere gratitude to my group of friends, who always stood by me and made this journey meaningful and joyful. Above all the *random* stuff we do, I appreciate your company, humor and friendship. May your paths bring you happiness, and may them join us together for even more great moments. Also, a big special note to Miriam, who stood by me throughout many difficult times and always believed in me. It wouldn't be possible without you. May your smile always fill the room, and your light shine bright. Thank you.

My gratitude also goes to Talkdesk, which allowed me to start a new learning phase as I'm about to end this one. A big thanks to my awesome Integrations team for providing me daily with new ideas, perspectives, knowledge and, above all, laughter. Thank you for including me and making me feel at home so quickly. A special thanks to my manager, João Fernandes, who I met still during my studies, for recognizing enough talent in me to join him, for being so comprehensive and supportive towards my situation, and for being a daily source of inspiration. Finally, a special note goes also to Ana Capelo, who accompanied me throughout this whole process (and still does) with positivism, calmness and sweetness like no other. Thank you all!

A sincere and deep thank you goes to my family. Thank you for believing in me and for being so supportive since day one. None of this would be possible without you, and I can only be grateful for having you in my life. I love you all.

A big final thanks to my mother. May you know, wherever you are, that we've made it.

João Paulo Moreira Barbosa

"Real generosity towards the future lies in giving all to the present."

Albert Camus

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Motivation . . . 2
- 1.3 Problem . . . 2
- 1.4 Objectives . . . 3
- 1.5 Strategy and Expected Results . . . 4
- 1.6 Report Structure . . . 4

2 State of the Art **7**
- 2.1 Code Smells . . . 7
  - 2.1.1 Techniques and Tools . . . 8
  - 2.1.2 Discussion and Open Issues . . . 11
- 2.2 Refactoring . . . 12
  - 2.2.1 Techniques and Tools . . . 13
  - 2.2.2 Discussion and Open Issues . . . 20
- 2.3 Live Software Development . . . 24
  - 2.3.1 Techniques and Tools . . . 26
  - 2.3.2 Discussion and Open Issues . . . 27
- 2.4 Software Visualization . . . 29
  - 2.4.1 Techniques and Tools . . . 29
  - 2.4.2 Discussion and Open Issues . . . 33

3 Problem Statement **35**
- 3.1 Current Issues . . . 35
- 3.2 Assumptions . . . 36
- 3.3 Hypothesis . . . 37
- 3.4 Research Questions . . . 38
- 3.5 Methodology . . . 40
- 3.6 Summary . . . 41

4 Proposed Solution **43**
- 4.1 Overview . . . 43
- 4.2 Candidate Identification . . . 44
  - 4.2.1 Reasoning . . . 44
  - 4.2.2 Implementation Details . . . 45
- 4.3 Development Experience . . . 48
  - 4.3.1 Reasoning . . . 49
  - 4.3.2 Implementation Details . . . 49

4.4 Architecture . . . 50
4.5 VS Code Extension . . . 53
4.5.1 Accept User Input . . . 53
4.5.2 Display Suggestions . . . 55
4.5.3 Highlight Suggestion . . . 56
4.5.4 Apply Suggestion . . . 57
4.5.5 Code Evolution . . . 57
4.6 Summary . . . 58

5 Empirical Evaluation **61**
5.1 Controlled Experiment with Robotized Refactoring Tool . . . 61
5.1.1 Experiment Design . . . 62
5.1.2 Running the Experiment . . . 69
5.1.3 Results . . . 71
5.1.4 Discussion . . . 81
5.2 Survey on Live Refactoring . . . 83
5.2.1 Experiment Design . . . 83
5.2.2 Results . . . 88
5.2.3 Discussion . . . 96
5.3 Summary . . . 97

6 Conclusions **99**
6.1 Summary . . . 99
6.2 Main Contributions . . . 104
6.3 Future Work . . . 105

References **109**

A Behavior Flow Diagram - PlantUML Code **119**

B Architecture Diagram - PlantUML Code **121**

C Questionnaire about Live Refactoring **125**

List of Figures

2.1 Activities in refactoring candidate identification. 13
2.2 Levels of liveness. 25
2.3 Tools representing software at source code level. 31
2.4 Class Blueprint. 31
2.5 2D representations of the same architecture. 32
2.6 3D visualization tools using real-world metaphors. 32
2.7 Mapping between UML and Geon Diagram. 32
2.8 Software evolution of the Jmol software. 33

4.1 Behavior flowchart. 44
4.2 Example function in TypeScript. 46
4.3 System architecture. 51
4.4 Command palette showing command provided by the refactoring tool. 55
4.5 Refactoring suggestions in VS Code gutters. 55
4.6 Refactoring suggestions in VS Code gutters when no candidates are available. . . 56
4.7 Highlighted suggestion. 56
4.8 Applied suggestion. 57
4.9 Applied suggestion's method. 57
4.10 Non-automatically extractable method message. 57
4.11 Code quality evolution. 58
4.12 Code quality evolution on hover. 59

5.1 Example of file with list of applied refactorings. 71
5.2 Example of collected metrics for a given file. 72
5.3 Metrics in a single revision and file for VS Code. 74
5.4 Metrics in a single revision and file for Deno. 74
5.5 Metrics in a single revision and file for Smart Refactoring Recommender. 75
5.6 Metrics in a single file across multiple revisions for VS Code. 76
5.7 Metrics in a single file across multiple revisions for Deno. 77
5.8 Metrics in a single file across multiple revisions for Smart Refactoring Recommender. 77
5.9 Metrics in a single revision across multiple files for VS Code. 79
5.10 Metrics in a single revision across multiple files for Deno. 79
5.11 Metrics in a single revision across multiple files for Smart Refactoring Recommender. 80
5.12 Age, education level and programming experience of participants. 89
5.13 Knowledge of participants about programming and software development. 89
5.14 Experience of participants with relevant tools. 90

5.15 Opinions on the ability to identify refactoring candidates. 90
5.16 Opinions on the tool's continuous feedback. 91
5.17 Opinions on single vs. multiple suggestions. 91
5.18 Opinions on the tool's availability and relevance. 92
5.19 Opinions on using VS Code's Command Palette. 92
5.20 Opinions on selection highlighting. 93
5.21 Opinions on having a semi-automated refactoring tool. 93
5.22 Opinions regarding code quality analysis. 94
5.23 Opinions on the tool's ease of use. 94
5.24 Opinions on the tool's impact and usability. 94
5.25 Opinions on the tool's best features. 95
5.26 Opinions on the tool's worst features. 95

List of Tables

2.1 Common smells affecting code structure. 9
2.2 Methods used in refactoring candidate identification and other activities. 14
2.3 Correspondence between Refactoring and Graph Transformation. 15
2.4 Examples of quality metrics. 18
2.5 Visualization techniques and representation metaphors. 30

3.1 Hypothesis tests considering the research questions. 40

4.1 Binary representation of code 46

5.1 Repository report. 66
5.2 Number of analyzed revisions, files and refactorings. 71
5.3 Statistics regarding difference in metrics after a refactoring is applied. 81

Abbreviations

SDLC	Software Development Life Cycle
LiveSD	Live Software Development
UML	UML Unified Modeling Language
IDE	Integrated Development Environment
GUI	Graphical User Interface
RQ	Research Question
MTBD	Model Transformation By Demonstration
VS Code	Visual Studio Code
AST	Abstract Syntax Tree
SV	Software Visualization
AI	Artificial Intelligence
AGI	Artificial General Intelligence
XAI	Explainable Artificial Intelligence
WSD	Word Sense Disambiguation
TCC	Tight Class Cohesion
CBO	Coupling Between Objects
LCOM	Lack of Cohesion of Methods
REST	Representational State Transfer
VDM	Vienna Development Method
VDM-SL	VDM Specification Language
CI	Candidate Identification
TS	Technique Selection
BP	Behavior Preservation
RE	Result Evaluation

Chapter 1

Introduction

1.1 Context 1
1.2 Motivation 2
1.3 Problem 2
1.4 Objectives 3
1.5 Strategy and Expected Results 4
1.6 Report Structure 4

This chapter describes the scope of this research work and briefly introduces current capabilities and limitations in refactoring recommenders.

1.1 Context

Software Engineering projects of relative complexity usually go through a so called *Software Development Life Cycle*, which is a process for building systems comprising of all necessary phases, steps and instructions to develop, test and alter software. It is a system to build and maintain systems [Ben12]. Several SDLC models have been developed over time, each providing different strengths and flaws for different situations [LLTT12]. This work relates to one of such methodologies, which is called *Agile* [Agi01].

Agile Software Development has incremental and iterative development as its foundation, in which the different phases of the SDLC are consistently revisited in "increments" with relatively short intervals of time. According to the Agile Manifesto, important principles of this methodology include [Agi01]:

1. Welcome changing requirements;
2. Iterative development;

3. Attention to good design, enhancing agility;

4. Adaptation to change.

Agile is about fast and incremental development, with short feedback loops, using flexible, simple and maintainable design, with the final goal of easily accommodating new requirements and objectives of a software system. One of the practices employed by agile methodologies to ensure quality and reusable software is *Refactoring*.

In short, refactoring aims at restructuring the internal structure of a system, with the main goal of improving its overall maintainability and agility. There is a wide variety of refactoring transformations, which depend on the current problem and context - including the presence of code smells, programming language, requirements, among others.

1.2 Motivation

As software complexity and team sizes are increasing, reusable and maintainable design gains equal importance in order to produce valuable products. Software's integrity and structure according to a design tends to decay over time as code is modified in response to new requirements [TPB+15], especially when developed by different people [Fow99]. Because of this, part of the software development cost is devoted to the system's maintenance.

Refactoring is one of the practices used in the maintenance process. Even though its application shows good empirical results, it comes with its own set of problems and open questions. There is research related with these different issues, namely why [MSAS06], when [MB16] and how [Opd92] one should refactor. This dissertation focuses on the first step in the refactoring process, identifying the refactoring candidate, that is, *what* software developers should refactor.

A good starting point to identify refactoring candidates are *code smells*, which represent surface signs of potentially deeper problems in the system. However, they just represent a symptom, not the disease: they're an indicative of the problem's location, but not the problem itself. Even when a code smell is identified, which might alone be a difficult task, programmers need to pick and validate the fragments related to the smell on which the refactoring techniques will be applied. Depending on the complexity of the system, this step may become long and cumbersome [MRA18, HH16], creating an undesirable bottleneck in the software development process.

1.3 Problem

Refactoring is hard. Not necessarily because refactoring techniques are difficult to implement and be available to programmers - some of them are, but others already have solid foundations since the very beginning of refactoring as an academic research problem [Opd92] and can be found in many of the currently available IDEs. Rather, refactoring is a difficult task because it should be guided by the need of something else: we improve the maintainability of the code to make it

easier to adapt to new requirements, ease the process of fixing bugs, enhance the readability of the code, etc.

Martin Fowler and Kent Beck [Fow99] termed this way of thinking as *Big Refactorings*, i.e., the application of refactoring with some additional, *true* purpose in mind. In this context, the recommendation of refactorings becomes a hard problem because, as the authors put it, *"we can't tell you exactly what to do, because we don't know exactly what you'll be seeing when you do it"* [Fow99]: even though we might have access to all software artifacts used in a system, there is still a knowledge gap between the information the artifacts convey by their structure and the true information they are meant to represent (the high-level, strategic reason why the software even exists in the first place). The latter is something that, at the moment, only humans are able to assimilate. Of course, even the ability of humans is dependent on their skills, experience and expertise [Fow99, SS18], and so refactoring recommenders are still useful tools to aid the programming activity. Nonetheless, we believe that finding the *best* refactoring recommendation, in a level of high reliability and certainty, is an AI-complete problem.

In 2012, E. Murhpy-Hill et. al. [MHPB12] published a study where they analyzed refactoring activities from more than 39,000 developers and were able to draw, among others, two important conclusions: developers perform refactorings frequently; and close to 90% of refactorings are performed manually (without any aid from tools). The authors reason currently available refactoring recommenders miss important features that would improve their impact on software development, namely awareness, opportunity and trust.

Our research of available literature indicate most works related to candidate identification are solely focused in finding refactoring opportunities. In other words, they do not take into consideration the agent with the primary and most important role in the refactoring process: the developer.

1.4 Objectives

From the difficulties and problems stated previously in Section 1.3 (p. 2), we reason that the ability to identify candidates with high confidence is a problem for Artificial General Intelligence (AGI) [GP07], which relates to the aptitude of learning and solving problems in a level comparable to those of humans, i.e., the ability to solve AI-complete problems.

While AGI is not yet available, we consider that a *smart* recommender is not one that solely tries to predict, detect and suggest refactoring opportunities, but also one that is human-centered (in the sense that it tries to help the programmer realize the true refactoring needs of the system).

Therefore, this dissertation seeks to explore ways to build a recommendation system that not only identifies refactoring candidates, but also suitably conveys that information to the developer. Given the novelty of the approach and techniques employed in this context, this work will focus only on the *Long Method* code smell and *Extract Method* refactoring.

The objectives of this work are centered around two main questions of research:

1. How good are refactoring candidates proposed by the recommendation system?

2. What is the impact of the recommendation system in the programming activity?

In order to answer these questions, we've studied the current state-of-the-art on important topics related to refactoring recommendation systems. We then build a refactoring tool that suggests refactoring opportunities. Finally, the tool's performance will be analyzed through empirical validation.

1.5 Strategy and Expected Results

The goal of this work is to provide insights on how to build smarter refactoring recommenders, that take into account the developer as the primary agent in the process of building more maintainable and readable source code. The contributions of this dissertation are divided in three parts.

Firstly, the state-of-the-art analysis, presented in Section 2 (p. 7), aims to introduce relevant topics to this dissertation, which include a description of the research area, previous works and contributions, ending in a discussion on why we think they are relevant and what are their main advantages and drawbacks.

Secondly, we also aim to develop a refactoring recommender that allows us to test our research questions (*cf.* Section 3.4, p. 38) and validate our hypothesis (*cf.* Section 3.3, p. 37). The final product is expected to be a Visual Studio Code extension for the TypeScript and JavaScript languages. Further details of our approach are detailed in Section 4 (p. 43).

Finally, the last contribution of this work is an empirical study, where we analyze how developers use the refactoring tool, and how the latter contributes to the overall development workflow, speed and quality of software.

1.6 Report Structure

In this introductory chapter we briefly stated, and hopefully transmitted, why we are doing this research: its importance in agile development, the major problems with current practices we attempt to fix, and how we plan to do it. Additionally, perhaps more importantly, we redefine and give an explanation to why refactoring recommendation systems should rethink their strategies and adopt more human-centric approaches.

Next, in the State of the Art chapter (*cf.* Chapter 2, p. 7), we explore the topics of research that will have a primary role on how we develop our solution. Each section is devoted to a single topic. At the start of each section, we expand on the origins, major definitions, contributions and research lines of the topic. After, in the techniques and tools section, we synthesize the main approaches used to address the problems in the field, which include a description of their strengths and drawbacks, as well as how previous research concretely tried to tackle these same issues. Finally, in the last section we summarize the main key points to take away from the research fields in the context of this dissertation, and discuss how we believe they might bring advantages to a refactoring recommender, considering also the concerns that can arise from applying them.

Chapter 3 (p. 35), Problem Statement, presents a more in-depth and detailed description of the ideas already presented in Section 1.3 (p. 2), which refer to the problems this dissertation tackles, building on the key points gathered from the state-of-the-art study. We provide a more extensive and formal definition of the problem, including the hypothesis and research questions that support this dissertation, as well as any possible assumptions we take into consideration.

Chapter 4 (p. 43), Proposed Solution, expands on the ideas presented in Section 1.5 (p. 4), presenting a more detailed description on the proposed solution we build to solve the problems of focus, its main objectives and requirements, and expected difficulties in the development process.

Chapter 5 (p. 61) presents a more detailed description about the evaluation process and the empirical study: how we conduct it, the main information we want to gather from these experiments, as well as how we evaluate the results.

Finally, the Conclusions chapter (*cf.* Chapter 6, p. 99) summarizes the work done throughout this investigation and the main conclusions we were able to draw.

Chapter 2

State of the Art

2.1 Code Smells 7
2.2 Refactoring 12
2.3 Live Software Development 24
2.4 Software Visualization 29

This chapter discusses the different and most relevant tools and studies about the concepts related to the main objective of this research. Our state of the art investigation focuses on the topics of *Code Smells*, *Refactoring*, *Live Software Development* and *Software Visualization*.

At the start of each section, we expand on the origins, major definitions, contributions and research lines of the topic. After, in the techniques and tools section, we synthesize the main approaches used to address the problems in the field, which include a description of their strengths and drawbacks, as well as how previous research concretely tried to tackle these same issues. All the methods presented in this research were picked with a main focus on diversity, i.e., showing the many different approaches and solutions used in the past. Finally, in the last section we summarize the main key points to take away from the research fields in the context of this dissertation, and discuss how we believe they might bring advantages to a refactoring recommender, considering also the concerns that can arise from applying them.

2.1 Code Smells

As people change code to accommodate new requirements or without a full comprehension of the intended design, the system's structure quality tends to decay [Fow99, TPB+15]. Loss of structure has a cumulative effect: as quality decreases, the harder it is to understand the system's design, which in turn makes it harder to maintain and change, leading to more defects.

Kent Beck was the first to describe, albeit informally, these structure defects as *code smells - "certain structures in code that suggest (sometimes they scream for) the possibility of refactoring"* [Fow99]. Code smells indicate implementation flaws that negatively affect software development lifecycle properties, such as understandability, testability, extensibility, and reusability - that is, code smells ultimately result in maintainability problems [Fow99, TPB+15].

More formally, T. Sharma et. al. [SS18] synthesized the following five key characteristics of a software smell:

Indicator. Indicator to or a symptom of a deeper design problem.

Poor solution. Suboptimal or poor solution to a problem.

Violates best practices. Smells violate best practices (such as design patterns), which might be domain dependent.

Impacts quality. Increasing difficulty in evolving, fixing and maintaining a software system.

Recurrence. Smells can also be defined as recurring problems.

Software smells are domain dependent, and different smells might have more or less impact on different areas of focus, such as implementation, design, architecture, databases, etc [SS18]. As such, the classification of code smells depends on the problem and their properties. T. Sharma et. al. [SS18] defined several classification frames for software smells, which include their effect (on development activities), violation of design principles, the software artifacts they're harmful to, and their granularity. An exhaustive, comprehensive and evolving list of smells is also provided by the authors, available online[1]. Considering code smells that impact code structure, some of the most common are found in Table 2.1 (p. 9).

Smells have many different ways of getting introduced in software systems, including [SS18]: developers' lack of skill or awareness, frequently changing requirements, the chosen technologies and development platforms, knowledge gaps (a lack of full comprehension of the intended system's design), processes used in development, schedule pressure, priority to features over quality, organizational politics, team culture and poor resource planning. Considering these factors, we can conclude smells are not necessarily introduced when modifications are applied to our system, but depend on many other external occurrences [SS18, TPB+15]. Therefore, for a sufficiently complex software system, smells are inevitably present, making its detection an important research subject.

2.1.1 Techniques and Tools

A large body of knowledge on software smell detection already exists. We divide these techniques in four categories: *Metrics*, *Rules and Heuristics*, *Change History Analysis* and *Artificial Intelligence*.

[1] A Taxonomy of Software Smells - `http://www.tusharma.in/smells/`. Last access on 21 January, 2020.

Table 2.1: Common smells affecting code structure [Fow99, MT04].

Code Smell	Description
God/Large Class	Occurs when a class is holding too much data and, consequently, much control over the system's behavior. These classes have a higher chance of having duplicated or less cohesive code.
Feature Envy	A method seems more interested in data from a class other than the one it is in. Usually implies that the method (or part of it) should be moved into that class.
Shotgun Surgery	When a modification needs to be introduced in the system, leading to a lot of small changes in different classes. When changes are all over the place, they're harder to find - and easier to miss.
Long Method	Occurs when a method is doing too much or is too long to understand.
Spaghetti Code	This smell refers to unmaintainable, structureless code. It doesn't exploit, and even prevents, the use of object-oriented practices and mechanisms.

Metrics

A typical metrics-based system converts source code into numerical values that represent key properties regarding the system's integrity, structure and maintainability. Examples of metrics include *TCC* (Tight Class Cohesion), *CBO* (Coupling Between Objects), *LCOM* (Lack of Cohesion of Methods), among others [SS18]. When computed metrics are above or below some predefined thresholds, they can be associated with a certain code smell.

Metrics-based methods provide the advantage of being relatively simple and convenient to implement. However, they're very dependent on the thresholds chosen and have difficulty in detecting several code smells.

S. Fernandes [Fer19] developed a tool that gives information regarding quality metrics to the developers while programming, also comparing the values with previous versions of software, allowing developers to better understand how the code is evolving. R. Marinescu [Mar05] proposed a technique where concrete code smells were detected by comparing the current metrics in the system to expected values in conformance with good design principles. A. Fard et. al. [FM13] created a tool, JSNose, that detected 13 JavaScript code smells using a metric-based approach that combined both static and dynamic analysis. J. Pantiuchina et. al. [PBTP18] implemented a technique to predict the appearance of code smells, by computing several metrics along different histories of the software system and applying machine learning methods to create a predictive model. R. Abílio et. al. [APFC15] applied metrics-based detection techniques to study how code smells can be detected in feature-oriented programming languages.

Rules and Heuristics

These approaches are an expansion of metrics-based techniques, able to define additional values/heuristics to help in smell detection. They allow more information to be extracted from

the source code and, consequently, generally provide a wider detection range of software smells.

When combined with software metrics, rules and heuristics are able to detect a high portion of known smells. The main drawback of these approaches is choosing the rules and heuristics to be applied: too general, they're not so different from metrics; too specific, they're prone to miss smells.

F. Palma et. al. [PDMG14] applied heuristics-based rules to detect both patterns and antipatterns in REST architectures. T. Sharma et. al. [SMT16] implemented a tool for C#, Designite, that focused on detecting smells at design level. A. Rama [Ram10] identified smells specifically concerned with modularity problems, mainly targeting legacy systems. N. Moha et. al. [MGDL10] introduced DECOR, a method that allows the definition of smells to be manually specified (at code and design level), allowing their detection in problem-dependent and domain-specific problems.

Change History Analysis

Previous works also attempted to detect code smells by analyzing code evolution, through commit logs or version control, extracting information about the system's structure. This information is used to build a model capable of predicting/detecting smells in code.

Change history analysis approaches might reach good performance levels when smells are associated with evolutionary changes. As we previously stated, however, this is not always the case, i.e., smells can be originated for reasons other than software changes. These techniques fail to detect defects in such cases.

J. Pantiuchina et. al. [PBTP18] computed several software metrics in different versions of a software system to predict the future presence of code smells. S. Fu et. al. [FS15] applied association rule mining techniques to detect code smells, given the evolution of the software system. Finally, F. Palomba et. al. [PBP+15] compared consecutive software versions using metrics and heuristics to detect changes - in classes, methods, types of variables, among others -, allowing the detection of code smells.

Artificial Intelligence

This category includes any approach that tries to reach a solution according to some predefined measure/function. Therefore, it includes both machine learning and optimization-based (e.g. genetic algorithms) techniques. A typical AI method defines a mathematical model that represents the smell detection problem and lets an algorithm find the best (possible) solution to that same model.

A big challenge in AI approaches is defining the mathematical model used to define a smell. Furthermore, in supervised learning, large amounts of data is necessary, which is still a problem in the context of refactoring. Finally, these techniques also suffer from the definition of thresholds, much like the metrics-based ones.

A. Maiga et. al. [MAB+12] used Support Vector Machines to detect antipatterns in a software system. J. Pantiuchina et. al. [PBTP18] used an unsupervised Random Forest to separate classes into according to their need of future refactorings. D. Sahin et. al. [SKBD14] proposed the treatment of code smell detection as a bilevel optimization problem. W. Kessentini et. al. [KKS+14] used parallel evolutionary algorithms that cooperatively searched for the presence of code smells. G. Czibula et. al. [CMC15] applied association rule mining to detect smells at design level. S. Bryton et. al. [BBM10] used Binary Logistic Regression to detect the Long Method code smell.

2.1.2 Discussion and Open Issues

A code smell is an indicator of a deeper maintenance problem in the system. They provide *surface* information on how parts of the software can cause issues in the future.

Thus, it is important to keep in mind that, even though code smells suggest the need of refactoring, they don't necessarily lead to one. Smells aren't bad on their own, they are indicative of a possible problem [Fow99, SS18].

For instance, a Long Method smell refers to methods that are too complex too understand, which *may* mean it has too many responsibilities - prominent, separable blocks of code that are unconnected to the method's main goal. Additionally, even if the method is handling a single responsibility, it *may* also mean that a particular task is still too complex, and would probably benefit from a subdivision into smaller, self-contained responsibilities.

In those cases, code smells are indicating an important vulnerability in the system. However, some long methods could be just fine, depending on the context. A good example of this situation are implementations of known, standard algorithms in programming[2]. From an agile development point of view, often these algorithms are not part of the *malleable* structure of the system, i.e., they don't affect maintainability and adaptability to new requirements. After the algorithm is thoroughly inspected with a reliable and extensive set of tests, these implementations are not very likely to need change (i.e. refactoring).

Code smells and refactoring techniques are nevertheless intimately connected. A code smell gives indication about a deeper problem in the system. The goal of refactoring is precisely to both fix and prevent these structural flaws, to *"help code keep its shape"* [Fow99]. Thus, it belongs to the refactoring process to validate if the code smell is a true positive, through the concrete detection of the problem, followed by consequent recommendation.

Nonetheless, code smells are helpful because they indicate parts of the system that need our attention, so they're good starting points for refactoring recommenders. Some of the challenges currently present in this topic are:

No clear definition. Given that code smells only suggest the need of refactoring, there's still a lack of a clear distinction between the differences of a smell and a quality problem [SS18].

[2]Example of the Edit Distance algorithm - https://github.com/TheAlgorithms/Java/blob/master/DynamicProgramming/EditDistance.java. Last access on 21 January, 2020.

High false-positive rates. Community believes code smells tools suffer from high false-positive rates [SS18]. This problem interestingly relates to the aforementioned lack of definition: if code smells are just indicators, how do we distinguish a false-positive smell from an actual maintenance problem? By definition, the latter belongs to the refactoring process, not to smell detection.

Inconsistent research results. Still related to the fact code smells lack standard definitions, previous investigations are inconsistent in their definition of a smell and, therefore, detection method. This hinders the correct evaluation and comparison of the available research [SS18].

Tools offer limited detection support. Most existing tools detect a small subset of known smells. Additionally, most of these tools are available to the Java language [SS18].

Research is mainly focused on code. Code smells can originate from evolving our system - making changes that accommodate new requirements and objectives. Code smell detectors and predictors usually focus on this type of smell origin. However, other factors, exterior to code, can be responsible for introducing maintenance problems (e.g. from business requirements, such as tight deadlines pressure) [SS18]. It is possible for a code smell to be introduced in the very beginning of a new class or method, and not necessarily when these artifacts suffer changes [TPB+15]. Thus, research on code smells should also explore how different factors, other than code, can be taken into account.

2.2 Refactoring

The research domain that addresses the problem of coping with increasingly complexity in a system by incrementally improving its design quality is called *restructuring* [CC90, MT04], or in the case of object-oriented software development, *refactoring* [Fow99, Opd92].

Refactoring is defined as the process of changing a software system's internal structure without altering its external behavior [Fow99]. Empirical evidence suggests that code design tends to decay over time [Fow99, TPB+15] and that refactoring can help in restoring and improving its quality [MSAS06].

Refactoring application encompasses a distinct set of activities, namely [MT04]:

1. Identify refactoring candidate(s);
2. Determine which refactoring techniques(s) should be applied;
3. Ensure the applied refactorings are preserving behavior;
4. Assess quality of the final result.

When we refactor, we expect to improve the design and reusability of code without changing its observable function, that is, we solely focus on improving the system's agility, readability and adaptability. Thus, refactoring becomes an invaluable tool for agile development. It is worth

noting that refactoring can be applied to different software artifacts, such as databases, system architectures and requirements [MT04].

This work focuses mainly on the first activity aforementioned, what should be refactored, i.e., picking the lines of code where refactoring should be applied. We identify three distinct activities in the identification of refactoring candidates: *prediction*, *detection* and *recommendation*.

Prediction is concerned with trying to find future refactoring needs and usually uses information related to code evolution, code smells and software metrics.

Detection of refactorings can be divided in two main categories: discovering applied refactorings through the analysis of code evolution (the inverse of prediction), and finding the presence of refactoring candidates in source code currently under maintenance. The former usually takes advantage of similar methods as prediction, while the latter is usually associated with the recommendation of refactorings.

Finally, recommendation is concerned with picking the best options, out of the detected ones, i.e. ordering the quality of the candidates.

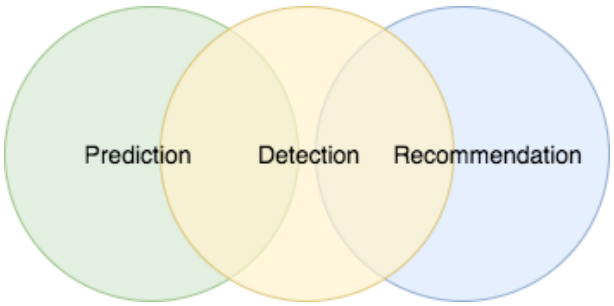


Figure 2.1: Activities in refactoring candidate identification.

2.2.1 Techniques and Tools

There is wide variety of techniques that have been proposed and are used in refactoring, mainly due to the fact that this analysis depends on the scope of the technique: for instance, finding candidates for the *Move Method* refactoring usually implies analyzing inter-class relationships, while in the *Extract Method* refactoring our scope can be reduced to a method of a class. As such, different refactorings need to use different refactoring techniques.

In this section, we list and discuss the current methods available in the refactoring process. The reader should take notice that the methods here presented are not exclusive of refactoring candidate identification, the activity of focus of this work, but can be used as such or at least aid in the process. Table 2.2 (p. 14) provides an overview of the methods used in candidate identification, as well as in other activities.

Formal methods

Formal methods have long been suggested and used as a technique to ensure refactoring applications are preserving behavior. William Opdyke was the first to suggest the use of

Table 2.2: Methods used in refactoring candidate identification and other activities. *CI* - Candidate Identification, *TS* - Technique Selection, *BP* - Behavior Preservation, *RE* - Result Evaluation.

	CI	TD	BP	RE
Formal Methods	○		○	
Rules and Heuristics	○	○		○
Graphs	○		○	
Change History Analysis	○	○		
Program Slicing	○		○	
Software Metrics	○			○
Artificial Intelligence	○			
Game Theory	○	○		

preconditions as a tool to preserve behavior, and included a set of example functions that could be employed in the refactoring domain [Opd92]. By including invariants, preconditions and postconditions, it is possible to guarantee that our operations are only changing the system's internal structure. Thus, invariant and condition-based detection provides a powerful technique to identify refactoring candidates that preserve behavior.

Formal methods approaches, however useful, come with their own set of limitations [MT04]:

1. *Full* behavior preservation might be a difficult task, depending on the language used - for instance, C programs can use pointer arithmetic, which may become problematic if the program behavior depends on the position of variable declarations.

2. If we use a more flexible, less formal notion of preservation, we are prone to miss refactoring candidates. For example, one can verify if behavior is maintained by rigorous testing: if all tests - in an extensive, reliable test suite - are successful after a refactoring, there is a high chance the refactoring is preserving behavior. However, one can also expect the existence of good refactoring candidates that, if applied, would not pass the tests (e.g. move a method into another class).

3. Some domain-specific software might require more than just input-output behavior preservation, and may include time, memory or safety constraints.

As previously stated, these approaches are concerned with guaranteeing behavior preservation when refactoring techniques are applied - however, one can easily put in these principles to the search, validation and detection of candidates. Invariant and condition-based techniques, in the scope of candidate detection, are useful to reduce search spaces and to validate refactoring candidates.

Invariants are used by Banerjee et. al. [BKKK87] in order to preserve behavior in databases: by identifying a set of rules in database schemas that should hold before and after a refactoring, the authors are able to reason about the quality and feasibility of the refactoring candidate. Ward et.al. [WB95] developed a tool that aids in the reconstruction of legacy software, using the

formal language WSL, by imposing preconditions that should remain valid after refactoring the system. Finally, D. B. Roberts [Rob99] showed that postconditions also present an advantage in these techniques because they help in reducing the search space and, therefore, improving the performance of composite refactorings.

Rules and Heuristics

These approaches make use of predefined criteria to reduce the search space that, consequently, allow a faster convergence into a solution. Contrary to formal methods, however, they are not necessarily restricted to conditions that preserve behavior. For example, many of the currently available refactoring recommendation tools provide filters (usually parameters that can be defined by the user) in order to reduce the search space, such as minimum and maximum number of lines a candidate should have [Sha12, BTN03, YLN09]. Other techniques employ semantic-based rules, such as looking for empty lines or comments in source code to divide it in blocks [YLN09, HH16].

Even though we place rules and heuristics as a separate technique from the others here presented, it is important to notice they still need to be employed in every currently available approach to refactoring. This is because knowing the *best* solution to apply to our code is still a hard problem [Fow99], i.e., there is still no absolute *formula* or set of *golden rules* that we can confidently apply to the detection and recommendation of refactoring candidates. Therefore, refactoring tools always suggest refactoring candidates according to some predefined criteria.

Graphs

Programs, and other software artifacts, can usually be represented as graphs, at different levels of detail (abstract syntax trees, program dependency graphs, UML diagrams, etc.). In other words, this means we can fully represent these software artifacts as graphs and maintain the same level of detail as in their original form. Thus, since a refactoring application corresponds to transformation of an artifact, we can make a correspondence between refactoring techniques and graph transformations [MT04], as shown in Table 2.3.

Table 2.3: Correspondence between Refactoring and Graph Transformation [MT04].

Refactoring	**Graph Transformation**
software artifact	graph
refactoring	graph production
composite refactoring	composition of graph productions
refactoring application	graph transformation
refactoring precondition	application precondition
refactoring postcondition	application postcondition
(in)dependence between refactorings in a sequence	parallel or sequential (in)dependence
conflict between refactorings applied in parallel to the same software artifact	confluency and critical pair analysis

Graph representations provide the additional advantage of containing useful information about the program, which can be leveraged through data and control flow analysis. Moreover, they can also be used as tool to ensure behavior preservation and consistency between multiple artifacts when a refactoring is applied [MVDJ05, BPPT03], with the ability to use pre and postconditions the same way techniques based on formal methods do [MT04, HHT96].

O. Tiwari et. al. [TJ19] introduced a new approach to identify targets for the Extract Method, by successively contracting edges in a structure dependency graph, according to distinct functionalities that can be captured in a method's graph structure. T. Sharma [Sha12] provides a mechanism to propose extract method candidates based on graph structures, on which suggestions are obtained by deleting the longest dependency edge in the graph. K. Hotta et. al. [HHK12] uses program dependency graphs to identify clones in source code where Form Template Method refactoring can be applied. G. Bavota et. al. [BDO11] identify Extract Class refactoring opportunities by exploiting structural and semantic relationships capturable in graphs, such as attribute references or method calls. T. Kanemitsu et. al. [KHK11] suggests Extract Method candidates by grouping nodes that are connected via edges not longer than a certain defined threshold.

Change History Analysis

Refactoring techniques based on change history analysis are usually concerned with the detection of past refactoring applications [CFYH18]. However, they have also been applied to the context of candidate prediction [PBTP18]. These techniques are split in four categories [CFYH18]:

Commit log mining. Identify refactorings by analyzing commit messages in version control systems - by searching for words like 'refactor' or 'extract'. This technique provides two advantages: it can be used in any project using a version control system, and it uses explicit intentions stated by the developers. However, research shows that these logs often contain unreliable or lack of necessary information [MHPB12].

Developer observation. Past refactoring applications are identified by observing how developers work by using, for instance, screen capturing tools. These techniques usually provide detailed information about development, but lack explicit information, in the context of refactoring, which leads to subjective judgement [CFYH18].

Tool usage logs. These techniques detect refactorings by collecting logs from refactoring support tools. Since these tools are used specifically for refactoring, the information gathered contains explicit information about these operations, which is a definite advantage. However, information is limited to the operations supported by the tool [CFYH18] and to how often it is used in the refactoring process (which, in most situations, is very little [MHPB12]).

History Analysis. History analysis techniques identify refactoring instances by analyzing a sequence of versions of software development artifacts. Since they use observable changes in

software artifacts, these techniques are less prone to subjective evaluation. However, they're also more likely to miss performed refactorings [CFYH18].

K. Stroggylos et. al. [SS07] used commit logs to detect changes marked as refactoring, and analyzed several metrics with the aim of evaluating the effectiveness and quality of the operation (before and after the commit). M. Boshernitsan et. al. [BGH07] created a visual tool to do refactoring by first conducting two studies on how developers produce and think about changes in software artifacts. R. Robbes et. al. [RL08] developed a toolset that stored changes happening in programs so other IDE extensions could take advantage of the recorded actions. In the history analysis category, N. Tsantalis et. al. [TME+18] proposed a rule-based approach that detects several refactorings (Extract Method, Move Method, Inline Method, among others) between revisions of a software project. S. Demeyer et. al. used software metrics between versions of a system to detect application of refactorings [DDN00].

The aforementioned techniques are focused on the detection of refactorings. As we previously stated, however, these methods can also be used for prediction. J. Ratzinger et. al. applied data mining using information extracted from commit logs, revisions and issue trackers to predict medium-term defects in the system [RSG08]. J. Pantiuchina et. al. [PBTP18] used several quality metrics to calculate recent and historical trends in a software project, using an unsupervised Random Forest to predict which classes are more likely to suffer from code smells in the future.

Program Slicing

Program slicing is a technique proposed by Mark Weiser whose original goal is to aid in the debugging process [Wei81]. A *program slice* consists of all statements that may affect the value of a given set of variables at a given set of statements. The variables and statements of interest are called the *slicing criterion* [Wei81]. Weiser started with the observation that, sometimes, *"only a portion of a program's behavior is of interest"* - in the context of debugging, it may mean the subset of code under maintenance [Wei81]. Thus, program slices usually capture a subset of code that indicate a specific target behavior.

In the context of refactoring, this means that candidates identified by program slicing can also preserve behavior [MT04] but, contrary to formal methods, whose main purpose is to maintain input-output behavior, program slices usually indicate a *semantic* behavior. In order to choose a slice that preserves a given behavior, however, one has to select the proper slicing criterion, which, depending on the refactoring technique, may be a difficult task [Ett07]. A possible and usually implemented solution is to depend on the user to pick the slicing criterion - for instance, by selecting a line of interest in source code -, which may be a limiting factor in automatic candidate detection [Ett07, Sha12, HH16].

K. Maryuama [Mar01] first used program slicing to automatically find Extract Method opportunities (however, it first relies on user input to choose the variable of interest, i.e. the slicing criterion). N. Tsantalis et. al. [TC09] adapted this approach, employing union of static slices to improve the quality of the detected refactoring candidates, while also ensuring

better behavior preservation by adding rules slices should obey to. A. Lakhoita et. al. [LD98] presented a transformation called tuck that allowed the decomposition of larger functions into smaller ones by using a three-step technique (wedge, split, fold) that relies on program slices as the initial candidates of the algorithm. J. Singh et. al. [SKM18] recently presented a technique that recommended Extract Method candidates using program slicing which is also able to automatically detect target methods in need of refactoring by computing slice-based cohesion metrics (first introduced by Weiser [Wei81]).

Metrics

Software metrics are also a common approach to handle refactorings. Numerical measures representing source code can evaluate how using a certain refactoring influences the overall system's structure, by comparing the metrics before and after its application. A list of some of the available metrics can be seen in Table 2.4.

Table 2.4: Some quality metrics that can be analysed to see if a software system is well designed and structured [Fer19].

Project Metrics	**Class Metrics**	**Package Metrics**	**Method Metrics**
Project total lines of code	Lines of code	Package total lines of code	Nested block depth
Number of packages	Number of children	Number of interfaces	Method lines of code
Number of external entities	Cohesion among methods	Number of classes	Number of parameters
Number of problematic classes	Number of fields	Number of entities	Number of methods called
Number of highly problematic classes	Number of methods	Abstractness	Number of accessed fields
	Number of static fields	Instability	
		Normalized distance	

In the context of refactoring, software metrics analysis techniques usually employ one of two options: either analyze measures to find code smells and suggest candidates that reduce/delete their presence; or look for refactorings that, in general, may improve quality of the software's structure. The difference is that, while the former's objective is to remove maintainability problems from the software system, the latter attempts to go beyond and also try to suggest refactorings not necessarily related to a code smell, but rather to the use of design patterns.

Software metrics offer a convenient and relatively easy way to implement solutions that extract information about the system. However, metrics usually reason about the overall quality of the code, meaning that we aren't able to retrieve more complex knowledge about the program. Therefore, we are prone to miss refactoring candidates that could carry more semantic meaning. Nonetheless, software metrics are a useful tool to measure the quality of refactoring candidates, either in detection or recommendation - in the sense that values above or below some predefined acceptable range are very likely to indicate a need of refactoring.

M. Salehie et. al. [SST06] proposed a metric-based approach to detect and locate design flaws in a program, by quantifying software metrics deviations from the common values of good design principles. G. Bavota et. al. [BDO11] identified Extract Class opportunities by analyzing both structural and semantic metrics about the system. T. Bodhuin et. al. [BCT07] created SORMASA,

a tool that uses genetic algorithms to suggest refactorings, considering several software metrics as the algorithm's objective function.

Artificial Intelligence

AI approaches have been employed for a long time in refactoring and its multiple activities. From heuristics and genetic algorithms to, more recently, machine learning techniques, they provide the advantage of being able to suggest candidates according to some defined objective function, which in turn allows a faster arrival to a better solution.

When compared to other approaches, AI techniques provide the advantage of usually letting the algorithm find candidates autonomously: in general, these methods are more focused in defining what a good candidate *should be*, instead of what they shouldn't. Therefore, in most techniques, there are no refactoring candidates that are automatically excluded by the algorithm used, but rather by the way we measure their utility. As such, the biggest challenge becomes choosing how do we define the quality of a refactoring candidate. Additionally, in the case of AI techniques centered in supervised learning, there is also the difficulty of obtaining good and reliable refactoring datasets: even though there are previous works that attempt to solve this issue [TME+18, HKFG18, KHFG16], the amount of available information is still not sufficient to truly be able to leverage the power of machine learning. Furthermore, some of these tools lack a solid validation of the identified refactorings, specially those that attempt to build the dataset automatically. Supervised learning techniques are usually used in the prediction of future refactorings, by combining machine learning with change history analysis.

J. Ratzinger et. al. [RSVG07] used information from version control system to extract data mining features, such as measures about the system's growth, relationships between classes, number of programmers modifying the same artifacts, etc., about the code under development in order to predict future refactoring needs, using decision trees, logistic model trees, rule learners and nearest neighbour algorithms. S. Xu et. al. [XSKX17] developed GEMS, a tool to recommend Extract Method refactorings, composed of three steps: (i) detect possible refactoring candidates using a block-based candidate generation algorithm, (ii) validate the detected candidates by the use of preconditions that they need to satisfy, and (iii) compute a feature vector for each candidate. Feature vectors are then used to train a probabilistic supervised learner, obtaining a model that measures how good a candidate is. C. Lung et. al. [LXZS06] proposed a technique that detects Extract Method candidates using clustering methods, targeting the identification of ill-structured or low-cohesive functions.

Game Theory

Game theory is a branch of mathematics that studies situations where players must decide between multiple choices with the objective of maximizing their return [Dre81]. In software engineering, there are many situations where contrasting goals collide, and a equilibrium between them needs to be reached [BOD+10]. For instance, system restructuring, like refactoring, must

be balanced with other activities, such as adding new function, testing, etc. As such, game theory approaches might take advantage of these conflicting processes in software engineering to suggest more sensible candidates. A major challenge of these techniques is defining these contrasting goals, and how an optimal balance can be found between them.

G. Bavota et. al. [BOD+10] studied the use of game theory to identify Extract Class refactoring opportunities, by having two agents that, through similarity measures, compete for methods of a single class, using both Nash equilibrium and Pareto optimum. In the end, the methods of the class are divided by those two competing agents, where the one that obtained less methods is recommended as an opportunity to extract as a new class. G. Bavota et. al. [BOD+14] later enhanced the previous (preliminary) approach based on an iterative algorithm, using more relevant similarity measures.

2.2.2 Discussion and Open Issues

This dissertation focuses on *refactoring candidate identification*. Candidate identification refers to *what* should be refactored, that is, the components of the software artifact that would benefit from modification (in the case of source code, it corresponds to the code lines). As previously discussed, there are other important activities of interest in refactoring application - namely behavior preservation, technique selection and final result evaluation -, which are not in the scope of this work.

When discussing refactoring, two important key points have to be taken into consideration:

- Agile development has as its core foundation the adaptation to change. Software is developed and improved iteratively, with short feedback times by customers, to converge on solutions.
- Refactoring is an agile practice that handles the restructuring of software artifacts with the sole objective of improving their maintainability and agility.

The objective of agile development is to easily adapt to change. Refactoring helps this process by altering the software in a way that such changes are easier to accommodate.

An important consequence of this definition is that refactoring depends heavily on context. When refactoring is applied with a higher goal in mind, the actions taken depend both on what we have now and what we need for the future, to fulfill the requirements. Of course, there are smaller refactorings that don't really need context (e.g. Replace Temp with Query), but these techniques usually don't impact the maintainability of the system on their own, but rather help in more complex problems - for instance, Replace Temp with Query provides the advantages of possibly preventing future duplicated code, as well as often being a vital step before the Extract Method refactoring [Fow99].

Nonetheless, in this dissertation, we discuss the needs and possibilities of a *smart* recommender. Therefore, we're interested in complex refactorings. As mentioned, these refactorings depend on context, on the higher abstraction they're representing [Fow99]. For

instance, when we apply Extract Method, it usually follows the definition of the new method's name, which should be representative of what the method is doing. A suitable name for the method is context dependent, because it usually relates to the containing class, its behavior and its overall objective.

A natural necessity (but not the only one) in refactoring to understand context is to understand the source code. Source code is built to model a certain reality, the one our software gives solution to. Code usually conveys meaning by the words we use - the name of variables, methods, classes, packages, etc. The ability to computationally identify the meaning of these words is called word sense disambiguation (WSD) [Nav09], which is considered an AI-complete problem [Nav09, Pit16, WWF20]. As such, since refactoring depends on WSD to understand context, we believe refactoring is also AI-complete.

From this conclusion, important consequences follow. First, it is not yet possible to truly measure the impact of a certain refactoring. Secondly, it also means that, no matter how smart and fine-tuned our algorithm is at identifying candidates, the final validation should remain on the developer side. Thus, refactoring recommenders should dedicate effort into providing useful and convenient information to programmers, instead of focusing solely on the techniques employed to detect and suggest refactoring opportunities. A smart recommender is not one that tries to surpass the developers, but rather empowers them.

The main challenges and gaps in the current research regarding refactoring candidate identification are:

No standard definition of terms. A starting difficulty in this state-of-the-art research was the lack of standard terms that would allow a more confident, reliable and fine-grained level of research on previous investigations. The most common term in candidate identification was, of course, "refactoring". Apart from that, the language used sometimes differed significantly, ranging from "finding" and "detecting", "suggestion" and "opportunity", "extraction" and "application", to just using the name of the refactoring method.

Lack of reliable datasets. A major issue in candidate identification research is the lack of a reliable, trustworthy dataset where we can benchmark our technique on. Specifically, two consequences arise from this problem:

- The evaluation and comparison of previous techniques is hard. Since there's no standard and extensive set of examples where the influence of refactoring can be studied impartially, investigators usually pick the code used to test the refactoring tool, which becomes a rather arbitrary and possibly (unintentionally) biased process.

  Previous works usually choose one or more open source projects (according to some factors such as maturity, age and activity of the project) with refactoring needs and try to improve the overall maintainability of the system, evaluating the result by comparing the application of refactorings with and without the tool. Others opt for a more qualitative description of the tool, and so are more interested in the feedback provided by the

developers using the refactoring recommender. Nonetheless, this evaluation is done on pieces of code previously chosen, so the conclusions regarding the recommender's performance are hard to generalize.

- Machine learning techniques have limited power and possibilities. Machine learning relies on data, which is usually provided by the type of datasets this research area is missing.

  This is specially concerning in supervised learning, which builds predictive models by feeding large amounts of data to an algorithm. When considering the prediction of future refactoring needs, most available research uses machine learning techniques to this end. Thus, since there are no reliable datasets to train the models, it is expected these techniques can only provide moderate success.

There are works that try to close this gap in refactoring candidate identification. On one hand, some tools have been developed to automatically retrieve refactoring applications by analyzing the evolution of source code in a given software project, and applying rules to classify a certain changes in code as refactoring. These techniques can build sizeable datasets, but lack validation of the instances found - refactoring is different from changing code, so the instances are not necessarily related to refactoring activities. To tackle this issue, on the other hand, other works try to build a dataset with the help of manual validation (usually made by the authors themselves). While these approaches are able to build more robust refactoring datasets, the amount of data retrieved is necessarily smaller. Additionally, these validations are subjective, in the sense that who validates them are people other than the developer, so we still can't be 100% sure that the instance can be considered refactoring (at least, when we speak about more complex applications).

Research focuses on detection and recommendation. Most of the available research in the literature is centered around describing new algorithms to recommend refactoring candidates. As we've discussed, while new techniques are important to improve the overall quality of suggestions, the recommendations still need to be validated by developers. Thus, we believe more research needs to be done in the direction of helping the person who dictates the final step.

Tools are not developer-centric. As stated in the previous item, most available tools focus on the algorithm that provides suggestions. According to E. Murphy-Hill et. al. [MHPB12], refactorings are performed frequently and close to 90% of them are performed manually (i.e. with no tools). The authors identified three key factors for the current underuse of refactoring tools:

Awareness. Developers are not always aware refactoring tools are available in their coding environment. Even if they are aware, most of the times developers remember the tool exists midway through a refactoring process, which then makes the tool irrelevant, because developers usually prefer to continue the refactoring manually.

The lack of awareness in developers reveals a lack of *presence* of refactoring tools.

Opportunity. Opportunity is related to awareness, but addresses the fact that developers may not always understand their specific necessity. For instance, developers might deem a method too long, but not know Extract Method refactoring can fix the particular issue. This happens because the refactoring tool is usually an option in the IDE (hidden in some pop-up menu), that is, these tools don't show they are a good option to solve the problem.

Thus, current refactoring tools show lack of *relevance*.

Trust. Developers may avoid the use of refactoring tools because it might lead to the introduction of errors or unintended side-effects. This problem is related to behavior preservation of refactorings which, as we've briefly discussed, is a difficult task, specially if we consider requirements other than just input-output preservation.

Furthermore, developers are not always aware of the limitations of the tool, so they don't know what key aspects of its use need to be verified. We believe that, specifically in the context of candidate identification, tools should put more effort in better conveying how they got to their suggestions. Current refactoring tools usually just show the final result of the application if accepted by the developer, possibly containing some additional quality metrics. This may carry little information on why that specific refactoring was suggested instead of others.

In summary, we believe refactoring tools also lack *understandability*.

This research, although admittedly preliminary (as stated by the authors), provides important insights on how developers use refactoring tools, with qualitative and in-depth descriptions on why these tools are underused. From our research on currently available refactoring techniques, the problems pointed out by the authors are still relevant, because these methods usually don't take into consideration the developer, just the algorithm. As such, we advocate more developer-centric approaches to combat these problems.

Little research on prediction. The major portion of research in candidate identification focuses on detection and recommendation, two of the three refactoring activities. We believe prediction in refactoring can help in providing the sense of awareness and opportunity of refactoring tools to developers (two key problems discussed in the previous topic). By predicting future refactoring needs, we are also able to conveniently show this information to developers, when such need arises in the code.

A major limitation of available techniques in refactoring prediction is that they usually adopt machine learning techniques. While we believe these methods can show promising results, they still are affected by the lack of refactoring datasets, which in turn affects their performance and overall chances of success.

Limited support. Most existing refactoring tools are built for the Eclipse IDE and the Java programming language [Fer19]. This happens mainly due to the fact that, historically speaking,

Java is one of the most prominent object-oriented programming languages, which directly relates to refactoring. Also, the Eclipse IDE has full Java support and is a coding environment with immense history and popularity as well.

2.3 Live Software Development

Software development encompasses a set of activities that lead to the creation of software artifacts. There are many available methodologies regarding the way software should be created and maintained, establishing the so called Software Development Life Cycle (SDLC). One of such methodologies is Agile Software Development, which favors incremental and iterative development, allowing shorter feedback loops [ARC+19]. A natural extension of this line of thought is *Live Software Development* (LiveSD).

Liveness was first introduced in the context of *Live Programming*, and is defined as "one form of a more general class of behaviors by a programming environment that provides information to programmers about what they are constructing" [Tan13, ARC+19]. Typical programming cycles encompass a series of activities where the developer writes code, runs the program and evaluates the results - traditionally, these phases are called edit-compile-link-run cycles (present in languages like C) or read-eval-print loops (in more interactive languages like Lisp or Python) [Tan13]. In principle, a live programming environment is one where all these activities happen at the same time: as the developer writes new code, the behavior of the program is changing and the results can be immediately analyzed.

There are some qualitatively different possibilities when we define the level of liveness in software.

Liveness level 1 serves as auxiliary information to the developer, with the sole purpose of aiding in the programming tasks.

Level 2 and level 3 correspond to the liveness levels present in most of current available software development techniques: in level 2, the developer does something, asks for a response and then gets the answer; in level 3, the developers does something and, after some time, gets a response (it eliminates the intermediate step of asking for something).

In level 4, there's no wait time between edits and computer response - the behavior of the program is changing along with the modifications introduced by the developer.

Level 5 and level 6 liveness work in the same fashion as level 4, but incorporate additional extrapolation tools that predict future programmer actions and needs. In level 5, it is expected that the environment stays a step ahead of the programmer, instead of lagging behind or just keeping up, hence this level is "tactically predictive" - such predictions, according to S. Tanimoto [Tan13], should be feasible through the use of machine learning techniques. In level 6 liveness, the system is able to make more intelligent predictions about the *true* desires and intentions behind the programmer and the software being built (it should be able to know what the final purpose of the software is, recommending actions accordingly), thus making predictions at a strategic level.

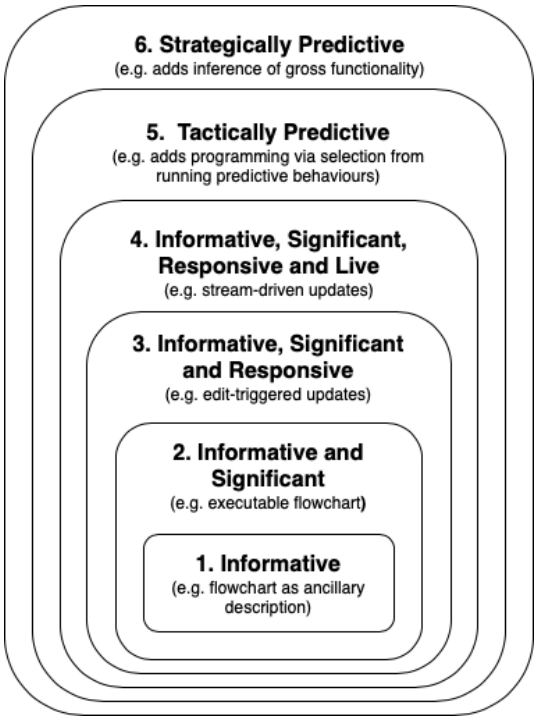


Figure 2.2: Levels of liveness, according to S. Tanimoto [Tan13].

Live Software Development extends these notions in programming to other activities of software development, such as requirements, testing, deployment and maintenance [ARC+19]. As such, agile and live software development are closely related - while the (traditional) goal of agile is to shorten feedback loops from months to weeks or hours, live development's objective is to shorten these intervals to minutes or seconds.

A. Aguiar et. al. [ARC+19] stated the following expected benefits of live information:

Immediacy. The behavior of software is not always easy to predict. Immediate feedback on how changes affect the outputs provided by the system would simplify the process of understanding it.

Exploration. Software development often relies on trial-and-error processes. Liveness, by means of immediacy, would provide safer and easier ways to iterate and adapt our system, allowing a faster convergence to a solution.

Stability. Continuous feedback on how the system works, including its internal state and behavior, would provide more control over the desired output, thus stabilizing the system faster.

With higher levels of liveness, in all activities of the software development life cycle, it is expected that convergence to solutions can be reached faster, because feedback intervals on the changes we introduce into the system will be, in general, shorter. Thus, we have more immediate and direct control on the behavior of software. The same authors highlighted the following key characteristics of liveness in software development [ARC+19]:

Holism. Live Software Development aims to reduce the barriers between the different activities involved in developing software. A holistic environment is one where the different activities are not really seen as different, but can rather be viewed as a whole.

Agnosticism. The use of domain-independent elements and representations would provide an easier and more flexible application of liveness, from which the specifics and concrete needs of each domain could build upon.

Abstractness. Abstractions are representations of software artifacts that provide simpler and more human-understandable comprehension of a system. In order for live information to be meaningful, it should also be able to be quickly grasped by developers.

2.3.1 Techniques and Tools

Elements of Live Programming have been around for decades already, with different levels of support in Lisp's Read-Eval-Print Loop, or in Smalltalk and its descendants [KRB18]. However, live tools have been lately getting more attention, providing useful tools for a programming environment.

Live Tools

There are multiple frameworks and tools available that enhance liveness in development environment and integrate immediate feedback, specially on the web. Chrome browser, along with its web development tools, allows web pages and JavaScript to reload without refreshing [Chr17]. Modern JavaScript frameworks, such as Vue.js and React.js, allow immediate feedback on code changes by using *hot reloading* [Rea17, Vue14]. NaturalMash, a tool created by S. Aghaee et. al. [AP14], is able to ouput the result of modifications in real time, as they're being made.

There are other tools which are non-web related. SOMETHINGit, a Smalltalk library, provides bridging mechanisms to combine dynamic Smalltalk, static Haskell and VDM-SL [ONY13]. LiveMTBD is a Eclipse plugin that extends the Model Transformation By Demonstration (MTBD) with a live model transformation engine to suggest transformation at model-edit time [SGW+11].

Live Programming Languages

Mainstream languages are also taking steps towards liveness. Java supports the replacement of some parts of an application while it is running through the Java Platform Debugger Architecture [Ora14]. Apple's language Swift supports live programming with its interactive playgrounds [KRB18]. Microsoft's Visual Studio also supports Read-Eval-Print Loops for C#, as well as live feedback on code through the Alive[3] extension [KRB18].

[3] Visual Studio's Alive Extension - https://devblogs.microsoft.com/visualstudio/find-your-favorite-visual-studio-extension/#alive

Other existing languages were purposefully created to support higher levels of liveness. Pharo[4] is a programming language and development environment derived from Smalltalk, where development tools and applications share the same runtime environment, allowing updates of any part of the application while it is running (reaching liveness level 4) [KRB18]. The Circa language uses a dataflow-based model of computation, allowing programmers to express their desires against the program's results, which then is used in a backpropagation algorithm to fix the system [Fis13]. Moon is a in-progress live programming language, built from scratch, that, when added to its IDE, promises to immediatly react to changes, providing visual feedback on the system's entities, state and evolution [LL13].

Live IDEs

Programming languages that were built with liveness in mind usually come with their own Integrated Development Environment (IDE) - such is the case of the aforementioned languages Pharo, Circa and Moon. More traditional IDEs allow higher levels of liveness through the use of extensions, e.g. the Alive extension for Visual Studio or the LiveMTBD extension for Eclipse.

2.3.2 Discussion and Open Issues

Live Software Development aims to significantly shorten feedback loops between the inputs and the outputs of development activities. In general, the goal of liveness is to close gaps in an essential aspect of software development: communication.

The behavior of software is not always easy to understand and, therefore, predict. Programs are usually intangible, in the sense that they only have meaning because we give them so. However, there is a difference between this meaning we wish to convey and the actual way we put it into software - such *translation gap* comes from different constraints concerning the development environment. For instance, the requirements of our software system, in its primordial form, are mostly transmitted via words, which are then translated into a suitable model in a program. In this case, the difficulty lies in transforming what we mean by words into what the programming language allows.

Understanding the translation gap usually requires significant cognitive load by developers [ARC+19, CZ11]. With higher levels of liveness, developers are able to better grasp how their actions impact the system, because the effects can be seen immediately, thus reducing the amount of time and effort they need to put in comprehending the software.

Liveness is a broad aspect of software development that spans different areas of research - namely Software Engineering, Human-Computer Interaction, Artificial Intelligence and User Experience [ARC+19] -, where much work is needed in order to fulfill its full capabilities. We now state some of the major challenges Live Software Development needs to address, as well as current issues on available tools and techniques:

[4]Pharo by Example - https://github.com/SquareBracketAssociates/UpdatedPharoByExample/

Complexity. In certain development activities, liveness may not always be needed [Tan13, ARC+19]. Overcomplexity might hinder the natural, expected work these activities need, which can be damaging to the system in terms of development and maintainability.

Unsuitable abstractions. Hiding certain elements of the system, while generally increasing human's comprehensibility, might unintentionally conceal important aspects for the task at hand. Furthermore, there is also the danger of the used abstractions becoming obsolete, because they no longer can faithfully represent the system as it evolves [ARC+19, GME05].

This problem is intimately related to Software Visualization, the next area of research studied in this state of the art (*cf.* Section 2.4, p. 29).

Transient semantics. Live environments need to be cautious about intermediate states of development, because the system may not always be valid or safe to be run [Tan13, ARC+19].

Latency. In a truly perfect live environment, changes should be automatic and immediate - a sort of symbiotic relationship between the developer and the environment. However, while such capabilities are not yet possible, environments must necessarily dedicate some computational power to the transformation of inputs into the desired outputs. As such, developers may have to cope with some latency between their actions and feedback.

Live tools should, therefore, take caution with the mechanisms they use, taking into consideration the fact that more powerful algorithms could significantly increase feedback time, thus potentially harming the usefulness of these tools.

Lack of support. The ecosystem built throughout the history of software development was not designed with liveness in mind. This make some of the aforementioned challenges, such as complexity and latency, even harder to handle.

Recent work, which also includes more traditional programming languages, have put some effort in bringing more liveness to development. Nonetheless, we consider that current practices hinder the possibilities that liveness can offer.

Thus, efforts should be made in rethinking and reinventing current available tools and artifacts. Research in live programming languages and live IDEs, as described previously, work towards this direction - but it is also worth noticing that many of these available tools are still in their infancy, which hardly allows a replacement of current practices.

Low levels of liveness. Most of the available tools offer low levels of liveness. Many of the activities in software work with liveness level 2 or level 3, due to some of the limitations explained in the previous item. Even so, the large majority of research specifically targeting more liveness have put their effort in reaching the fourth level.

We believe that research should also aim at liveness level 5 and level 6, in order to truly take advantage of what Live Software Development can offer.

2.4 Software Visualization

As software becomes increasingly complex, gathering incrementally larger amounts of information and interactions between software elements, at all granularity levels, its understandability and, consequently, maintainability become more difficult to manage [CZ11]. Indeed, maintaining software is known to be the most expensive phase of the development life cycle [BGE95], because a big part of the time spent in this process must be devoted to understanding the system.

Software is virtual and intangible [KM02, GME05]. Humans are better at gathering information from graphical images than from numerical/textual data [Bie89]. Therefore, visualization techniques can help developers make a clearer representation of the software, reducing development time and cost and increasing chances of success [SWM00].

Software visualization addresses three different software aspects [CZ11, GME05]:

Static. Visualization of static elements of software focuses on information that is valid in every software execution.

Dynamic. Provides graphical information about a particular run or instance of software.

Evolution. Adds the time dimension to static software elements.

Software visualization works at different levels (lines of code, classes, architecture, etc). As such, different levels of abstraction usually require different visualization techniques.

2.4.1 Techniques and Tools

Software visualization makes use of suitable abstractions and metaphors to provide a more tangible view of the system, closer to what humans are used to deal with. P. Caserta et. al. [CZ11] provide extensive, detailed information about the different available techniques (*cf.* Table 2.5, p. 30), along with their pros and cons. Here, we'll describe some of these tools, regarding different software levels and aspects.

Code-line-level Visualization

Tools in this category deal with software visualization at the source code level. While these techniques are arguably very low level, they still provide a higher degree of abstraction than regular source code editors.

S. Eick et. al. developed the SeeSoft (*cf.* Figure 2.3, p. 31), a visualization tool that condenses lines of code into colored squared pixels: each line of code occupies one pixel in height, while its length is proportional to the length of the code line; colors are attributed depending on code structure. Indentation of code is also preserved to better represent its structure. It also allows the visualization of several source code files at once by putting them side-by-side [ESS92].

Table 2.5: Types of visualization techniques and representation metaphors [CZ11].

	Level	Focus	Section	Visualization Technique	Representation
Time T Visualization	Line	Line properties	2	Seesoft	2D colored pixel
				Sv3d	3D colored cuboid
	Class	Functioning, Metrics	3	Class BluePrint	2D layers and graph
	Architecture	Organization	4.1	Treemap	2D/3D colored nested boxes
				Circular Treemap	2D/3D colored nested circles
				City/Cities	3D city metaphor
				Sunburst	2D colored radial display
				Solar System	3D solar system metaphor
				Voronoi Treemap	2D colored irregular shapes
		Relationships	4.2	Dependency Structure Matrix	2D table
				UML	2D diagrams
				Geon	3D geon diagrams
				Solar System	3D solar system metaphor
				Landscape	3D landscape metaphor
				Hierarchical Edge Bundles	2D graph with bundled edges
				City/Cities	3D city metaphor with edges
				3D Clustered Graph	3D clustered graph
		Metrics	4.3	Polymetric views	2D graph
				Solar System	3D Solar system metaphor with edges
				UML MetricView	2D UML diagrams with charts on top
				Treemap metrics	2D nested boxes with color and texture
				City	3D City metaphor
				UML Area Of Interest	2D diagrams with area of interest
Visualizing Evolution	Line	Changes	5.1	Code Flow	cable-and-plug wiring metaphor
	Class		5.2	TimeLine	3D building metaphor
	Archi.	Organizational Changes	5.3.1	Hierarchical Edge Bundles	2D graph with bundled edges
		Metrics Evolution		Evolution Matrix	2D matrix
			5.3.2	RelVis	2D Kiviat diagrams and graph
				City/Cities	3D city metaphor with animation

A. Marcus et. al. [MFM03] created sv3D (*cf.* Figure 2.3, p. 31), a visualization technique inspired in SeeSoft. Since sv3D outputs source code line information in 3D space, it allows more information to be displayed, taking advantage of visual parameters, such as height, depth, x-position, y-position and color. One should be careful, however, because more information being displayed can eventually to information overload [CZ11].

Class-level Visualization

Class-level visualization techniques help understand the inner functioning of a class, which may help in understanding its position and responsibilities in the overall system.

Class Blueprint (*cf.* Figure 2.4, p. 31) is a visualization method that displays the general structure of a class, control flow among methods and how attributes are accessed. The purpose of this tool is to show different patterns in the class, allowing a better and quicker understanding of its responsibilities and relationships between methods and attributes [LD01, DL05].

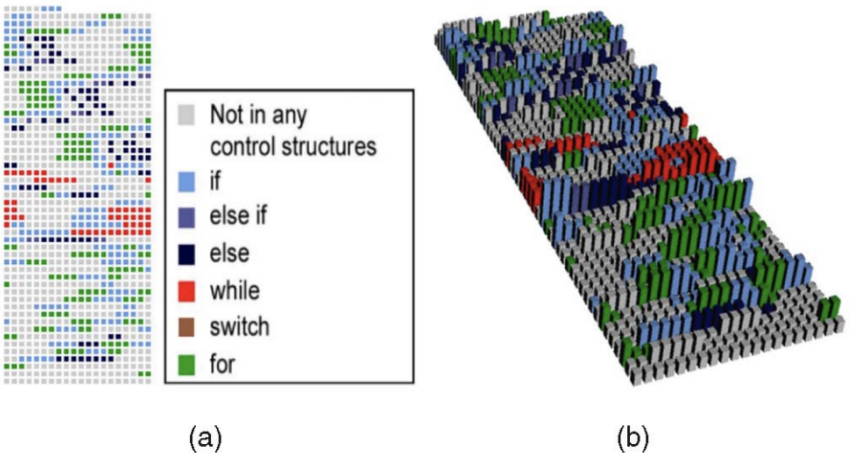


Figure 2.3: Tools representing software at source code level: (a) SeeSoft; (b) sv3D. [CZ11].

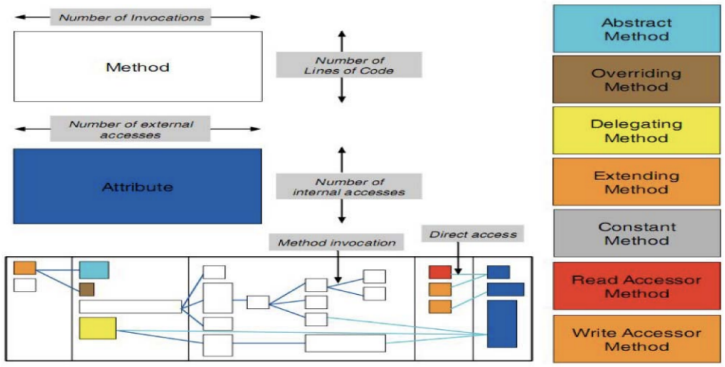


Figure 2.4: The Class Blueprint [CZ11].

Architecture-level Visualization

In the context of general software visualization, the ability to graphically translate our system's architecture is arguably the most important [CZ11].

Regarding software organization, many techniques have been tested using either 2D or 3D representations. In the 2D spectrum, multiple tools have been proposed using graphs, treemaps and sunbursts (*cf.* Figure 2.5, p. 32). By adding other properties to these structures, such as color or variable size, they're able to provide insightful information about the system.

3D visualization techniques usually take advantage of familiar concepts to humans (*cf.* Figure 2.6, p. 32). These real-world metaphors relies on the human natural understanding of the physical world, including spatial factors in perception and navigation. For instance, the cities metaphor represents a software system using multiple cities - where each city can represent packages or classes, and objects inside the city (e.g. buildings) can represent methods, attributes, etc [DF98]. Alternatively, previous works also used solar systems to represent software: each solar

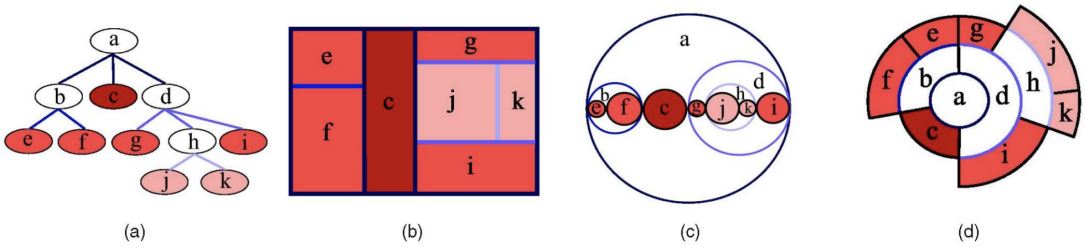


Figure 2.5: These figures display the same system: (a) Graph; (b) Treemap; (c) Circular Treemap; (d) Sunburst [CZ11].

system represents a package, while planets orbiting around represent classes inside that package [GYB04].

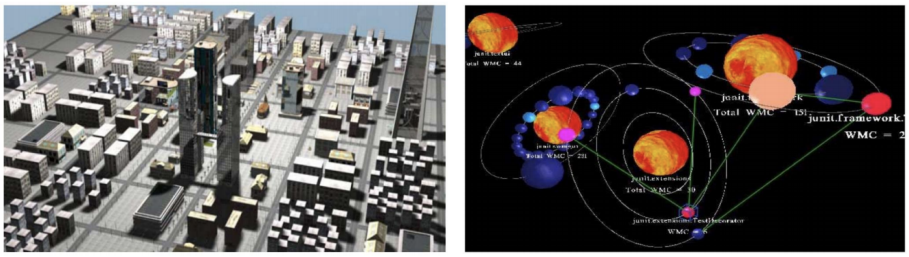


Figure 2.6: 3D visualization tools using real-world metaphors: (a) City; (b) Solar System [CZ11].

Other visualization techniques focus on showing relationships between software entities. P. Irani et. al. [IW00], based on structural object perception, used geon diagrams to represent corresponding UML diagrams (thus mapping 2D information into 3D). Geons consists of regular 3D solids (cones, cylinders, ...) with additional information on how they're interconnected (*cf.* Figure 2.7).

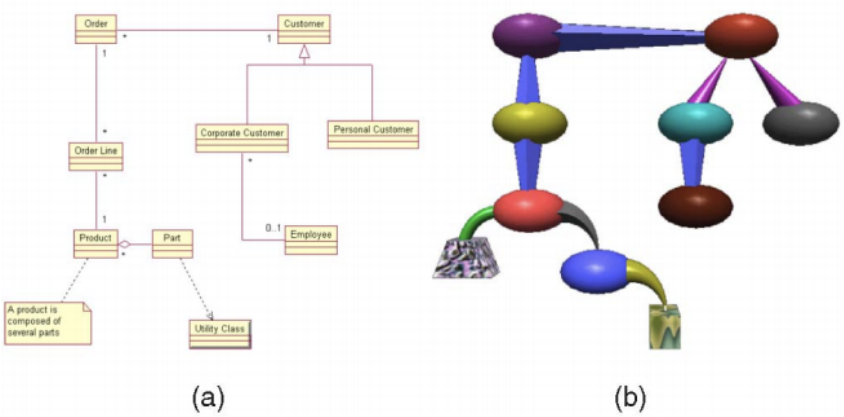


Figure 2.7: Mapping between UML and Geon Diagram: (a) UML; (b) Geons [CZ11].

Finally, regarding software evolution techniques, R. Wettel et. al. [WL08] created a time-dependent visualization tool that shows how classes are evolving along with software releases (*cf.* Figure 2.8). Using a building metaphor, where every building represents a class, and each brick a method inside the class, one is able to understand how a class is evolving: for instance, when a method is removed, the corresponding brick spot is also left empty (reappearing if the method is placed in the class ever again); the color of the bricks also depicts for how long they've been inside a class.

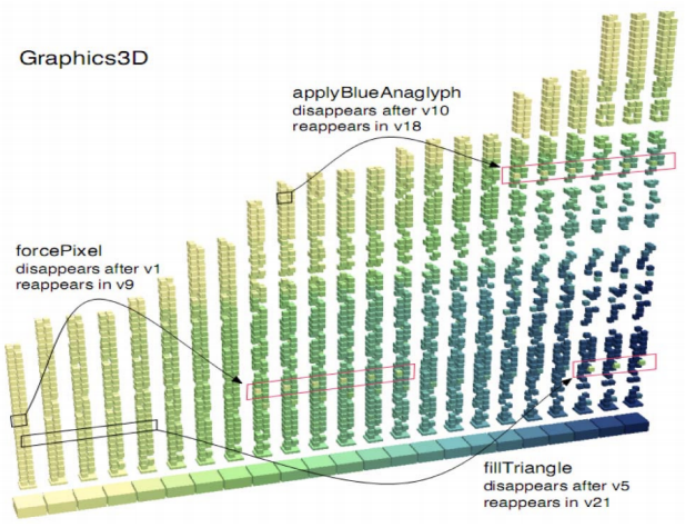


Figure 2.8: Software evolution of the Jmol software [CZ11].

2.4.2 Discussion and Open Issues

Understanding software at source code level is difficult because, as the size of the software increases, the number of interactions and responsibilities of the system's different elements tends to become increasingly complex and hard to grasp. This problem is even more serious when developers have to deal with code they didn't originally develop [CZ11].

Analyzing code usually carries cognitive load. Humans reduce this burden by building abstractions - more tangible representations of what the code represents, its meaning. The goal of software visualization is precisely to build clearer, simpler and more human-understandable representations of software systems.

The main issues regarding software visualization refer to *expressiveness* and *effectiveness* [GME05]. Expressiveness is about the medium used to graphically represent our abstraction (the metaphor used and the level of detail it allows). Effectiveness is refers to how successful is the representation in enhancing the understandability of the information. Visualization systems, therefore, should consider the following key characteristics [GME05]:

Scope of representation. A major issue in software visualization is to define the elements of software we're trying to represent. Overcomplex abstractions can create understandibility

problems, which is exactly what these techniques try to solve. A good scope definition lays the ground work for other choices to build upon.

Medium of representation. The type and level of detail of the information being abstracted (set in the scope definition) dictate the medium of graphical representation.

Visual metaphor. The metaphors used in the medium of representation impact expressiveness. Visual metaphors should provide a natural mapping between the software artifacts and the metaphorical objects they represent. When picking a suitable metaphor, two issues have to be considered:

Consistency. The mapping between software artifacts and representations should be consistent, i.e., a software artifact should correspond to a single metaphor and vice-versa.

Semantic richness. The chosen metaphor should be able to provide as much detail as the elements of software being represented. There should be enough suitable representations in the metaphor to map to a desired software artifact.

Abstractness. If the software elements have different levels of granularity and abstraction, the software visualization tools should also be able to follow the necessary detail (e.g. hide package-level artifacts if the programmer is visualizing a method inside a class).

Ease of navigation and interaction. The medium of representation should have a natural feel on how to navigate and interact. Users should be able to quickly understand the level of abstraction they're working on, and how to move between different levels of detail.

Level of automation. Ideally, the visualization tools should be able to update themselves automatically, without any need of manual input.

Software visualization is an open problem, in the sense that there is no *best* answer on how to cope with the aforementioned key issues these tools have to face. Nonetheless, visualization techniques still offer powerful insights, providing quick and easy understanding of software to developers.

One of the key premises of software visualization is that understanding software at source code level is a demanding task for developers. In the context of refactoring, as we've discussed (*cf.* Section 2.2, p. 12), most of the available tools usually show the result of applying a certain refactoring by displaying how the code will look like if applied. As such, it is expected that recommending more complex refactorings will result in more difficulties in understanding it.

Refactoring tools, in order to show recommendations in a meaningful manner, should explore the advantages software visualization techniques can offer. We believe higher levels of abstraction can improve the understandability of the refactoring decisions, thus making refactoring tools more trustworthy and reliable.

Chapter 3

Problem Statement

3.1 Current Issues 35
3.2 Assumptions 36
3.3 Hypothesis 37
3.4 Research Questions 38
3.5 Methodology 40
3.6 Summary 41

In this chapter, we describe more accurately the problems related to refactoring recommendation. Specifically, we state the current issues (*cf.* Section 3.1, p. 35) found during our research that will be tackled during this dissertation, including some of the assumptions made (*cf.* Section 3.2, p. 36). We then finally discuss our main hypothesis (*cf.* Section 3.3, p. 37) and consequent research questions (*cf.* Section 3.4, p. 38).

3.1 Current Issues

Throughout the state-of-the-art analysis (*cf.* Section 2, p. 7), several areas of research were addressed, which included a full description of the topic, techniques and tools created from past investigations, an in-depth discussion of their advantages and drawbacks and, finally, how these topics relate to our main area of research, which is *Refactoring*.

We explained why we believe refactoring is an AI-complete problem and, as such, the true validation and application of refactorings should remain on the developer's side. From our investigation on refactoring, we emphasized the lack of support current tools offer to developers (*cf.* Section 2.2.2, p. 20). Thus, we believe there is a need for refactoring recommenders to take **more human-centered approaches**.

In order for refactoring tools to be important in everyday developing activities, we identify three key characteristics these tools need to aim for:

Presence. Developers are not aware enough that refactoring tools exist in their programming environment. These tools are usually hidden under some pop-up menu, and needed to be explicitly called by the developer to start analyzing the code.

The lack of immediate presence reduces the effectiveness of refactoring tools because, in more complex refactoring operations, they are disruptive to the normal programming workflow. Understanding the system requires cognitive load, a mental representation of what the code is doing; initiating a refactoring process, that sometimes requires additional configuration steps, breaks the fluidity and chain-of-thoughts of the developer.

Thus, refactoring recommenders need to be more readily available, allowing an easier understanding of their usefulness and capabilities.

Relevance. Even if developers are aware of refactoring recommenders being available in their environment, they might not always be certain how these tools can be used.

The ability to provide applicable, pertinent suggestions to the developer is here referred as relevance. It is intimately related to presence, but is more concerned with the *opportunity* they provide to developers - a refactoring tool can have high presence (e.g. always visible in the IDE), but low relevance (if most of the time the tool's capabilities are not applicable to the problem at hand).

Refactoring recommenders should also help developers understand in which situations they are useful and can be applied.

Understandability. Available refactoring tools often show possible applications at code level, highlighting the lines of code that should be refactored - with the possible additions of showing the result after the refactoring, as well as some quality metrics.

While we believe these features are important, relying solely on them places more burden on the developer: understanding the semantics of a certain piece of code is not always easy; by relying just on the output of a certain refactoring application, developers must spend time grasping the utility of the suggestion, and why other possible applications were not recommended.

Therefore, we believe refactoring recommenders should also dedicate effort into explaining their suggestions more carefully.

3.2 Assumptions

We believe this work is relevant because it explores some important gaps in current refactoring recommenders. In order to tackle the issues listed above, we take into consideration two key assumptions about refactoring and development in general:

1. Our first assumption is that refactoring is an AI-complete problem. A more in-depth explanation on why we believe so is available in Section 2.2.2 (p. 20).

   In summary, we stated that refactoring, among other aspects, relies on knowing what source code is conveying, through the names placed on methods, classes, packages, etc. In order to grasp this kind of information (the translation between code and the reality it tries to model), it is necessary for refactoring recommenders to be able to apply word sense disambiguation (WSD), which is considered an AI-complete problem. Thus, refactoring is also AI-complete.

   Even though we find this assumption strong and probably correct, there is no formal proof to support such statement (the same, however, could be said about WSD).

2. From the first assumption, we conclude that developers must have a central role in the refactoring process. As such, the second assumption we make is that programmers would still see benefit in having a refactoring recommender to aid them in their development activities.

   While previously gathered statements on developers indicate that they would indeed appreciate more relevant refactoring tools, the actual use of recommenders is very reduced in more complex refactorings [MHPB12], which originates a rather contradictory and hard to analyze situation. Nonetheless, we assume this lack of use is related to current flaws in refactoring recommenders, and not related to reduced interest from developers.

3.3 Hypothesis

During our discussion on Live Software Development (*cf.* Section 2.3.2, p. 27), we discussed how liveness can reduce the gap between our actions and their impact on software, through more immediate feedback, lowering the cognitive load necessary for developers to understand code. In the context of refactoring, we believe higher liveness can have a positive effect on the overall development activity. Thus, we advocate for *Live Refactoring*.

This dissertation is based on a hypothesis that serves as the building block for this work's development and implementation. The respective hypothesis is:

Live Refactoring improves software quality and development experience.

Notice this hypothesis has two different components to be addressed. The first refers to software *quality*. We argue that liveness can be of aid to developers in the refactoring process, for projects in different stages of development and levels of maturity. In other words, software quality is improved by using live refactoring tools continually.

In the second component, we are taking into account that higher liveness improves the *presence*, *relevance* of refactoring recommenders, which in turn provide a better development experience. Our reasoning is as follows:

- Liveness provides a natural mechanism to improve presence in refactoring recommenders because it implies faster feedback on our actions. Contrary to current refactoring tools, where the developer has to manually request the code analysis, a recommender with higher liveness will examine source code as it is being developed. As such, it can provide more immediate reactions and suggestions on actions produced by developers.

  It is worth noticing that software visualization also plays a key role in providing higher levels of presence, because it allows the analysis to be directly visible for the developer - if the code is being analyzed constantly, but no feedback is shown, then the recommender still lacks the necessary mechanisms to display its existence.

- Relevance is improved by higher levels of liveness because developers are able to see in (near) real-time how their actions are impacting the information provided by the recommender system. At the same time, by gaining awareness on how the refactoring tool is changing, developers can also reason about the properties the recommender is analyzing and, therefore, in which actions the recommender can help.

In addition to liveness, we also believe that identifying multiple refactoring candidates can improve the *understandability* current recommenders lack. Notice that by multiple candidates we don't mean just different alternatives, but rather independent alternatives, that is, they address different parts of code. By showing multiple suggestions, the developer is able to better understand the assumptions (i.e. bias) of the tool and, therefore, better judge their validity. Our approach is explained in the solution proposal chapter (*cf.* Section 4, p. 43).

3.4 Research Questions

In order to provide insights on the claims we intend to test during this work, and given the novelty of the research problems here presented, we will focus on a small subset of refactoring applications. Specifically, this work focuses on the *Extract Method* refactoring, along with the *Long Method* code smell.

As previously stated (*cf.* Section 3.3, p. 37), the main hypothesis is composed of two different issues to be addressed in this work.

First, we intend to research whether live refactoring tools can improve the code's overall quality by continuous use. Refactoring recommenders, first and foremost, have to provide suggestions that may be of value for the developer. As such, we aim to analyze the quality of the recommendations. Thus, our first research question is:

RQ1. *"Do refactoring tools with higher levels of liveness improve software's code quality in every stage of development?"*

Secondly, we wish to study if current gaps noticed in refactoring tools, which fail to address the central role of the developer in the refactoring process, can be tackled using higher levels of

liveness. Therefore, our analysis focuses on the impact of live refactoring in the overall developer's experience. Our second research question is:

RQ2. *"Do refactoring tools with higher levels of liveness improve development experience?"*

By improving development experience we mean developers find easiness in using live tools, as well as usefulness and simplicity in the recommendations provided - which in turn encourages their use as a more common practice.

We've identified three key characteristics we believe refactoring tools should have, and we wish to study if it is possible to achieve those characteristics by the use of higher levels of liveness and having multiple refactoring suggestions. These features are presence, relevance and understandability. Notice, however, that the results of these measures don't dictate the success of this question: developers can still have a better experience even if presence, relevance and understandability were not improved. Therefore, these characteristics need to be analyzed separately.

In this work, we solely study how liveness can impact presence and relevance. As such, we identify two sub-research questions to be analyzed:

RQ2.1. *"Is the level of presence improved by liveness?"*

RQ2.2. *"Is the level of relevance improved by liveness?"*

Still related to development experience and, more specifically, to the understandability of the refactoring tool, we've stated that multiple suggestions would provide developers with a better understanding on how the recommender works. This concern is different from the ones pointed out in RQ2. because it deals with the identification and recommendation of multiple refactoring alternatives, rather than the use of liveness. Therefore, we identify a third research question:

RQ3. *"Is the level of understandability improved by offering multiple suggestions?"*

We now identify some hypotheses that allow creating a hypothesis test based on the research questions described (*cf.* Table 3.1, p. 40). The null hypotheses (**H0**) present the cases where the tool created doesn't add any value or advantage to its users. The alternative hypotheses (**H1**) present the cases in which the tool developed is relevant to its different users, taking into account the different research questions described.

Table 3.1: Hypothesis tests considering the research questions.

Null Hypothesis	Alternative Hypothesis
RQ1 - *Software Quality*	
H0: Software's code quality didn't improve with liveness.	***H1:*** Software's code quality improved with liveness.
RQ2 - *Development Experience*	
H0: Development experience didn't improve with liveness.	***H1:*** Development experience improved with liveness.
RQ3 - *Multiple Suggestions*	
H0: Multiple suggestions didn't improve the tool's understandability.	***H1:*** Multiple suggestions improved the tool's understandability.

3.5 Methodology

This dissertation, first and foremost, aims to provide insights on how to build a *smart* recommendation system for code refactoring. We believe refactoring tools should use more human-centered approaches to surpass current issues. In order to build these new insights, the work is divided in three major contributions:

State of the Art. In the first phase of this dissertation, we introduce the main concepts and areas of research we deem important for this work. Our state of the art investigation focuses on the topics of *Code Smells*, *Refactoring*, *Live Software Development* and *Software Visualization*.

Research about previous investigations regarding code smells and refactoring is a *must* in any work related to the refactoring activity. On the other hand, research regarding liveness and software visualization in this context is scarcer, due to the fact these topics are not directly related to refactoring, but rather to the easiness and fluidity of developing software (an important matter of this dissertation). Throughout Chapter 2 (p. 7), we delve deeper into the definition and concepts of each of these topics, stating our main takeaways of previous investigations.

Refactoring Recommender (Extract Method). With this work, we wish to build a refactoring recommendation system which will be used as the validation tool to answer the research questions this dissertation focuses on. As stated, our recommender will solely focus on the Extract Method refactoring application.

Empirical Validation. The true analysis and validation of our research questions will come through empirical evaluations using our developed recommender. We use different validation techniques for different research questions of interest, which are further detailed in Chapter 5 (p. 61).

We'll analyze the impact on software's quality by gathering code metrics of multiple refactoring applications of suggestions provided by the tool, comparing the results before and after. This study is conducted by analyzing multiple open source projects in TypeScript, across multiple revisions and files. We reason that, given enough examples of refactoring applications and their consequences on source code, we can better estimate the bias of the tool and gain more confidence on the expected results of using it - contrary to, for instance, surveying developers on specific examples. This validation method refers to **RQ1**.

Our second and third research questions, **RQ2** and **RQ3**, focus on the experience of development, i.e., how the refactoring tool fits in the workflow of the developer. We believe that answers to these questions, which regard the use of more human-centric approaches to build refactoring tools, are necessarily answered by humans. Therefore, our approach to validation relies on a survey where participants are asked about their opinion on our solution's usability and overall experience.

3.6 Summary

In this chapter, we identified the key issues, gathered from our state-of-the-art analysis (*cf.* Section 2, p. 7), that will be explored during this dissertation, which we consider to be of utmost importance in order to build smarter refactoring recommendation systems. We specified *presence*, *relevance* and *understandability* as key features to be addressed, so that refactoring tools are able to suggest more human-centered feedback.

In addition, we also described the main hypothesis (*cf.* Section 3.3, p. 37) that supports this work, as well as all its research questions (*cf.* Section 3.4, p. 38), complemented by possible hypothesis tests that can be done to validate them (*cf.* Table 3.1, p. 40).

Finally, we briefly discussed our methodology in order to tackle these issues and validate our approach (*cf.* Section 3.5, p. 40). The work is divided into state-of-the-art analysis, the development of a VS Code extension, and the validation of the latter through empirical validation.

Chapter 4

Proposed Solution

4.1 Overview 43
4.2 Candidate Identification 44
4.3 Development Experience 48
4.4 Architecture 50
4.5 VS Code Extension 53
4.6 Summary 58

4.1 Overview

This chapter describes in more detail the proposed solution to assert the falsifiability of the hypothesis stated in Chapter 3 (p. 35). The general flow of the tool is available in Figure 4.1 (p. 44). The diagram image was automatically generated using PlantUML[1], a text-based UML generator. The used code is available in Appendix A (p. 119). We explain how that behavior is achieved, and how we separated the different concerns of the system in order to achieve the desired goal.

The refactoring tool has two major components to be addressed, namely *Candidate Identification* (*cf.* Section 4.2, p. 44) and *Development Experience* (*cf.* Section 4.3, p. 48). We then describe in more detail the tool's architecture and implementation (*cf.* Section 4.4, p. 50), which will materialize in a VS Code[2] Extension for the TypeScript[3] and JavaScript[4] languages (*cf.* Section 4.5, p. 53).

As discussed in Chapter 3 (p. 35), we believe refactoring tools need to increase their *presence*, *relevance* and *understandability* in order to improve their quality and overall usability. As such, throughout the sections in this chapter, we explain the reasoning behind our choices, considering these three factors as the primary problems to be addressed.

[1] PlantUML: https://plantuml.com/. Last access on 30 April, 2020
[2] Visual Studio Code: https://code.visualstudio.com/. Last access on 29 January, 2020
[3] TypeScript: https://www.typescriptlang.org/. Last access on 29 January, 2020
[4] JavaScript: https://developer.mozilla.org/docs/Web/JavaScript. Last access on 29 January, 2020

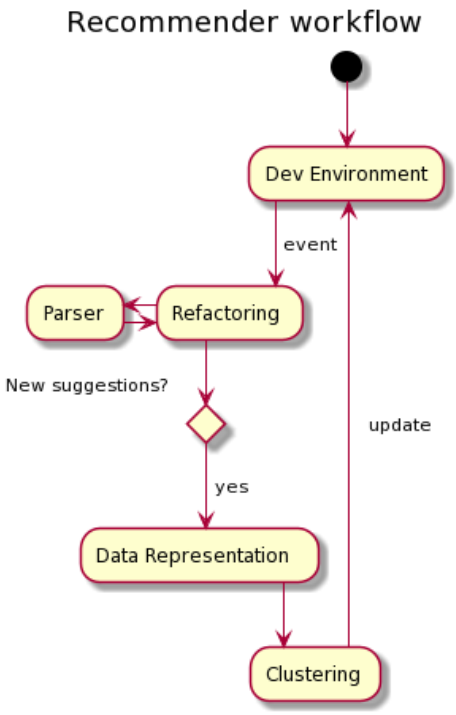


Figure 4.1: Flowchart describing the solution's general behavior.

4.2 Candidate Identification

The first necessity in a refactoring recommender is to find candidates to suggest to developers. We analyzed current practices in this refactoring activity in order to build a clearer picture on why these techniques are used, how they've been used in the past, and what are their main advantages and disadvantages (*cf.* Section 2.2, p. 12).

4.2.1 Reasoning

From our analysis, we picked Artificial Intelligence as the approach to the problem. More precisely, we use unsupervised learning approaches in order to cluster tightly coupled lines of code inside a method, with the objective of maximizing the use of *Single Responsibility Principle* [MRCN03] in each cluster. Thus, each cluster represents a possible refactoring candidate.

Our understanding is that Machine Learning techniques provide some noteworthy advantages, namely: (i) it is believed that machine learning methods allow tactically predictive feedback (i.e. liveness level 5) to be feasible [Tan13, ARC+19]; (ii) many of these techniques don't have to keep the data used to build the prediction model and, as such, if there is the need to store information, the amount of storage space needed can be significantly reduced.

As previously discussed (*cf.* Section 2.2.2, p. 20), most of the refactoring tools currently available, when analyzing and suggesting refactorings to apply, usually only show the result of that application (e.g. by previewing how the code will look like if the suggestion is applied), possibly along some code metrics. These recommendation systems may carry little information

on how the tool reached those results or why a specific recommendation is preferred over others, which, consequently, offers insufficient explanations and insights to the programmer.

While these limitations are certainly a challenge, even for AI techniques, we believe that the use of unsupervised learning to cluster lines of code surpasses these tools in terms of information provided: for each group, the developer can not only understand what those lines of code have in common, but also the main differences to other groups. When compared to techniques that show a single suggestion to apply, we believe programmers are better able to reason about how the recommendations were built, i.e., why a certain line of code was grouped in a certain cluster instead of another.

Moreover, investigation regarding Explainable Artificial Intelligence techniques (XAI) is gaining ever more interest and relevance among scientific research to make AI reliable and easily understood [Lip16, MSK+19, DVK17], which could mean that AI will be able to provide informative and well explained insights to developers in the future.

In summary, we believe that current unsupervised learning techniques bring more *understandability* to refactoring tools. It's important to keep in mind that, when talking about understandability, we're not referring to an explanation of why the suggestion is valuable, but rather why the recommender made such suggestion - in other words, how to better help the developer understand the bias behind the suggestions.

In addition, AI (not restricted to unsupervised learning) is also likely to bring better solutions to the refactoring process in the future, due to the ability to provide level-5 liveness, less storage space needs and stronger capacity to explain suggestions.

4.2.2 Implementation Details

We here discuss the details and decisions behind specifics of unsupervised learning. More specifically, we explain our approach in four major factors of clustering techniques: data representation, algorithms, distance function and cluster validity.

Data Representation

The first necessary step for clustering is data representation: we want to transform a function/method into a mathematical description that is suitable to the application of unsupervised learning techniques. In machine learning, an usual way to achieve this is through feature vectors, which are a n-dimensional representation of an object of interest. We chose our object of interest to be each individual statement inside the method. Thus, clusters will be composed of distinct statements, each of which usually occupies a single line of code in TypeScript.

We build the feature vectors by considering the variables used in each of the statements, as well as any structural dependencies (in the case of complex statements). Variables are an important feature to track because, albeit not perfectly, they show a natural *semantic* dependency between multiple statements. Structural features are also important because they show their *logical*

dependencies. Thus, using these two type of features, we can build clusters that allow a better separation of responsibilities inside a method.

We use a binary representation, meaning a '1' states that a certain feature is present, while a '0' states the absence of it. Consider the piece of code presented in Figure 4.2 and respective representation in Table 4.1.

```
1 function buildName(firstName: string, lastName?: string) {
2     if (lastName) {
3         return firstName + " " + lastName;
4     }
5     return firstName;
6 }
```

Figure 4.2: Example of a function in TypeScript.[5]

Table 4.1: Binary representation of Figure 4.2.

(statement)	firstName	lastName	level-0	level-1
1	0	1	1	1
2	1	1	1	1
3	1	0	1	0

The statements of interest are lines 2, 3 and 5. Each statement will have the value '1' in *firstName* and *lastName* when they're specifically used, and '0' otherwise. The *level* features represent structural dependencies - the if-statement (including the conditional) uses the *level-1* feature. Notice that structural features are cumulative, i.e., if another complex statement was to be produced inside the if-statement, then all statements inside it would also inherit the *level-1* feature. Accordingly, all statements have the *level-0* feature which, in practice, makes it discardable when the clustering technique is applied, since it does not give any relevant information in terms of separability. After producing a binary representation of the method, feature vectors can now be fed into the clustering algorithm.

Clustering Algorithm

During the development phase, we analyzed the pros and cons of three different unsupervised learning techniques: K-Means [Mac67], DBSCAN [EKSX96] and OPTICS [ABKS99].

In the context of live refactoring, where continuous feedback is provided to the developer, K-Means clustering has two main disadvantages: (i) the number of clusters to be extracted, k, is a parameter of the algorithm, which means we should be able to know beforehand how many

[5]TypeScript Functions: https://www.typescriptlang.org/docs/handbook/functions.html. Last access on 26 April, 2020

different responsibilities are being handled inside the method in order to produce valuable results, which can be a difficult problem. A workaround to solve this issue is to run the clustering algorithm several times with different values of k and analyze the multiple results through similarity measures of clusters to pick the best option, but this is also troublesome because these multiple runs could take a long time to compute, excluding the possibility of live feedback; (ii) another problem with this technique is that, generally, its initialization method picks the starting points of the clusters randomly, which makes the final results non-deterministic, i.e., multiple runs for the same parameters might yield different results. This is troublesome in the context of live feedback, because it means that the information displayed to the developer could change even if no modifications were made to the code.

The other two algorithms, DBSCAN and OPTICS, are very similar in their nature: both are deterministic, use the same parameters in the algorithm, and are density-based clustering methods (which means they expect regions where features are concentrated, separated by areas that are sparse). In fact, the OPTICS algorithm is an extension of DBSCAN, purposefully created to address one of DBSCAN's weaknesses, which is detecting meaningful clusters in data of varying density.

Source code can naturally have regions of varying density, depending solely on how the developer built it. Thus, considering all of the aforementioned results from our analysis, we chose OPTICS as the final clustering algorithm in our tool.

Distance Function

Another important matter to discuss is the distance function. A distance function measures how different (or inversely, how similar) two statements are in the representation space. The choice of the distance function is important because it influences the parameters of the algorithm, and even possibly the representation for the data. In our solution, we chose Hamming Distance[6], which is equivalent to the Manhattan distance on binary data.

Let A and B be the set of all the features of two statements with feature length m. Then, the distance between the two statements is defined as:

$$d(A,B) = \sum_{i=1}^{m} |A_i - B_i|$$

In other words, the value we obtain using the Hamming distance corresponds to the number of features the two statements don't share in common. Consider statements 1 and 2 in Figure 4.1 (p. 46): the only feature not shared is *firstName*, so they will have a distance of 1.

The reason we chose this measure as our distance function is directly related to how we represent the data and the chosen algorithm. The OPTICS algorithm has two entry parameters: ε, which describes the maximum distance (radius) to consider when finding core points, and $MinPts$, describing the number of points required to form a cluster.

[6]Hamming Distance - https://en.wikipedia.org/wiki/Hamming_distance/. Last access on 3 June, 2020

When we talk about *MinPts*, it is easy to find a meaningful value for the parameter, in the scope of software development: the minimum points to form a cluster is directly related to the minimum number of lines to form a new refactoring candidate. The minimum number of lines to suggest refactorings is often a customizable option available in recommendation tools, which is also possible to offer in our solution.

The other parameter used by OPTICS, ε (epsilon), however, is not so easy to give a meaningful translation to developers. As we've stated, it describes the maximum distance to consider when finding core points in data - a point is said to be a core point if there are at least *MinPts* with a maximum distance of ε around it. If we use, for instance, euclidean distance as our distance function, it may be hard for the developer to customize this option with a meaningful value. This is specially true if the representation of data is non-binary. On the other hand, when we use a binary representation, along with the hamming distance measure, the ε parameter gains a clearer meaning: we can see it as the maximum desired number of different features between statements in a cluster.

Cluster Validity

After we run the clustering algorithm and obtain the suggestions, we need to evaluate the quality of what was returned. This is known as *cluster validity*. Cluster validation techniques usually consider the properties of the cluster (intra-cluster validity) as well as the properties of the other clusters (inter-cluster validity).

In this tool, suggestions are evaluated according to the Silhouette Coefficient [Rou87], which takes into account the similarity of the data points inside a cluster (its cohesion) compared to other clusters (its separation). We selected this measurement because its properties can be directly related to the use of the principle of single responsibility - ideally, a method with a single responsibility has high cohesion and high separation.

The silhouette is computed for every data point in a cluster and ranges from -1 to +1: a value of -1 means that the point was misclassified and should belong to a different cluster, while a value of +1 means it is a perfect match. A value of 0 indicates that the point is on edge of two clusters and could be assigned to either (or none). By averaging the silhouette of each point, we get the overall classification of the cluster (silhouette coefficient). Averages closer to +1 mean that the cluster is well formed and indicate a noticeable difference to other clusters.

We use the values provided by the silhouette coefficient of each cluster to indicate to the developer the refactorings that are most likely to bring the most impact in the quality of the method and, therefore, overall software quality.

4.3 Development Experience

Our approach to help developers on identifying problems quickly uses the concept of *Live Refactoring*. The *liveness* of a system refers to its ability to provide continuous feedback about the state of the system. Thus, a live refactoring environment is one that is constantly analyzing the

software with the aim of predicting, detecting and suggesting refactoring candidates, even while the programmer is working.

4.3.1 Reasoning

We wish to analyze if shorter feedbacks on how the code is evolving, or, in other words, higher levels of liveness, bring a positive impact in the programming activity.

As discussed in Section 2.3 (p. 24), programs don't have meaning on their own. Software is built to model a certain reality or necessity that can be sufficiently translated into a sequence of instructions executable by a machine. The meaning of programs is usually brought by the purpose we convey to it, through the use of indicative names and references to real-world activities in variables, methods and classes.

One of the major difficulties in this process, however, is handling the translation gap: how to transform our model (which is usually defined with words we use in our daily lives) to a model that is comprehensible to the machine. Understanding this translation gap may require significant cognitive load by developers. With higher levels of liveness, the effects of a certain action or modification in the code can be immediately noticed by the developer, which can greatly impact the amount of time and effort needed to *fill the gap*.

Accordingly, we believe that higher levels of liveness can aid developers in the refactoring process - see Section 3.1 (p. 35) and Section 3.3 (p. 37) for more details. We expect liveness to help in two of the main aspects our solution tries to tackle - presence and relevance -, as follows:

- Presence refers to the tool's availability to developers. Most of the currently available tools are hindered in their potential to help in the refactoring process because they can disrupt its normal workflow and fluidity. In an environment where continuous feedback is provided, developers should be able to better understand the recommender's capabilities, while also obtaining valuable insights in earlier stages of the development.

- Relevance has to do with the developer's understanding of how the tool can be used, i.e., in which situations it is helpful. Liveness can improve the relevance of recommendation systems because developers are able to see changes in the tool's suggestions as they're coding, allowing to better distinguish which actions have effects on the analysis and, therefore, where it can be applied.

4.3.2 Implementation Details

An important aspect to consider when building tools with shorter feedback intervals is that, inevitably, necessary computations to produce the desired results will consume more time and processing power. There is a trade-off between the ability to provide higher levels of liveness and the possible deterioration of the environment's performance. As such, these tools should consider how they can give valuable output in a timely fashion without affecting the overall development experience.

The time complexity for the algorithm chosen in our solution, OPTICS, is highly dependent on the parameters used to run it, being $O(n^2)$ the worst case scenario. While a faster algorithm is always desirable, given that it is used to analyze methods/functions, which usually yield a low number of features to be processed, we found it to be sufficiently performant during its development and testing. Nonetheless, other mechanisms can be employed in order to ensure that the recommendation system is not taking up all of the programming environment's resources. In our solution, we've implemented a *throttle* mechanism with a 250 *ms* rate limit, which means that the algorithm will only run every quarter of second (if changes are being applied to the program). Another possibility would be to use a *debounce* mechanism, that would run the algorithm after a certain time interval of inactivity. We preferred the first option because debouncing the analysis would ultimately cause a delay in the feedback to the developer, which we deemed unnecessary given the development environment was fluid enough to handle the cluster processing in-between modifications to the source code.

Another key factor when building live systems is related to how the information is shown to the programmer. Overcomplex representations might bring too much focus to the suggestions provided by the tool, damaging its effectiveness in aiding in the development process. Additionally, the use of unsuitable abstractions might unintentionally lead the developer to neglect important issues, invisible in these representations. These challenges are related to the areas of Software Visualization, Human-Computer Interaction and User Experience, which, although intimately related to Live Software Development, are not within the scope of this dissertation.

In our solution, we attempted to suggest refactorings to the developer non-intrusively - that is, occupy the least possible space in the environment yet increasing the tool's presence and relevance - in a way that would still make them understandable and intuitive to apply. A more visual and detailed explanation is available further in this chapter, when we describe our recommender implementation in VS Code (*cf.* Section 4.5, p. 53).

4.4 Architecture

An overview of the system's architecture is given by a high-level diagram in Figure 4.3 (p. 51), which graphically represents the layers, packages and relationships between different components. The diagram was also generated using PlantUML (*cf.* Appendix B, p. 121).

This section is the detailed explanation of what are the responsibilities of each component, how they interact, and why we opted for this design.

The system uses a four-layered architecture, where each layer deals with a different concern, namely *presentation*, *middleware*, *logic* and *core*, as follows:

Presentation. The presentation layer handles the client-side of the system, which is the programming environment. Its major concerns are: (i) get input from the developer and (ii) provide output to the developer. It sets up the system to catch events from the environment (e.g. *character inserted* or *mouse position changed*) and also displays the results from the tool back to the programmer (e.g. show a warning message or underline a certain part of the code).

Middleware. This layer provides an abstraction for communication between the presentation and logic layers. This way, we can have different UI requirements (e.g. events to listen or development environments) but still disassociate them from the analysis and processing to get the desired results, providing a better decoupled system structure.

The middleware is responsible for: (i) listening to events, sending them to the logic layer for processing, (ii) parse source code to give the necessary information to the logic layer and (iii) receive the results, sending them to the presentation layer.

Logic. The logic layer is responsible for the refactoring applications and also data representation and clustering algorithms. It is in this layer that suggestions, according to the events received, are produced. It is also responsible for asking the middleware layer about information regarding source code.

Core. Core components are shared by all other sub-systems and carry information that is accessible to all. Thus, the core layer functions as the *language* of the system: instances of core components can be understood and processed by every layer.

For instance, clusters are created in the logic layer but are used in the presentation layer to show that information to the developer. Variables are produced by the parser (in the middleware layer) to be used in the computations of the logic layer. The same applies to statements and methods.

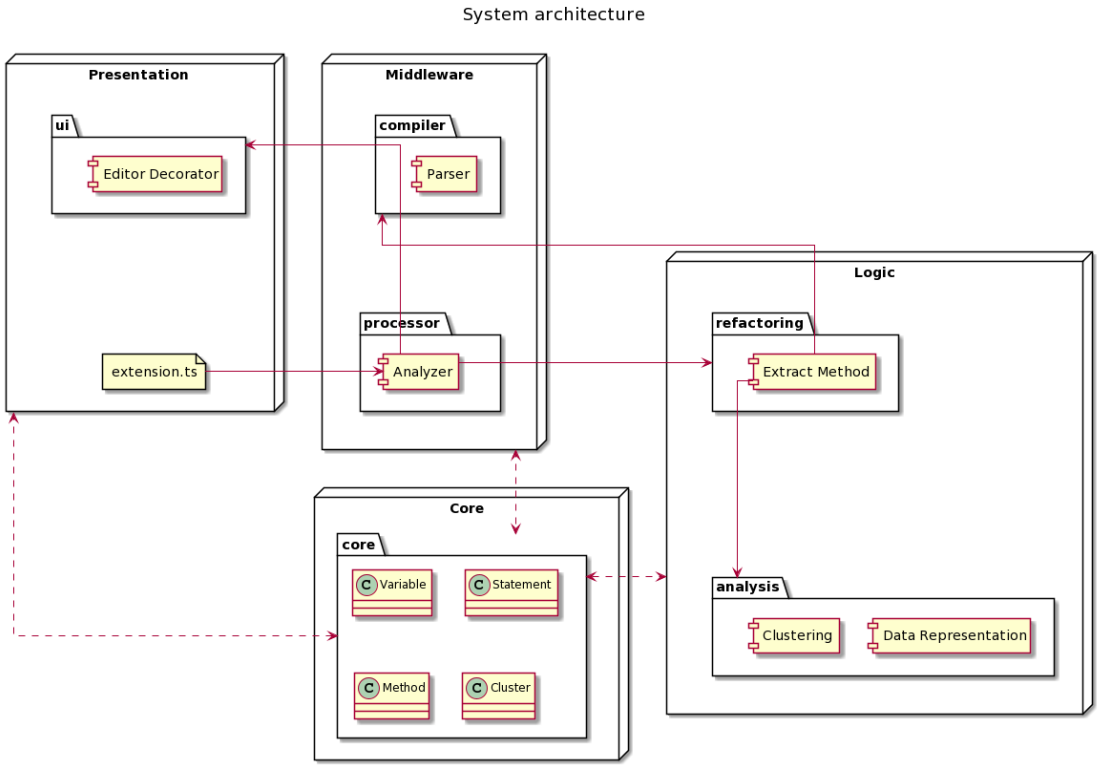


Figure 4.3: System architecture.

After describing the main sub-systems of our solution, we'll now provide more detail into the components of each of these, which are divided in packages. We start by the presentation layer, then middleware, then logic, finishing in the core layer.

Extension. The *extension.ts*, although not a package, is the entry file for every VS Code extension[7]. In this file, we activate our extension's functionalities, providing event listeners to user actions. When an event is fired, it sends that information to the processor package in the middleware layer.

Editor Decorator. The editor decorator receives actions from the analyzer (middleware layer), as well as the necessary data, transforming those events into visual output, which is dependent from the development environment in use. It is responsible to show changes in the development environment to the user, such as highlight lines of code, underline errors, show 'light-bulb' suggestions, etc.

Compiler. The compiler package is responsible for parsing source code and transforming it into readable data, i.e., an instance of a core class (representing variables, statements or methods). Having a component responsible for transforming source code allows the abstraction of that knowledge to other layers, which means that, for instance, the logic layer is independent from the source code it is analyzing. This allows the system to better scale to support more languages in the future.

Processor. The processor package is one of the most important components because it is the main link between the presentation and logic layers. Whenever an event is fired by the user, that event is sent to this package, which then redirects it to the appropriate refactoring technique in the logic layer. Additionally, when the logic layer computes the suggestions, it is also responsible for sending it back to the development environment, while also triggering the correct visual displays in the editor decorator.

Refactoring. Every refactoring that the system supports (only Extract Method, in our case) goes into the refactoring package. Its main responsibility is to fetch the necessary information about source code (from the compiler package) so that it can be processed in the analysis package.

As an example, when applying the Extract Method, we're mostly interested in variables and structural dependencies inside a method (*cf.* Section 4.2, p. 44). On the other hand, Extract Class refactoring probably takes into consideration relationships inside the class, so it needs to gather that information as well.

Analysis. The transformation of source code into numerical data and later clustering process takes place in the analysis package. It receives requests from the refactoring package to

[7] VS Code - Extension Anatomy: https://code.visualstudio.com/api/get-started/extension-anatomy#extension-entry-file. Last access on 1 May, 2020

represent data (we used a binary representation), and also provides the interface to run the unsupervised learning algorithms (in our case, OPTICS).

Core. The core package holds every object that is understood by every other component. Variables, statements and methods are mostly used to get the information needed to build our recommendations, while clusters represent the end results that are suggested to the user.

4.5 VS Code Extension

The solution we've implemented to suggest refactoring applications with live feedback was built as a VS Code extension, which is the main source of interaction between the tool and the developer. In this section, we explain the components that directly refer to the use of the extension and the overall experience in the programming environment. The major functionalities the tool offers are five:

Accept User Input. Our recommender listens to events fired from VS Code to start producing refactoring candidates and displaying those results in the UI. The system listens to events provided by user input.

Display Suggestions. The identified candidates are displayed in VS Code to the developer, in such way that they're easily distinguishable and also clearly show the lines of code they relate to.

Highlight Suggestion. The developer is also able to further highlight a cluster of interest, to better analyze its structure or to apply the change suggested by it. Additionally, we also inform the user if it is not possible to apply the change automatically.

Apply Suggestion. When a cluster is accepted, the tool automatically applies the Extract Method refactoring, after which the user can pick the name of the new method.

Code Evolution. Developers also have access to a page inside the VS Code to analyze how their code is evolving, according to several software quality metrics.

4.5.1 Accept User Input

The two ways of listening to user input in our tool is via listeners and commands. We here explain what each of these components does and how they help the overall user experience.

The first type of events, from listeners, are produced by actions from the user in his regular programming experience. Thus, these events are the source of the ability to provide live feedback, since they register actions as the developer is programming.

Text Editor Changed. This event is fired whenever, inside VS Code, developers change the active page of development (for instance, by switching tabs). This allows the tool to notify the middleware layer that the source code that will be parsed has changed. It also would allow us

to stop the tool if the new language isn't supported, but we don't need to care for that because it is already provided out-of-the-box by the code editor.

Event: *vscode.window.onDidChangeActiveTextEditor*[8]

Selection Changed. The second event, and the main source of liveness, is related to a change in the position of the cursor inside VS Code. The detection of a change in the cursor's position satisfies the two needs of our recommender: it triggers when the developer inserts new characters, or when the developer changes position (e.g. by clicking the arrow buttons or mouse).

Both triggers are important because the insertion of new characters may imply that the suggestions should change, and a different position of the cursor may imply that there's a new context to be analyzed (in our case, the context is the method being modified). This way, we can always know if there is a need to update our suggestions, thus providing an opportunity for live feedback.

Event: *vscode.window.onDidChangeTextEditorSelection*[9]

File Saved. Whenever the developer saves a modified file, our tool triggers the metrics analyzer, which will output several software quality metrics, which are then saved into a file. The metrics are used for the code evolution functionality provided by the tool.

In order to avoid taking too much storage space, we only keep the 20 most recent metrics, for each file. Thus, if a file already has 20 entries and the user saves it to disk (triggering a new quality analysis), the oldest entry in that file is discarded.

Event: *vscode.workspace.onDidSaveTextDocument*[10]

The other events our tool listens to are those provided by commands. Commands are a way to trigger actions in VS Code[11], and are available inside the development environment by opening the Command Palette (using 'Ctrl+Shift+P' or 'Cmd+Shift+P'), as in Figure 4.4 (p. 55). After opening the palette, developers can search for the commands provided by our recommender. The first command shown in Figure 4.4 (p. 55) is responsible for highlighting clusters and applying the method extraction, while the second open a new tab in VS Code displaying the evolution of code quality metrics.

[8] VS Code API - onDidChangeActiveTextEditor : https://code.visualstudio.com/api/references/vscode-api#2468. Last accessed 1 May, 2020.

[9] VS Code API - onDidChangeTextEditorSelection: https://code.visualstudio.com/api/references/vscode-api#2470. Last accessed 1 May, 2020.

[10] VS Code API - onDidSaveTextDocument: https://code.visualstudio.com/api/references/vscode-api#2741. Last accessed 1 June, 2020.

[11] VS Code - Commands : https://code.visualstudio.com/api/extension-guides/command. Last accessed 1 May, 2020.

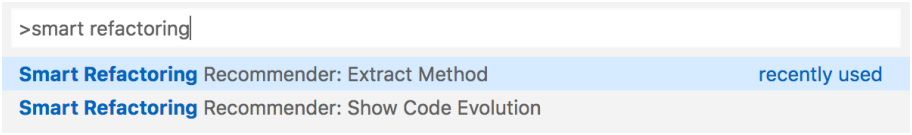


Figure 4.4: Command palette showing command provided by the refactoring tool.

4.5.2 Display Suggestions

An important aspect for a refactoring tool to be effective is how it provides the suggestions to the developer. This issue is related to other areas of research, such as Software Visualization and User Experience, which are out of scope of this dissertation (*cf.* Section 4.3, p. 48). Nonetheless, considering the functionalities VS Code has to offer, we've attempted to provide a intuitive and visual appealing programming experience. Our tool shows suggestions as the programmer is coding by showing the resulting cluster in the code editor's gutters, next to the lines of code (*cf.* Figure 4.5). Each different color refers to a different cluster and, therefore, a different refactoring candidate.

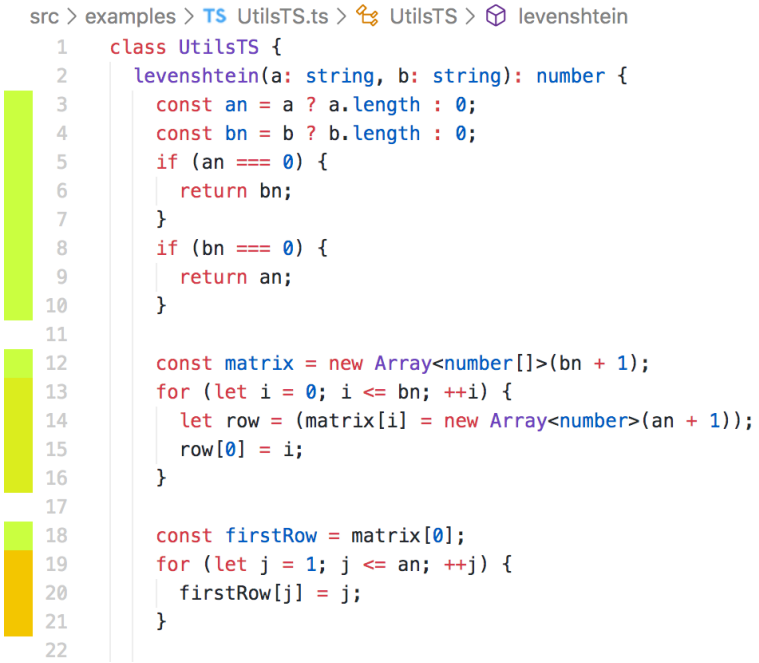


Figure 4.5: Refactoring suggestions in VS Code gutters. In this example, three recommendations (clusters) are shown.

The colors used in our tool use a gradient from green to red. Each cluster is assigned a color according to its validity, measured by the silhouette coefficient (*cf.* Section 23, p. 48), in ascending order, i.e., clusters with high separability (coefficient closer to +1) are assigned to colors closer to red.

If no suggestions are available for the user, i.e. no clusters were found, then the gutter takes a green color, spanning from the beginning to the end of the method (*cf.* Figure 4.6, p. 56).

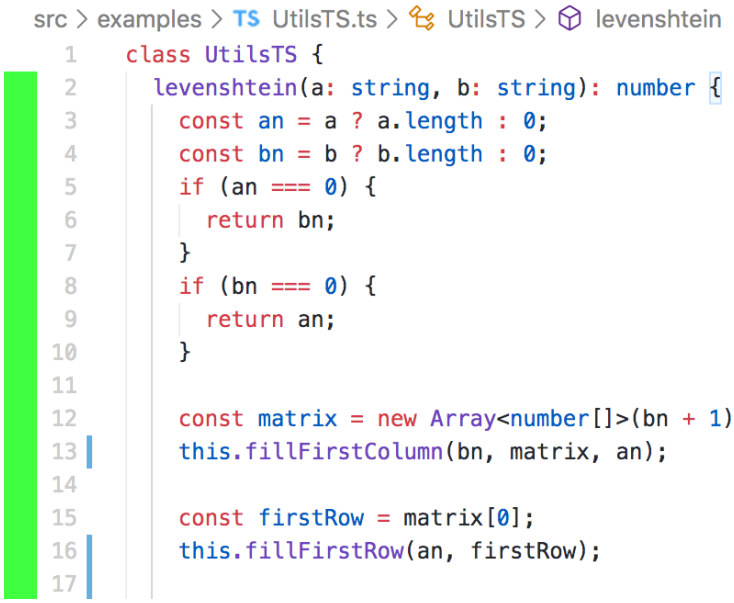


Figure 4.6: Refactoring suggestions in VS Code gutters when no candidates are available.

4.5.3 Highlight Suggestion

Whenever the user runs the "Extract Method" command made available by our extension (*cf.* Figure 4.4, p. 55), it will open a new menu with a list all the clusters/suggestions the tool produced, as shown in Figure 4.7.

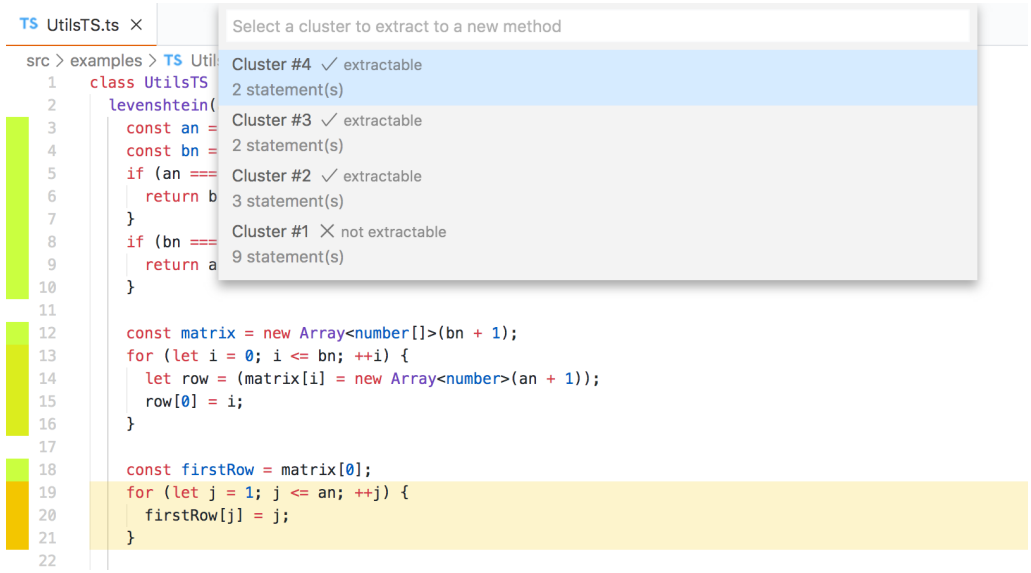


Figure 4.7: Highlighted suggestion.

For each suggestion, the following information is displayed to the developer: cluster number, number of statements affected, and if the operation is automatically extractable by our tool. We consider a cluster to be automatically extractable if all the statements are consecutively together. We also show clusters that aren't automatically extractable because they can give

valuable information to the developer nonetheless (but the refactoring operation must be done manually).

Finally, as the developer is traversing the multiple options available, the suggestion currently active will be highlighted in the code, using the same color in the gutter, to distinguish it from the rest of the recommendations.

4.5.4 Apply Suggestion

The final feature provided by our tool is the ability to automatically apply the method extraction using the produced suggestions. Considering the example in Figure 4.7 (p. 56), if the user presses 'Enter', we apply VS Code's built-in Extract Method action[12], which will replace the highlighted code with a method call. The user is then prompted to rename the method, as shown in Figure 4.8.

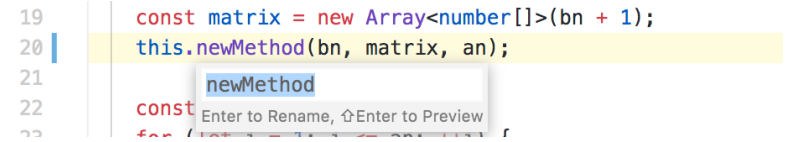


Figure 4.8: After a suggestion is applied, the developer can choose the new method's name.

The new method created will contain the lines of code that were just replaced in the original method (*cf.* Figure 4.9).

```
44  private newMethod(bn: number, matrix: number[][], an: number) {
45    for (let i = 0; i <= bn; ++i) {
46      let row = matrix[i] = new Array<number>(an + 1);
47      row[0] = i;
48    }
49  }
```

Figure 4.9: A method is automatically created with the extracted behavior.

If developers press 'Enter' on a non-extractable cluster, they will receive an information message saying that the desired operation is not allowed (*cf.* Figure 4.10)

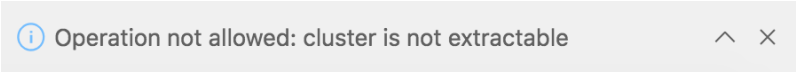


Figure 4.10: If a cluster is not automatically extractable, the user will receive an info message.

4.5.5 Code Evolution

Our tool also provides a second command, "Show Code Evolution" (*cf.* Figure 4.4, p. 55). When the user activates this command, a new view is shown in VS Code with several code quality metrics (chosen from and calculated according to S. Fernandes et. al. [Fer19]) for the active file

[12] VS Code - Refactoring: https://code.visualstudio.com/docs/editor/refactoring. Last accessed 2 May, 2020.

at the time of the command trigger. The view displays the evolution of these metrics throughout various points in time in a multi-line chart, as shown in Figure 4.11.

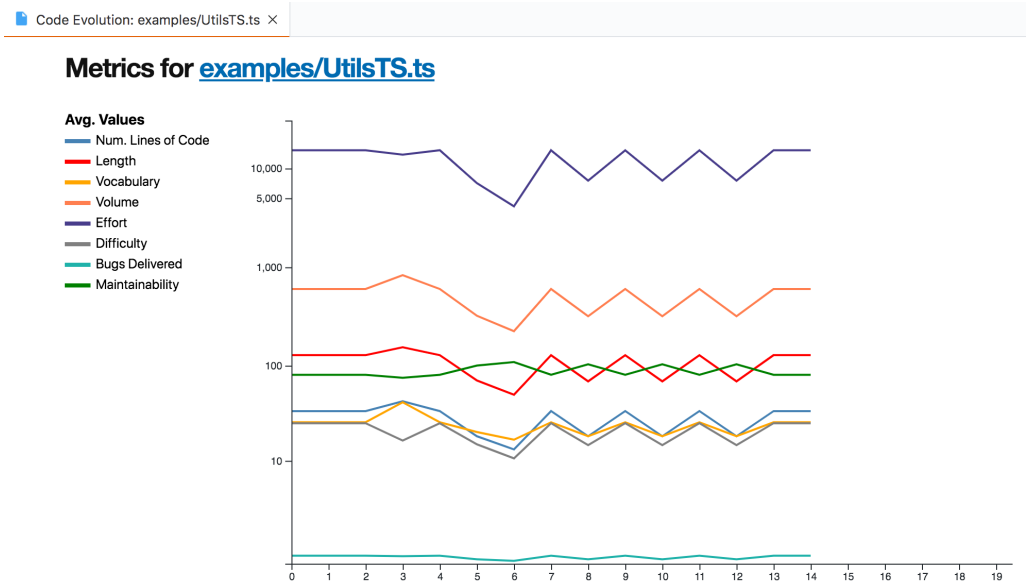


Figure 4.11: Code quality evolution.

The *y*-axis displays the absolute value of each metric, using a logarithmic scale due to the wide range of values these measurements can take. The *x*-axis shows the range of 0-19, where each unit represents a point in time when the file was analyzed. The chart is automatically updated with new values whenever the file it analyzes is saved to the disk by the user, as discussed in the beginning of this chapter (*cf.* Section 4.5.1, p. 53), up to a maximum of 20 (most recent) entries.

The user is also able to the see the values of the metrics by hovering the graph near the entry of interest. A vertical line will be displayed, showing the entry the metrics are referring to, as well as a small rectangle with the values (colored according to the metric they represent) (*cf.* Figure 4.12, p. 59).

The code evolution view was produced using VS Code's Webview[13] capabilities and D3.js [14].

4.6 Summary

In this chapter, we've described in more depth the proposed solution, from design decisions to implementation details, and how we think it can impact future refactoring recommenders in order to produce more reliable and valuable development tools.

The first three sections are directly related to the flaws we've encountered in current refactoring recommenders - lack of presence, relevance and understandability -, and how we wish to solve

[13]VS Code - Webview API: https://code.visualstudio.com/api/extension-guides/webview. Last access on 3 June, 2020

[14]D3.js: https://d3js.org/. Last access on 3 June, 2020

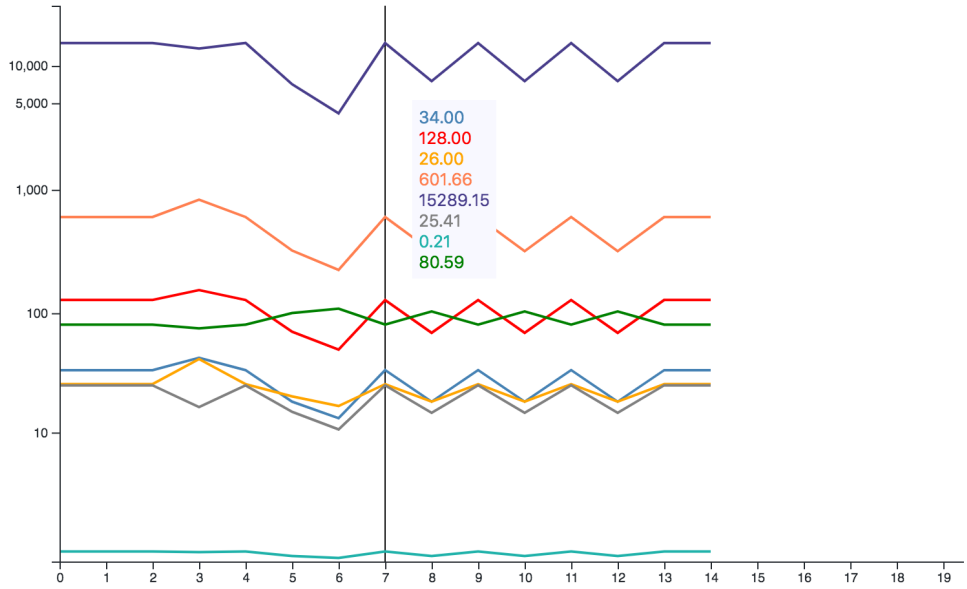


Figure 4.12: Code quality values are displayed when the user hovers the chart.

these issues. Section 4.2 (p. 44) focuses on the identification of refactoring candidates, while Section 4.3 (p. 48) explains how these recommenders can improve their usability using liveness.

Finally, the last two sections are more related to the system design itself. Section 4.4 (p. 50) explains how the different concerns are divided in our solution, as well as why we believe it makes it easier to develop, and better to maintain. Section 4.5 (p. 53) focuses on how the solution works as a VS Code extension, i.e., the functionalities developers should expect to see when they use the tool.

Chapter 5

Empirical Evaluation

5.1 Controlled Experiment with Robotized Refactoring Tool 61
5.2 Survey on Live Refactoring 83
5.3 Summary 97

The objective of this dissertation was to develop a tool capable of reducing the effort in understanding and maintaining a software system. In order to validate the performance of the tool relative to its hypothesis and research questions, described in Section 3.3 (p. 37) and Section 3.4 (p. 38), respectively, we developed experiments that attempt to provide an answer to these issues. Since our main hypothesis can be divided in two different components to be addressed, we also divided our empirical validation in two experiments, according to their usefulness and effectiveness in bringing valuable insights to our questions.

5.1 Controlled Experiment with Robotized Refactoring Tool

The objective of our first experiment is to test whether the continuous use of a live refactoring tool improves, in general, code quality. In other words, we wish to measure how our tool affects software in terms of maintainability and complexity. It is important, therefore, to keep in mind that this experiment is not designed to understand how liveness helps developers, but only evaluate if it would be a good asset to have in their activity. The main research question in focus is **RQ1** (*cf.* Section 3.4, p. 38).

Our experiment consists in building a robot that applies refactorings recommended by the tool across multiple methods, files and revisions. Given sufficient instances where refactorings were applied, and that we can measure their impact on software, we are able to gain more confidence on the results achieved by the live refactoring tool.

5.1.1 Experiment Design

A controlled experiment investigates a testable hypothesis where *independent variables* are manipulated to measure their impact on *dependent variables*. In an experiment using the scientific method, we are interested in the *effect* that a *factor* has on an attribute of interest, through the use of *treatments*, on the elements of study called *subjects* or *objects*. In the context of this work, we attribute the factors to refactoring suggestions provided by the tool, treatments are the application of those same suggestions, and we wish to measure the effect of the latter in software's code quality. The objects of interest, given the focus on the Extract Method refactoring, are class methods.

5.1.1.1 Reasoning

The purpose of this experiment is to validate whether code quality benefits from continually applying refactoring suggestions and, therefore, developers benefit from using a live refactoring tool continuously showing opportunities. Notice that, in order to achieve the goal of this experiment, it is not necessary to rely on developers: despite their central role in the refactoring process, the quality of a certain refactoring application is independent of that same role. We assume, however, that the tool picks the suggestions on behalf of the developer, which can possibly be different from the ones a real developer would apply.

We believe our approach to this study is innovative in this area of research and provides great advantages, compared to the classical reliance upon developers to lead the experiment, specially when applied in larger scales.

The experiment consists in simulating a developer in his refactoring activity. Thus, we created a robot that analyzes source code and applies refactorings suggested by our tool. The robot was built upon features provided by VS Code, namely:

- Executing CLI commands
  - Retrieve version control information;
  - Switch between project versions;
  - Find all files to be analyzed.
- VS Code API
  - Open files;
  - Select lines of interest in file;
  - Apply and save changes;
  - Activate our tool analysis;
  - Access to the suggestions provided by the tool.
- TypeScript Server

- Analysis of source code to indicate the location of methods;
- Application of refactorings.

The robot searches all valid files (i.e. TypeScript or JavaScript files) in a given repository of a software project and, on each file, it analyzes each method inside a class, automatically applying refactoring suggestions provided by the tool. This process is repeated across multiple revisions of a project. Algorithm 1 shows the pseudo-code that dictates the general behavior of the robotized refactoring tool.

Algorithm 1: Analyze software project `P` on folder `F`

```
AnalyzeProject (P, F)
    inputs: A software project P; a folder of the project F
    foreach revision in revisions(P) do
        foreach file in files(F) do
            foreach method in methods(revision, file) do
                while available refactorings do
                    1. apply refactoring 2. record result
                end
            end
        end
    end
```

At first glance, we can see that the computational complexity and requirements of the algorithm are a factor to take into consideration. The time complexity of the robot's algorithm is $O(n^3)$ in the best case (when there are no refactoring suggestions), which is certainly not ideal if we want to run this experiment on a large scale. However, it is also worth pointing out that the algorithm should be parallellizable, depending on how the effects of the refactorings are being measured. For instance, if we only measure the impact of a refactoring at the method level, then we can parallellize the algorithm up to the third loop (traversal of methods in a class), making the refactoring applications the bottleneck of the analysis. On the other hand, if we measure the results at the file level, then all refactoring applications produced inside that file need to be taken into account - and so the traversal of the third loop is not paralellizable anymore. The application of refactorings cannot be paralellized because the suggestions themselves have a dependency chain, i.e., a suggestion may appear only after another one was previously applied.

Another important aspect worthy of attention is the phase where refactorings are applied. The information we have access to is contained in the state of the files at a certain revision, i.e., we only have the source code that entered into the codebase in a well defined date (the time it was committed). This means that we are not able to use the refactoring tool in intermediate steps, e.g. midway through the development of a method. Therefore, the robot created for this experiment, which tries to simulate the usage of the refactoring tool, relies on an approximation of the behavior of developers. There are multiple ways we can attempt to address this problem:

1. The first option is to actually have recorded behavior of developers at the time of a certain commit to the project's version control system. This first option, while closer to approximate regular usage, is not feasible at the moment, primarily due to the fact that there are no sizeable records of all actions produced by a developer to reach a certain feature in the source code. Secondly, it would also carry its own additional issues to be solved, such as when we should apply the suggestions - the developer is not always refactoring, so overapplying refactorings might also not be a good approximation.

2. Another alternative would be to better understand the structure of the projects under analysis, regarding their contribution policy. For instance, if a project is using feature branching[1] and all commits in that branch are discarded (or squashed) when they enter the common codebase, then we can reason that commits that happened in that branch have smaller intervals of time between them. This knowledge could be useful to gain insights on the localization of certain refactoring suggestions, and better understand and estimate their impact on code quality when merged into the main codebase. This option, however, requires considerable initial effort in studying and picking the best projects for the objective of the experiment.

3. A third option is to make no assumptions whatsoever when it comes to the developer's behavior or how the project is structured. The information we have access to is the codebase and its evolution across multiple versions. In this context, we are simulating a developer that finished all tasks appointed on a certain revision, and is now refactoring the final code before submitting into the main codebase. This approach is the least representative of an ideal use of refactoring tools - continually, instead of after all features are built. However, it takes no assumptions or possible heuristics about the projects under analysis.

In this experiment, we chose the third option as our approach to build the robot. As previously stated, this option does not make any assumptions about the project or their developers. Given the novelty of this experiment in the field, we believe that assumption-free tests are the ones that can provide us with more confidence in their results, since it is expected that they're more easily reproducible and fair (i.e. only one factor is changing while all other conditions remain unchanged) - the other alternatives here presented require more work and investigation to be held as safe approaches.

That said, it is important to understand how the used approach can be analyzed and its results justified. The great advantage provided by a robot that automatically applies refactorings is that we can build sizeable, measurable datasets with these refactoring applications: as these datasets grow, we can also be more confident about the general behavior of the tool.

Another aspect to take into consideration is the diversity of these datasets. We've used the robot to analyze multiple software projects, from which we can collect information of projects with different purposes, on different stages of development, with different levels of maturity and

[1] Feature Branch: https://martinfowler.com/bliki/FeatureBranch.html. Last access on 12 June, 2020.

active developers, and with different refactoring needs. We can increase our confidence on the suggestions provided by the tool, if results are regular across this increased diversity.

In summary, our experiment was built around the idea that having many measurable refactoring applications provides more confidence on their results than having feedback from a group of developers. There is a wide variety of open source projects available to run these robotized experiments, where much data can be gathered from. Patterns found across distinct, diverse projects and codebases allow us to build clearer and sounder conclusions about the use of the tool.

The following sections, regarding the experiment design, are based on the reasoning and justifications provided in this section. We describe with more detail the concrete elements of this experiment, and how they relate to the objective of this empirical validation.

5.1.1.2 Objects of Interest

The objects/subjects of interest in an investigation are the ones that receive the treatment. The treatments provided by our tool are possible refactoring applications confined to the Extract Method, which is always applied in the scope of functions or methods. Thus, the objects of interest are functions and methods. More specifically, we're interested in those written in the JavaScript and TypeScript languages.

As we've discussed in the previous section, we're interested in building a sizeable dataset of refactoring applications. There are many open source projects available online, so there is certainly room to build a considerable group of objects of study. However, we're also interested in building a diverse dataset, that is, not too biased towards a type of project, requirements or level of maturity. As such, we've picked multiple open source projects, in different stages of development and different purposes, to retrieve our objects of interest for this experiment.

All the information about the projects described below was retrieved before 13 June, 2020. The projects we've chosen are:

VS Code[2] Visual Studio Code is a source code editor, available for Windows, Mac OS and Linux. It has built-in integration with TypeScript, JavaScript and Node.js, but also supports many other languages and runtimes, such as C++, Python, Go, Java, Unity, among others.

VS Code is one of the oldest and most mature projects written in TypeScript, dating its first commit to 13 November 2015, but still has a very active community, features under development, and plans for the future. At the time of this writing, it is one of the most starred projects on GitHub, and the most starred in the TypeScript language.

[2]Official page: https://code.visualstudio.com/. Github repository: https://github.com/Microsoft/vscode/. Last access on 12 June, 2020.

[3]Official page: https://deno.land/. Github repository: https://github.com/denoland/deno. Last access on 12 June, 2020.

Deno[3] Deno is a simple, modern, secure runtime for JavaScript and TypeScript, built on top of the Rust language and the V8 engine, which provides a single executable for a productive scripting environment. Deno was thought of and created by the mind behind Node.js.

The first commit in Deno's repository dates to 14 May 2018, which is significantly more recent than VS Code's. Deno also has less commits and contributors, and just recently introduced their 1.0 version of the software. Thus, we can understandably reason that this project is still in a more infant, immature stage of its development. That said, Deno is also the second most starred TypeScript project in GitHub and has a lot of expectations surrounding it.

Smart Refactoring Recommender Smart Refactoring Recommender is the VS Code extension we've developed for this work (written in TypeScript), whose features and details were described in the previous chapter (*cf.* Chapter 4, p. 43). This project's first commit was on 22 February 2020 and was produced by a single developer. Additionally, contrary to the previous projects, it has a hard deadline to fulfill its requirements.

As described, we've picked projects with different purposes and levels of maturity, in order to try to minimize the possible bias in the results our robotized tool will produce. This is further confirmed in Table 5.1, which describes the projects in terms of number of commits, contributors, files and lines of code, clearly showing non-identical complexities and stages of development.

Table 5.1: Repository report for the projects under analysis.

Project	VS Code	Deno	Smart RR
Revisions	67,112	3,739	88
Contributors	1,166	376	1
Files	3,918	950	36
Files (TS/JS)	2,787	629	22
Lines of Code	992,759	100,541	3,067
Lines of Code (TS/JS)	600,095	59,291	2,291

5.1.1.3 Factors

The experiment consists on observing the effects of a factor in software's code quality. As previously stated, our factors are the refactoring suggestions provided by the tool. The insights and details on how these factors are generated are thoroughly explained in the Candidate Identification section (*cf.* Section 4.2, p. 44) of our proposed solution. As such, their expected properties and attributes will not be detailed in this section again. We point out, however, some of the key aspects - explained in the solution chapter - that we deem important for the context of this experiment and for reproducibility of its results, namely:

Determinism. Given a certain state of a method, our tool always produces the same suggestions.

Independence. Given a certain state of a method, the suggestions are independent, i.e., they always relate to distinct lines of code.

Chain dependency. After a modification is made to the code, the refactoring suggestions may change. This includes the application of a certain refactoring suggestion. Thus, despite the suggestions' independence at a given state, the order of how refactorings are applied may change the final results.

Score. Suggestions are measured according to the silhouette coefficient and, for this experiment, are applied in descending order. It is important to keep this feature in mind due to the aforementioned chain dependency between refactorings.

5.1.1.4 Attributes of Interest

The goal of this experiment is to study the effect of a given factor (refactoring suggestion) on software's code quality. Therefore, the attribute of interest of this study is code quality.

We need, however, to properly define what we mean by code quality. In the context of refactoring, code quality refers to the levels of maintainability, readability and flexibility of a software product. As such, our goal in this experiment is to observe how refactoring applications impact those attributes.

Another important aspect to take into consideration is how will code quality be measured: the levels of maintainability, readability and agility are in themselves intangible attributes, and need a proper way to be described and conveyed. As we've previously pointed out during the state-of-the-art analysis on Refactoring (*cf.* Section 2.2, p. 12), there are no formulas or set of rules that we can rely upon to validate the goodness of refactoring applications, which means that our measurements will inevitably use certain heuristics or assumptions. That considered, we've picked software metrics as our tool of measurement. Software metrics reason about the overall quality of the code, according to certain attributes. In our understanding, given enough data from multiple sources, the overall quality measured by metrics is a good indicative of the overall effectiveness of the refactoring tool. Additionally, metrics reports have straightforward implementations and can be easily reproduced in other experiments.

5.1.1.5 Measurements

In the previous section, we've indicated software metrics as our primary tool for measuring code quality. It now follows logically the need to pick and describe the appropriate attributes to be measured.

We're interested in metrics that are related to the Extract Method refactoring and will most likely suffer changes by applying our tool's suggestions. The chosen metrics are eight, and are defined and calculated (for each method) according to S. Fernandes et. al. [Fer19]:

Lines of Code. Total number of lines of code (excluding blank or commented lines).

Length. Count of operands and operators.

Vocabulary. Count of unique operands and unique operators.

Volume. Size of an algorithm implementation.

Effort. Effort to understand or modify the software.

Difficulty. Difficulty in understanding or modifying software (also known as error proneness).

Bugs Delivered. Estimation of the number of possible errors in an implementation.

Maintainability. Degree or index of maintainability of a software system (takes into account multiple metrics).

An ideal refactoring tool will decrease the values of all the described metrics, except for maintainability where an increase is desired. As such, we wish to observe the behavior of these values as refactorings are applied. In this experiment, these metrics are recorded after every refactoring application (for every method, in every file, in every revision).

5.1.1.6 Threats to Validity

In order to make sure that results are reliable and trustworthy, it is important to understand possible problems with the experiment's design. In this section, we identify threats to internal and external validity, and how these were addressed in order to minimize their impact.

Internal Validity. Internal validity is the extent to which we can establish a cause-and-effect relationship between the treatment and its outcome. Our treatment is only one: refactoring applications. However, some variables can be harder to control and affect the reliability of the results.

Number of refactoring applications. Our position on this experiment is that, given a considerable and diverse number of examples of refactoring applications and their outcomes, the conclusions drawn from the results are more robust and provide more confidence, compared to other alternatives. However, considering the novelty of this method, we can't estimate how many examples would be necessary in order to increase our trust. To address this issue, we've chosen software projects with different levels of maturity and requirements, which allowed us to study a substantial number of refactorings in different contexts. Nonetheless, the issue remains identified but ultimately unanswered.

Refactoring application order. As the robot scans a software project and analysis its methods, the application of refactoring suggestions is done by descending order according to their score. Given that, after a refactoring application, the suggestions provided may change, the order of these applications inevitably influences the results. Again, we believe this issue is solved by having enough examples of refactoring applications to draw conclusions from, so

that this influence can be mitigated. Additionally, one should be able to run the refactoring tool multiple times on the same method, with different orders/preferences - which, ultimately, increases the size of the dataset.

Software metrics. Metrics are used in order to evaluate the effect of a certain refactoring application on the software's code quality. Code quality is composed of three attributes - maintainability, readability and agility -, which are intangible. Therefore, the use of metrics as a measurement tool can only give us an approximation of those same attributes. Nevertheless, software metrics analysis is a broad and active area of research, with proven results, which can give us confidence in its reliability in the general assessment of code quality.

External Validity. External validity refers to how well the results and conclusions from this study are applicable to other scenarios. We identify two possible threats related to developers and software projects, respectively.

Developer behavior. The robot created for this experiment, as previously pointed out, simulates the usage of our refactoring tool as a regular software programmer. Still, this simulation is only an approximation of the behavior of an actual developer and, as such, it is possible that some refactoring occurrences are being missed with this method. In order to deal with this problem, we picked the approach that makes less assumptions about developers' behavior and software projects, so that results are easier to generalize and less prone to the influence of other external factors. We believe, nonetheless, that there are other alternatives to handle this issue, but require more investigation.

Project selection. Our objects of study are methods/functions from available software. We've selected projects with different levels of maturity, software requirements and overall complexity, in order to increase the diversity of refactoring applications in our analysis and, consequently, our confidence in the results. Notwithstanding, our observations may not apply to all available TypeScript/JavaScript projects. We point out, however, that this issue relates to this experiment due to time constraints, and not necessarily to a flaw in its design: in the best case scenario, we could analyze all the projects to which this refactoring tool may apply, which would dismiss any threat.

5.1.2 Running the Experiment

The experiment was performed using a MacBook Pro with 16GB RAM and Intel Core i7 processor. Results collection from all projects took one week, running approximately eight hours a day.

The robot created for this experiment was exposed as a VS Code extension (as is our refactoring tool), due to several reasons. First, developing the robot directly in VS Code reduces the difficulty in linking to certain necessary dependencies, such as the TypeScript server and VS Code's API, which are built-in in the source code editor. Secondly, it also allows a closer simulation of how

developers would use the tool, since we can mimic several actions - e.g. open a file, select lines of code, save changes, etc. Finally, the use of VS Code allowed us to visually analyze the progress and effectiveness of the refactoring suggestions, which in turn allowed possible errors and edge cases within the refactoring tool to be caught, analyzed and fixed earlier.

On the other hand, using VS Code brought its own issues, which should be considered on future experiments. For instance, whenever a refactoring application (that was deemed possible to apply) failed, VS Code offered no programmatic feedback whatsoever, which means that, from the robot's perspective, one could never be sure if a refactoring was successfully applied. Our workaround for this issue was to compare the entire text of the file before and after our refactoring attempt - if it changed, the refactoring suggestion was applied. Another issue we've found while running this experiment was that, sporadically, VS Code's TypeScript server would be shut down and never restart until VS Code itself was restarted (an error message could be seen inside the code editor), resulting in no refactoring applications since these are dependent on the server. Although instances of this issue were scarce, it would cause any possible refactoring applications to be missed in that period, while also creating an undesirable dependency on us, researchers, to manually restart VS Code anytime that happened. Finally, it's worth pointing out that many of the possible actions inside VS Code are asynchronous, which could cause some delays in collecting data - for instance, when a file was opened, or the revision was changed, VS Code could take some extra time to update its view, which in turn would also cause some delay in our robot (contrary to, say, a bash script that does the same thing).

In order to start the robot analysis, we open VS Code with a project, in a folder of interest (the one with the content we're interested in). After, we open the editor's Command Palette and run the appropriate command. We can then input the revision's hash from where the analysis will start (an empty string would mean from the beginning). After these steps the robot will process every method in every file in sequential order, changing to the next revision after completion, restarting the process. As a final note, it is also important to make sure that, before running the robot, the current project's revision is the one where we wish the analysis to be ended (usually the last revision).

After starting our experiment, we quickly realized that it wouldn't be possible to gather all available data from these projects, due to the computational time required. For instance, VS Code's project took approximately one day to get results from all files in a single commit (out of about sixty thousand commits). Thus, given the time constraints for this work, we were obliged to narrow down the number of refactoring applications we would be able to analyze. To do so, we reduced the number of commits to be analyzed, and restricted the files to a subfolder of the project. The folders were picked according to how often files inside that folder were modified throughout the project's history, using a git heatmap[4]. Table 5.2 (p. 71) shows the amount of information we were able to gather during this experiment.

We collected 19,294 refactoring applications, across 152 revisions and 137 files. Notice that the number of analyzed instances were significantly reduced, compared to Table 5.1 (p. 66), but

[4]Git Heatmap: https://github.com/jez/git-heatmap. Last accessed on 14 June, 2020.

Table 5.2: Number of analyzed revisions, files and refactorings.

Project	VS Code	Deno	Smart RR	Total
Revisions	60	50	42	**152**
Files	102	19	16	**137**
Refactorings	16,127	2,286	881	**19,294**

we were still able to gather a considerable amount of refactoring applications. This shows the true potential of the method used in this experiment which, with less restrictive time constraints and a few improvements, can provide us with large amounts of data to analyze and draw conclusions from.

We can also see that the number of refactorings per project are unbalanced. This phenomenon inevitably occurs since we specifically picked projects with different complexity and maturity. These issues fall in the context of data understanding and data preparation, which are out of the scope of this dissertation. We encourage, however, future investigations (focused in this type of experiments) to better address and tackle these occurrences.

5.1.3 Results

After running the experiment, we obtained, for each project, the following assets:

- A CSV file with a list of all refactorings applied in that project. Each entry includes the revision, file and the name of the method under analysis. It also includes three additional attributes: (i) *accepted clusters*, which is the number of successfully applied refactorings in that method; (ii) *processed clusters*, which is number of refactoring attempts, including the ones successfully applied - thus, the number of processed clusters is always greater or equal to the number of accepted clusters; and (iii) *score* which refers to the value of the silhouette coefficient of the extracted cluster. Figure 5.1 shows an example of such file for a given project.

```
revision,file,method,accepted_clusters,processed_clusters,score
8f35cc4768,./browser/pluginHostStatusBar.ts,hide,1,1,0
8f35cc4768,./browser/pluginHostStatusBar.ts,update,1,1,1
8f35cc4768,./browser/pluginHostStatusBar.ts,update,2,3,0
8f35cc4768,./browser/pluginHostStatusBar.ts,setStatusBarMessage,1,1,0.4
8f35cc4768,./browser/pluginHostStatusBar.ts,setStatusBarMessage,2,2,0
8f35cc4768,./browser/pluginHostStatusBar.ts,setEntry,1,1,0
8f35cc4768,./browser/extHostOutputService.ts,close,1,2,0
8f35cc4768,./browser/pluginHostQuickOpen.ts,show,1,1,0.5616666666666668
8f35cc4768,./browser/pluginHostQuickOpen.ts,show,2,2,0
8f35cc4768,./browser/pluginHostQuickOpen.ts,show,3,3,0
8f35cc4768,./browser/pluginHostQuickOpen.ts,_show,1,2,0
8f35cc4768,./browser/pluginHostQuickOpen.ts,_show,2,3,0.8166666666666667
8f35cc4768,./browser/pluginHost.api.impl.ts,_setLanguageConfiguration,1,1,0
8f35cc4768,./browser/pluginHost.api.impl.ts,Modes_CommentsSupport_register,1,1,0
```

Figure 5.1: Example of file with list of applied refactorings.

- A folder with a list of CSV files describing software metrics. The metrics were collected on robot's demand, i.e., we could trigger the analysis only when required. We chose to gather the values after each refactoring application.

  In order to correctly map a metrics file to appropriate entry in the refactorings file, one has to look at the name of the metrics file, which follow the convention *<revision>-<file>-<iteration>*. For instance, the file `0a2f0cbc5c-languageFeatures.ts-5.csv` refers to revision `0a2f0cbc5c`, file `languageFeatures.ts` and 5th refactoring application done in that file. Using this information we can link it to the refactoring file, where we can also look at the name of the method for more detailed information.

  Each metrics file contains values for all methods of the file under analysis (so it does file-level analysis, and not method-level). Each entry of the metrics file contains the name of the method, and values for multiple metrics. These metrics were collected using an external tool, so it has more metrics than the ones used - detailed in Section 5.1.1.5 (p. 67). An example of such file with the collected metrics can be seen in Figure 5.2.

```
method,numLines,numCodeLines,numBlankLines,numCommentLines,numConditionals,numLoops,cyclomatic,length,vocabulary,volume,effort,
constructor,11,11,0,0,0,0,0,29,16,116.0,223.7142857142857,1.9285714285714288,0.5185185185185185,12.428571428571427,0.0122840400
createDebugAdapter,9,9,0,0,0,0,0,22,14,83.76180828526729,217.01923055728344,2.590909090909091,0.3859649122807018,12.05662391984
daExecutableFromPackage,7,7,0,0,0,0,0,24,17,98.09910819000814,143.8786920120119,1.4666666666666666,0.6818181818181819,7.9932606
createSignService,3,3,0,0,0,0,0,5,5,11.609640474443681,11.609640474443681,1.0,1.0,0.6449800263576005,0.0017090624242341599,140.45
$runInTerminal,51,36,12,3,0,0,0,115,53,658.7108522747677,7711.736807119233,11.707317073170733,0.08541666666666667,428.429822617
createVariableResolver,3,3,0,0,0,0,0,18,14,68.53238859703687,44.80963869806257,0.6538461538461539,1.5294117647058822,2.48942437
constructor,11,11,0,0,0,0,0,29,16,116.0,223.7142857142857,1.9285714285714288,0.5185185185185185,12.428571428571427,0.0122840400
createDebugAdapter,9,9,0,0,0,0,0,22,14,83.76180828526729,217.01923055728344,2.590909090909091,0.3859649122807018,12.05662391984
daExecutableFromPackage,7,7,0,0,0,0,0,22,15,85.95159310338741,204.13503362054507,2.375,0.42105263157894735,11.340835201141395,0
newMethod,3,3,0,0,0,0,0,12,9,38.039100017307746,48.907414307796711,1.2857142857142858,0.7777777777777777,2.7170785726648394,0.00
createSignService,3,3,0,0,0,0,0,5,5,11.609640474443681,11.609640474443681,1.0,1.0,0.6449800263576005,0.0017090624242341599,140.45
$runInTerminal,51,36,12,3,0,0,0,115,53,658.7108522747677,7711.736807119233,11.707317073170733,0.08541666666666667,428.429822617
createVariableResolver,3,3,0,0,0,0,0,18,14,68.53238859703687,44.80963869806257,0.6538461538461539,1.5294117647058822,2.48942437
constructor,11,11,0,0,0,0,0,29,16,116.0,223.7142857142857,1.9285714285714288,0.5185185185185185,12.428571428571427,0.0122840400
createDebugAdapter,9,9,0,0,0,0,0,22,14,83.76180828526729,217.01923055728344,2.590909090909091,0.3859649122807018,12.05662391984
daExecutableFromPackage,7,7,0,0,0,0,0,22,15,85.95159310338741,204.13503362054507,2.375,0.42105263157894735,11.340835201141395,0
newMethod,3,3,0,0,0,0,0,12,9,38.039100017307746,48.907414307796711,1.2857142857142858,0.7777777777777777,2.7170785726648394,0.00
createSignService,3,3,0,0,0,0,0,5,5,11.609640474443681,11.609640474443681,1.0,1.0,0.6449800263576005,0.0017090624242341599,140.45
$runInTerminal,44,30,12,2,0,0,0,96,49,539.0121450350599,4906.392602242213,9.1025641025641,0.10985915492957747,272.5773667912341
newMethod_1,10,9,0,1,0,0,0,23,13,85.11011351724513,401.23339229558417,4.714285714285714,0.2121212121212213,22.290744016421343,
createVariableResolver,3,3,0,0,0,0,0,18,14,68.53238859703687,44.80963869806257,0.6538461538461539,1.5294117647058822,2.48942437
```

Figure 5.2: Example of collected metrics for a given file.

5.1.3.1 Data Processing

We next processed the aforementioned gathered information to retrieve the necessary insights to fulfill the goal of this experiment. Our main tools for this analysis were Pandas[5] and Matplotlib[6], in the Python language.

The goal of this experiment is to study how code quality is impacted by the use of our refactoring tool. As such, we observed how software metrics behaved with the application of multiple, successive refactoring suggestions - in other words, what was the tendency of each metric. To achieve this, we mapped every refactoring occurrence to the corresponding metrics file, while maintaining its order of application, i.e., for each revision and file, the order in which refactorings were applied to a single method was respected. After that, we were able to calculate and plot several charts showing the behavior of these metrics.

[5]Pandas: https://pandas.pydata.org/. Last access on 15 June, 2020.
[6]Matplotlib: https://matplotlib.org/. Last access on 15 June, 2020.

5.1.3.2 Analysis by revision and file

We start our analysis by studying the effects of the refactoring applications on a specific file and revision. Figures 5.3, 5.4, 5.5 show multiple boxplot charts describing our results, from randomly picked revisions and files, one for each project. Note that, although the figures were randomly selected, the analysis was made for multiple revisions and files on each project - we believe, however, that they are representative of the general behavior and tendency we've found in all other scenarios.

Each boxplot chart represents the tendency of each one of metrics measured in this experiment. The name of the metric is placed above the respective chart, along with an arrow that represents the desired tendency. In the *y*-axis, we have a linear scale that represents the absolute value of the recorded metric - keep in mind, however, the range of values changes between charts. The *x*-axis is a discrete scale in which each number represents a refactoring iteration, i.e., a refactoring suggestion applied by our robot. The iteration 0 represents the original file, without any modifications. The green triangle in each of the boxplots is the average value for that metric (considering all methods in the file) at a given iteration. The red line spans across all these averages for visual enhancement of the results.

The charts indicate that, in general, the application of refactoring suggestions provided by our tool have the desired effect in the metrics: the number of code lines, length, vocabulary, volume, effort, difficulty and bug delivery tend to decrease, while maintainability usually increases. We can also observe that the effectiveness of the refactoring application is independent of its iteration, i.e., there are no noticeable patterns or changes in behavior as more refactorings are applied.

Despite observing the desirable tendency in most refactoring applications, we can also see some examples where that isn't true. For instance, the length, volume and difficulty in Figure 5.5 (p. 75) show some fluctuations, which indicates that there is also room for improvement and more work to be done.

Finally, it is also possible to notice some effects on the charts' outliers. Some of them are properly handled by the tool, making such outlier disappear. Others, however, remain in the chart after all refactoring iterations or might appear after certain refactoring applications. This shows that, assuming that metrics are always a direct representation of code quality, our tool might not always be effective at handling the most prominent refactoring needs. Thus, we're not claiming our approach to the problem is a silver-bullet.

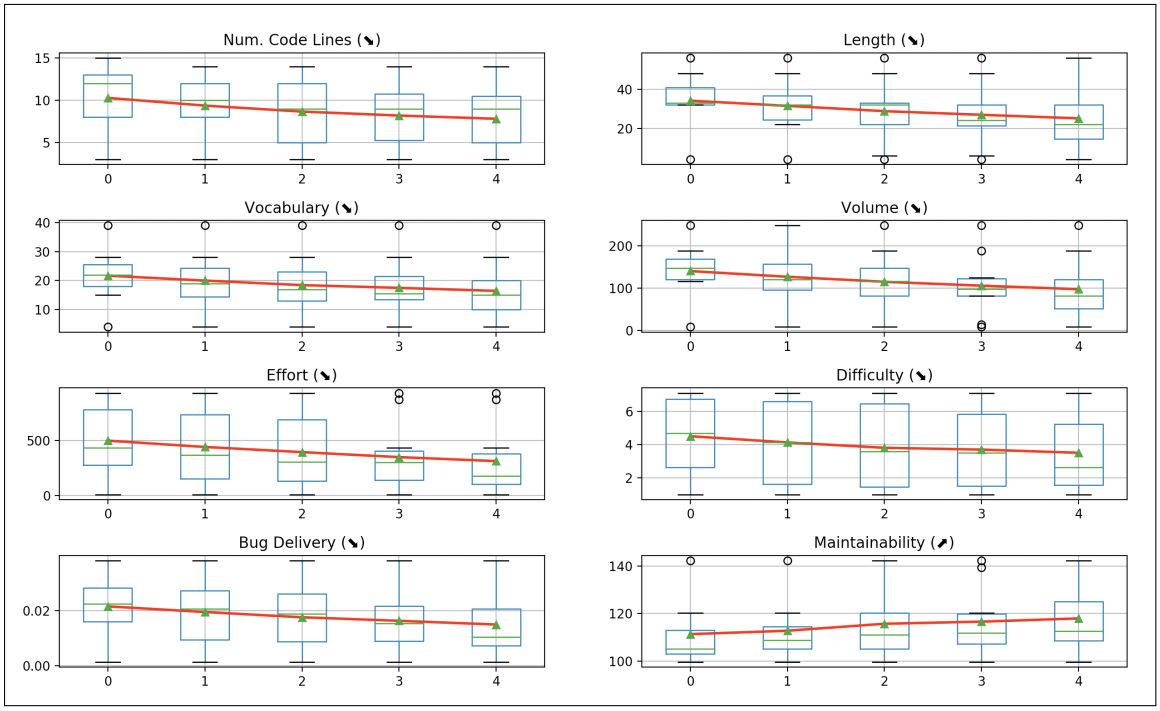


Figure 5.3: Metrics in a single revision and file for VS Code.

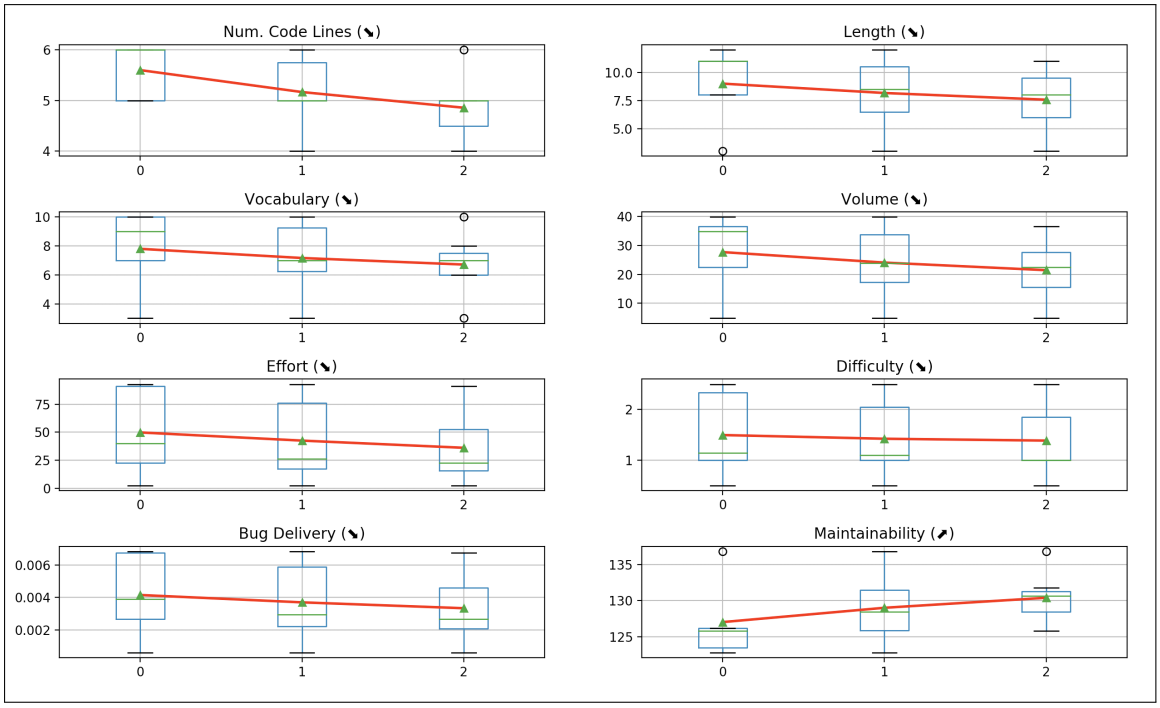


Figure 5.4: Metrics in a single revision and file for Deno.

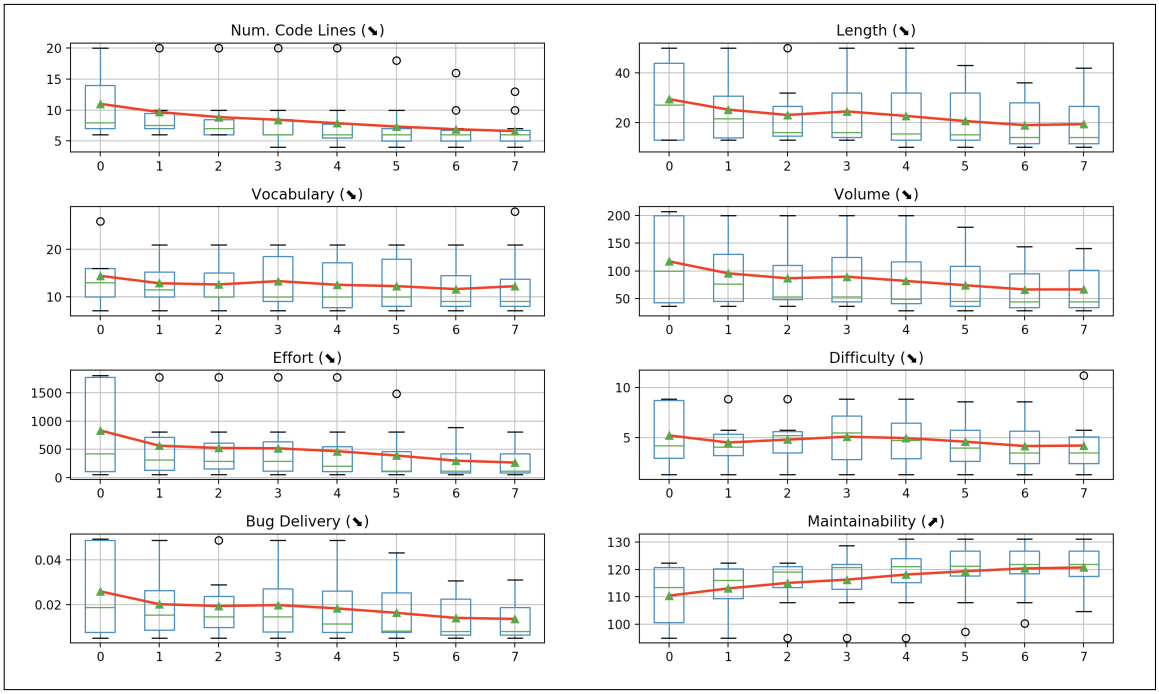


Figure 5.5: Metrics in a single revision and file for Smart Refactoring Recommender.

5.1.3.3 Analysis by file

Our next step is to study the effects of refactoring applications on a specific file, across multiple, consecutive commits. Our goal here is to analyze how possible changes to a file (more distant in time, perhaps by different developers) could impact the tool's performance. The randomly selected instances can be seen in Figures 5.6, 5.7 and 5.8.

Due to the dimensions of the chart, only two metrics are shown in the figures, namely: number of code lines and maintainability. We chose these two metrics because they're among the most important in the context of the Extract Method refactoring and, additionally, the desirable tendency for each is different - descending and ascending, respectively. The value of the metrics here shown always correspond to the average value of all methods in the file.

Each figure is composed of two charts. The top chart shows the average number of code lines, while the bottom chart shows the average maintainability. The y-axis is a (non-zero) linear scale representing the value of the metric. The x-axis is a discrete scale where each value represents a revision (in the same order as in the project, from left to right). Each revision is also distinguished by a different color.

As observed in the previous analysis, by revision and file, the values of the metrics usually follow the preferable tendency. When comparing any two consecutive revisions, one of the two applies: the file suffered changes or it didn't. We can see if a file was changed if either the values or the behavior of the metrics differ between one another. Whatever the case, we can see that the refactoring tool still provides useful suggestions, which are capable of increasing the software's code quality.

We can also see that some refactoring applications (specially the first in each commit) are usually the same, which means that the lines of code in question still remain in the codebase after several revisions. In these cases, a live refactoring tool would allow the developer to perceive the quality defect in earlier stages of development, possibly mitigating future defects that could potentially have arisen from that occurrence - such as, for example, duplicated code.

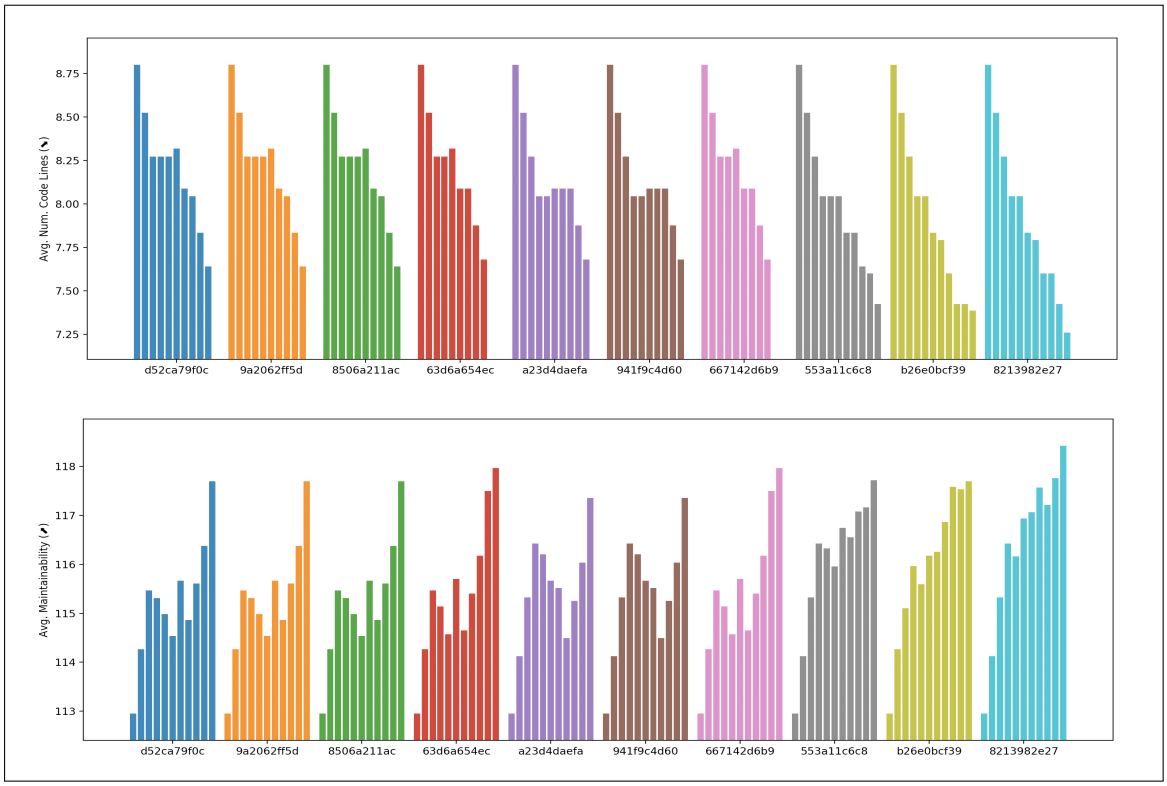


Figure 5.6: Metrics in a single file across multiple revisions for VS Code.

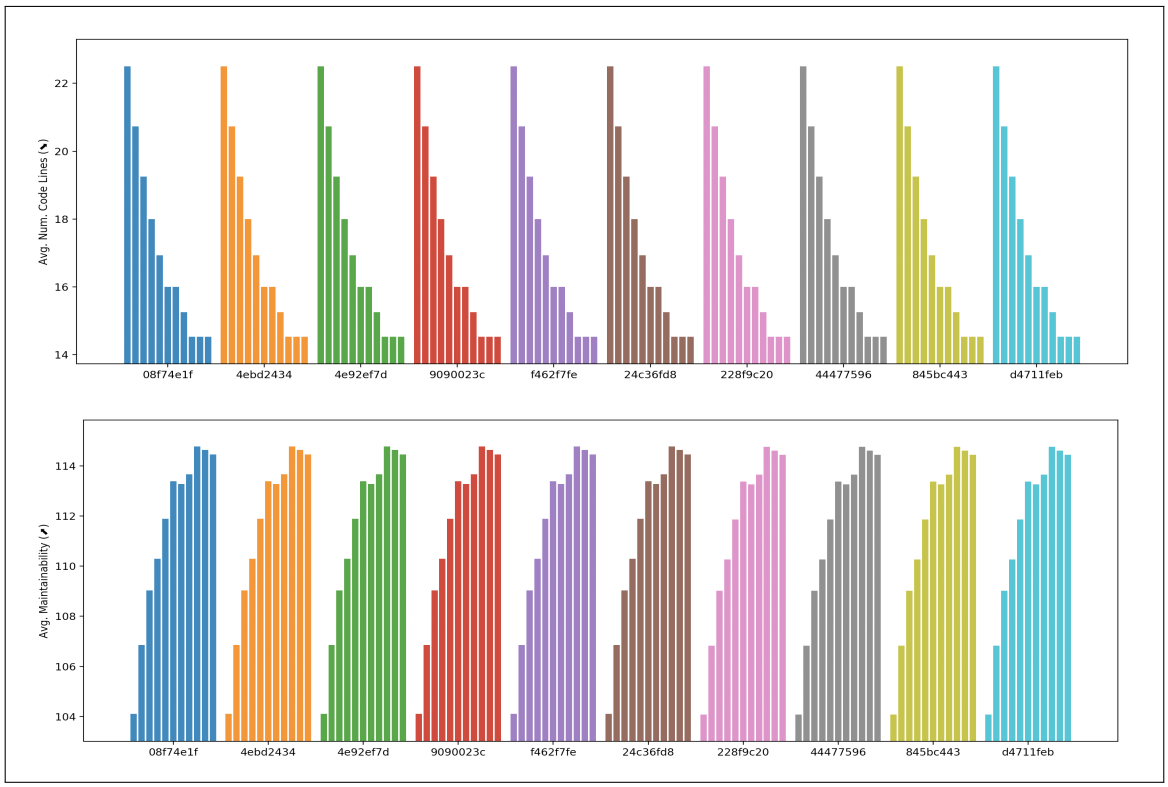


Figure 5.7: Metrics in a single file across multiple revisions for Deno.

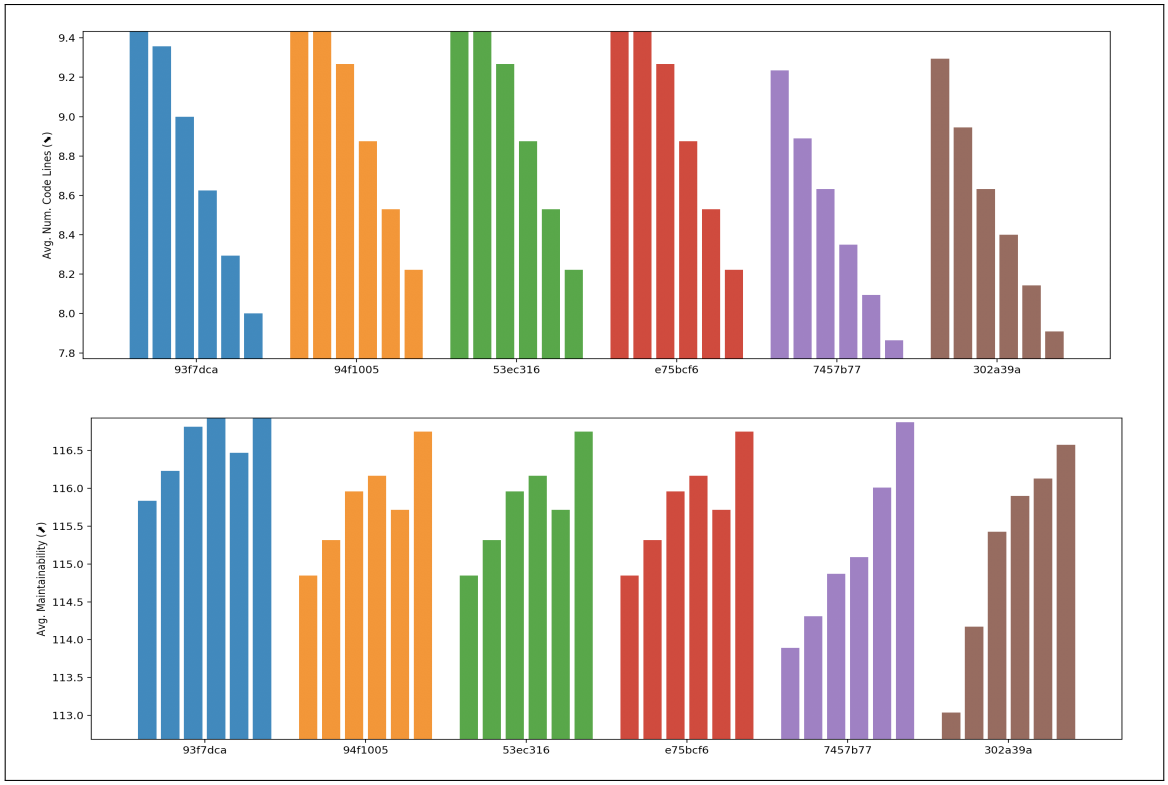


Figure 5.8: Metrics in a single file across multiple revisions for Smart Refactoring Recommender.

5.1.3.4 Analysis by revision

In our final stage of this experiment, regarding the study of code quality, we observed the effects of our refactoring tool in a dimension orthogonal to the previous analysis by file. In other words, our observations are centered in a single revision, across multiple files. This allows us to have a broader inspection of the behavior of the tool in objects with different purposes and, consequently, different refactoring needs. Figures 5.9, 5.10 and 5.11 are representative of the overall findings in the three projects. As in the previous analysis, we only show metrics for code lines and maintainability to reduce the charts' size.

Three-dimensional charts were used for more graphical comprehensibility. The top chart shows the average number of code lines,while the bottom chart shows the average maintainability. The x-axis is a discrete scale representing each of the analyzed files. The y-axis is also discrete, showing the refactoring iteration. Finally, z-axis shows the average value of all methods in the file, for the metric in focus.

Files analyzed and displayed in the chart are also represented by a different color, having different shapes. Each of these shapes contains, either on top or bottom, a dashed line and a solid line that show the file's original metrics and their values after applying refactorings, respectively. Solid lines also contain multiple red dots, which represent the value of the metrics after a certain refactoring iteration. Thus, the vertical distance from a red dot to the dashed line constitutes how much the metric changed, in that respective iteration.

Similar to the two previous studies, metrics generally change according to the desirable tendency: the average number of code lines decreases and the maintainability index increases, as refactorings are applied. We conclude, therefore, that our refactoring tool is robust to multiple use cases.

It is also possible to observe that, in general, our tool is able to improve metrics independently from their starting point (original values), which tells us that our algorithm is able to adapt to different situations. On the other hand, we can also see that there is no direct relationship between the original metrics and the amount they change, i.e., files with numerically worst initial metrics might not always improve as much as other files with (apparently) less issues to tackle.

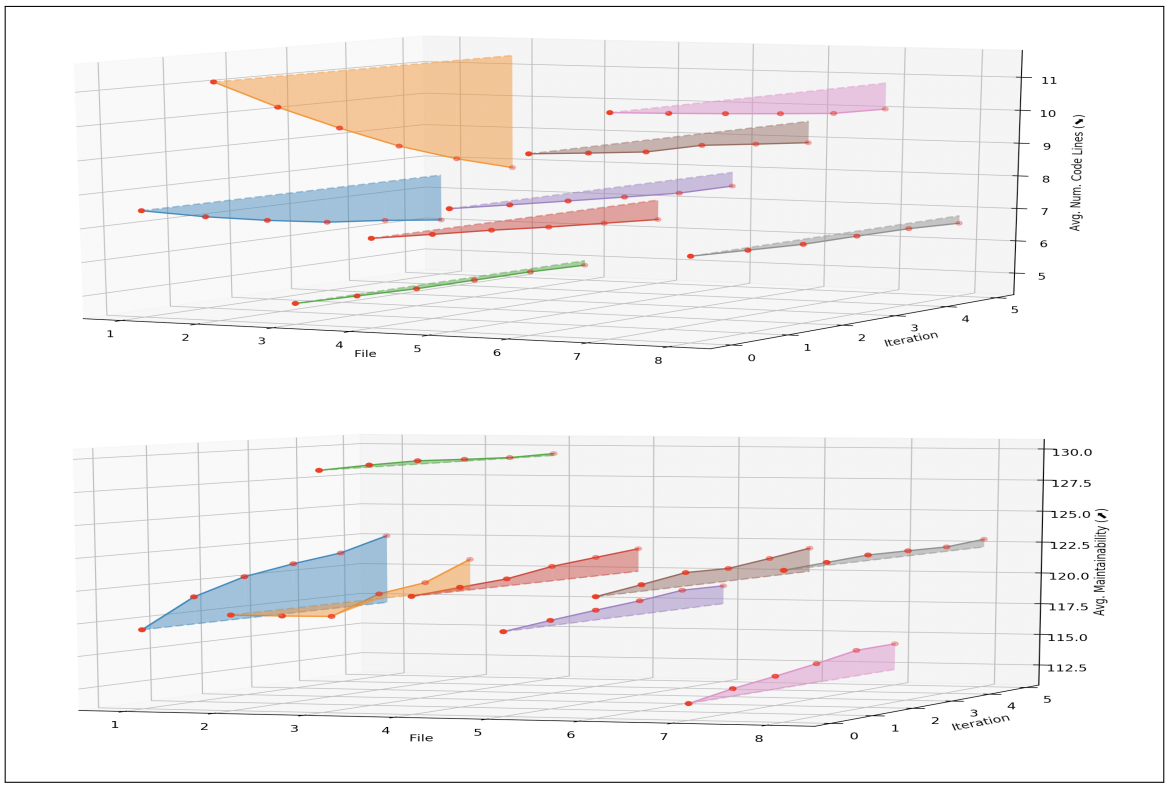


Figure 5.9: Metrics in a single revision across multiple files for VS Code.

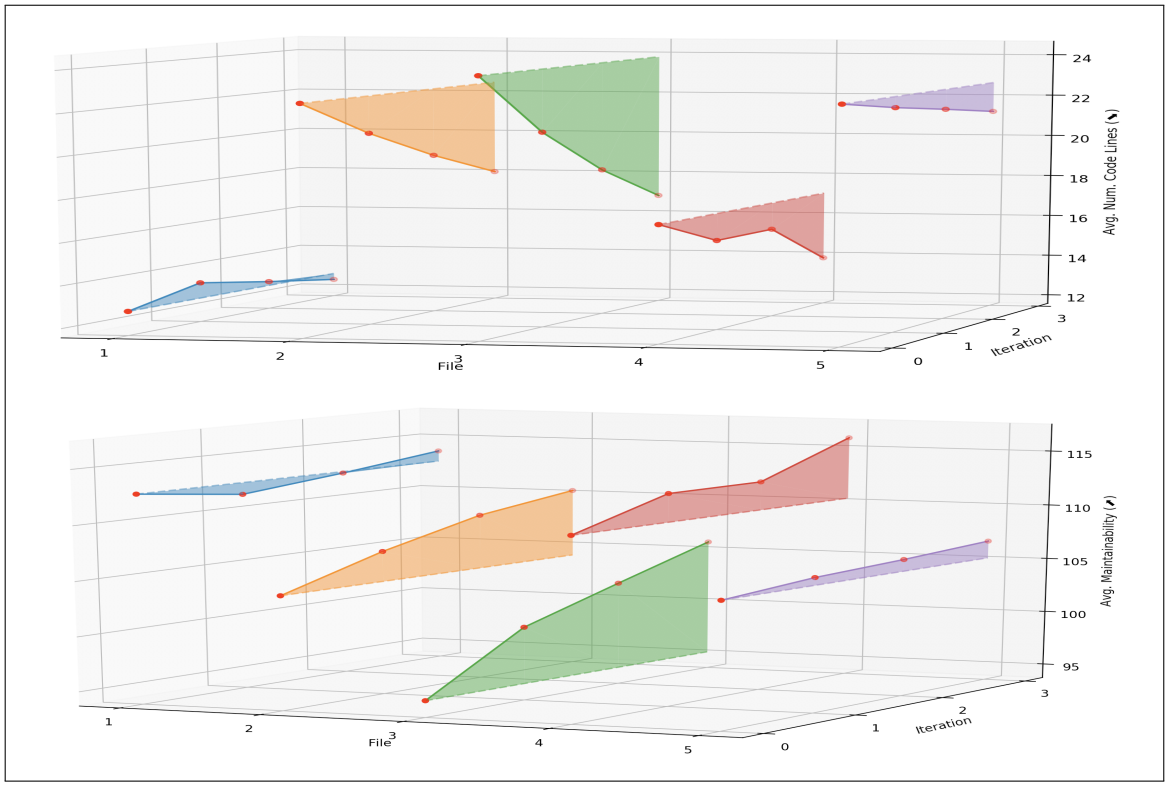


Figure 5.10: Metrics in a single revision across multiple files for Deno.

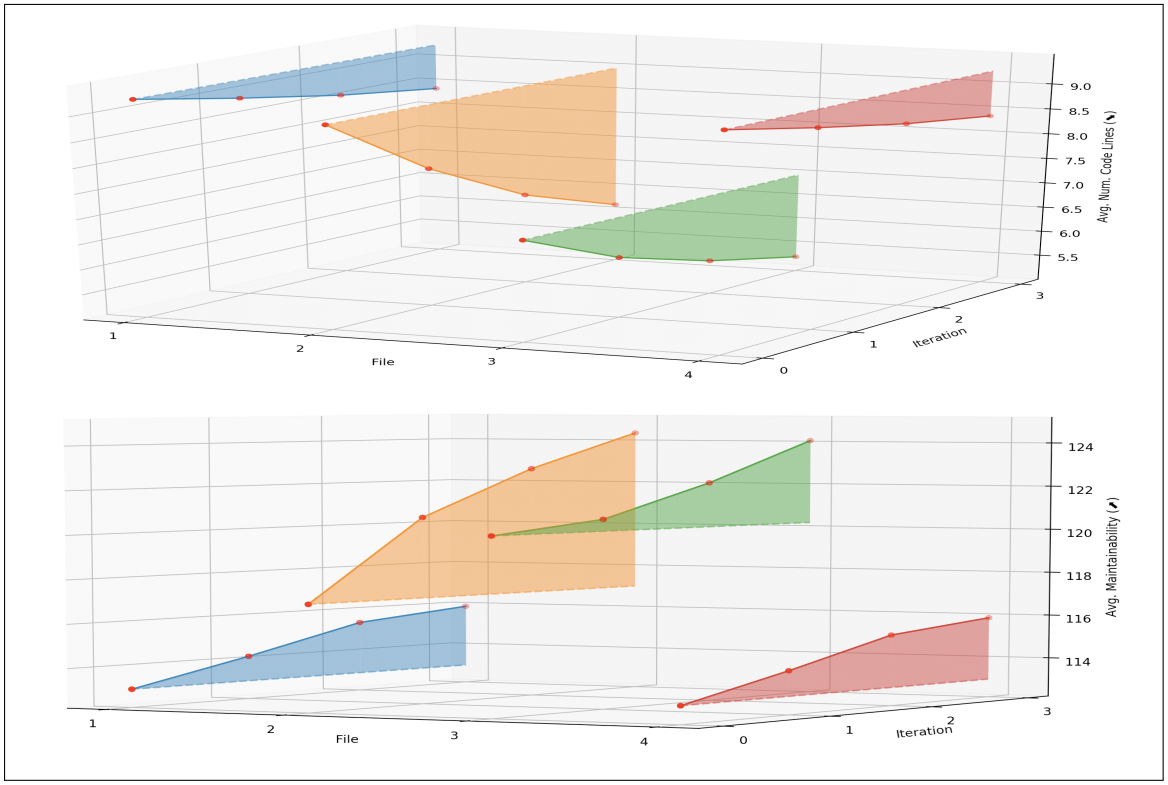


Figure 5.11: Metrics in a single revision across multiple files for Smart Refactoring Recommender.

5.1.3.5 Other findings

This section contains some additional results and observations that were gathered during this analysis and serve as complementary information that have a more general perspective on our findings.

Refactoring Impact. Our robot recorded metrics after each refactoring iteration. From this data, we calculated the difference between metrics before and after a refactoring application, which allows us to have an overview of how much metrics change at each step. Table 5.3 (p. 81) shows the statistics gathered from this analysis for each metric used, which include the average, standard deviation and quartiles.

As it was seen in the previous sections, on average, metrics tend to follow the desired tendency. We can also observe that the standard deviation has higher absolute values than the mean, which in general indicates that these changes are numerically spread out and might even have a negative impact on code quality - as previously observed. We believe, however, that standard deviation is also highly inflated due to the outliers present in these statistics. This is further corroborated by considering that the 75% quartile either shows improvements or values very close to zero - which means that close to 75% of our refactoring applications bring a positive impact -, and by the fact that the max values are considerably lower than the absolute min values in every metric.

The reader should take notice that the maintainability metric has a different desirable tendency from the others and, as such, the analysis should be inverted, i.e., the 25% quartile shows a zero value, and the max value is significantly higher than the min value.

Table 5.3: Statistics regarding difference in metrics after a refactoring is applied.

Statistics	mean	std	min	25%	50%	75%	max
Num. Code Lines	-0.328	0.776	-15.5	-0.382	-0.126	-0.022	7.067
Length	-1.149	3.157	-59.0	-1.409	-0.512	0.087	25.633
Vocabulary	-0.428	1.133	-9.2	-0.607	-0.231	0.094	13.9
Volume	-6.597	17.106	-283.048	-7.413	-2.8	0.114	132.247
Effort	-100.244	387.712	-7764.892	-62.127	-24.166	-1.677	1169.483
Difficulty	-0.098	0.42	-10.573	-0.121	-0.037	0.025	2.873
Bugs Delivered	-0.001	0.004	-0.099	-0.001	-0.0	0.0	0.023
Maintainability	0.658	1.291	-3.903	0.0	0.285	0.811	22.52

Automatic Extraction Success Rate. As we've stated in Section 5.1.2 (p. 69), for each refactoring application, we recorded the number of accepted clusters and the number of processed clusters for each method. The number of accepted clusters refers to how many refactorings were successfully applied, while the number of processed clusters refers to the number of refactoring attempts (including the accepted ones). For each entry in the csv file with the recorded refactorings, if these values differ, it means that we were not able to apply the refactoring suggestion with the highest score and had to select the next alternative (in sucession, until a valid suggestion was found).

In our solution proposal chapter (*cf.* Section 4, p. 43), we stated that suggestions with overlapping boundaries were considered not extractable by our tool. However, there are other cases where a refactoring might not be applicable even if there is no overlapping - e.g. having a return statement inside an if statement. Thus, in order to understand how often those cases occur in our suggestions, we've gathered the aforementioned accepted and processed clusters.

We can calculate the success rate simply by subtracting the number of processed clusters to the number of accepted clusters and divide it by number of applications. The value we've obtained is of **0.45**. This means that close to 50% of the suggestions we deemed as extractable were not successfully applied, and shows that there are many more edge cases to take into consideration when evaluating the suggestion's extractability. As such, we consider this issue to be one important future improvement to our refactoring recommender.

5.1.4 Discussion

We start this discussion by summarizing the various aspects that were addressed throughout this whole Section 5.1 (p. 61). The goal of this first experiment was to evaluate whether liveness is a good feature to have in a refactoring tool and how it affects software in terms of maintainability and agility. In the context of this experiment, it is not our intention to understand the impact of

continuously displaying refactoring suggestions to developers, but rather how using those same suggestions, earlier and more often, can help improve code quality. As such, our observations are focused in studying the impact of refactoring applications.

Considering this perspective, we decided to build a robot in VS Code that simulates a developer using the refactoring tool. Our reasoning for this experiment design is that we can increase our confidence and trust on the results by analyzing refactoring applications from multiple projects (with different ages, requirements and goals), if enough examples can be extracted - when compared, for instance, to an experiment where developers are asked to use the tool and give their opinion on the viability of the suggestions. Note that, since our main assumption in this dissertation is that developers must be the final judges of whether a refactoring suggestion should be applied (*cf.* Section 3.2, p. 36), it would certainly be preferable to have their feedback rather than simulating it. However, considering the need for a significant number of developers to participate to produce valuable results, and also the time constraints for this work, the amount of information gathered would necessarily be considerably less. Additionally, it would also require, as done in this experiment, opinions regarding different projects and situations, which in turn would require participants to spend time in comprehending the context and requirements of these scenarios. Thus, our approach - which is, to our knowledge, innovative in the field - provides a simpler and more efficient method to collect more trustworthy results.

As we've discussed throughout the design experiment (*cf.* Section 5.1.1, p. 62), approximations had to be considered in order to produce a robot that simulates a developer using our refactoring tool, namely:

Developer Behavior. In order to understand how our refactoring tool impacts code quality, when used by developers in their regular activity, it would be desirable to know how often developers apply refactorings, and also in which situations (e.g. before developing any new features, after guaranteeing tests are passing, etc). This would require having this information beforehand, which doesn't seem feasible at the moment because, to our knowledge, there are no available datasets with reproducible programming activities (e.g. every action in a certain commit) and also, during those activities, knowing without a doubt that a refactoring was applied.

Thus, our robotized experiment relies on an approximation of this behavior. Since we only have access to the state of source code at the end of each commit, we take this sequence of commits as the different actions in the programming activity carried out by developers, in which refactorings are applied in between these actions. We picked this approximation because it works exclusively with what we have access to, and makes no assumptions whatsoever on how developers would use this tool, making our experiment more reproducible and fair.

Metrics as Code Quality. Our approach to assess the impact of refactoring applications in code quality was to collect values for code metrics that are relevant to the Extract Method. As we've discussed during the state-of-the-art analysis (*cf.* Section 2.2, p. 12), software metrics are

a useful and easy way to measure code quality at a surface level. With a considerable amount of refactoring applications, we are also able to assess the tool's general impact.

Considering all assumptions and approximations, we find that, in general, our refactoring suggestions are able to bring a positive impact on code quality. Thus, we believe that the results positively support the hypothesis that software's code quality is improved by using tools that promote refactoring by continuously showing opportunities to developers, i.e. with higher levels of liveness.

We found that the evolution of metrics tend to follow the desirable trend in a strict fashion, i.e., successive refactorings are able to improve quality independently of how many were applied before. We observed the same behavior when our analysis focused in a single file across multiple revisions, and also in a single revision across multiple files, which indicates that our tool is able to perform well in different contexts (files) and scenarios with consecutive changes (revisions).

It was also found that our tool might not always be able to mitigate the most prominent refactoring needs, as there were some instances in which methods with worst metrics remained with the same values after their analysis was completed. This can additionally be shown by noticing that there is no strict correlation between the metrics' initial values and the impact of the refactoring - some files with initial worst average metrics values suffered less improvements than others where those values were healthier. In general, we observed this phenomenon due to two different reasons: (i) our tool could not find an appropriate refactoring that would in fact address the most noticeable issues, and (ii) some of the high-score suggestions couldn't be automatically extracted and would require human intervention to be fulfilled.

5.2 Survey on Live Refactoring

In the previous experiment, we focused on evaluating the impact of the tool's refactoring suggestions on the code quality of software, leaving aside the developer's role in the refactoring process. Our goal was to understand if using liveness in refactoring tools, i.e. providing short feedback loops for refactoring suggestions, would be a valuable asset for developers in improving software's maintainability.

To fulfill the goals set for the experiment, we asked developers to answer a survey regarding our refactoring tool, in which the main focus of the analysis is their opinion on the usability and user experience provided by the recommender. Additionally, we will use the answers to better understand how live tools impact the tool's presence, relevance and understandability, as explained throughout Chapter 3 (p. 35). Thus, the main research questions in focus are **RQ2** (and its sub-questions) and **RQ3** (*cf.* Section 3.4, p. 38), since they directly relate to development experience.

5.2.1 Experiment Design

Survey research can be defined as a method for collecting data about characteristics, actions or opinions from a particular group of persons, often by asking participants to answer a questionnaire.

One of the situations in which this method is appropriate to use is when there is no interest or possibility of completely controlling the independent or dependent variables [FOSM00]. Since this experiment focuses on development experience, which is subjective and intangible, we believe that survey research is a fitting method for the problem at hand. Throughout the following sections we expand on this idea, while also detailing the questionnaire's structure and the purpose of the questions it contains.

5.2.1.1 Reasoning

We now aim to assess the usability and user experience of a live tool by placing the developer in the central role of the refactoring process. In other words, our evaluation focuses on whether developers see value in having a live refactoring tool, which provides continuous feedback, in their regular programming activities. Contrary to the previous controlled experiment, we are now exclusively interested in understanding how liveness helps developers, instead of its impact on code quality.

Thus, we believe that the answers to the questions in focus must be necessarily provided by developers. As such, we built a questionnaire (*cf.* Appendix C, p. 125) where participants were asked to observe multiple static and animated images, and give their opinion to questions about what they've seen. The questions were thought of and constructed with the final goal of providing an answer to our research questions.

The participants' responses are mainly in the form of a linear scale, from 1 to 5, corresponding to the categories *"Highly disagree"*, *"Disagree"*, *No opinion"*, *"Agree"* and *"Highly Agree"*. There are also some questions in the form of single or multiple choice.

5.2.1.2 Subjects

Our study is not focused on any particular group, with specific software skills or background, since our live tool is not targeting any distinct type of developers. As such, participants eligible to engage in this experiment could be either students or developers in the professional environment. Knowledge about refactoring was not necessary, neither experience in using VS Code or JavaScript/TypeScript. However, having some programming experience should be mandatory in order to properly answer the questionnaire.

It is important, nonetheless, to understand the distribution of the participants to avoid generalizing the results to populations that were not properly represented in the survey, due to lack of answers in that profile. Therefore, to avoid possible biases, all participants were asked to describe their technical background, namely:

- Programming experience;
- Familiarity with programming terms (classes, methods, etc);
- Familiarity with software development terms (refactoring, debugging, etc);

- Experience in JavaScript or TypeScript;
- Experience with Visual Studio Code;
- Experience with software metric analysis tools.

Additionally, participants were also able to share their age and education level in the answers. Answers regarding this information were entirely optional.

5.2.1.3 Attributes of interest

Our goal in this experiment is to assess how developers feel about using a live refactoring tool, considering its usability and usefulness in the refactoring process. In other words, how it impacts **development experience**.

Note, however, that development experience, in itself, has no particular method or set of measurements with which we can undoubtedly and objectively describe it. As such, our assessment is ultimately provided by the developer's overall opinion on the tool and its essential properties. Therefore, our main attribute of interest is the developer's point of view on whether the tool is able to provide helpful insights and worth using.

Nonetheless, as explained in Section 3.4 (p. 38), we also wish to understand how liveness impacts some specific attributes found in live refactoring tools, namely:

Presence. Are developers aware of the tool and its suggestions? Are they able to quickly identify them?

Relevance. Do developers know in which situations the tool can be used?

Understandability. Are developers able to understand the tool's suggestions?

5.2.1.4 Tasks

Here we describe the multiple sections that participants were asked to answer in this survey, and how they relate to the research questions. As previously stated, sections regarding the participant's profile and technical background aim to provide a better understanding on their distribution. The section of the questionnaire in focus on this section is concerned with opinions from developers regarding the refactoring tool. It can be found in Appendix C (p. 125) with the title *"Part 1: Extract Method Finder"*.

The section is divided in four major components, which may be further subdivided, as follows:

1. **Visualization**

   In this first part, participants were asked to answer questions related to visual aspects of the refactoring tool. In the beginning, participants are able to read that our tool is constantly analyzing source code and making live suggestions as it is being modified. It is also explained how colors are assigned to the suggestions, according to their severity. Then, we divide the section in three main aspects:

(a) **Continuous Feedback**

Participants are presented with an animated image where a class method is being built from scratch, and are able to see our tool's suggestions changing as the code is modified. Our main focus is to understand if developers are able to locate the suggestions, how colors affect their awareness and the effect of continuous feedback. Thus, it mainly relates to the tool's *presence*.

(b) **Multiple vs. Single Suggestion**

Two pictures are shown, one in which a single suggestion is provided, and another where multiple options are available. Participants are then asked about their opinion on the benefits of having multiple suggestions. We are also interested in measuring the *understandability* of the tool.

(c) **Refactoring Availability**

This subsection shows an animated image where the editor's cursor traverses different data structures. Our tool only shows feedback in a subset of them (functions and class methods). We aim to understand if developers are able to quickly understand in which data structures the tool is available. Thus, this part is mainly related to the tool's *relevance*.

2. **Selection and Application**

In this section, we describe how suggestions can be selected and then applied using VS Code's features. The information available for each of the suggestions, the code highlighting feature and the option to change the method's name after its extraction are also here explained. After, the participant can see an animated image where all these features are displayed. Our main goal is to understand if developers see value in having a semi-automated refactoring recommender available in their coding environment, and what features are most beneficial.

3. **Code Quality History**

In our tool, we've provided an additional command that allows users to see how code quality evolved in a specific file, through a webview displaying a multi-line chart. This section starts by giving a brief description about the chart and its metrics. Then, an animated image shows source code being modified and its effects on the file's code quality history. We aim to understand if developers see value in having an historical record about the software's evolution regarding code quality, and also if liveness is a good asset for these scenarios.

4. **Final Remarks**

This section asks the participant about their overall opinion on the tool and which features they considered the most (or less) valuable. The questions are more general and take mainly into consideration development experience.

Notice that all previous sections also consider development experience, but are more focused in specific aspects of the tool and, as such, they can't be used on their own to assess the tool's usability - for instance, the participants may have appreciated the visual aspect of the tool, but still would not consider using it as part of their regular programming activity (or vice-versa). Nonetheless, all results are important to understand how developers see benefit in the tool and which possible future improvements are necessary.

5.2.1.5 Threats to Validity

Several aspects need to be taken into consideration when analyzing the results of this survey and making sure they're reliable. Our main goal is to understand if our tool is able to provide a good development experience. Similar to the previous experiment, we divide these threats into internal and external validity.

Internal Validity. In order to understand if there's a cause-effect relationship, the following aspects must be taken into consideration:

Software Engineering knowledge. Participants should, preferably, have some knowledge in Software Engineering, so that they can better understand why they would even need a refactoring tool in the first place. We didn't, however, rule out any participants with less experience in the area because we're interested in understanding what participants think about the usability and user experience of the tool, in which deep knowledge in Refactoring isn't necessarily a requirement. As such, we dedicated a part of the survey to better understand the participants' experience in programming and related areas. We also sent out the survey through channels where the vast majority of recipients are likely to have some software knowledge.

Experience with Refactoring Tools. In order to understand if participants see benefit in a live tool compared to a more traditional recommender, it is important to know whether they have some experience with the latter. Otherwise, while results may be positive, they may not necessarily indicate that liveness provided an advantage, since there was no baseline of comparison. To tackle this issue, just as in the previous threat, a section of the survey was dedicated to understand if developers have used refactoring tools before.

Environment. Our survey was made available online for anyone to participate. This means that all participants selected the environment where the experiment took place. As such, different elements and aspects of their surroundings could play a role in their response - e.g. possible external distractions, co-answering the questions, etc. This factor is always a risk since participants are not constrained to any specific environment and have access to the questions anytime they want.

External Validity. The generalization of the results to larger populations takes into consideration:

Sample Size. The number of persons who willingly participate in this survey is a factor that should always be taken into consideration, so that we are able to have some confidence.

As we'll see in the results section (*cf.* Section 5.2.2, p. 88), the number of participants was 31, which is a relatively small sample size. As such, any analysis must take this into consideration when interpreting our findings.

Sample Characteristics. The composition of our sample is also of importance to understand if participants are homogeneously distributed across the population, or if there is some unbalanced subset represented. In order to understand this, participants are asked to answer the technical background section of the survey, from which we can build a clearer picture on their software knowledge and experience.

5.2.2 Results

In this section, we describe the results gathered from 31 participants of the survey. We start by characterizing the participants (*cf.* Section 5.2.2.1, p. 88) and then discuss the results on the questions regarding the tools (*cf.* Section 5.2.2.2, p. 88).

5.2.2.1 Participants' Profiles

We here provide some insights on the population who volunteered to participate in the survey, so that we are able to better understand its distribution in terms of programming experience and knowledge.

Our participants are mainly in the 18-24 age range, accounting for 65% of the population, while the 25-40 comes in second with 23%. There were no participants under the age of 18. On the education level, most of the participants have either a Bachelor's or a Master's degree as the highest completed degree of education, representing 74% of all answers. Additionally, the majority of participants has 3-5 years (48%) or 5-10 years (23%) of programming experience, followed by participants with over 10 years (19%). The full description of this distribution can be seen in the ring charts in in Figure 5.12 (p. 89).

Also, most of the participants considered themselves to have good knowledge about programming and software development terms. On the other hand, the distribution of knowledge in JavaScript or TypeScript is more spread out (*cf.* Figure 5.13, p. 89).

Finally, participants were also asked about their experience regarding VS Code, and if they've ever used refactoring or software metrics analysis tools before. 100% already had experience in VS Code, and the vast majority had already used refactoring or metrics tools before (*cf.* Figure 5.14, p. 90).

5.2.2.2 Tool Evaluation

In this section, we discuss the input provided by the participants on their opinion about the tool, divided by the multiple tasks found in the survey (*cf.* Section 5.2.1.4, p. 85).

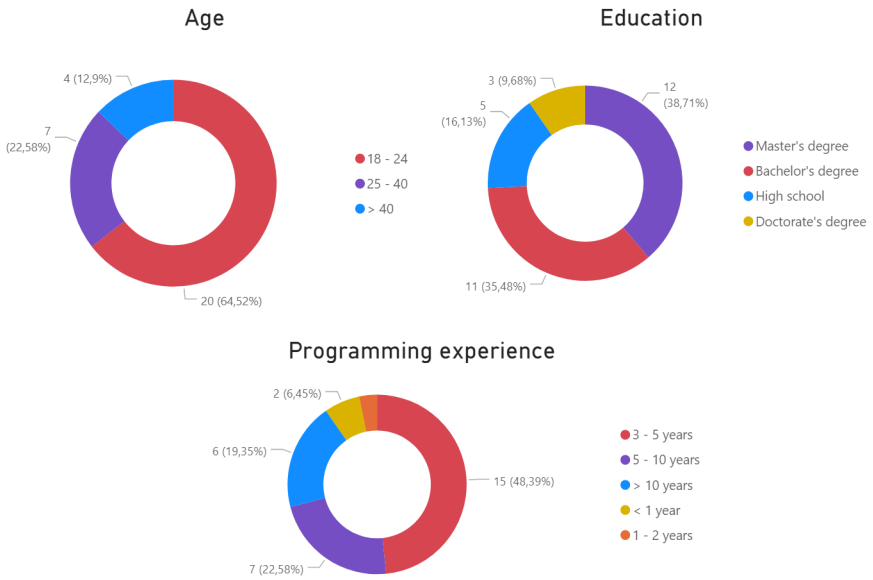


Figure 5.12: Age, education level and programming experience of participants.

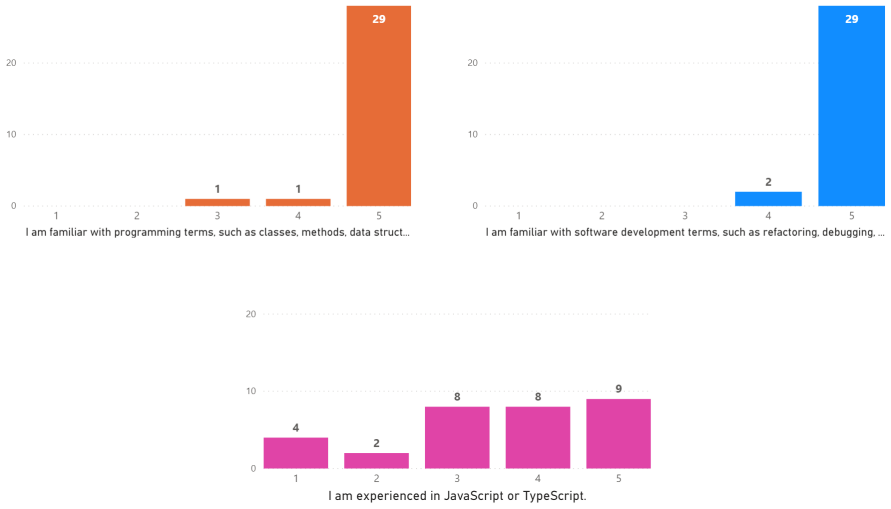


Figure 5.13: Knowledge of participants about programming and software development.

Visualization. In general, participants were able to identify the suggestions provided by our tool quickly (90%), which indicates an increase of the tool's presence. Additionally, most of them agreed that the gradient color scheme allowed them to rapidly identify the most prominent refactorings (87%), according to our tool. Figure 5.15 (p. 90) shows the full distribution among participants.

Regarding the continuous feedback next to line numbers, most of the participants were either neutral or didn't feel that it would be too distracting for their development activity (90%). In contrast with the previous results, however, the mode in this group was the neutral opinion, which may indicate that there is room for improvement in this visualization technique. Still, the results

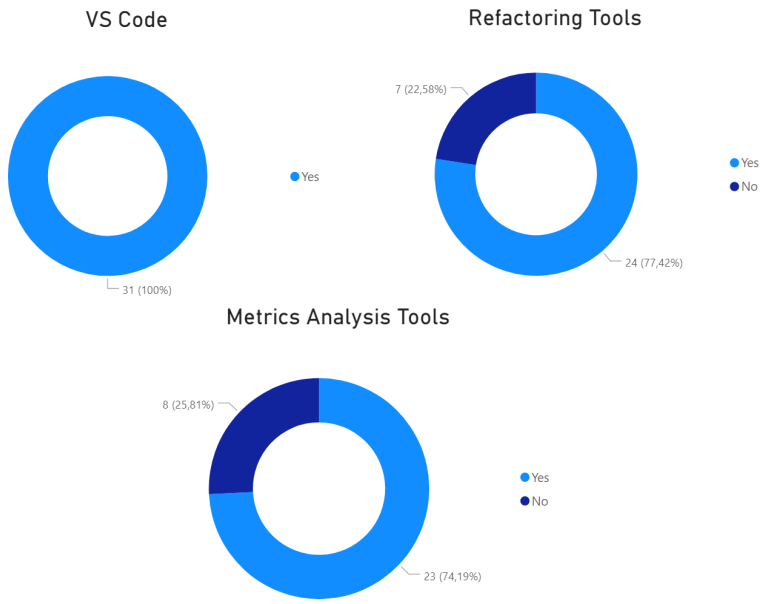


Figure 5.14: Experience of participants with relevant tools.

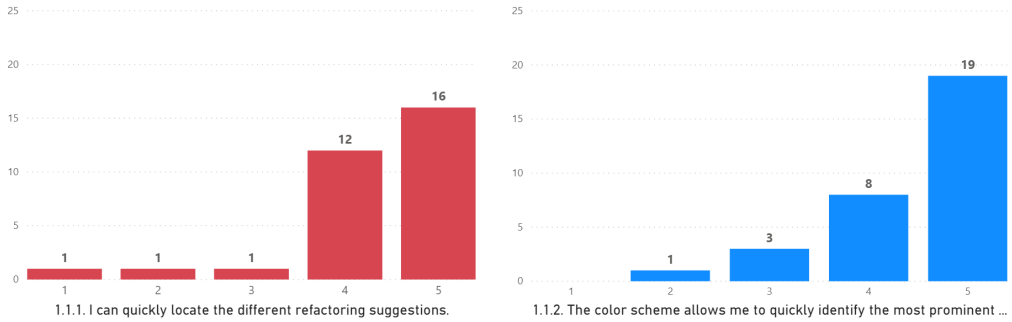


Figure 5.15: Opinions on the ability to identify refactoring candidates.

were overall positive and suggest that our approach to software visualization was successful. The bar chart on the left in Figure 5.16 (p. 91) conveys this information.

Participants also provided their opinion on how frequently should the suggestions be updated. Results are shown in the right chart in Figure 5.16 (p. 91). Our tool was built to update its suggestions anytime a change was made to the code, i.e., continuously - this was one the most preferred options, accounting for 29%. Other participants would prefer feedback to be updated once they change to another line or once they stopped typing (61% total). This supports the idea that full liveness on feedback to users might not always be the best option from the developers pespective (*cf.* Section 2.3.2, p. 27). However, people who preferred continuous feedback still hold a significant percentage. As such, we believe further investigation would be required to understand the best feature. Another, possibly better, option would be to offer the frequency rate as a customizable feature, as one of the participants suggested with a custom response to this

question.

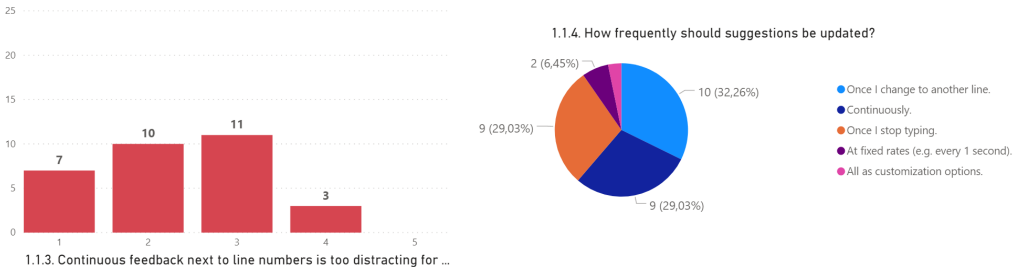


Figure 5.16: Opinions on the tool's continuous feedback.

Participants were also asked about their opinion on if tools should provide a single suggestion (the most prominent) or multiple suggestions. The results can be seen in Figure 5.17, where the red chart displays the results of how multiple suggestions impact their awareness, while the blue chart displays the results regarding its impact on understandability.

Most of participants agreed or highly agreed (94%) with having multiple suggestions available, so that they know which other alternatives can be taken in the refactoring process. Regarding the tool's understandability, results were more spread out, with only 65% agreeing that the tool improved their understanding of the tool's reasoning behind the suggestions, while 29% remained with neutral opinion. While we believe our results show that multiple suggestions improve the tool's understandability, other alternatives or refinements have to be done in order for it to be accepted by larger audiences.

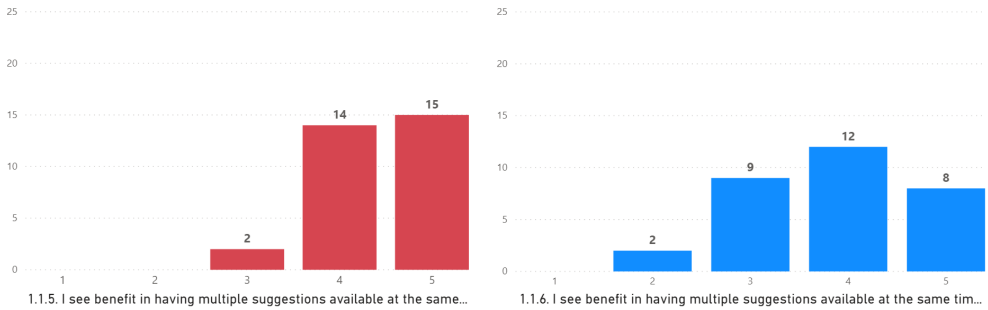


Figure 5.17: Opinions on single vs. multiple suggestions.

In the last part of this section, participants were asked if they could understand when the refactoring tool was available, which directly relates to the tool's relevance. As it can be seen in Figure 5.18 (p. 92), opinions ranged from highly disagree to highly agree. Still, the majority of participants (71%) were able to understand when the refactoring tool was available and when it was not. We conclude, therefore, that our tool increased its relevance by immediately showing feedback when it was useful for the programming activity in context.

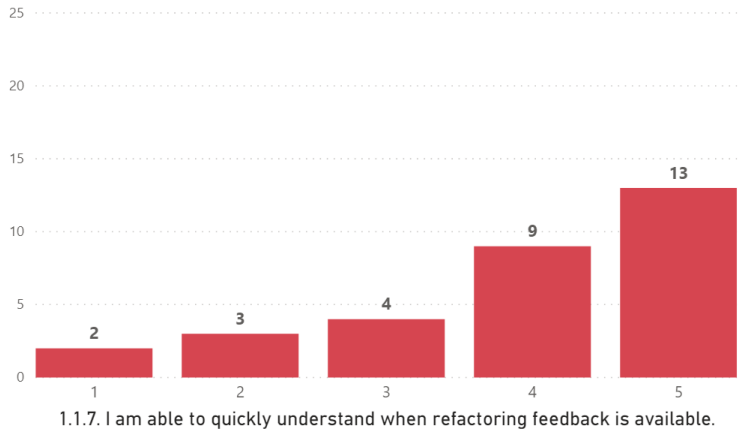


Figure 5.18: Opinions on the tool's availability and relevance.

Selection and Application. In order to select and apply refactoring suggestions, users must select the command provided by our tool through VS Code's Command Palette. In general, participants found that this feature is easy and intuitive enough to have in their regular development activity, with 90% agreeing or highly agreeing with its use (*cf.* Figure 5.19).

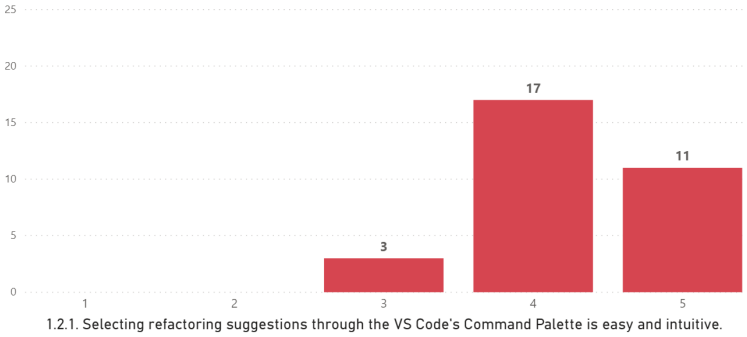


Figure 5.19: Opinions on using VS Code's Command Palette.

Participants also agreed with that highlighting the lines of code belonging to a particular suggestion is a helpful feature to have (94%), as seen in Figure 5.20 (p. 93).

Finally, most of the participants (87%) agree that having a semi-automated refactoring tool, i.e. a tool in which developers select a suggestion and then it is automatically applied, would be a good asset to have in their regular programming activity - despite opinions being more distributed than in the previous questions, with some disagreement -, as seen in Figure 5.21 (p. 93).

Code Quality History. Figure 5.22 (p. 94) shows participants' opinions regarding the various aspects about code quality history analysis in our tool. It is possible to observe that opinions are distributed across different levels of agreement but, in general, their perspective tends to be positive on this feature.

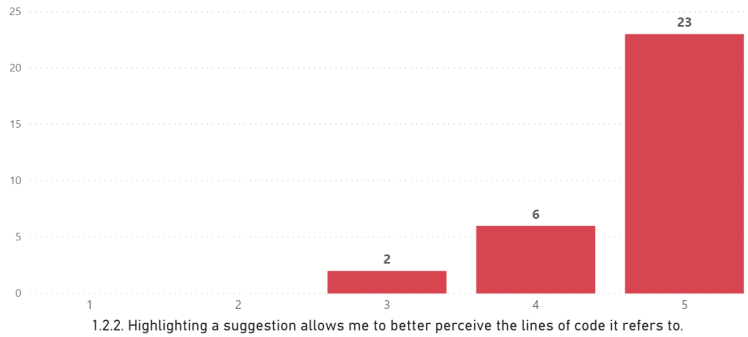


Figure 5.20: Opinions on selection highlighting.

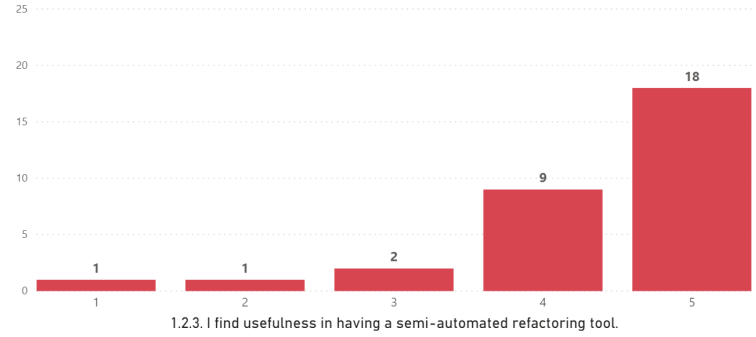


Figure 5.21: Opinions on having a semi-automated refactoring tool.

74% of participants agreed that software metrics are a good indicate of overall code complexity and quality (red chart), and 71% deem that knowing the absolute values of those metrics is important (orange chart). Also 74% find value in having an historical record of such values for inspection (blue chart). Finally, 68% of the participants agree that having live feedback on the evolution of code quality history would be a good feature to have, while 19% remained neutral about it (purple chart). This indicates that the majority would appreciate live feedback on history analysis, but more effort and investigation needs to address the possible alternatives for providing such feature.

Final Remarks. This section collects the participants' general impressions on the live refactoring tool they evaluated. We find that, in general, participants had a positive opinion about the overall usability and user experience of the tool.

As it can be seen in Figure 5.23 (p. 94), 97% of participants agreed that the tool's features were simple and easy to understand.

Also, participants most of the participants (84%) believe that our tool would positively impact their development workflow, and 77% of them would consider using this tool (*cf.* Figure 5.24, p. 94).

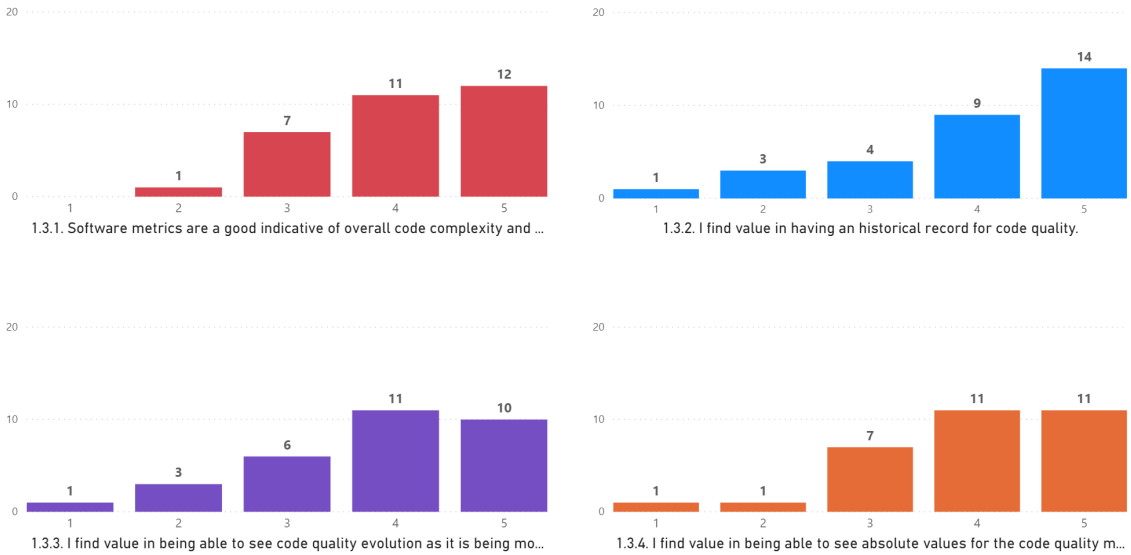


Figure 5.22: Opinions regarding code quality analysis.

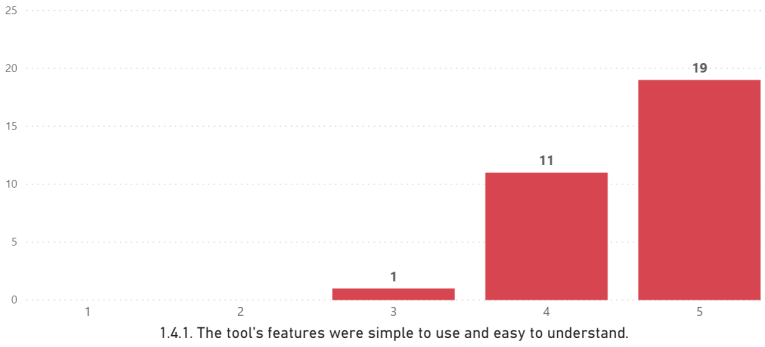


Figure 5.23: Opinions on the tool's ease of use.

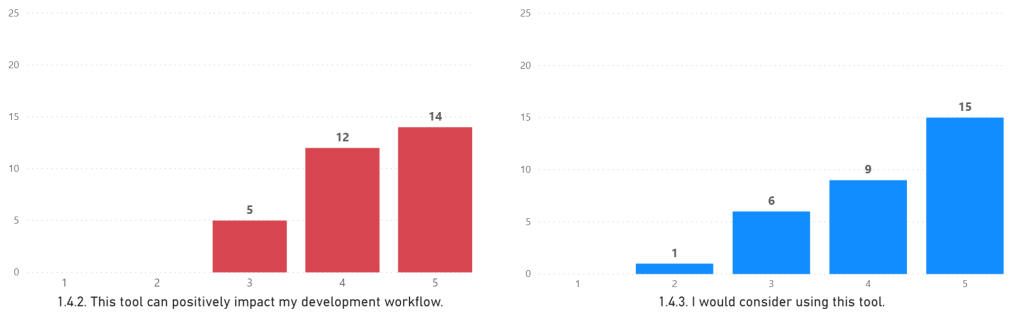


Figure 5.24: Opinions on the tool's impact and usability.

Finally, we asked participants to select which features were the most important. Most of the participants highly rated live feedback, suggestion severity based on color gradient, multiple

refactoring suggestions and having a semi-automated refactoring tool, as it can be seen in Figure 5.25. Only a small portion selected code quality history analysis as one of the best features in the tool.

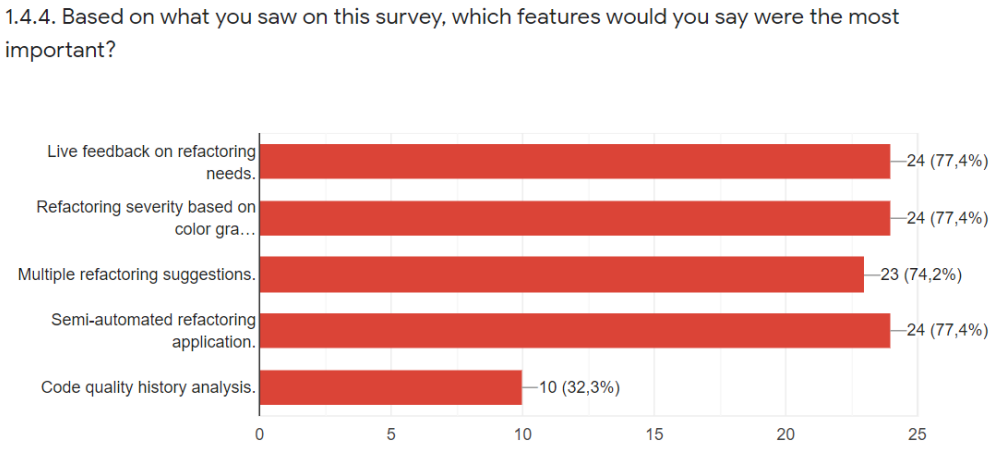


Figure 5.25: Opinions on the tool's best features.

Participants were then, inversely, also asked to select which features could be improved. As could be expected by the previous results, participants picked code quality history analysis to be a candidate for future improvements (*cf.* Figure 5.26). However, live feedback was also one of the top picks for the participants, even though it was also considered one of the best. This indicates that, although advantages can be seen in using liveness, there is immense room for improvements and new possibilities.

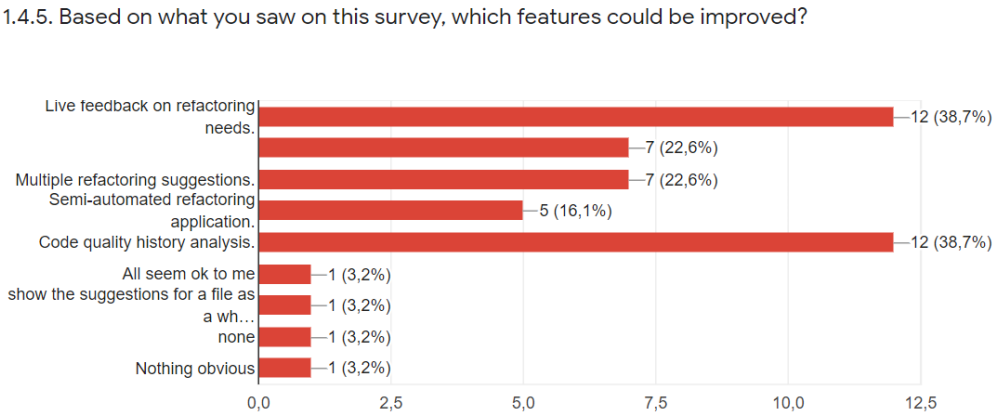


Figure 5.26: Opinions on the tool's worst features.

Additional Remarks. Throughout the different sections of the survey, participants had access to a free text area where they could give a more qualitative input on the tool and their opinion. We highlight the following suggestions regarding future improvements:

- Ability to select a threshold regarding the color gradient indicator, so that users may choose when it should appear (e.g. only show yellow to red suggestions, displaying nothing if they're green);
- Less bright colors to distinguish suggestions;
- Clusters could be ordered by their appearance in the code (we order them according to their severity);
- Code quality history should have thresholds of recommended values for each metric;
- Code quality history could have a way to see the state of the code on a particular historical instance.

5.2.3 Discussion

The goal of this experiment was to understand if developers consider liveness in refactoring tools to be a valuable asset for their activity. To do so, we developed a survey where participants were asked to answer a few questions about the refactoring tool, using static and animated images of its main features.

In summary, we believe that our results support the hypothesis that higher levels of liveness improve development experience. Our findings show that participants find value in the tool that was presented to them during this survey, and that it can positively impact their development workflow and, as such, they would consider using it. This is further corroborated by the fact that most participants found great value in having live feedback on refactoring needs and also in having a semi-automated refactoring tool, as shown in Figure 5.25 (p. 95).

Participants also believe that our tool is quickly able to inform them about refactoring needs, which indicates that our recommender is able to bring more presence to the environment, compared to traditional refactoring tools. Additionally, participants found value in having a color gradient scheme to classify the severity of refactoring suggestions and also didn't find, in general, the way information was provided to be too distracting for their regular programming activity. See Figure 5.15 (p. 90) and Figure 5.16 (p. 91) for more information. This also shows how Software Visualization and User Experience can play a significant role in providing more liveness to refactoring tools.

Another feature that was greatly appreciated by participants was the ability to see multiple refactoring suggestions (*cf.* Figure 5.25, p. 95). Results regarding single vs. multiple suggestions show that participants mostly found value in having multiple suggestions due to its ability to provide more awareness on the multiple options and refactoring opportunities available. When asked about the impact of having multiple suggestions in comprehending the tool's bias, which directly relates to the tool's understandability, answers were more spread out, with some disagreement and other neutral stances. This information can be seen in Figure 5.17 (p. 91). Nonetheless, despite the larger interest on awareness, we believe results show that multiple

suggestions are able to provide more understandability to developers. Still, it also shows that future work can be done in order to improve this tool's aspect.

Furthermore, participants agreed, in general, that they were able to easily understand in which situations the refactoring tool was available Figure 5.18 (p. 92), which supports our hypothesis that immediate feedback imbues refactoring tools with more relevance. On the other hand, results on this question were more spread out than in others (with some disagreement and neutral stances), which indicates that there is also room for improvement.

Finally, the opinions from participants show that our tool's code quality history analysis isn't one of the most appreciated features in this tool. According to Figure 5.22 (p. 94), in general, developers see value in having an historical record for this information and that, more particularly, having live feedback on this history would be a good feature. This indicates that our tool was not able to deliver up to their expectations, and should definitely be reworked. Nonetheless, our study shows that code quality history is a feature that would be prized by developers.

5.3 Summary

Our main hypothesis for this work is that *"Live Refactoring improves software quality and development experience"* (*cf.* Section 3.3, p. 37). This hypothesis can be divided in two sub-components to be addressed: (i) software quality and (ii) development experience. As such, two experiments were developed and executed, in order to better understand how our hypothesis can be supported, and how our research questions (*cf.* Section 3.4, p. 38) can be answered.

The first experiment carried out in this work focuses on understanding the impact of liveness in software quality. To do so, we developed a robotized refactoring tool that was capable of processing multiple methods, across different files and projects (*cf.* Section 5.1, p. 61). Our findings show that, in general, by having a live refactoring tool, that translates in developers applying refactoring suggestions earlier and more often, bring a positive impact in the software's code quality.

The second experiment focuses on the usability and user experience of the tool. We believe that answers to this component are inherently subjective and must come from users and their impression about using the tool. As such, our experiment consisted on a survey, handed out mainly to software developers - either students or professional workers -, in which several features of our tool were displayed for their evaluation (*cf.* Section 5.2, p. 83). Results show that live tools are generally able to bring a good development experience.

In conclusion, we believe our experiments are able to positively support our main hypothesis, and that we are able to adhere to the alternative hypothesis (*cf.* Table 3.1, p. 40) for all research questions.

Chapter 6

Conclusions

6.1 Summary 99
6.2 Main Contributions 104
6.3 Future Work 105

This chapter summarizes the key ideas of this dissertation, considering the work done, which is presented in Section 6.1. We then describe the main contributions of this investigation (*cf.* Section 6.2, p. 104), as well as any identified future work and research opportunities (*cf.* Section 6.3, p. 105).

6.1 Summary

As software products evolve, their maintainability becomes increasingly complex. One of the methodologies used to handle their progress and delivery is Agile Software Development, which has as core values fast and incremental building of software artifacts, allowing shorter feedback loops and easier accommodation of new requirements. One of the important practices to preserve the agility, maintainability and flexibility in the system is *Refactoring*.

The goal of refactoring is to change the internal structure of software, without altering its external behavior, with the sole purpose of simplifying the system's design - thus reducing the effort necessary to be modified when it needs to adapt to new objectives.

The name of this dissertation is *"Towards a Smart Recommender for Code Refactoring"* and, as such, we studied the existing ideas, practices and solutions regarding the recommendation of refactorings, to identify their current strengths and gaps, with the final goal of understanding how these recommenders can be smarter. In our understanding, the smartness of recommenders should be adressed from two different perspectives: (i) quality of the recommendations and (ii) quality of communication. The first is concerned with how suggestions are built and which algorithms

should be used, while the latter focuses on how those suggestions should be handed out to the end user.

Our investigation shows that the vast majority of recommenders is centered around the quality of recommendation component. In fact, the state-of-the-art research, in topics related to the refactoring activity (*cf.* Section 2.2, p. 12), shows that current tools focus mainly on describing algorithms that can be used by recommenders to provide refactoring suggestions. On the other hand, the way suggestions are transmitted to the developer is often disregarded: most of these recommenders analyze code on demand by having the developer manually select the tool in order to start its execution, which may not always be intuitive or effortless. Also, often times, if developers feel the need to use the tool, they already know there is a need for refactoring and might have a good idea on how to improve the code. The main consequence of this way of working is that, in practice, refactoring tools end up having limited use and, consequently, usefulness.

Considering the gaps found during this investigation, this work addresses the second component of smart recommenders, the quality of communication. We reason that refactoring recommenders should dedicate more effort into better conveying its information to developers because, in the end, the validity of the recommendations depend on their approval. Our approach has the core assumption that refactoring is an AI-complete problem Section 3.2 (p. 36). In a simple way, code refactoring relies on word sense disambiguation (WSD), which is considered an AI-complete problem and, as such, so is refactoring. Other factors also play a role in understanding the refactoring needs, even though they're not always represented in source code, such as software requirements.

Consequently, the correctness of refactoring applications must ultimately be judged by developers. We advocate that developers are the primary agent of change and validation in the refactoring activity. As such, refactoring tools should put more effort in improving their quality of transmission. A smart recommender is not one that surpasses developers, but rather empowers them.

After deciding which identified issues would be tackled in this investigation, it followed how to tackle them. As we've discussed in Section 2.3.2 (p. 27) and Section 3.3 (p. 37), we believe *Live Software Development* provides a good solution to mitigate those issues. Thus, in this dissertation, we studied the impact of LiveSD in the refactoring process. Given time constraints to perform this work, our research focused mainly on the candidate identification component of the Extract Method. We analyzed if higher levels of liveness lead to more present and relevant recommendation tools. This activity is named *Live Refactoring*. In addition, we also studied the impact of having multiple refactoring suggestions and how they affected the user's understanding on how suggestions are made. As such, the main hypothesis (*cf.* Section 3.3, p. 37) of this work was formulated as:

Live Refactoring improves software quality and development experience.

This hypothesis was then divided into three main research questions (*cf.* Section 3.4, p. 38). The first two had the goals of providing more clarity and understanding on the effects of liveness

in terms of code quality and development experience, respectively. The third research question studied the impact of multiple suggestions.

We then implemented our solution, a refactoring tool for VS Code that supports JavaScript and TypeScript, where we explained our reasoning behind the algorithms to find refactoring suggestions and how we wish to raise development awareness using liveness (*cf.* Chapter 4, p. 43). We also described our system's architecture and how its structure can be advantageous for future solutions that tackle similar issues (*cf.* Section 4.4, p. 50). Finally, we also detail how those features were built on VS Code and which components of the code editor were most important and useful to provide the end result, as well as demonstrating the presentation of our tool (*cf.* Section 4.5, p. 53).

The next step, after concluding the solution's implementation, was to validate our hypothesis and research questions. To do so, we carried out experiments that provide empirical validation, in support or opposition, to our initial topics of investigation (*cf.* Chapter 5, p. 61). Since our main hypothesis has two components to be addressed, namely code quality and development experience, we built two distinct experiments that allowed us to better validate each one: (i) to study the impact of liveness in code quality, we created a controlled experiment where refactoring were applied by a robotized simulation of a developer (*cf.* Section 5.1, p. 61), and (ii) to understand the effects of liveness in development experience, we handed out a survey where developers were asked to give their opinion on the tool's usability, mainly targeting its live components (*cf.* Section 5.2, p. 83).

After the conclusion of all previous steps, we are now able to discuss our research questions and whether they support the null hypothesis or the alternative hypothesis (*cf.* Table 3.1, p. 40), in each case. This discussion presupposes the understanding of the threats to validity of each experiment - Section 5.1.1.6 (p. 68) and Section 5.2.1.5 (p. 87) -, and provides a more formal and expanded analysis of our summary regarding the empirical validations carried out (*cf.* Section 5.3, p. 97).

RQ1. *"Do refactoring tools with higher levels of liveness improve software's code quality in every stage of development?"*

In our controlled experiment Section 5.1 (p. 61), we were able to gather a significant amount of refactoring applications from suggestions provided by our tool, using different software projects in distinct levels of maturity, software requirements and overall complexity. Results show that the applied changes were able to improve code quality metrics consistently across multiple revisions, files and methods. Despite the fact that some refactoring applications could worsen some metrics, the final code quality was always better than the initial. It was also shown that some significant refactoring needs remained in the projects during multiple revisions, which indicates that having this feedback earlier and faster would indeed provide an advantage for the developer, when building the next blocks of the project's application. An extended analysis can be seen in the experiment's discussion section (*cf.* Section 5.1.4, p. 81).

Thus, we believe our empirical results show that the null hypothesis for this research question can be rejected, in favor of the alternative hypothesis: software's code quality is improved with

liveness.

RQ2. *"Do refactoring tools with higher levels of liveness improve development experience?"*

This research question is aimed at understanding if developers believe live refactoring tools can be helpful to their regular programming activity. As we reasoned during the design of the experiment created to address this question (*cf.* Section 5.2.1.1, p. 84), the goal of better understanding the tool's usability and usefulness from the developers' perspective must necessarily be fulfilled by developers. As such, the experiment comprised a survey where participants (developers with different backgrounds) were asked to observe static and animated images, concerning our implemented solution, and give their opinion on what they've seen (*cf.* Section 5.2, p. 83). The results we were able to gather from participants, who voluntarily took part in the survey, show that most developers considered our tool to have value for their activity and believed it could positively impact their development workflow. It was also found that the majority of participants were of the opinion that continuous feedback was one of the most important features in our solution, as well as having a semi-automated refactoring tool - which indicates that live refactoring tools have a place in software development.

We also identified two characteristics that refactoring tools should aim towards, in order to provide a better user experience for developers, and we proposed to investigate if those characteristics could be achieved by higher levels of liveness. Since they directly relate to liveness and development experience, they were addressed as two separate sub-research questions. These are:

RQ2.1. *"Is the level of presence improved by liveness?"*

Presence can be defined as the recommender's visibility to developers. Developers are not always aware that refactoring tools exist in their programming environment. This lack of immediate presence reduces the effectiveness of refactoring tools because, in more complex refactoring operations, they are disruptive to the normal programming workflow.

We believe liveness is able to bring more presence to the coding environment because, by definition, it is concerned with providing more feedback to the end user. Indeed, survey results show that participants were generally able to quickly identify recommendations provided by our tool. Other factors also weigh in on this matter - such as the color gradient scheme to classify refactoring need, as well as an unobtrusive solution to continually show these recommendations -, which fall in the scope of Software Visualization and User Experience. Nonetheless, we believe that this question can be positively answered - liveness improves the level of presence of refactoring tools.

RQ2.2. *"Is the level of relevance improved by liveness?"*

Relevance can be defined as the recommender's pertinence to developers. This characteristic is closely related to presence, but focuses on the recommender's ability to convey when it can be used. By showing this information, developers are able to better reason in which situations the tool can be of help.

We attempted to increase our tool's relevance by only showing suggestions in the data structures of interest (functions and class methods). Liveness is used in this scenario because we immediately react to actions from developers, and toggle the recommendations' visibility accordingly. Participants of the survey generally agreed they were able to easily grasp when the refactoring tool was available. We also found some disagreement and neutral stances, which suggests that the approach used could be improved, or maybe that more answers to survey would be required to gain more certainty on the subject. Even so, we believe results point towards the positive effect of liveness in refactoring tools.

In summary, we believe liveness is a good solution to tackle current issues regarding development experience. Additionally, it also provides more presence and relevance to refactoring tools. The full extension of the discussion regarding how results support our claims can be seen in Section 5.2.3 (p. 96).

In conclusion, it is our opinion that the empirical results demonstrate that the null hypothesis for this research question can be rejected, in acceptance of the alternative hypothesis: development experience is improved with liveness.

RQ3. *"Is the level of understandability improved by offering multiple suggestions?"*

In our last research question, our goal was to understand if developers are able to better understand refactoring recommendations by displaying multiple suggestions. Our state-of-the-art research shows that current refactoring tools usually just show the final result of the application, if accepted by the developer, possibly containing some additional quality metrics. This may carry little information on why that specific refactoring was suggested instead of others.

We identify this feature as understandability, which can be defined as the recommender's ability to convey its bias in finding suggestions to the developer. Notice that our goal is not to explain why a certain suggestion is acceptable - that falls in the scope of *explainability*. Rather, we're interested in showing how the suggestions were built and which possible factors could contribute to it. A developer may understand why a recommender would identify a certain refactoring suggestion, but still disagree with its validity and explanation.

We believe refactoring recommenders can increase their levels of understandability by showing multiple suggestions, since it allows the developer to better understand the different patterns found with the algorithm. Just as in **RQ2**, answers to this research questions should be gathered directly from developers. As such, we dedicated a section of our survey to the use of multiple suggestions (*cf.* Section 5.2, p. 83). From the results, our understanding is that participants generally saw benefit in having multiple suggestions because it increases their awareness on the opportunities available. When asked about their ability to better understand patterns in the suggestions, the majority of participants agreed that having multiple suggestions helped them. However, a significant number of answers remained on the neutral stance, which indicates that our tool can be improved and better solutions can be implemented in order to bring more

understandability to the coding environment. Nonetheless, the majority of answers considered having multiple suggestions to be positive, and it was also regarded as one of the most important features of our tool.

Thus, we believe our empirical results allow us to accept the alternative hypothesis and reject the null hypothesis: multiple suggestions improve the tool's understandability.

6.2 Main Contributions

Our research comprises several different topics, ranging from Code Smells to Software Visualization. We believe this work offers an extensive review on current refactoring practices, and provides insightful knowledge regarding their strengths and weaknesses. From this review, we concluded that an essential aspect of recommenders is often disregarded: development experience. Recommenders should take into consideration the central role of the developer in the refactoring process, in which they are the ultimate agent of change. This means that *smart* recommendation system must not only know what suggestions to give, but also how to give them.

As we've mentioned multiple times throughout this work, **a smart recommender is not one that surpasses developers, but rather empowers them**. We consider this idea to be the main contribution of this work, and we hope it can open new paths of investigation in the future.

Other important contributions have been made in this work:

State-of-the-art Analysis. The systematic literature review performed during this work (*cf.* Chapter 2, p. 7) provides extensive information on the topics of Code Smells (*cf.* Section 2.1, p. 7), Refactoring (*cf.* Section 2.2, p. 12), Live Software Development (*cf.* Section 2.3, p. 24) and Software Visualization (*cf.* Section 2.4, p. 29). Each section is composed of (i) an introduction, where the concept is introduced, defined and its purpose explained; (ii) a description of different techniques and tools developed; and (iii) a discussion, where strengths and flaws are considered and critically analyzed.

This review was also the building block for the later work produced in this investigation. Namely, it allowed us to identify current problems and formulate our hypothesis (*cf.* Chapter 3, p. 35), while also deeply impacting our approach to the implementation of our solution.

Refactoring Tool. In order to validate our hypothesis, we developed a refactoring recommender, which was built as a VS Code extension. Throughout the chapter where the solution is described (*cf.* Chapter 4, p. 43), we address multiple concerns and aspects that should be taken into consideration when implementing new refactoring tools. We provide detailed explanations on why both unsupervised learning and liveness produce better outcomes, and how these techniques can be applied - described in Section 4.2 (p. 44) and Section 4.3 (p. 48), respectively. The solution's architecture is also explained and provides some insights and suggestions on how recommenders can be built with scalability in mind (*cf.* Section 4.4, p. 50). We believe that our architecture provides good flexibility, and could

easily accommodate new refactoring techniques and features. Finally, we address the multiple features offered by VS Code and their importance in the implementation phase (*cf.* Section 4.5, p. 53).

Empirical Validation. Our work comprises two different experiments that were carried out to validate our hypothesis (*cf.* Chapter 5, p. 61). Both validation methods were designed according to their goal - the research question they intend to validate. We provide thorough explanations and descriptions regarding their design and how they can fulfill the goals of our investigation, while also identifying the possible threats to their validity. We also present the results gathered from these experiments, concluding with a final discussion on their consequences.

Innovative Empirical Method. To our knowledge, the validation method carried out during our controlled experiment (*cf.* Section 5.1, p. 61) is innovative in the field. We produced a robotized refactoring tool that simulates developer behavior during their refactoring activity. Then, we let the robot analyze multiple software projects, applying refactoring suggestions and collect code metrics before and after the applications.

The main benefit of this approach is the ability to collect considerable amounts of data, across software projects with different contexts, goals and requirements. In this work, we were able to gather information about approximately 20,000 refactoring applications, which is already far greater than most research using different methods. We believe this method has the potential to achieve much greater numbers, likely numbering in the millions, through the optimization of its algorithm and the use of parallel programming. The more instances we can use to evaluate the impact of our tool, the more confidence we can have on the results.

Other key components need to be addressed to grow this method's robustness and reliability, including the assumptions and heuristics identified when designing this experiment. A full description of these components can be read in Section 5.1.1 (p. 62) and Section 5.1.4 (p. 81). Nontheless, this validation method remains a good option to take into consideration when more data is needed to draw conclusions from.

6.3 Future Work

Throughout this work, we've identified multiple flaws or possible improvements that could be tested and implemented to refine our solution, as follows:

- Our tool attempted to calculate if a refactoring candidate was extractable and display that information to developers when they're picking a suggestion (*cf.* Figure 4.7, p. 56). Our controlled experiment showed that our calculations were insufficient, and that approximately 45% of candidates deemed extractable, in reality were not (*cf.* Section 56, p. 81). This rate is too high to be acceptable, so this improvement would be one of the most important in future developments.

- The code quality history analysis of our tool, although generally accepted as a good feature (*cf.* Figure 5.22, p. 94), was noticeably less rated than other features provided (*cf.* Figure 5.25, p. 95). Thus, new approaches to display this type of data should be considered.
- The tool should offer more customization. This issue wasn't addressed during this work, but participants of the survey (*cf.* Section 5.2, p. 83) suggested multiple times the ability to control certain features to better fit their personal taste. As such, we believe this would be a good improvement to our tool. A few examples follow:
  - Define the minimum number of lines to build a cluster.
  - Specify the number of instances to be recorded in code quality history analysis.
  - Ability to select a threshold for the color gradient (e.g. only show colors from yellow to red).
  - Ability to select the frequency of suggestions updates - continuously, after the user stops typing, after changing to another line, etc.
- Our tool uses unsupervised learning techniques to identify refactoring necessities. A whole section in the proposed solution chapter is dedicated to explain why and how certain decisions were made when developing this feature (*cf.* Section 4.2, p. 44). Nonetheless, some open questions remain about the true potential of these clustering techniques, including:

  Data Representation. Our tool uses a binary representation of variables and structural dependencies found in source code, because it allowed us to assign a semantic meaning to the parameters of our algorithm (*cf.* Section 20, p. 47). Still, we believe that more investigation, exclusively dedicated to candidate identification, could provide improved solutions for this matter.

  Clustering Algorithm. In this work, three unsupervised learning algorithms were studied and compared (K-Means, DBSCAN and OPTICS). Future works should try to provide a deeper comparison between these three techniques, as well as including others that weren't used in this investigation.

  Distance Function. The distance function used was very dependent on data representation and clustering algorithm. By exploring other approaches to these two, other distance functions may be more appropriate.

  Cluster Validity. Our solution uses the Silhouette Coefficient to measure how good are the clusters that were found. We believe more work has to be done in this particular matter, to better understand what other alternatives are available and how they correlate to code quality.
- Liveness is a core feature of our solution which, in our opinion, brought solid improvements to some of the identified flaws in current refactoring tools. We believe there is still

much work to be done in this area that future research could use as a building block for their investigations. Specifically, we need better understanding on how liveness can affect the performance of our applications, and also dedicate more effort to comprehending how research areas like Software Visualization, Human-Computer Interaction and User Experience can play a more significant role in bringing shorter feedback loops to developers.

We also developed a new validation method to understand the impact of liveness in code quality (*cf.* Section 5.1, p. 61). As was discussed during the experiment's design (*cf.* Section 5.1.1, p. 62) and discussion (*cf.* Section 5.1.4, p. 81), a few assumptions and approximations had to be made, so that it was feasible to run the experiment given our time constraints. Future research should attempt to address these open issues.

Finally, we want to leave this important note for future research, more specifically for refactoring recommenders, but also for recommenders in general. The major question in focus during this investigation was not about *what* developers need, but *how* they need it. As such, we want to encourage future works to continue tackling both. Only by having these two components thoroughly addressed, then we can say we have a truly smart recommender.

References

[ABKS99] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. OPTICS. *ACM SIGMOD Record*, 28(2):49–60, jun 1999.

[Agi01] Manifesto for Agile Software Development, 2001. Available at https://agilemanifesto.org/.

[AP14] Saeed Aghaee and Cesare Pautasso. End-User Development of Mashups with NaturalMash. *Journal of Visual Languages & Computing*, 25(4):414–432, aug 2014.

[APFC15] Ramon Abilio, Juliana Padilha, Eduardo Figueiredo, and Heitor Costa. Detecting Code Smells in Software Product Lines – An Exploratory Study. In *2015 12th International Conference on Information Technology - New Generations*, pages 433–438. IEEE, apr 2015.

[ARC+19] Ademar Aguiar, André Restivo, Filipe Figueiredo Correia, Hugo Sereno Ferreira, and João Pedro Dias. Live software development. In *Proceedings of the 3rd International Companion Conference on Art, Science, and Engineering of Programming - Programming '19*, pages 1–6, New York, New York, USA, 2019. ACM Press.

[BBM10] Sergio Bryton, Fernando Brito e Abreu, and Miguel Monteiro. Reducing Subjectivity in Code Smells Detection: Experimenting with the Long Method. In *2010 Seventh International Conference on the Quality of Information and Communications Technology*, pages 337–342. IEEE, sep 2010.

[BCT07] Thierry Bodhuin, Gerardo Canfora, and Luigi Troiano. SORMASA: A tool for suggesting model refactoring actions by metrics-led genetic algorithm. *1st Workshop on Refactoring Tools (WRT)*, 8:23–24, 2007.

[BDO11] Gabriele Bavota, Andrea De Lucia, and Rocco Oliveto. Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software*, 84(3):397–414, mar 2011.

[Ben12] Richard Bender. Systems Development Life Cycle : Objectives and Requirements. Technical report, 2012.

[BGE95] F Brito e Abreu, M Goulao, and R Esteves. Toward the Design Quality Evaluation of Object Oriented Software Systems. In *Proc. of 5th International Conference on Software Quality*, pages 1–12, 1995.

[BGH07] Marat Boshernitsan, Susan L. Graham, and Marti A. Hearst. *Aligning development tools with the way programmers think about code changes*. 2007.

REFERENCES

[Bie89] Irving Biederman. "Recognition-by-components: A theory of human image understanding": Clarification. *Psychological Review*, 96(1), jan 1989.

[BKKK87] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data - SIGMOD '87*, pages 311–322, New York, New York, USA, 1987. ACM Press.

[BOD+10] Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, Giuliano Antoniol, and Yann-Gael Gueheneuc. Playing with refactoring: Identifying extract class opportunities through game theory. In *2010 IEEE International Conference on Software Maintenance*, pages 1–5. IEEE, sep 2010.

[BOD+14] Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, Andrian Marcus, Yann-Gael Gueheneuc, and Giuliano Antoniol. In medio stat virtus: Extract class refactoring through nash equilibria. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 214–223. IEEE, feb 2014.

[BPPT03] Paolo Bottoni, Francesco Parisi-Presicce, and Gabriele Taentzer. Coordinated Distributed Diagram Transformation for Software Evolution1 1Partially supported by the EC under Research and Training Network SeGraVis. *Electronic Notes in Theoretical Computer Science*, 72(4):59–70, mar 2003.

[BTN03] FM Bravo, V Tenebras, and M De Nantes. A logic meta-programming framework for supporting the refactoring process. 2003.

[CC90] E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, jan 1990.

[CFYH18] Eunjong Choi, Kenji Fujiwara, Norihiro Yoshida, and Shinpei Hayashi. A Survey of Refactoring Detection Techniques Based on Change History Analysis. aug 2018.

[Chr17] Google Chrome. Chrome DevTools | Tools for Web Developers | Google Developers, 2017. Available at https://developers.google.com/web/tools/chrome-devtools/.

[CMC15] Gabriela Czibula, Zsuzsanna Marian, and Istvan Gergely Czibula. Detecting software design defects using relational association rule mining. *Knowledge and Information Systems*, 42(3):545–577, mar 2015.

[CZ11] Pierre Caserta and Olivier Zendra. Visualization of the Static Aspects of Software: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 17(7):913–933, jul 2011.

[DDN00] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. *ACM SIGPLAN Notices*, 35(10):166–177, oct 2000.

[DF98] Andreas Dieberger and Andrew U. Frank. A City Metaphor to Support Navigation in Complex Information Spaces. *Journal of Visual Languages & Computing*, 9(6):597–622, dec 1998.

[DL05] Stéphane Ducasse and Michele Lanza. The class blueprint: visually supporting the understanding of classes. *IEEE Transactions on Software Engineering*, 31(1):75–90, jan 2005.

[Dre81] M Dresher. *The Mathematics of Games of Strategy: Theory and Applications*. Dover Books on Mathematics Series. Dover, 1981.

[DVK17] Finale Doshi-Velez and Been Kim. Towards A Rigorous Science of Interpretable Machine Learning. feb 2017.

[EKSX96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press, 1996.

[ESS92] S.C. Eick, J.L. Steffen, and E.E. Sumner. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992.

[Ett07] Ran Ettinger. Refactoring via program slicing and sliding. Technical report, 2007.

[Fer19] Sara Fernandes. *Supporting Software Development through Live Metrics Visualization*. PhD thesis, University of Porto, jul 2019.

[Fis13] Andrew Fischer. Introducing Circa: A dataflow-based language for live coding. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 5–8. IEEE, may 2013.

[FM13] Amin Milani Fard and Ali Mesbah. JSNOSE: Detecting JavaScript Code Smells. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125. IEEE, sep 2013.

[FOSM00] Henrique Freitas, Mírian Oliveira, Amarolinda Zanela Saccol, and Jean Moscarola. O método de pesquisa survey. *Revista de Administração da Universidade de São Paulo*, 35(3), 2000.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[FS15] Shizhe Fu and Beijun Shen. Code Bad Smell Detection through Evolutionary Data Mining. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, volume 2015-Novem, pages 1–9. IEEE, oct 2015.

[GME05] Denis Gračanin, Krešimir Matković, and Mohamed Eltoweissy. Software visualization. *Innovations in Systems and Software Engineering*, 1(2):221–230, sep 2005.

[GP07] Ben. Goertzel and Cassio. Pennachin. *Artificial General Intelligence*. Cognitive Technologies. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[GYB04] Hamish Graham, Hong Yul Yang, and Rebecca Berrigan. *A solar system metaphor for 3D visualisation of object oriented software metrics*, volume 35. 2004.

[HH16] Roman Haas and Benjamin Hummel. Deriving Extract Method Refactoring Suggestions for Long Methods. Lecture Notes in Business Information Processing, pages 144–155. Springer International Publishing, 2016.

[HHK12] Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. Identifying, Tailoring, and Suggesting Form Template Method Refactoring Opportunities with Program Dependence Graph. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 53–62. IEEE, mar 2012.

[HHT96] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. GRAPH GRAMMARS WITH NEGATIVE APPLICATION CONDITIONS. *Fundamenta Informaticae*, 26(3,4):287–313, 1996.

[HKFG18] Péter Hegedűs, István Kádár, Rudolf Ferenc, and Tibor Gyimóthy. Empirical evaluation of software maintainability based on a manually validated refactoring dataset. *Information and Software Technology*, 95:313–327, mar 2018.

[IW00] Pourang Irani and Colin Ware. Diagrams based on structural object perception. In *Proceedings of the working conference on Advanced visual interfaces - AVI '00*, pages 61–67, New York, New York, USA, 2000. ACM Press.

[KHFG16] Istvan Kadar, Peter Hegedus, Rudolf Ferenc, and Tibor Gyimothy. A Code Refactoring Dataset and Its Assessment Regarding Software Maintainability. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, mar 2016.

[KHK11] Tomoko Kanemitsu, Yoshiki Higo, and Shinji Kusumoto. A visualization method of program dependency graph for identifying extract method opportunity. In *Proceeding of the 4th workshop on Refactoring tools - WRT '11*, WRT '11, page 8, New York, New York, USA, 2011. ACM Press.

[KKS+14] Wael Kessentini, Marouane Kessentini, Houari Sahraoui, Slim Bechikh, and Ali Ouni. A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection. *IEEE Transactions on Software Engineering*, 40(9):841–861, sep 2014.

[KM02] C. Knight and M. Munro. Virtual but visible software. In *2000 IEEE Conference on Information Visualization. An International Conference on Computer Visualization and Graphics*, pages 198–205. IEEE Comput. Soc, nov 2002.

[KRB18] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. The road to live programming. In *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*, pages 1090–1101, New York, New York, USA, may 2018. ACM Press.

[LD98] Arun Lakhotia and Jean-Christophe Deprez. Restructuring programs by tucking statements into functions. *Information and Software Technology*, 40(11-12):677–689, dec 1998.

[LD01] Michele Lanza and Stéphane Ducasse. A categorization of classes based on the visualization of their internal structure. *ACM SIGPLAN Notices*, 36(11):300–311, nov 2001.

[Lip16] Zachary C. Lipton. The Mythos of Model Interpretability. *Communications of the ACM*, 61(10):35–43, jun 2016.

[LL13] Remo Lemma and Michele Lanza. Co-evolution as the key for live programming. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 9–10. IEEE, may 2013.

[LLTT12] Yu Leau, Wooi Khong Loo, Wai Yip Tham, and Soo Fun Tan. Software Development Life Cycle AGILE vs Traditional Approaches. Technical Report Icint, 2012.

[LXZS06] Chung-Horng Lung, Xia Xu, Marzia Zaman, and Anand Srinivasan. Program restructuring using clustering techniques. *Journal of Systems and Software*, 79(9):1261–1279, sep 2006.

[MAB+12] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Esma Aïmeur. Support vector machines for anti-pattern detection. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, page 278, New York, New York, USA, 2012. ACM Press.

[Mac67] J MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Berkeley, Calif., 1967. University of California Press.

[Mar01] Katsuhisa Maruyama. Automated method-extraction refactoring by using block-based slicing. *ACM SIGSOFT Software Engineering Notes*, 26(3):31–40, may 2001.

[Mar05] Radu Marinescu. Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, volume 2005, pages 701–704. IEEE, 2005.

[MB16] Antonio Martini and Jan Bosch. An empirically developed method to aid decisions on architectural technical debt refactoring. In *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16*, pages 31–40, New York, New York, USA, may 2016. ACM Press.

[MFM03] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3D representations for software visualization. In *Proceedings of the 2003 ACM symposium on Software visualization - SoftVis '03*, page 27, New York, New York, USA, 2003. ACM Press.

[MGDL10] Naouel Moha, Y.-G. Gueheneuc, Laurence Duchien, and A.-F. Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, jan 2010.

[MHPB12] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering*, 38(1):5–18, jan 2012.

[MRA18] Panita MEANANEATRA, Songsakdi RONGVIRIYAPANISH, and Taweesup APIWATTANAPONG. Refactoring Opportunity Identification Methodology for Removing Long Method Smells and Improving Code Analyzability. *IEICE Transactions on Information and Systems*, E101.D(7):1766–1779, jul 2018.

[MRCN03] R C Martin, J M Rabaey, A P Chandrakasan, and B Nikolic. *Agile Software Development: Principles, Patterns, and Practices*. Alan Apt series. Pearson Education, 2003.

[MSAS06] Raimund Moser, Alberto Sillitti, Pekka Abrahamsson, and Giancarlo Succi. Does Refactoring Improve Reusability? In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 4039 LNCS, pages 287–297. Springer Verlag, 2006.

[MSK+19] W. James Murdoch, Chandan Singh, Karl Kumbier, Reza Abbasi-Asl, and Bin Yu. Interpretable machine learning: definitions, methods, and applications. *Proceedings of the National Academy of Sciences of the United States of America*, 116(44):22071–22080, jan 2019.

[MT04] Tom Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, feb 2004.

[MVDJ05] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, jul 2005.

[Nav09] Roberto Navigli. Word sense disambiguation: A survey. *ACM Computing Surveys*, 41(2):1–69, feb 2009.

[ONY13] Tomohiro Oda, Kumiyo Nakakoji, and Yasuhiro Yamamoto. SOMETHINGit: A prototyping library for live and sound improvisation. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 11–14. IEEE, may 2013.

[Opd92] William F Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, USA, 1992.

[Ora14] Java Platform Debugger Architecture, 2014. Available at https://docs.oracle.com/javase/7/docs/technotes/guides/jpda/.

[PBP+15] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. Mining Version Histories for Detecting Code Smells. *IEEE Transactions on Software Engineering*, 41(5):462–489, may 2015.

[PBTP18] Jevgenija Pantiuchina, Gabriele Bavota, Michele Tufano, and Denys Poshyvanyk. Towards just-in-time refactoring recommenders. In *Proceedings of the 26th Conference on Program Comprehension - ICPC '18*, pages 312–315, New York, New York, USA, 2018. ACM Press.

[PDMG14] Francis Palma, Johann Dubois, Naouel Moha, and Yann-Gaël Guéhéneuc. Detection of REST Patterns and Antipatterns: A Heuristics-Based Approach. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8831, pages 230–244. Springer Verlag, 2014.

[Pit16] Fabian Pittke. *Linguistic refactoring of business process models*. Logos Verlag Berlin GmbH, 2016.

[Ram10] Girish Maskeri Rama. A desiderata for refactoring-based software modularity improvement. In *Proceedings of the 3rd India software engineering conference on India software engineering conference - ISEC '10*, page 93, New York, New York, USA, 2010. ACM Press.

[Rea17] Facebook React. React – A JavaScript library for building user interfaces, 2017. Available at https://reactjs.org/.

[RL08] Romain Robbes and Michele Lanza. SpyWare. In *Proceedings of the 13th international conference on Software engineering - ICSE '08*, page 847, New York, New York, USA, 2008. ACM Press.

[Rob99] D. B. Roberts. *Practical analysis for refactoring*. PhD thesis, 1999.

[Rou87] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20(C):53–65, nov 1987.

[RSG08] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. On the relation of refactorings and software defect prediction. In *Proceedings of the 2008 international workshop on Mining software repositories - MSR '08*, page 35, New York, New York, USA, 2008. ACM Press.

[RSVG07] Jacek Ratzinger, Thomas Sigmund, Peter Vorburger, and Harald Gall. Mining Software Evolution to Predict Refactoring. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 354–363. IEEE, sep 2007.

[SGW+11] Yu Sun, Jeff Gray, Christoph Wienands, Michael Golm, and Jules White. A Demonstration-based Approach to Support Live Transformations in a Model Editor. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6707 LNCS, pages 213–227. 2011.

[Sha12] Tushar Sharma. Identifying extract-method refactoring candidates automatically. In *Proceedings of the Fifth Workshop on Refactoring Tools - WRT '12*, pages 50–53, New York, New York, USA, 2012. ACM Press.

[SKBD14] Dilan Sahin, Marouane Kessentini, Slim Bechikh, and Kalyanmoy Deb. Code-Smell Detection as a Bilevel Problem. *ACM Transactions on Software Engineering and Methodology*, 24(1):1–44, oct 2014.

[SKM18] Jagannath Singh, Pabitra Mohan Khilar, and Durga Prasad Mohapatra. Code refactoring using slice-based cohesion metrics and aspect-oriented programming. *International Journal of Business Information Systems*, 27(1):45, 2018.

[SMT16] Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. Designite - A Software Design Quality Assessment Tool. In *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities - BRIDGE '16*, pages 1–4, New York, New York, USA, may 2016. ACM Press.

[SS07] Konstantinos Stroggylos and Diomidis Spinellis. Refactoring–Does It Improve Software Quality? In *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*, pages 10–10. IEEE, may 2007.

[SS18] Tushar Sharma and Diomidis Spinellis. A survey on software smells. *Journal of Systems and Software*, 138:158–173, apr 2018.

[SST06] Mazeiar Salehie, Shimin Li, and Ladan Tahvildari. A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, volume 2006, pages 159–168. IEEE, 2006.

[SWM00] M.-A.D. Storey, K. Wong, and H.A. Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2-3):183–207, mar 2000.

[Tan13] Steven L Tanimoto. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 31–34. IEEE, may 2013.

[TC09] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of Extract Method Refactoring Opportunities. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 119–128. IEEE, 2009.

[TJ19] Omkarendra Tiwari and Rushikesh K Joshi. Extract Method Refactoring by Successive Edge Contraction. *arXiv:1908.04636 [cs]*, aug 2019.

[TME+18] Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*, ICSE '18, pages 483–494, New York, New York, USA, 2018. ACM Press.

[TPB+15] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and Why Your Code Starts to Smell Bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 403–414. IEEE, may 2015.

[Vue14] Vue. Vue.js, 2014. Available at https://vuejs.org/.

[WB95] M. P. Ward and K. H. Bennett. Formal methods to aid the evolution of software. *International Journal of Software Engineering and Knowledge Engineering*, 05(01):25–47, mar 1995.

[Wei81] Mark Weiser. Program slicing. In *Proceedings - International Conference on Software Engineering*, ICSE '81, pages 439–449. IEEE Press, 1981.

[WL08] Richard Wettel and Michele Lanza. Visual Exploration of Large-Scale System Evolution. In *2008 15th Working Conference on Reverse Engineering*, pages 219–228. IEEE, oct 2008.

[WWF20] Yinglin Wang, Ming Wang, and Hamido Fujita. Word Sense Disambiguation: A comprehensive knowledge exploitation framework. *Knowledge-Based Systems*, 190:105030, feb 2020.

[XSKX17] Sihan Xu, Aishwarya Sivaraman, Siau-Cheng Khoo, and Jing Xu. GEMS: An Extract Method Refactoring Recommender. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, volume 2017-Octob, pages 24–34. IEEE, oct 2017.

[YLN09] Limei Yang, Hui Liu, and Zhendong Niu. Identifying Fragments to be Extracted from Long Methods. In *2009 16th Asia-Pacific Software Engineering Conference*, pages 43–49. IEEE, dec 2009.

REFERENCES

Appendix A

Behavior Flow Diagram - PlantUML Code

This code was used to generate the flowchart diagram regarding the system's behavior (*cf.* Figure 4.1, p. 44).

```
@startuml
skinparam linetype polyline
skinparam linetype ortho

title Recommender workflow

(*) -down-> "Dev Environment"
-down->[   event] "Refactoring"
-left-> "Parser"
-right-> "Refactoring"



if "New suggestions?"
    -down->[  yes] "Data Representation"
    -down-> "Clustering"
    -up->[update] "Dev Environment"
endif
@enduml
```

Appendix B

Architecture Diagram - PlantUML Code

This code was used to generate the architecture diagram (*cf.* Figure 4.3, p. 51).

```
@startuml
skinparam linetype polyline
skinparam linetype ortho

title System architecture

node "Presentation" as node_presentation {
    package "ui" {
        component [Editor Decorator]
    }

    file "extension.ts" as extension
}

node "Middleware" as node_middleware {
    package "processor" {
        component [Analyzer]
    }

    package "compiler" {
        component [Parser]
    }
}

node "Logic" as node_logic {
```

```
    package "refactoring" {
        component [Extract Method]
    }

    package "analysis" {
        component [Data Representation]
        component [Clustering]
    }
}

node "Core" as node_core {
    package core #FFFFFF [
        {{
          class Variable
          class Statement
          class Method
          class Cluster
        }}
    ]
}

' Relationships
extension -> [Analyzer]
Analyzer -> refactoring
Analyzer -> ui
[Extract Method] --> analysis
[Extract Method] --> compiler
node_presentation <.> core
node_logic <.> core
node_middleware <.> core

' Positioning constraints
node_presentation -[hidden]-> node_middleware
node_middleware -[hidden]-> node_logic
node_core -[hidden]up-> node_presentation
node_core -[hidden]up-> node_middleware
node_core -[hidden]up-> node_logic
compiler -[hidden]down-> processor
ui -[hidden]down-> extension
refactoring -[hidden]down->analysis
```

```
@enduml
```

Appendix C

Questionnaire about Live Refactoring

The following form is a copy of the survey handed to developers that participated in the survey (*cf.* Section 5.2, p. 83).

Live Refactoring

This survey aims to evaluate two refactoring tools: "Extract Method Finder" and "Semi-automated Extractor", developed in the context of two master thesis done at FEUP: "Towards a Smart Recommender for Code Refactoring" and "Towards a Live Refactoring Recommender Based on Code Smells and Quality Metrics", respectively.

Both tools were built as a VS Code extension for JavaScript and TypeScript. They're recommendation systems that allows developers to visualize, select and apply refactoring suggestions.

No extensive knowledge about refactoring is required to answer this survey. However, basic programming experience is recommended.

All answers are anonymous and will be used exclusively for academic purposes. If you're interested and consent, your participation will be acknowledged in our work.

Thank you for your collaboration,
João Barbosa & Sérgio Salgado

(Estimated completion time: 15-30 mins)
*Obrigatório

Participant's Profile (Optional)

This section will only be used to better understand the distributions among the participants. No filling is required if you don't want to share your personal information.

1. Age

Marcar apenas uma oval.

- < 18
- 18 - 24
- 25 - 40
- > 40

2. Education

Highest completed degree of education. If currently enrolled in one, choose the highest one completed.

Marcar apenas uma oval.

- ◯ High school
- ◯ Bachelor's degree
- ◯ Master's degree
- ◯ Doctorate's degree
- ◯ Outra: _____

Technical Background

Tell us a bit more about yourself, regarding knowledge in software development and refactoring systems.

3. Programming Experience *

Marcar apenas uma oval.

- ◯ No experience
- ◯ < 1 year
- ◯ 1 - 2 years
- ◯ 3 - 5 years
- ◯ 5 - 10 years
- ◯ > 10 years

4. I am familiar with programming terms, such as classes, methods, data structures, etc. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

5. I am familiar with software development terms, such as refactoring, debugging, unit testing, etc *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

6. I am experienced in JavaScript or TypeScript. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

7. I have used Visual Studio Code before. *

Marcar apenas uma oval.

- ◯ Yes
- ◯ No

8. I have used refactoring tools or extensions before. *

Marcar apenas uma oval.

- ◯ Yes
- ◯ No

9. I have used software metric analysis tools before. *

Marcar apenas uma oval.

- ◯ Yes
- ◯ No

Part 1: Extract Method Finder

The first part of this survey aims to evaluate the "Extract Method Finder".

The application focuses on the Extract Method refactoring. We recommend this simple 2-min read if you want to learn or need a refresh about this technique: https://refactoring.guru/extract-method.

1.1. Visualization

This section focuses on the first component of the tool: visualizing refactoring recommendations.

The tool is always active and analyzing the source code as it is modified, providing real-time updates to the developer.

Suggestions are shown as colored blocks, to the left of the editor's line number. Each suggestion uses a different color.

Colors follow a gradient from green to red and are assigned according to the suggestion's severity: closer to red means the refactoring is more evident. If there are no suggestions, only a green color is shown, spanning from the beginning to the end of the block.

This section evaluates three main aspects:
- Continuous Feedback
- Multiple vs. Single Suggestion
- Refactoring Availability

Continuous Feedback

The animated image below shows a method being implemented. Our tool updates its suggestions in real-time as changes are being made to the code.

10. 1.1.1. I can quickly locate the different refactoring suggestions. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

11. 1.1.2. The color scheme allows me to quickly identify the most prominent suggestions. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

12. 1.1.3. Continuous feedback next to line numbers is too distracting for my regular development activity. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

13. 1.1.4. How frequently should suggestions be updated? *

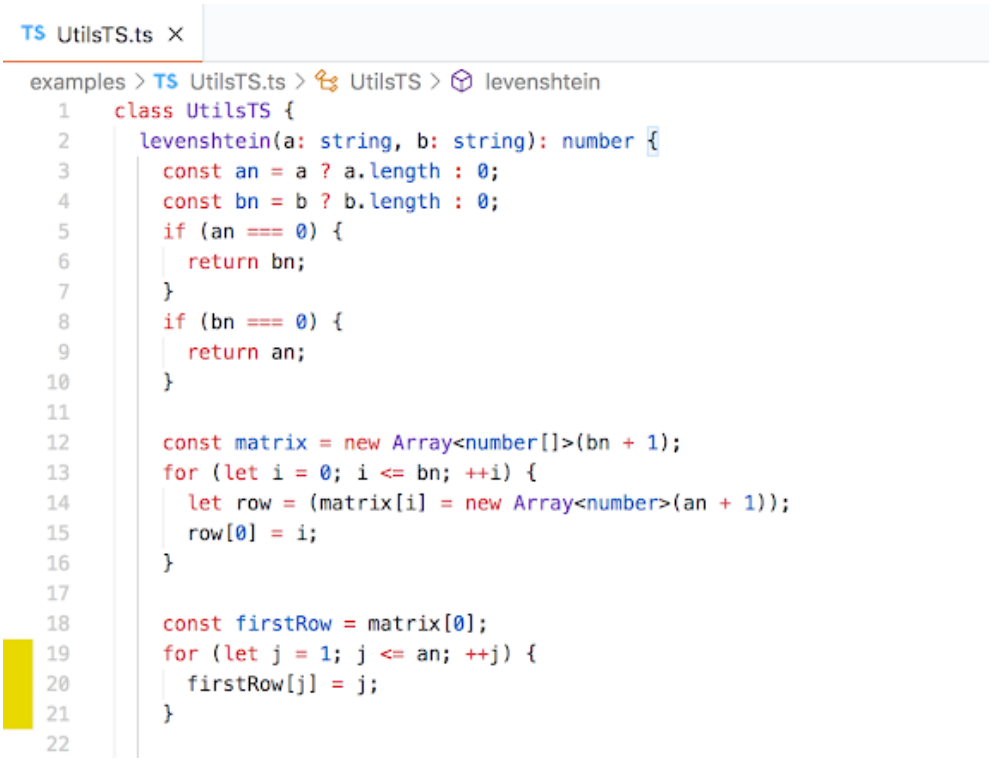
Marcar apenas uma oval.

- ◯ Once I stop typing.
- ◯ Once I change to another line.
- ◯ At fixed rates (e.g. every 1 second).
- ◯ Continuously.
- ◯ Outra: _____

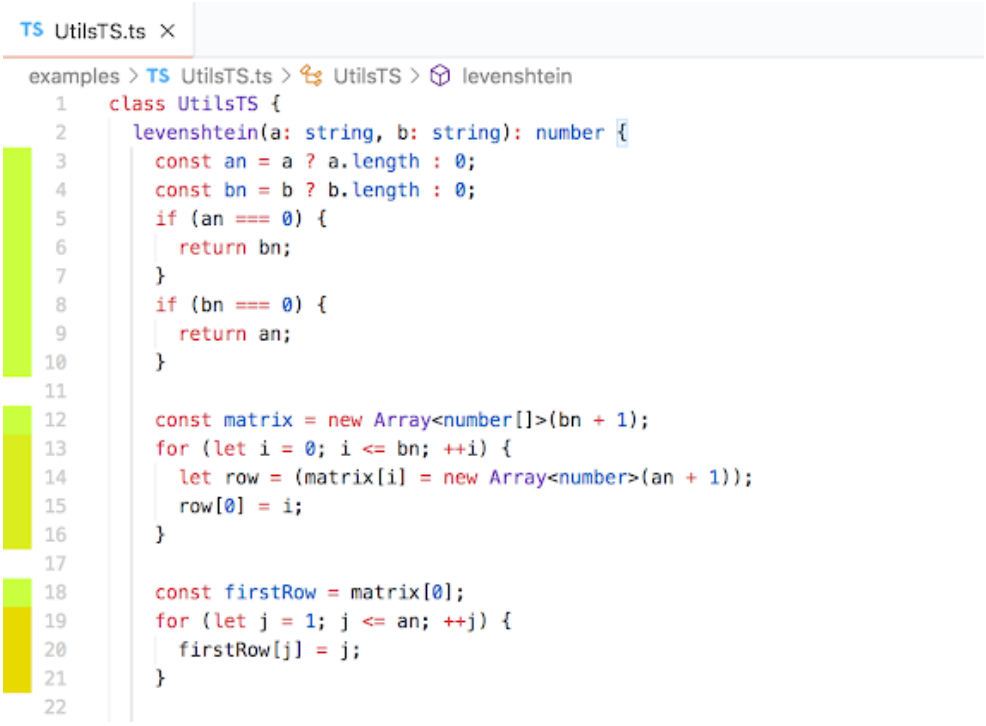
Multiple vs. Single Suggestion

Consider the two images below. The first image shows a more traditional refactoring recommender, with a single suggestion. The second image shows all suggestions available in the method 'levenshtein', of the class 'UtilsTS'.

Single suggestion



Multiple suggestions



14. 1.1.5. I see benefit in having multiple suggestions available at the same time, since it raises my awareness on the options I have. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

15. 1.1.6. I see benefit in having multiple suggestions available at the same time, since it helps me better understand the tool's reasoning. *

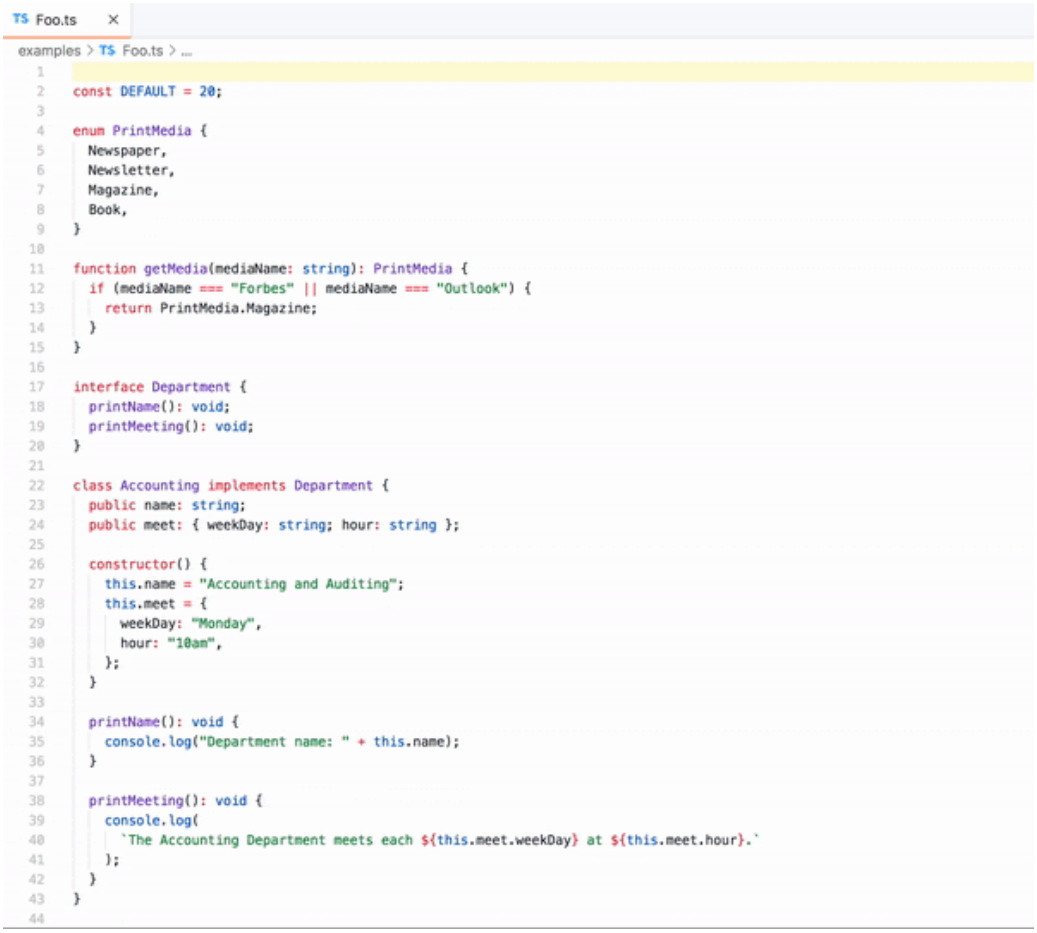
For instance, consider line 18 (second image). Do you agree with the assignment? If not, can you understand why/how it may have reached a different outcome?

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

Refactoring Availability

On the animation below, we move our cursor from top to down, across all lines of code. Our tool only shows feedback in certain blocks.



16. 1.1.7. I am able to quickly understand when refactoring feedback is available. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

17. 1.1.8. In which data structures was our tool giving feedback? *

Marcar tudo o que for aplicável.

- [ ] Global variables
- [ ] Enums
- [ ] Functions
- [ ] Interfaces
- [ ] Class declarations
- [ ] Class attributes
- [ ] Class constructors
- [ ] Class methods

18. Additional remarks

Write here any additional feedback that wasn't addressed in the previous questions.

1.2. Selection and Application

This section is concerned with the selection and application of refactoring suggestions provided by the tool.

Both actions are available through a command reachable by VS Code's Command Palette, which opens a new selection menu.

For each suggestion, the developer is able to see if it is automatically extractable and the number of lines it contains. Also, code lines of the suggestion in focus will be highlighted in the editor.

After a suggestion is accepted, the developer is prompted to write the name of the new extracted method.

19. 1.2.1. Selecting refactoring suggestions through the VS Code's Command Palette is easy and intuitive. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

20. 1.2.2. Highlighting a suggestion allows me to better perceive the lines of code it refers to. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

21. 1.2.3. I find usefulness in having a semi-automated refactoring tool. *

Semi-automated means it modifies the code automatically, but first a suggestion must be selected by the developer.

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

22. Additional remarks

Write here any additional feedback that wasn't addressed in the previous questions.

1.3. Code Quality History

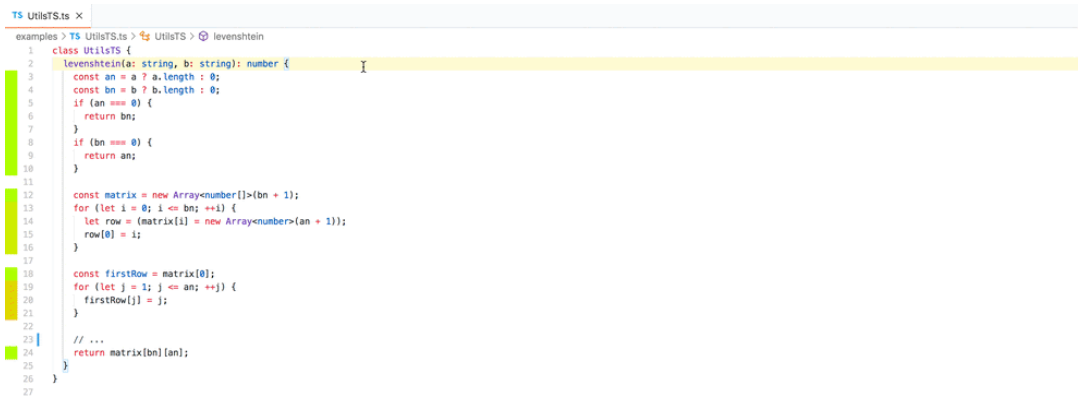
Our tool provides a second command, in VS Code's Command Palette, which allows developers to analyze the file's evolution in terms of code quality.

The command opens a webview displaying a multi-line chart. The chart updates every time the file is saved.

The chart displays a subset of software metrics, picked according to their relevance for Extract Method refactoring. Each software metric has a label and is represented by a unique color.

The developer is also able to hover recorded instances, which displays the absolute values of these metrics (colored according to the label).

(Note: knowledge about code quality metrics is not necessary to answer this section. In short, metrics are numerical values that are indicative of the software's overall complexity and/or quality. If you want to know more about the metrics we use, you can check https://www.verifysoft.com/en_halstead_metrics.html)



23. 1.3.1. Software metrics are a good indicative of overall code complexity and quality. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

24. 1.3.2. I find value in having an historical record for code quality. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

25. 1.3.3. I find value in being able to see code quality evolution as it is being modified. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

26. 1.3.4. I find value in being able to see absolute values for the code quality metrics. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

27. Additional remarks

Write here any additional feedback that wasn't addressed in the previous questions.

1.4. Final Remarks

This section of the survey is aimed at better understanding your findings on the tool's overall experience and usability, as well as possible improvements to be made.

28. 1.4.1. The tool's features were simple to use and easy to understand. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

29. 1.4.2. This tool can positively impact my development workflow. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

30. 1.4.3. I would consider using this tool. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

31. 1.4.4. Based on what you saw on this survey, which features would you say were the most important? *

Marcar tudo o que for aplicável.

- [ ] Live feedback on refactoring needs.
- [ ] Refactoring severity based on color gradient.
- [ ] Multiple refactoring suggestions.
- [ ] Semi-automated refactoring application.
- [ ] Code quality history analysis.
- [ ] Outra: _____

32. 1.4.5. Based on what you saw on this survey, which features could be improved? *

Marcar tudo o que for aplicável.

- [ ] Live feedback on refactoring needs.
- [ ] Refactoring severity based on color gradient.
- [ ] Multiple refactoring suggestions.
- [ ] Semi-automated refactoring application.
- [ ] Code quality history analysis.
- [ ] Outra: _____

33. Additional remarks

Write here any additional feedback that wasn't addressed in the previous questions.

_____

_____

_____

_____

_____

Part 2: Semi-automated Extractor

This section of the survey showcases our second tool, presented below, which consists of a refactoring tool that supports the execution of three refactorings based on extraction: Extract Method, Extract Class and Extract Variable. These refactorings are executed based on background live analysis of quality metrics, whose values can indicate the presence of 'code smells' lying within the source code.

A code smell is a characteristic in code, which usually indicates the presence of a deeper problem. If you need to refresh your memory about what a code smell is, we recommend reading the following:

What is a code smell? - https://martinfowler.com/bliki/CodeSmell.html
Types of code smells. - https://sourcemaking.com/refactoring/smells

2.1. Interface

This section is dedicated to questions relative to the tool's interface and respective usability. The images below show the tool's interface presenting a report about the results retrieved from the analysis of the 'Observable.ts' source code file, in both the compressed and expanded view. If you wish to view the file, although not needed to answer the questions, it is located at: https://github.com/dojo/intern-only-dojo/blob/master/src/Observable.ts

The interface is divided into three sections:
1. File-related metrics: area where line, node and Halstead metrics (not shown in the picture, see below for more information) are shown. This section shows metrics calculated across the entire code file.
2. Method metrics: area where metrics relative to each individual method contained in the file is shown. Small colored squares reflect the metric values on each method, without the need to expand this area of the interface.
3. Refactoring suggestions: area which shows the amount of suggestions found for each kind of supported refactoring, with more information about the top suggestion when clicked.

Halstead metrics identify measurements of software properties and the relationships between them and are inferred using the amount of operands and operators in a given code structure. Although knowledge about them is not needed to answer the following questions, you can read more about them at https://www.geeksforgeeks.org/software-engineering-halsteads-software-metrics/

Background colors are coded to show the severity of each metric, ranging from red (for critical values), to green (near-optimal/optimal values).

If needed, full resolution pictures are provided at:
- Compressed View: https://i.imgur.com/tupGifx.png
- Expanded View: https://i.imgur.com/VC4yLZ3.png

Interface compressed report

Live Refactoring

Showing results for file: src\middlewares\middleware.ts

Metric	Current Value
Line-related Metrics	
Number Of Lines	153
Number Of Comment Lines	0
Number Of Code Lines	119
Number Of Blank Lines	34
Node-related Metrics	
Number Of For Statements	3
Number Of While Statements	0
Number Of Conditional Statements	10
Number Of Classes	6
Number Of Methods	15
Number Of Functions	0
Number Of Class Fields	0

Method Metrics

Function Metrics

Refactoring Suggestions

Extract Method Suggestions (14 suggestions)

Extract Class Suggestions (0 suggestions)

Extract Variable Suggestions (12 suggestions)

Interface expanded report

Live Refactoring

Showing results for file: src\middlewares\middleware.ts

Metric	Current Value
Line-related Metrics	
Number Of Lines	153
Number Of Comment Lines	0
Number Of Code Lines	119
Number Of Blank Lines	34
Node-related Metrics	
Number Of For Statements	3
Number Of While Statements	0
Number Of Conditional Statements	10
Number Of Classes	6
Number Of Methods	15
Number Of Functions	0
Number Of Class Fields	0

Method Metrics

Middleware.bypassCrawlers	
Maintainability	85.83
Method Complexity	1.00
Number of Statements	1.00
LCOM	0.00
Middleware.getExceptionResources	
Maintainability	85.83
Method Complexity	1.00
Number of Statements	1.00
LCOM	0.00
Middleware.getHandler	
Maintainability	82.24
Method Complexity	1.00
Number of Statements	1.00
LCOM	0.00
Middleware.getName	
Maintainability	94.66
Method Complexity	1.00
Number of Statements	0.00
LCOM	0.00
MandatoryQueryParams.getName	
Maintainability	85.83
Method Complexity	1.00
Number of Statements	1.00
LCOM	0.00
ValidateParams.getName	
Maintainability	85.83
Method Complexity	1.00
Number of Statements	1.00
LCOM	0.00
NamespaceValidator.getName	
Maintainability	85.83
Method Complexity	1.00
Number of Statements	1.00
LCOM	0.00
Middleware.handler	
Maintainability	89.41
Method Complexity	1.00
Number of Statements	0.00

LCOM	0.00

MandatoryQueryParams.handler	
Maintainability	60.18
Method Complexity	4.00
Number of Statements	9.00
LCOM	0.00

ValidateParams.handler	
Maintainability	52.39
Method Complexity	7.00
Number of Statements	19.00
LCOM	0.00

NamespaceValidator.handler	
Maintainability	64.95
Method Complexity	3.00
Number of Statements	7.00
LCOM	0.00

Middleware.isProd	
Maintainability	84.25
Method Complexity	1.00
Number of Statements	1.00
LCOM	0.00

Middleware.register	
Maintainability	81.11
Method Complexity	1.00
Number of Statements	1.00
LCOM	0.00

AppMiddleware.register	
Maintainability	58.25
Method Complexity	4.00
Number of Statements	11.00
LCOM	0.00

ResourceMiddleware.register	
Maintainability	79.77
Method Complexity	1.00
Number of Statements	1.00
LCOM	1.00

Function Metrics

Refactoring Suggestions

Extract Method Suggestions (14 suggestions)

In order to execute the following refactor operation automatically, please execute the following command: Live Refactoring: Extract Method

Our algorithm considers the extraction of the following fragment in method *handler* to be the most valuable of all found candidates:

```
const queryParams = Object.entries(this.resource.getQueryParams());
const query = req.query;
for(let [n, p] of queryParams) {
            if(n in query) {
                const param = new p(query[n] as string)
                if(!param.isValid()) throw new InvalidParamException(n);
                params[n] = param;
            }
        }
```

This fragment to be extracted has the following metrics:

- **Method Complexity** (method-wide metric. the lower, the better): **4.00**
- **Number of Statements** (method-wide metric. the lower, the better): **8.00**
- **Lack of Cohesion in Methods** (class-wide metric. the lower, the better): **0.00**

and changes the highest values of these metrics on all methods in class *ValidateParams* to the following:

- **Method Complexity** (method-wide metric. the lower, the better): **4.00**
- **Number of Statements** (method-wide metric. the lower, the better): **11.00**
- **Lack of Cohesion in Methods** (class-wide metric. the lower, the better): **0.00**

Extract Class Suggestions (0 suggestions)

Extract Variable Suggestions (12 suggestions)

34. 2.1.1. Most of the metric values on this file are within healthy limits. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

35. 2.1.2. The color scheme used allows me to quickly identify problematic methods without spending time looking through them individually. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

36. 2.1.3. Despite the long list of metrics, I am able to process the information fairly quickly. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

37. 2.1.4. The first section of the interface, file-related metrics, is emphasized enough and its information is of easy interpretation. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

38. 2.1.5. The second section of the interface, method-related metrics, is emphasized enough and its information is of easy interpretation. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

39. 2.1.6. The third section of the interface, refactoring suggestions, is emphasized enough and its information is of easy interpretation. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

40. 2.1.7. The amount of information presented is overwhelming. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

41. 2.1.8. The information given is enough for me to know if any action is needed on the code. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

42. 2.1.9. The preview colored squares on the 'Method Metrics' section allows for even faster analysis of methods. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

43. 2.1.10. This report format of presentation fits the purpose of this tool. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

Post-Changes Interface

When changes in the file occur, the tool displays the two more recent values, with the most recent being on the right, storing them for the current section.
The image below shows the impact of an Extract Method refactoring on the metric values for the same file as the previous image.

If necessary, the full resolution image is available here: https://i.imgur.com/FR7v7r7.png

Post-changes impact on the interface report

Metric ValidateParams.handler	Value Before	Current Value
Maintainability	52.39	57.95
Method Complexity	7.00	4.00
Number of Statements	19.00	12.00
LCOM	0.00	0.00

44. 2.1.11. I can clearly understand the values on the left side correspond to old values, while the ones on the right correspond to the newest ones. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

45. 2.1.12. Only showing the two more recent values of each metric for a programming session is enough. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

2.2. Visual Studio Code

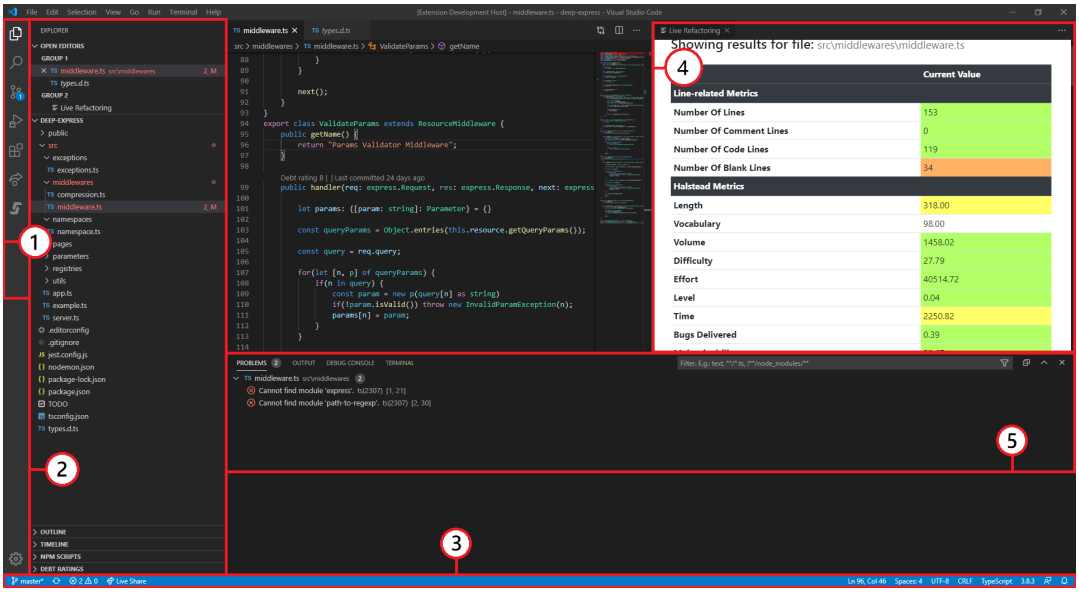
This section contextualizes how our tool is incorporated into the VSCode UI.

The image below shows five contribution points in the VSCode UI, where implementing UI fragments is the most common. Their nomenclature is as follows:

1 - Tree View Container
2 - Tree View
3 - Status Bar
4 - Webview
5 - Diagnostics Report

If necessary, the full resolution image is available here:
https://i.imgur.com/8HOMRtV.png

VSCode main interface



46. 2.2.1. The webview interface is distracting. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

47. 2.2.2. I would prefer for the information given by the tool to be shown in other contribution points, eliminating the need for a webview. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

48. 2.2.3. Extensions whose features are scattered thoughout the VSCode UI can be confusing to use. *

Marcar apenas uma oval.

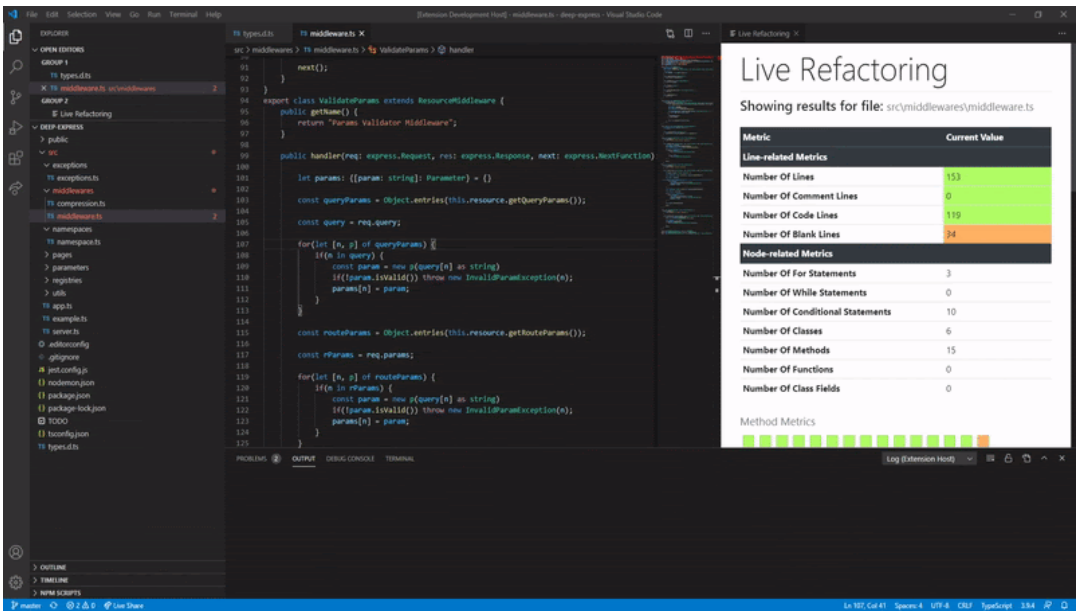
	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

Workflow

The video below shows the usage of the tool, from the moment metrics are quickly scanned, to the assessment and execution of the top Extract Method suggestion. Apart from Extract Method, our tool directly supports the Extract Class and Extract Variable refactorings.

The file shown, 'middleware.ts', contains 6 classes, 14 methods across them, with around 150 lines of code.

Workflow on the Semi-automated Extractor



49. 2.2.4. The workflow to execute the top Extract Method suggestion looked simple. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

50. 2.2.5. Instructions presented on the interface on how to execute a refactoring are clear. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

51. 2.2.6. The processing time of post-changes metrics and refactoring suggestions (around five seconds for this file) is too high. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

52. 2.2.7. The webview is distracting for the programmer. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

53. 2.2.8. I would prefer a button in the interface to run the refactoring, instead of manually executing a command. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

2.3. Refactoring

Refactoring is a common software engineering practice, where the programmer changes the internal aspect of a system, without changing its external behavior.

As you may have noticed from the previous video, apart from a metrics report, our tool supports the automated execution of three refactorings: Extract Method, Extract Class and Extract Variable. These refactorings are automatically evaluated according to the values of a specific set of quality metrics (some not shown on the report).

54. 2.3.1. All three supported refactorings (Extract Method, Extract Class and Extract Variable) are refactorings I use regularly. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

55. 2.3.2. A tool which automatically finds the best refactoring of each kind, for any file and in close to real time, is something I see value in. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

56. 2.3.3. Having semi-automated execution of the best found refactorings is useful, i.e. the user triggers the execution, but execution is performed automatically. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

57. 2.3.4. Having a preview of what the refactoring is going to change before executing is useful. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

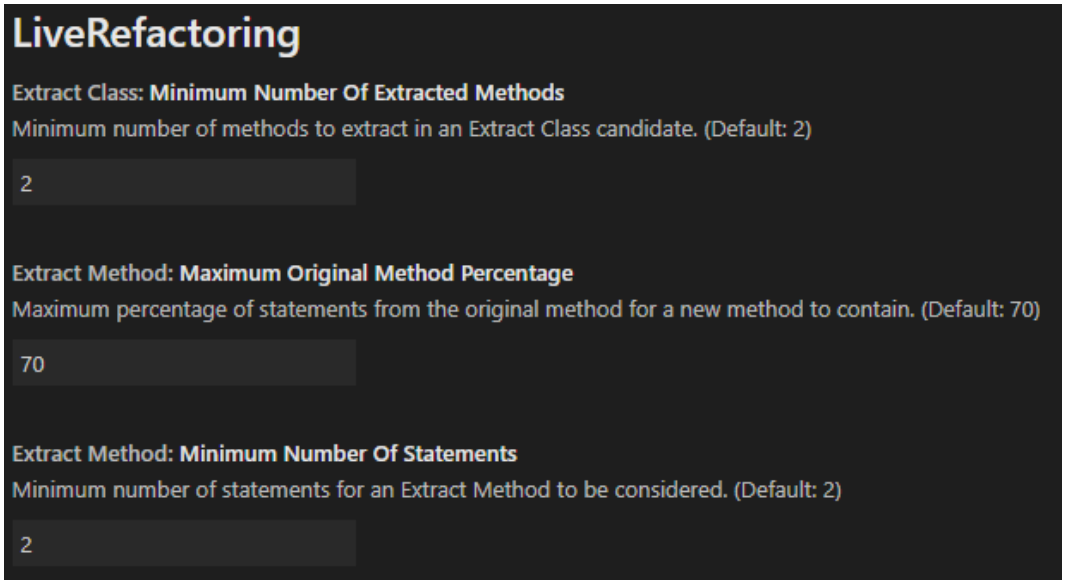
Customization

As each project's context is different from one another, our tool allows programmers to customize the refactoring suggestions and the information shown on the interface report.

The image below shows some of the options the user can customize to alter the tool's behavior.

If necessary, the full resolution image is available here: https://i.imgur.com/ALrKnai.png

Options menu



58. 2.3.5. Offering user customization for refactoring tools is useful. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

59. 2.3.6. The description of each option and its impact on the suggestions is clear. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

2.4. Final remarks

The final section of the survey aims to understand your opinion on this tool and which changes should be made to make it better.

60. 2.4.1. This tool's features are simple and fast to understand. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

61. 2.4.2. This tool can positively impact my development workflow. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

62. 2.4.3. I would use this tool. *

Marcar apenas uma oval.

	1	2	3	4	5	
Highly disagree	◯	◯	◯	◯	◯	Highly agree

63. 2.4.4. Based on what you saw on this survey, what would you say were the best features of this tool? *

Marcar tudo o que for aplicável.

- [ ] Metrics and refactoring report (Webview interface)
- [ ] Comparison between old and new metrics
- [ ] Live computation of metrics
- [ ] Live evaluation of refactoring opportunities
- [ ] Semi-automated refactoring execution
- [ ] Extension customization
- [ ] Outra: _____

64. 2.4.5. Based on what you saw on this survey, what would you say were the features which could be improved upon? *

Marcar tudo o que for aplicável.

- [ ] Metrics and refactoring report (Webview interface)
- [ ] Comparison between old and new metrics
- [ ] Live computation of metrics
- [ ] Live evaluation of refactoring opportunities
- [ ] Semi-automated refactoring execution
- [ ] Extension customization
- [ ] Outra: _____

All done :)

Thank you for your time and valuable feedback!

And remember, refactor early and continually.