

UNIVERSITY OF PORTO • FACULTY OF ENGINEERING

Doctoral Thesis

A Live Environment for Continuous Software Inspection and Refactoring

Author:

Sara Filipa Couto Fernandes

Supervisor:

Ademar Aguiar, PhD
Associate Professor

Co-Supervisor:

André Restivo, PhD
Assistant Professor

Doctoral Program in Informatics Engineering

May 19, 2025



FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

A Live Environment for Continuous Software Inspection and Refactoring

Sara Filipa Couto Fernandes

Doctoral Program in Informatics Engineering

May 19, 2025

Abstract

Writing software is hard; reviewing, changing, or adapting old software is even more challenging. A bad design can quickly lead to rotting software, with each modification heading to a rigid and fragile codebase. Evolution may become extremely costly if we do not refactor code at the right time.

Often, developers choose to refactor too late when most symptoms are already impossible to ignore. They check their source code, looking for code smells and the most appropriate refactorings, trying to bring sanity into their design. They then realize they need help from specialized, overly complex, and hard to use tools. If only they tended to refactor their code sooner, they might have gained more peace of mind earlier.

In this process of writing and evolving code, changing hats from coding to refactoring should be done often, if not constantly. We argued that a live refactoring environment, which presents refactorings in real-time, would help developers be continually aware of possible refactoring opportunities, making it easier to apply them earlier and faster when the codebase is still under control. We developed such a live environment by considering and evaluating in real-time several code-quality metrics, detecting smells, providing feedback, and presenting possible refactoring candidates unobtrusively and elegantly to the developers — all without leaving the comfort of their development environment.

By enhancing an existing IDE with live refactoring capabilities, we showed that we can help developers understand, adapt, and maintain their systems in a more controlled and prompt fashion, allowing them to produce better code faster.

The following contributions resulted from this work: (i) an extensive analysis of the state-of-the-art on the main topics of our project, (ii) a live refactoring environment capable of continuously inspecting code to detect, suggest, and apply refactorings, (iii) an empirical validation using different approaches that helped us gathering data that allowed to confirm our hypothesis, and (iv) a set of scientific publications validating all the work done.

While our work presents significant contributions, there are areas for further exploration. We could enhance specific aspects of our Live Refactoring Environment, including broader refactoring support or reducing processing time for complex code. Moreover, future work could also involve predicting the impact of refactorings on quality metrics and enhancing usability, including tests with color-blind users.

Resumo

Desenvolver *software* é complicado. Revê-lo, alterá-lo ou adaptá-lo é ainda mais difícil. Um mau *design* pode, rapidamente, levar à degradação do *software*, em que cada modificação pode criar uma *codebase* rígida e frágil. Neste sentido, evoluir um sistema destes torna-se quase impossível e a única solução é aplicar *refactorings*, no momento certo, para o melhorar.

Na maioria das vezes, os programadores decidem refactorizar o seu código em etapas avançadas do desenvolvimento do mesmo, o que faz com que a maioria dos problemas nele presentes sejam já impossíveis de serem ignorados. Assim, os programadores tentam inspecionar o seu *source code* em busca de *code smells* e possíveis *refactorings*, de forma a trazer mais clareza ao *design* dos seus programas. É aqui que os programadores se apercebem que necessitam de ajuda, na forma de ferramentas específicas, que por vezes são excessivamente complexas e difíceis de usar. Apercebem-se, igualmente, que caso tivessem cuidado do seu código atempadamente talvez não tivessem estes problemas e poderiam continuar a desenvolver os seus sistemas sem quaisquer “dores de cabeça”.

Neste processo de escrever e evoluir código, a metáfora de “*mudança de chapéus*” entre *coding* e a aplicação de *refactorings* deve ser adotada frequentemente, se não constantemente. Neste sentido, hipotisámos que um *live refactoring environment*, em que são apresentadas sugestões de *refactoring* em tempo real, é capaz de ajudar os programadores a estarem continuamente cientes das oportunidades de *refactoring*, tornando-as mais fáceis de executar quando ainda se tem o *source code* sob controlo. Este ambiente de desenvolvimento tem em conta diversas métricas de qualidade de código, em tempo-real. Com esta abordagem, verificámos ser possível detetar *code smells*, fornecendo *live feedback* e apresentando visualmente possíveis candidatos de *refactoring*, de forma discreta e elegante para os programadores, sem estes terem de deixar o conforto do seu IDE.

Mostramos que, ao aprimorar um IDE já existente recorrendo a mecanismos de *refactoring*, em tempo-real, podemos ajudar os programadores a compreender, adaptar e manter o seu *software* de uma forma mais controlada e rápida, permitindo-lhes produzir melhor código, com maior qualidade, mais rapidamente.

Neste sentido, com este trabalho, resultaram as seguintes contribuições: (i) uma extensa análise do estado da arte referente aos tópicos principais deste projeto, (ii) um *live refactoring environment* capaz de inspecionar o código para identificar, sugerir e aplicar possíveis *refactorings*, (iii) uma validação empírica composta por diferentes abordagens que nos permitiu obter dados que suportassem a veracidade da nossa abordagem e (iv) um conjunto de publicações científicas que resumem todo o trabalho efetuado.

Embora este trabalho apresente contribuições significativas, existem áreas que podem ser futuramente exploradas. Podemos melhorar aspetos específicos do nosso *Live Refactoring Environment*, como incluir suporte para mais *refactorings* ou reduzir o tempo de processamento para código mais complexo. Além disso, trabalhos futuros podem envolver a previsão do impacto destes *refactorings* nas métricas de qualidade, podendo ainda haver o aperfeiçoamento da usabilidade da ferramenta, incluindo testes que incluam utilizadores daltónicos.

Acknowledgements

A special acknowledgment goes to my exceptional Supervisor, Ademar Aguiar, and Co-Supervisor, André Restivo. Their unwavering commitment to sharing their knowledge and providing guidance, especially during my moments of existential uncertainty, has been invaluable.

I want to extend my heartfelt gratitude to the esteemed Faculty of Engineering at the University of Porto. I sincerely appreciate the dedicated professors and staff members who have enriched my Ph.D. journey. Among all, I have to mention the names of Sandra Reis and Pedro Silva. Sandra was undoubtedly an essential help, always ready to listen, help, and laugh at my craziness. Pedro, always very busy, never denied help, even when help seemed impossible.

Additionally, my gratitude extends to Inês Teixeira and Liliana Antão for their invaluable advice and shared ideas. I owe them a debt of gratitude for their constant support through my mood swings and their constant availability whenever I needed assistance, regardless of the time.

I additionally convey my deepest appreciation to my cherished family, especially my mother, father, and brother, for their unending support for my aspirations and choices.

I am also profoundly grateful to my steadfast friends and colleagues who have stood by me over these years. Your friendship, motivation, mutual support, shared laughter, and tolerance of my idiosyncrasies have been essential in my journey.

I also have to thank my students for all the hours they allowed me to share knowledge and all the help they provided along this path. I will never forget the help they gave me by participating in the multiple experiences that were made, as well as all the hours of laughter and relaxation that allowed me to escape from the stress of this long journey.

Finally, I must thank FCT - Fundação para a Ciência e Tecnologia for the support they have given me over the last few years through the doctoral grant they have awarded me. Without FCT, it would have been impossible to face the difficulties that result from a doctoral program.

Sara Fernandes

“I am Iron Man”

Robert Downey Jr., Iron Man, Marvel

Contents

1	Introduction	1
1.1	Context	2
1.2	Motivation	4
1.3	Problem	4
1.4	Objectives	5
1.5	Strategy	6
1.6	Main Contributions	6
1.7	Document Structure	7
2	Literature Review	9
2.1	Software Inspection	10
2.1.1	Software Quality Assessment	10
2.1.2	Changes History Analysis	11
2.1.3	Code Review	12
2.1.4	Discussion and Open Issues	13
2.2	Code Smells	15
2.2.1	Heuristics	16
2.2.2	Source Code Analysis	16
2.2.3	Artificial Intelligence Techniques	19
2.2.4	Slicing Techniques	20
2.2.5	Discussion and Open Issues	20
2.3	Refactoring	23
2.3.1	Source Code Analysis	26
2.3.2	Code Metrics and Heuristics	30
2.3.3	Change History Analysis	33
2.3.4	Artificial Intelligence Techniques	36
2.3.5	Game Theory	38
2.3.6	User Direct Action	39
2.3.7	Slicing Techniques	40
2.3.8	Combined Techniques	40
2.3.9	Discussion and Open Issues	41
2.4	Code Quality Metrics	48
2.4.1	Assess Quality Attributes	49
2.4.2	Abstract Syntax Tree	51
2.4.3	Software Revisions Analysis	52
2.4.4	Models Definition	52
2.4.5	Discussion and Open Issues	53
2.5	Software Visualization	55

2.5.1	Code-Line-Level Visualization	56
2.5.2	Class-Level Visualization	57
2.5.3	Architecture-Level Visualization	58
2.5.4	Other Approaches	60
2.5.5	Discussion and Open Issues	61
2.6	Liveness	62
2.6.1	Live Programming Environments	63
2.6.2	Live Programming Languages	68
2.6.3	Live Tools and Approaches	70
2.6.4	Discussion and Open Issues	73
2.7	Summary	77
3	Research Problem	79
3.1	Open Issues	80
3.2	Guidelines	81
3.3	Thesis Statement	82
3.4	Research Questions	82
3.5	Research Scope	83
3.6	Methodology	84
3.6.1	Research Methods	84
3.6.2	Proposed Development Approach	86
3.6.3	Validation Approach	86
3.7	Previous Research	88
3.8	Main Contributions	88
3.9	Summary	89
4	Live Refactoring Environment	91
4.1	Development Environment	92
4.2	Refactoring Detector and Applier	93
4.2.1	Architecture	93
4.2.2	Code Quality Metrics	95
4.2.3	Supported Refactorings	97
4.2.4	Artificial Intelligence	110
4.3	Refactoring Visualizer	116
4.4	Liveness Features	119
4.5	Limitations	120
4.6	Summary	120
5	Empirical Experiments	123
5.1	Live Extract Method Experiment	124
5.1.1	Experimental Groups	125
5.1.2	Programming Tasks	125
5.1.3	Outcomes	126
5.1.4	Results and Discussion	127
5.2	Live Refactoring Experiment	143
5.2.1	Project and Participants	143
5.2.2	Outcomes	144
5.2.3	Results and Discussion	145
5.3	Automated Experiment	164

5.3.1	Crawler Implementation	164
5.3.2	Automated Refactorings	166
5.3.3	Outcomes	171
5.3.4	Results and Discussion	171
5.4	Answering the Research Questions	184
5.5	Threats to Validity	187
5.5.1	Internal Validity	187
5.5.2	External Validity	188
5.5.3	Construct Validity	189
5.5.4	Conclusion Validity	189
5.6	Summary	190
6	Conclusions	191
6.1	Thesis Statement	191
6.2	Research Questions	192
6.3	Outcomes	194
6.3.1	Literature Review	194
6.3.2	Live Refactoring Environment	195
6.3.3	Empirical Experiments	195
6.3.4	Scientific Publications	196
6.4	Future Work	196
	References	199
A	Scientific Publications	225
A.1	Publications from this Research Project	225
A.1.1	Live Software Inspection and Refactoring	225
A.1.2	A Live Environment for Inspection and Refactoring of Software Systems	226
A.1.3	A Live Environment to Improve the Refactoring Experience	226
A.1.4	LiveRef: a Tool for Live Refactoring Java Code	227
A.1.5	Towards a Live Environment for Code Refactoring	227
A.2	Publications from Previous Work	228
A.2.1	Supporting Software Development through Live Metrics Visualization	228
A.2.2	Helping Software Developers through Live Software Metrics Visualization	228
A.3	Unpublished Work	229
A.3.1	Empirical Evaluation of a Live Environment for Extract Method Refactoring	229
A.4	Co-Supervised Master Theses	230
A.4.1	Towards a Smart Recommender for Code Refactoring	230
A.4.2	Towards a Live Refactoring Recommender Based on Code Smells and Quality Metrics	230
A.4.3	Mining Software Repositories to Improve Refactoring Assistants	231
A.4.4	Spatialized Live Refactoring	231
B	User Guide	233
B.1	Installation	233
B.2	Activation	234
B.2.1	Tool Workflow	234
B.2.2	Tool Configuration	236

C	Experiments Artifacts	239
C.1	Experiments Protocol	239
C.2	Participation Consent Declaration	259
C.3	Live Extract Method Experiment	261
C.3.1	Experiment Guide	261
C.3.2	Experiment Questionnaire	282
C.4	Live Refactoring Experiment	298
C.4.1	Experiment Guide	298
C.4.2	Experiment Questionnaire	309
C.5	Automated Experiment	318
C.5.1	Questionnaire on Extract Method	318
C.5.2	Questionnaire on Extract Class	321
C.5.3	Questionnaire on Extract Variable	324
C.5.4	Questionnaire on Move Method	326
C.5.5	Questionnaire on Introduce Parameter Object	329
C.5.6	Questionnaire on Replace Inheritance with Delegation	331

List of Figures

1.1	Software Development Life-Cycle	2
2.1	Architecture visualization interface to inspect a software system	11
2.2	Comments representing the reviews made on source code	12
2.3	Visual components of ICC-Inspect	13
2.4	InCode automatically marking code design problems	17
2.5	The different types of code cloning	18
2.6	AntiCopyPaster identifying Extract Method opportunities	28
2.7	An example of a phase count view	29
2.8	ExtC dependency graph	31
2.9	Example of the visual interface provided by RAID	36
2.10	Visualization of different refactoring candidates	38
2.11	Drag-and-drop of an Extract Method refactoring	39
2.12	Identification of Feature Envy code smells using JDeodorant	40
2.13	Tech Debt Tracker results	51
2.14	3D visualization of a software system	55
2.15	Comparison between SeeSoft and Sv3D	57
2.16	An example of the Class Blueprint technique.	58
2.17	Code city applying the city metaphor	59
2.18	An example of a geon visualization methodology	60
2.19	Health screen of different classes and possible code smells	60
2.20	Visual representation of a Message Chain smell	61
2.21	Liveness hierarchy proposed by Tanimoto	63
2.22	Example of a program being executed using Circa	64
2.23	Moon development environment	65
2.24	Warnings launched when there is a gap between design and implementation	66
2.25	PacMan being developed on SuperGlue	69
2.26	Quokka interface showing the interactions between endpoints	71
4.1	Workflow synthesizing the developed Live Refactoring Environment	92
4.2	Architecture of the developed Live Refactoring Environment	94
4.3	An example of a graph	115
4.4	An example of a dendogram	115
4.5	Example of the visual approach used by Barbosa	116
4.6	Color gradient used in our visual solution	117
4.7	An example of our visual approach being used on IntelliJ	117
4.8	An example of our refactoring menu	118
4.9	Visual approach for colorblind users	118

4.10	Conversion of the color gradient to the symbols' scale	118
5.1	Synthesis of the groups that participated in our first empirical experiment.	125
5.2	Characterization of the participants from the first experiment	129
5.3	Metrics evolution by experimental group on Task 2	130
5.4	Metrics evolution by experimental group on Task 3 File 1	131
5.5	Metrics evolution by experimental group on Task 3 File 2	132
5.6	Answers to our SUS Questionnaire on the first experiment	140
5.7	Answers to other usability questions on our tool on the first experiment	141
5.8	Participants satisfaction on the first experiment	143
5.9	Characterization of the participants from the final experiment	152
5.10	Metrics evolution over the four-week project development time	155
5.11	Answers to our SUS Questionnaire on the final experiment	162
5.12	Answers to other usability questions on our tool on the final experiment	162
5.13	Metrics evolution in each project	182
5.14	Metrics evolution in each project (continuation)	183
5.15	Personal opinions on the refactorings applied	184
6.1	Mockup of a possible interface to display specific code quality metrics	198
B.1	How to install LiveRef	233
B.2	Live Refactoring menu when using IntelliJ 2021	234
B.3	Popup dialog and Live Refactoring menu when using IntelliJ 2022	234
B.4	Displaying refactorings using multiple colors	235
B.5	Refactoring menu listing multiple refactoring candidates	235
B.6	Message displayed when no refactorings are found	235
B.7	Option to stop the tool when using IntelliJ 2021	236
B.8	Option to stop the tool when using IntelliJ 2022	236
B.9	Configuration Menu of LiveRef	236

List of Tables

2.1	Comparison of software inspection approaches and tools	14
2.2	Comparison of code smells detection approaches and tools	21
2.3	Code smells and their respective refactoring solutions	24
2.4	Linkage between metrics and design properties	31
2.5	Measure of each quality attribute	31
2.6	Comparison between refactoring approaches and tools	43
2.7	Metrics supported by CodeMR divided into four categories	50
2.8	Comparison between code quality metrics approaches and tools	53
2.9	Summary of possible visualization methods and metaphors	56
2.10	Comparison between software visualization approaches and tools	62
2.11	Comparison between live environments	74
2.12	Comparison between live languages	75
2.13	Comparison between live programming tools	76
4.1	Code quality metrics supported by our approach, grouped by type	97
4.2	Binary representation of the code example	111
4.3	An example of a distance matrix	114
4.4	Number of Extract Method and Extract Class candidates and the time needed to present them by approach	116
4.5	Summary of the refactorings and metrics included in this research project	121
5.1	Average number of refactorings applied and time efficiency by experimental group	132
5.2	Null and alternative hypotheses of the tests used to compare the experimental groups from the first experiment	133
5.3	Shapiro-Wilk test on the data collected on the preliminary experiment	133
5.4	Wilcoxon Rank Sum test results after applying the Bonferroni Correction method to check if the experimental groups from the preliminary experiment were symmetric	135
5.5	Null and alternative hypotheses used on the Permutation tests for the preliminary experiment	136
5.6	Permutation tests performed to compare the experimental groups of the preliminary experiment after applying the Bonferroni Correction method	138
5.7	Usability questions used to assess reliability	141
5.8	Cronbach's alphas measured between Group A1 and A2	142
5.9	Shapiro-Wilk test on the data collected on every experimental group	146
5.10	Null and alternative hypotheses used on several tests used to characterize our experimental groups and their data	146
5.11	Spearman Coefficient test to verify if the final code quality and project grade were related to the groups' GPA when comparing all groups	147

5.12	Spearman Coefficient test to verify if the final code quality and project grade were related to the groups' GPA when restricting the groups' GPA	147
5.13	Number of participants and their GPA per experimental group	150
5.14	Number of refactorings and undo operations applied by type and experimental group	152
5.15	Final evaluation of the projects developed on our experiment per experimental group	156
5.16	Shapiro-Wilk test results when the GPA is higher or equal to 14	156
5.17	Null and alternative hypotheses used on the tests used to compare the experimental groups of the second experiment	158
5.18	Wilcoxon Rank Sum test results after applying the Bonferroni Correction method to check if the experimental groups were equivalent	158
5.19	Null and alternative hypotheses used on the Permutation tests	159
5.20	Results collected from applying the Permutation test and the Bonferroni Correction method to our data	160
5.21	Usability questions used to assess reliability on the final experiment	163
5.22	Cronbach's alphas measured to assess the reliability of our refactoring tool	164
5.23	Number of refactorings applied in each project	176
5.24	applied in each file of each project	177
5.25	Average improvement of each metric per refactoring type	179
5.26	Average improvement of each metric	180
A.1	Summary of the publications resulting from this research project	225
A.2	Summary of the publications resulting from previous work	228
A.3	Summary of the unpublished papers resulting from this research project	229
A.4	Co-supervised Master's theses	230

List of Algorithms

1	Identification possible extractable code fragments	100
2	Identification of <i>Extract Method</i> candidates	101
3	Identification of <i>Extract Class</i> candidates	103
4	Assessing all non-void calls from a method	104
5	Identification of <i>Extract Variable</i> candidates	104
6	Identification of <i>Move Method</i> candidates	106
7	Identification of <i>Replace Inheritance with Delegation</i> candidates	108
8	Identification of <i>Introduce Parameter Object</i> candidates	108
9	Identification of <i>String Comparison</i> candidates	109
10	<i>String Comparison</i> refactoring applier.	109
11	Sequence of code that starts a new refactoring inspection whenever a refactoring or code action happens	120
12	Sequence of code that starts a new refactoring inspection whenever we change the focus between text files	120
13	Fetching and Cloning GitHub Repositories.	165
14	Crawling the GitHub projects.	166
15	<i>Extract Method</i> applier created for our robotized tool.	167
16	<i>Extract Variable</i> applier created for our robotized tool.	168
17	<i>Move Method</i> applier created for our robotized tool.	169
18	<i>Replace Inheritance with Delegation</i> applier created for our robotized tool. . . .	170
19	<i>Introduce Parameter Object</i> applier created for our robotized tool.	170

Abbreviations

ANA	Average Number of Ancestors
API	Application Programming Interface
AST	Abstract Syntax Tree
AVG	Average
CAM	Cohesion Among Methods of Class
CC	Cyclomatic Complexity
CDM	Call-based Dependence between Methods
CIS	Class Interface Size
CSS	Cascading Style Sheets
CSM	Conceptual Similarity between Methods
DAM	Data Access Metrics
DCC	Direct Class Coupling
DNDRefactoring	Drag-and-Drop Refactoring
DSL	Domain-Specific Languages
DSC	Design Size in Classes
EC	Extract Class
EEDETECTOR	Extra Edit Anomaly Detector
EM	Extract Method
EV	Extract Variable
FBP	Flow-Based Programming
GPA	Grade Point Average
GPT	Generative Pre-trained Transformer
GrPPI	Generic Reusable Parallel Patterns Interface
GUI	Graphical User Interface
HD	Halstead Difficulty
HE	Halstead Effort
HL	Halstead Length
HV	Halstead Volume
HT	Halstead Time
HPC	High-Performance Computing
HTML	HyperText Markup Language
HVSM	Historical Versions of Sequence Metrics
ID	Replace Inheritance with Delegation
IDE	Integrated Development Environment
IPO	Introduce Parameter Object
JS	JavaScript
JSX	JavaScript XML
LCOM	Lack of Cohesion

LLMs	Large Language Models
LOC	Lines of Code
M	Maintainability
MEDETECTOR	Missing Edit Anomaly Detector
ML	Machine Learning
MOA	Measure of Aggregation
MFA	Measures of Functional Abstraction
MM	Move Method
MTBD	Model Transformation By Demonstration
NOM	Number of Methods
NOP	Number of Polymorphic Methods
NP	Maximum Number of Possible Connections
NOH	Number of Hierarchies
OPTICS	Ordering Points to Identify the Clustering Structure
PHP	Hypertext Preprocessor
PSI	Program Structure Interface
QDF	Quality Decline Factor
QMOOD	Quality Model for Object Oriented Design
RQ	Research Question
SCC	String Comparison
SDLC	Software Development Life-Cycle
SQUANER	Software Quality Analyzer
SRP	Single Responsibility Principle
SSM	Structural Similarity between Methods
SUS	System's Usability Scale
SVM	Support Vector Machine
SVN	Subversion
T2	Task 2
T3-F1	Task 3 File 1
T3-F2	Task 3 File 2
UI	User Interface
UML	Unified Modeling Language
VR	Virtual Reality
VSCoDe	Visual Studio Code
WYSIWYG	What You See Is What You Get
XML	Extensible Markup Language

Chapter 1

Introduction

1.1	Context	2
1.2	Motivation	4
1.3	Problem	4
1.4	Objectives	5
1.5	Strategy	6
1.6	Main Contributions	6
1.7	Document Structure	7

Developing high-quality software can be challenging, primarily due to the complexity and dimension of the source code. Then, changing this code to meet new requirements or to fix errors becomes even more demanding. In addition, developers need to understand their code to quickly identify indicators that the code can be better represented [LF95, ND86, MS06].

Independently of the techniques used to identify and mitigate these issues, developers often need to restructure their code [MG18, Fow18, MCSW07, SLZ08, ADA18].

When navigating the complexities of software development, the analogy of wearing “*two hats*” becomes particularly important for this topic [Fow18]. In linear software development, such as the *Waterfall* model, which consists of a sequential and phased structure, where each phase is a prerequisite for the subsequent one [PWB09], developers predominantly wear the “*coding hat*” until the end of each phase [PWB09]. In this case, the delivery time is longer, and the feedback is delayed to further development stages [Cle03].

In contrast, with an *Agile* methodology, developers embrace an iterative and incremental approach to software development. Rather than following a linear sequence, *Agile* divides the project into small and manageable iterations, making the project requirements easier to change and ensuring that any problems in the software are identified and addressed earlier than if they used a

sequential approach [Mar03, Gel08]. Here, developers tend to regularly switch from “*coding hat*” to “*refactoring hat*” since they continuously change, adapt, and maintain their software [Mar03].

With real-time feedback, the “*two hats*” analogy becomes even more straightforward. In contrast to traditional methods, real-time feedback increases productivity by shortening the feedback loop, facilitating error detection, testing changes, and visualizing the impact of modifications during coding [Al 12, MS06, ARC⁺19, Han03, Tan13]. Using their metaphorical “*refactoring hat*” alongside a real-time strategy, developers benefit from an immediate and dynamic feedback loop. This accelerates issue identification and resolution, notably reducing debugging time and enabling swift incremental code refinement.

1.1 Context

The *Software Development Life-Cycle* (SDLC) encompasses all the necessary steps and requirements to implement, test, and maintain software [Som10, SBF14]. Figure 1.1, (p. 2) summarizes all these steps.

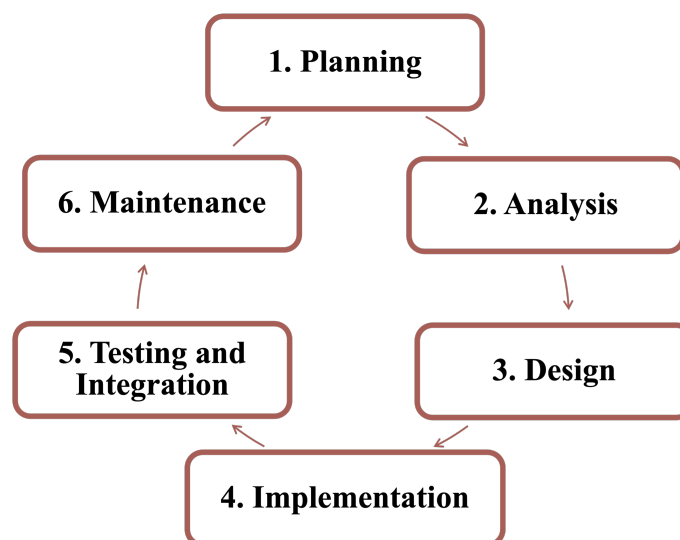


Figure 1.1: Software Development Life-Cycle.

1. **Planning.** In this phase, the development team works with stakeholders to understand the project objectives and assess the necessary resources, timelines, and potential risks. A comprehensive project plan is created, outlining the roadmap for the entire development process.
2. **Analysis.** This phase involves gathering and documenting detailed requirements from stakeholders. It aims to define both functional and non-functional specifications for the software.
3. **Design.** This phase translates the requirements into a high-level system architecture. Detailed technical specifications are developed for each component, including the user interface design.

and database structure. The goal is to create a blueprint that guides the development team in building the actual software.

4. **Implementation.** In this phase, the software is built based on the technical specifications defined in the system design. Developers write the actual code, adhering to coding standards and best practices. Regular code reviews should be conducted to maintain code quality and identify potential issues early in the process.
5. **Testing and Integration.** Testing is a critical phase where various types of testing, such as unit tests or integration tests, are used. The goal is to identify and fix defects and bugs to ensure the software meets the specified requirements. Testing validates the functionality, performance, and reliability of the software. Integration or deployment involves releasing the software to a production environment. It may include tasks such as data migration, and system configuration changes to ensure a smooth transition from development to the live environment. Deployment is a crucial step in making the software available to end-users.
6. **Maintenance.** Post-deployment, the maintenance phase begins. It involves addressing issues or bugs and implementing updates, patches, or new features as needed. Ongoing support is provided to users and stakeholders to ensure the continued functionality and stability of the software.

Therefore, developing software systems involves implementing the necessary features to meet their requirements. The goal is to create code with minimal complexity to facilitate its future inspection, modification, and reuse without requiring proportional effort or cost [Som10]. While developing their software, developers may need to change specific features or fix any potential defects in their code to align all the functionalities with users' needs [KHFG16].

Refactoring code might be one activity on the *implementation* stage of SDLC. This process consists of restructuring the internal structure of the software to improve its clarity, flexibility, and ease of maintenance. However, it is essential to note that refactoring should keep the previously implemented behaviours the same. The concept of *code smell* is another important topic to consider in the development stage. A *code smell* is a visible sign that there may be underlying programming issues present in the code. By applying specific *refactoring* methods, *code smells* can be addressed and mitigated, and ultimately the quality of the software can improve. In short, *refactoring* and *code smells* are concepts that go hand in hand. They are an integral part of developing good code, and their proper implementation can significantly enhance the software performance, readability, adaptability, and maintainability [Fow18, ADA18].

Measuring specific *code quality metrics* can also be a valuable way for developers to know the quality status of their software. By assessing particular properties, developers can evaluate the quality of different aspects of a software system and identify any underlying issues, such as code smells [Hal77a, LMD06, BBM96, RMT09, Mar04].

Besides, applying *software visualization* techniques can also help developers learn more about their software and identify specific blocks of code that should be reworked [SWM00, WLR11].

1.2 Motivation

Developers often face the challenge of ensuring their software systems meet specified requirements and features. This can be particularly difficult when dealing with complex systems with many lines of code and various behaviours already implemented. To overcome these challenges, developers must use good programming practices and adapt their systems to maintain quality. However, this process can be challenging and time-consuming, as understanding and modifying these systems can lead to a significant effort. So, correctly keeping software can be a struggle for developers, even when they are committed to producing high-quality code [MG18, MS06, CAGA11]. As Hoare stated:

“There are two ways of constructing a software design: One way is to make it so simple that there are no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.” [Hoa81]

Besides, developers frequently strive to minimize the time and effort required to create high-quality software by identifying and mitigating potential code defects. *Live programming* refers to providing immediate feedback to developers on the performance and behaviours of their systems while coding them.

Several development environments provide developers with assistance in improving their software. However, these environments could provide better real-time support, increasing the developers’ awareness of their code problems and reducing the time needed to achieve better, self-explanatory, and straightforward software [CAGA11]. For instance, Yamashita and Moonen [YM13] stated that developers want to receive feedback about their software as soon as possible. Moreover, Yuriy Tymchuk [Tyn17] tried to show that software maintainability cannot be validated by executing software but rather through its static analysis. He focused on proving that software developers benefit from live quality feedback on their programs. With his research, he verified that *90% of the developers* concluded that receiving live feedback about their system’s quality is helpful since it allows them to find possible design problems and programming mistakes, helping save time in developing good software.

1.3 Problem

Unfortunately, producing high-quality code can be a challenging goal to achieve due to various hidden problems that can negatively impact its readability, adaptability, and maintainability. These issues can hinder our ability to produce software that meets the needs and expectations of the users. Therefore, we must address these issues and work towards creating code that is as efficient and effective as possible [BT14, Fow18, KHFG16].

One way to address code defects is to regularly inspect code for potential problems and guide developers on fixing them. This can be done through regular code reviews, implementing testing protocols, and providing training or resources for developers to improve their coding skills. By

proactively identifying and addressing code defects, we can minimize the negative impacts on the functionality and performance of the software. However, we face a possible challenge:

How can we inspect software systems, provide immediate and continuous feedback about their quality, and guide developers towards better solutions?

Software inspection is a valuable practice for identifying potential issues in a system, but it can be challenging when the system is large and complex. Code smells may indicate deeper problems that can be addressed through refactoring, which can be a complex process, especially when considering the specific context of a program [Fow18]. Therefore, it is crucial to approach software inspection and refactoring with care, as they can be time-consuming and require a particular approach to effectively reduce code smells and improve code quality [Opd92]. These issues arise since developers possess varying perspectives and approaches when examining and analyzing software. Some individuals may be able to identify specific blocks of code causing problems, while others may not recognize those problematic segments. This diversity of perspectives can lead to differing opinions on addressing and solving particular code issues. Ultimately, all parties involved must work together and communicate to effectively troubleshoot and mitigate any possible problems [Fow18].

The research conducted by Murphy-Hill and Black [MHB08, MHB10], focused on the refactoring practices of approximately 40,000 software developers, indicated that they tend to manually refactor their systems, which can significantly extend the time required to produce a high-quality programming solution. Alternatively, some developers use static analysis tools to analyze the code after completing their programming tasks, prolonging the software development process. Overall, the authors concluded that manual refactoring and reliance on static analysis solutions could negatively impact the efficiency and effectiveness of the software development cycle.

1.4 Objectives

This research project aimed to enhance the feedback about the code quality of the software being analyzed and assist developers in writing better code. To accomplish this, we verified how to integrate and apply refactoring techniques within a live development environment. Our research project also aimed to determine if using code quality metrics and inspecting code in real-time can correctly identify refactoring opportunities.

We can break our primary goal down into three distinct sub-goals:

1. Increasing the efficiency and accuracy of code quality feedback;
2. Improving the overall quality of code;
3. Speed-up the process of identifying and addressing code issues (*i.e.*, code smells).

By addressing these objectives, we aimed to achieve the ultimate goal of facilitating the development of high-quality code. With this in mind, we studied the current state-of-the-art on the

main topics of this research project (cf. Chapter 2, p. 9) to propose a possible solution to reduce our problem (cf. Chapter 3, p. 79). Besides, this solution was validated to check if the sub-goals mentioned above were fulfilled or not (cf. Chapter 5, p. 123).

1.5 Strategy

To address the issues mentioned in Section 1.3, (p. 4), we proposed the development of a *refactoring recommender*. This recommender can analyze code quality metrics and identify potential refactoring candidates. It then provides specific refactoring recommendations to improve the code's comprehensibility, adaptability, and maintainability.

To ensure the success of this solution, we prioritized the needs of software developers. By involving them in the process and providing continuous feedback and support, we can empower them to create code that meets high-quality standards. Thus, we have incorporated real-time mechanisms into our refactoring environment to create a *Live Refactoring Environment*, enabling continuous and instant refactoring support and guidance.

Overall, our *Live Refactoring Environment* provides a comprehensive and efficient solution to the problem of poor code quality, ensuring that software systems are robust, reliable, and easy to maintain.

1.6 Main Contributions

Based on the idea of developing a *Live Refactoring Environment* that acts as a real-time refactoring recommender, the results of this research project can be divided into the following components:

State-of-the-Art. To keep abreast of the latest developments in our project's key areas, we reviewed the current state-of-the-art in software inspection, code smells, refactoring, code quality metrics, live programming, and software visualization (cf. Chapter 2, p. 9). This literature review included an in-depth analysis of the various approaches, techniques, and tools available for each topic and examined their advantages and any remaining open issues. By doing this, we could stay informed about the latest advances in these fields and ensure that our project used the most current and effective methods.

Live Refactoring Environment. We proposed the development of a *Live Refactoring Environment* to reduce the problem described in Section 1.3, (p. 4). This environment is capable of reviewing code and measuring specific code quality metrics such as the cyclomatic complexity, the number of lines of code, the Halstead metrics, or the maintainability degree, to identify specific refactoring opportunities like *Extract Class*, *Extract Method*, or *Move Method*, in real-time. In Section 3.6, (p. 84) and Chapter 4, (p. 91), we make a more detailed description of our work.

Empirical Experiments. We have also conducted empirical experiments using different case studies with real participants to validate our *Live Refactoring Environment*. These experiments

aimed to verify our solution’s effectiveness and determine if our solution was valid. The details of our validation methodology, including the specific goals and expected steps, are described in more detail in Section 3.6.3, (p. 86) and Chapter 5, (p. 123).

Scientific Publications. The fourth contribution from this research project was a compilation of scientific publications originated from various phases of our project (*cf.* Appendix A, p. 225)). These publications predominantly focus on the problem we sought to address, our suggested solution, and the methodologies used to implement and validate it.

1.7 Document Structure

In Chapter 2, (p. 9), *Literature Review*, we delve into the various topics we considered while developing our solution. This chapter provides background information on each concept and discusses the key contributions, techniques, and tools related to it. We aimed to comprehensively overview each of their strengths and limitations. By the end of this chapter, we summarize all the insights we have gained through our search on the current state-of-the-art in these fields.

In Chapter 3, (p. 79), *Research Problem*, we present the main aspects of our solution by outlining the issues in the state-of-the-art that our project aimed to address. We formally define the problem by stating our hypothesis, research questions, the steps we took in the development process, and any guidelines we had considered during it. In this chapter, we also summarize our development methodology, considering the critical points delineated in the previous chapters and the scope of our project. Additionally, we outline our validation process, including the methods we used to gather and evaluate data.

In Chapter 4, (p. 91), *Live Refactoring Environment*, we describe all the work done to create our real-time refactoring recommender. Here, we detail the development environment chosen to implement our solution, how we structured it, and what we used to develop it. We list all the code quality metrics used to help identify and sort the refactorings included in our approach. We describe all the refactorings, why we decided to have them in our approach, and how we did it. Besides, we describe the visual methodology developed to suggest each refactoring and the liveness applier that helps shorten the refactoring-loop.

In Chapter 5, (p. 123), *Empirical Experiments*, we summarize the experiments carried out to validate our hypothesis (*cf.* Section 3.3, p. 82) and answer our research questions (*cf.* Section 3.4, p. 82). We carried out two experiments with several participants divided into different experimental groups to check if a *Live Refactoring Environment* helps create better code faster. We also performed a robotized experiment that crawls multiple GitHub projects to verify if our approach can improve their code quality. Finally, in this chapter, we also present and discuss all the results collected from all experiments.

In Chapter 6, (p. 191), *Conclusions*, we summarize all the work done. We start by revisiting the hypothesis and research questions that we focused on. Then, we enumerate the primary outcomes resulting from this project. Lastly, we outline some future work that would improve our solution.

8 Introduction

In Appendix A (p. 225), *Scientific Publications*, we overview all scientific publications created from this research project. This encompasses not only papers directly stemming from the current project but also includes prior papers from earlier work, unpublished manuscripts, and master theses from co-supervised students who work with the main topics of our project.

In Appendix B (p. 233), *User Guide*, we introduce the user guide for the created *Live Refactoring Environment*, detailing instructions for proper installation, usage, and configuration.

In Appendix C (p. 239), *Experiments Artifacts*, we outline a comprehensive set of guidelines adhered to in the experiments conducted to validate our research project.

Chapter 2

Literature Review

2.1	Software Inspection	10
2.2	Code Smells	15
2.3	Refactoring	23
2.4	Code Quality Metrics	48
2.5	Software Visualization	55
2.6	Liveness	62
2.7	Summary	77

In this research project, we have explored a range of interconnected topics. One focus was on *Software Inspection* (cf. Section 2.1, p. 10), which systematically reviews software systems to identify programming defects that can compromise quality, hinder comprehension, and reduce adaptability and maintainability. Related to this, our work also addressed the concepts of *Code Smells* (cf. Section 2.2, p. 15), *Refactoring* (cf. Section 2.3, p. 23), and *Code Quality Metrics* (cf. Section 2.4, p. 48), which play crucial roles in understanding and improving software design and maintainability.

In addition, we have also explored topics like *Software Visualization* (cf. Section 2.5, p. 55) and *Live Programming* (cf. Section 2.6, p. 62). These areas focus on enhancing developers' feedback, support, and guidance, enabling them to work more effectively and make informed decisions about their code.

Thus, this chapter provides the foundational background for these subjects and explores various techniques and tools associated with each area. These approaches are categorized based on their objectives, methodologies, or specific applications. We also analyze the strengths and limitations of each one of them. Finally, the chapter summarizes the key insights derived from this literature review.

2.1 Software Inspection

Software inspection in software engineering refers to looking for possible defects in a software system to increase their quality using a well-defined proceeding [Bry99, BW93, AAA⁺11].

Usually, the inspection process includes reviewing every significant source of a project, from its requirements to its test plan. It starts by selecting a system for review. Then, a previously selected team meets to inspect the product. They choose a moderator to take care of the meeting, and each inspector has to point out possible defects in the product. This is also known as “*Human Inspection*” [TG02, RZ79, AFE84].

In the *late 70s*, *Michael Fagan* developed the most well-known inspection process that served as the basis for other inspection processes created since then. In this model, the process can be divided into several stages [Fag76, TG02]:

Planning. The moderator plans the inspection meeting. It is also when the team determines if a product is suitable to be reviewed.

Overview Meeting. The developers describe the background and specific areas of the product.

Preparation. The inspectors learn the most important aspects of the project being analyzed.

Inspection Meeting. The inspectors find defects in the project and talk about them.

Rework. The developers try to correct all the defects pointed out at the inspection meeting.

Follow-Up. The moderator and the inspectors check the developers’ rework. In this stage, they verify if the corrections were the best and that no new defects exist.

Our research project does not focus on the different stages of the inspection process but instead on one of the software review elements: the *code inspection*. **Code inspection** is an essential component of an inspection checklist. The inspectors or software developers audit the source code or just a sample to find possible defects that should be fixed. In this case, a defect is a block with more or less lines of code that does not correctly implement the specified requirements or does not use the best programming practices possible, which can make the system hard to read, understand, or improve [Li95, RZ79, ASK14]. In an extensive software system, reviewing code as soon as possible is the key to have long-term source code with high quality [KBG16, Bri88].

To learn more about this topic and its current state-of-the-art, we have reviewed some literature about it. We divided the approaches and tools based on their main goals. In this way, we found three major objectives for inspection tools and methodologies: *assess the software quality*, *analyze the changes history*, and *inspect the source code*.

2.1.1 Software Quality Assessment

One of the main goals of software inspection approaches and tools is to assess the quality of software systems [TG02, Bri88]. On this subject, we found two distinct inspection techniques that verify code, not to directly find defects but to assess the quality of the systems under analysis.

Hist-Inspect. Inspection tool for Eclipse IDE that easily configures thresholds of software metrics and reviews the code to provide feedback to developers about the stability of specific modules and the growth or deterioration of particular properties like coupling or cohesion. This tool has a set of views that graphically represent each module’s evolution of specific software metrics. With these views, programmers do not have to recover and analyze each system version’s values, metrics, and properties [MHM⁺11].

Panas *et al.* [PQV07] stated that high-performance computing systems encourage an inspection of their design and development rather than their requirements and specifications. They then proposed a tool for large C and C++ programs that provides feedback about their structure and code quality. With this tool, developers receive feedback through intuitive visual forms. For instance, if developers want to verify if the architecture and structure of their systems are equivalent to the mental model they created, they can use several interfaces representing the respective program. Figure 2.1, (p. 11) shows the architecture visualization interface that illustrates software and its interactions as small cities.

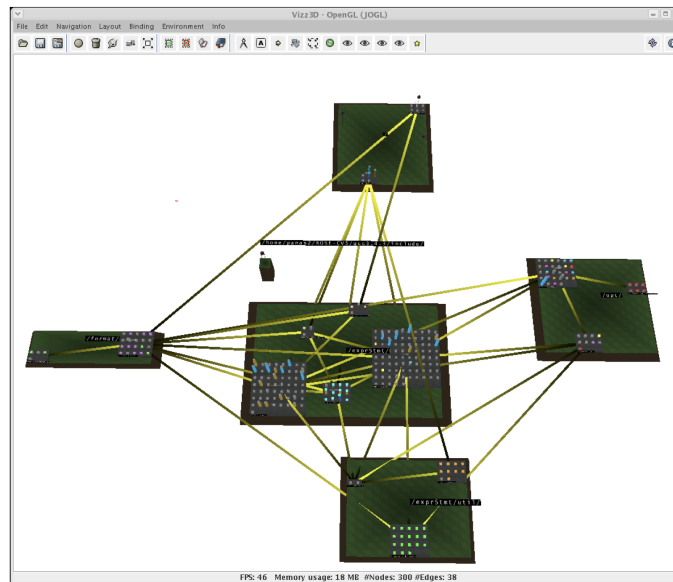


Figure 2.1: Architecture visualization interface to inspect a software system (Figure 3 of [PQV07]).

2.1.2 Changes History Analysis

Another goal of software inspection is to analyze the evolution of software systems over different development iterations [ZSK14]. On this subject, we found two distinct inspection approaches. As it happens with the techniques in Section 2.1.1, (p. 10), these approaches are also based on reviewing code to analyze the evolution of software programs over time.

Critics. Code review tool developed for Eclipse, which inspects systematic changes in software systems. During peer reviews, programmers check possible differences between several system versions. However, detecting and examining changes across several files is hard for them. This tool aims to mitigate this problem by (i) searching for possible systematic changes using a template

customized by the developers and (ii) detecting inconsistent modifications. Thus, Critics can enhance the inspection productivity of developers in examining large and complex edits when reviewing source code [ZSK14].

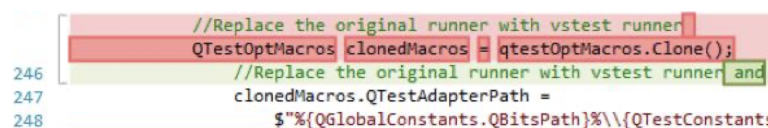
Rahman *et al.* [RRRC16] proposed a code review and recommendation tool for Vendasta technologies¹, that analyzes pull requests to find possible defects on their source code. With it, when developers start a new pull request on GitHub, the request itself is automatically examined, and it is graded with a letter from A to E, where A is the best grade possible. This grade is calculated by measuring several metrics to help detect possible code defects.

2.1.3 Code Review

Most software inspection tools aim to review code to identify possible programming defects, bugs, or blocks of code that do not correctly implement the specified requirements [RZ79, ASK14]. On this topic, we found distinct approaches and tools.

RefDistiller. Refactoring-aware code review tool for Eclipse that helps developers identify behavioral modifications on source code manually refactored. It compares two versions and checks each refactoring applied between those versions. RefDistiller uses two reconstruction techniques: (i) a template-based checker to find missing edits on the refactorings employed and (ii) a refactoring separator to identify extra changes that may deteriorate the behavior of the program. Therefore, by helping programmers know if the modifications they did are refactoring methods, it provides more confidence to them about their software programs [ASK14, ASM⁺18].

CFar. Collaborative code review tool that analyzes software systems to provide feedback to developers on their programs. Another goal of this tool is to reduce the time and effort needed to do high-quality reviews that detect the most defects possible. CFar automatically examines a program and creates comments about the analysis mapped to the relevant lines of code. The generated comments are very similar to the ones developers made on their source code. Then, each comment is highlighted on the source code with colors in a range of red and green, regarding its importance (*cf.* Figure 2.2) [HMC⁺18].



```

246 //Replace the original runner with vstest runner
    QTestOptMacros clonedMacros = QTestOptMacros.Clone();
247 //Replace the original runner with vstest runner and
    clonedMacros.QTestAdapterPath =
248     $"%{QGlobalConstants.QBitsPath}%\\{QTestConstant:

```

Figure 2.2: Comments representing the reviews made on source code (Figure 1 of [HMC⁺18]).

BovInspector. Tool that inspects source code to find possible examples of buffer overflow. Given a program, BovInspector first does a warning reachability analysis and, after, executes the code considering the reachable warning detected in the previous step. Each reachable warning is approved and categorized by verifying if all the path conditions and buffer overflow constraints can be satisfied simultaneously. For each valid reachable warning, BovInspector fixes the case of buffer overflow by suggesting three possible automatic solutions: (i) adding boundary checks,

¹End-to-end commerce platform that sells digital solutions to small businesses, <https://www.vendasta.com>

(ii) changing the current API, replacing it with a safer one, or (iii) increasing the size of the buffers [GWL16].

ICC-Inspect. Real-time tool for Android applications that streams inter-component communication (ICC) information to a dynamic visualization framework. This framework (*cf.* Figure 2.3) represents the systems with interactive ICC call graphs that allow developers to examine particular fragments of their Android application while being executed, helping them to understand and debug the different behaviors of their Android programs [JC18].

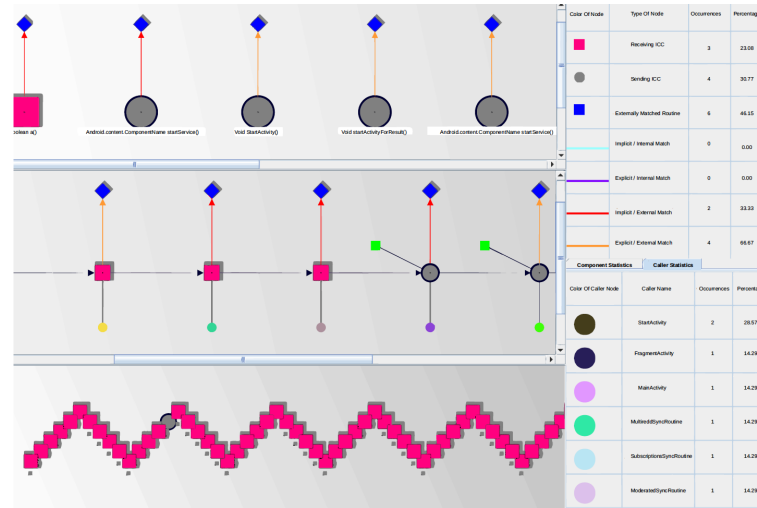


Figure 2.3: Visual components of ICC-Inspect (Figure 2 of [JC18]).

2.1.4 Discussion and Open Issues

Software inspection aims to find defects in software systems [BW93]. As mentioned, our work will focus on code review to identify possible problems or indicators of deeper problems in the code. Even when their outputs are correct, several software programs are implemented using incorrect programming practices, making the systems hard to understand and maintain [Li95, RZ79]. To reduce the effort and time needed to inspect code, the inspection actions should be executed as soon as possible [KBG16].

In this section, we analyzed some inspection tools and approaches to improve software systems. Most of them focus on providing visual information about the quality of software systems to developers by analyzing code quality metrics over different versions of a specific program or reviewing code to detect possible code smells and identify refactoring opportunities. These tools take on topics such as quality metrics, code smells, and refactoring due to the intrinsic link between software inspection and the current demand to increase the quality of software systems. So, it should also be taken into consideration that some of the following sections (*cf.* Section 2.2, p. 15, Section 2.3, p. 23, and Section 2.4, p. 48) present tools and approaches that can also be considered as belonging to this concept, mainly related to the analysis of *code quality* and *source code*.

Despite seeing many positive points in these tools, we also found some disadvantages that could represent challenges to the development of source code inspection approaches:

Discontinuous Inspection Solutions. Just one of the analyzed tools reacts automatically to any change made by developers on their source code, which means that the remaining ones need to be manually executed to provide feedback about the software systems. The goals of these tools are great since they aim to improve the quality of software programs and improve the comprehension and maintenance capability of programmers. However, slow inspection approaches that make discontinuous analyses may increase the time needed to provide feedback to developers about the inspected code, making it less fluid and continuous. With slower feedback, developers are not able to shorten the total development time since they will take more time scanning the implemented code.

Intrusive Presence on IDEs. Many tools directly provide feedback about the inspected code on the text editor, which can be a little intrusive. This happens mainly in the tools that analyze software systems in real and programming-time. While developers are programming, the tools inspect their code, giving information about it in the IDE itself on the block of code where possible defects were found. This way of providing feedback can be intrusive and make programmers lose their train of thought or development flow.

In these cases, the advantages of using liveness on these tools would be reduced, as developers would waste some of their development time paying attention to the information presented by the tool. Therefore, we believe that information related to possible problems detected when the source code was inspected should be given to developers while they are programming, but in a more straightforward and less distracting way.

Some of the challenges mentioned above can also be possible problems we can encounter with other tools and methods used to detect code smells, identify refactoring opportunities, or analyze code quality metrics.

Table 2.1, (p. 14) summarizes the literature review on the approaches and tools found in the concept of software inspection.

Table 2.1: Comparison of software inspection approaches and tools.

	<i>Goal</i>	<i>Target</i>	<i>Visual</i>	<i>Real-Time</i>	<i>Plugin</i>
[MHM ⁺ 11]	Software Quality Assessment	-	✓	-	✓
[PQV07]	Software Quality Assessment	C, C#	✓	-	✓

Continued on next page

Table 2.1: Comparison of software inspection approaches and tools. (Continued)

	<i>Goal</i>		<i>Target</i>	<i>Visual</i>	<i>Real-Time</i>	<i>Plugin</i>
[RRRC16]	Changes Analysis	History	-	-	✗	-
[ZSK14]	Changes Analysis	History	-	-	✗	✓
[ASK14, ASM ⁺ 18]	Code Review		-	-	-	✓
[HMC ⁺ 18]	Code Review		-	✓	-	✓
[GWL16]	Code Review		-	-	-	✓
[JC18]	Code Review		Java (Android)	✓	✓	✓

2.2 Code Smells

Poorly designed software degrades in quality, making it harder to understand, adapt, and maintain [TPB⁺15, KHFG16]. These issues often stem from code smells – surface indicators of deeper design problems, first introduced by *Kent Beck* and *Martin Fowler*. Code smells impact readability, testability, adaptability, and maintainability [Fow18, TPB⁺15].

Sharma and *Spinellis* [SS18] identified five key aspects of code smells:

Indicator. A code smell can express a symptom of a design problem.

Poor solution. A code smell can express a weak solution to a specific problem.

Bad practices. A code smell violates the best practices of software development.

Quality decay. A code smell makes a software system hard to read, increasing the difficulty of changing or maintaining it. Thus, its quality is lower than it should be.

Recurrence. A code smell tends to be a recurring problem that occurs when developers try to implement specific features in a short period.

Code smells vary in impact and appear in source code, design, or architecture [SS18]. They arise from developer inexperience, shifting requirements, or diverse technologies, making them inevitable in complex systems [SS18, TPB⁺15]. The most well-known catalog is from *Martin Fowler*, detailing a wide range of code smells [Fow18].

Over time, different techniques and tools have been developed to reduce code smells in a software system. Next, we summarize some of them, divided by the methodology used to detect these underlying coding problems. We found that the majority of code smell detection tools and approaches are based on assessing *heuristics*, inspecting the *source code*, using *artificial intelligence* algorithms, or *slicing techniques*.

2.2.1 Heuristics

One way to detect code smells is by using specific heuristics to measure particular programming aspects [LTOT08]. On this subject, we found distinct approaches and tools that will be described below.

c-JRefRec. Eclipse plugin that detects Java *Feature Envy* smells. *Feature Envy* is a classical code smell that occurs when a method overuses data from other classes rather than from its own [Fow18]. To detect *Feature Envy* smells, c-JRefRec takes into account four distinct heuristics analyzed statistically and semantically: (i) the number of methods implemented in a class, excluding abstract methods, (ii) the number of incoming or outgoing edges connected between the members of a specific class, (iii) the number of classes using the methods or properties of a particular class, and (iv) the number of classes whose methods or properties are used by methods of another class [UOH17].

TAJSlint. Tool to detect JavaScript code smells like *Argument Count Mismatch* or *Primitive Property Assignment*, using specific rules and metrics such as assessing the number of functions' parameters, number of function's declarations, or the cyclomatic complexity of a block of code [AL20].

Li [LTOT08, LT11] developed a tool for Emacs and Eclipse that detects similar or cloned code in Erlang programs. This approach is integrated with Wrangler, a pre-existing refactoring tool for Erlang, also developed by Li [LTOT08]. To execute this tool, the developers should specify four parameters: (i) the minimum number of expressions included in a code clone, (ii) the maximum number of new parameters of the common method, (iii) the minimum number of parameters of a cloned class, and (iv) the similarity score limit [LT11].

InCode. Eclipse plugin designed to continuously detect design problems in software projects. Upon starting the Eclipse workbench, InCode initiates background analysis of the currently active source file in the editor. As it identifies design problems, it places red markers adjacent to the affected classes or methods (*cf.* Figure 2.4, p. 17). These markers are also visible in the overview ruler on the right side of the editor, similar to the markers used for compiler errors or warnings. These markers are dynamic, with new markers appearing or existing markers disappearing as new code is written or modified. InCode recognizes that relying solely on metrics is insufficient for detecting genuine design problems. Therefore, it focuses on identifying four specific design problems associated with an improper distribution of intelligence among classes: *God Class*, *Data Class*, *Feature Envy*, and *Code Duplication*. Rather than requiring developers to interact directly with metrics, InCode enables them to reason about the quality of their design at a conceptual level that is more convenient and intuitive [MGV10].

2.2.2 Source Code Analysis

Another method to detect possible code smells is direct source code analysis. By checking the code on the text editor, different tools and approaches can identify blocks of code that should be refactored since they might represent deeper problems on the software systems [Fow18].

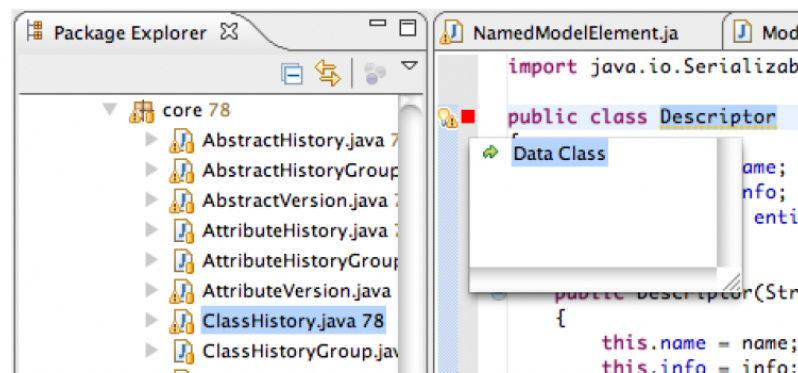


Figure 2.4: InCode automatically marking code design problems (Figure 1 of [MGV10]).

WebScent. Tool that detects embedded code smells in web applications, mainly in PHP servers. Embedded code smells violate principles like modularity or separation of concerns, consequently reducing the maintainability degree of the code. WebScent finds code smells such as mixed code for HTML, CSS, or JavaScript, duplicated JS code, CSS in HTML, or HTML syntax errors. To identify these types of code smells, the tool detects the problems on the source code and then tries to find them on the server-side [NNN⁺12].

ADoctor. Also known as the *Android Code Smell Detector*, is a tool specifically designed to identify 15 Android-specific code smells, such as *Durable WakeLock*, *Inefficient Data Structure*, *Internal Setter*, *Leaking Thread* and *Member Ignoring Method*. Its functionality revolves around the syntactic analysis of the source code, leveraging the extraction of Abstract Syntax Trees (ASTs) from the Java classes within the codebase. Subsequently, aDoctor employs detection algorithms built upon the Visitor pattern to identify these code smells [PDNP⁺17].

RAIDE. Eclipse plugin, specifically designed to detect test smells and facilitate their refactorings in Java projects using the JUnit framework. It combines the capabilities of the tsDetect tool [PAN⁺20], which is an automated code smell tool for Java code that locates smells in test code for detecting test smells and leverages the graphic components of JDeodorant [FTC07, TCC08, Nik] for refactoring smelly test code. With RAIDE, developers can efficiently identify test classes, methods, and lines of code that are impacted by specific test smells. This helps developers assess the quality of their test code by comparing it against established test patterns. By providing refactoring solutions, RAIDE empowers developers to enhance the quality and maintainability of their test code [SMR⁺20].

PyNOSE. PyCharm plugin developed to detect code smells in Python code. To identify Python-specific test smells such as *Magic Number Test*, *Redundant Assertion*, *Conditional Test Logic*, *Duplicate Assert*, and *Test Maverick*, the authors used a tool named PythonChangeMiner [GLB⁺21], which they also created. This tool was employed to analyze the test suite histories and identify frequent change patterns. This tool was built upon the algorithm developed by Nguyen *et al.* [NND⁺19] for Java code. It is specifically designed to handle Python code and extract relevant information from the histories of test suites. By leveraging this tool, the authors could

detect common change patterns that indicate the presence of test smells [WGS⁺21].

Stolee and Elbaum [SE11] developed a tool that detects code smells for pipe-like mashups on Yahoo! pipes environment. This tool identifies several specific code smells and uses them to suggest refactoring techniques to solve them. The smells supported by this approach are [SE11]:

Laziness Smell. Pipe-like mashup code smell related to the *Lazy Class* smell. It represents classes that do not do enough actions [Fow18]. In this case, *Laziness Smells* identify pipes that contain modules that do not collaborate with the pipe output.

Redundancy Smell. Pipe-like mashup code smell that consists of duplicated strings or modules in a pipe, increasing the complexity of the system.

Environmental Smell. Pipe-like mashup code smell that occurs when pipes are not updated when the external environments are modified.

Population-Based Smell. Pipe-like mashup code smell that focuses on the repository of the system where it can exist patterns in different modules were used and reused by the pipes.

This tool has an interface that allows developers to verify their pipes' status and which refactorings they can apply. Thus, its main objective is to reduce the system's complexity by increasing its abstraction [SE11].

Fenske et al. [FMS⁺17] created a tool for FeatureIDE² that detects three different types of code clones (cf. Figure 2.5). *Type-1 Clones* are highlighted in red, representing the exact duplicated code. *Type-2 Clones* are highlighted in yellow, showing the statements that were renamed but are copies of other statements. *Type-3 clones* are highlighted in green and represent new statements added to the code clone. To handle these code clones, the authors implemented a linting recommending refactorings like Rename or Pull Up To Common Feature.

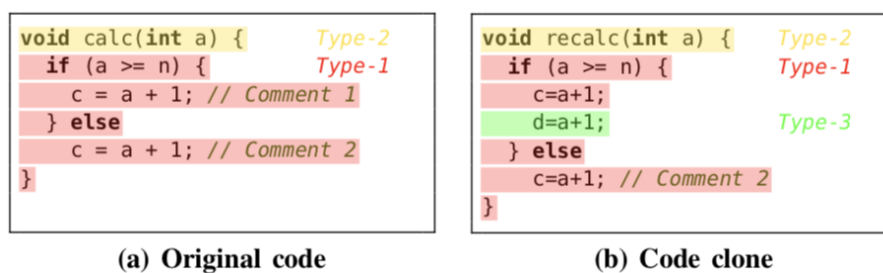


Figure 2.5: The different types of code cloning (Figure 3 of [FMS⁺17]).

Gadient et al. [GGFN18] introduced a linting tool that used Android Lint³, a static analysis framework provided by the official Android Studio IDE. The linting tool employs a detector-based approach, where each detector is implemented as a Java class that utilizes Android Lint interfaces to access the app's ASTs, which are constructed from XML, source code, or bytecode. To detect

²FeatureIDE, <http://fosd.net/fide>

³Android Lint, <https://sites.google.com/a/android.com/tools/tips/lint>

security smells like *Persisted dynamic permission*, *Broken service permission*, *Custom scheme channel*, *Incorrect protection level*, or *Implicit pending intent*, the authors implemented specific detection strategies using Android Lint's APIs. One notable advantage of integrating their approach with Android Lint is that the linting tool seamlessly integrates into the Android Studio IDE. Besides, through notifications, developers can receive real-time feedback during app development, providing immediate awareness of security code smells in their code.

2.2.3 Artificial Intelligence Techniques

We can also detect code smells using artificial intelligence algorithms. This detection category can include any approach or tool that tries to find possible solutions according to predefined functions or values. Therefore, it defines a mathematical formula to detect code smells and lets the chosen algorithm find the best sequence of smells that should be mitigated by a specific refactoring or sequence of refactoring techniques [Bar20].

Nucci *et al.* [DPT⁺18] think that some code smell detection tools are subjective since they use different approaches, and most do not use machine learning algorithms. Machine Learning methods can reduce the subjectivity of the detection tools because they can correctly distinguish smells from non-smells. They applied 32 different machine learning algorithms, like *Random Forest* or *Naive Bayes*, in several projects to detect four code smells (*Data Class*, *Large Class*, *Feature Envy*, and *Long Method*). With this work, they created a dataset with smells that detection tools can use to help identify them on source code.

Palomba [Pal15] proposed a textual approach to detect *Long Method* smells. This technique is based on textual analysis, while most current practices only identify smells based on the structural information provided by the software system under analysis. Only doing a structural analysis, tools, and detection methodologies can classify methods as long, even if they only implement one feature. Then, the procedure suggested by Palomba takes a specific method, extracts the code blocks from it, and separates the identifiers and comments on each block to clean the text from non-relevant words. The approach computes the similarity matrix for each pruned pair of blocks using *Latent Semantic Indexing*. If there are blocks with a similarity lower than α , where α is 0.4, the method is classified as long.

Liu *et al.* [LJX⁺21] introduced a novel approach to detect Java code smells using deep learning techniques. Their key insight was that deep neural networks and advanced deep learning methods had the potential to automatically select relevant features from source code and establish complex mappings between these features and predictions. Their networks were based on metrics used to detect specific code smells by previous works. They used this approach to detect well-known smells like *Feature Envy*, *Long Method*, or *Large class*.

Das *et al.* [DYD19] also proposed an approach to identify two Java code smells (*Brain Class* and *Brain Method*) using a deep learning approach. Their system used a dataset comprising thirty open-source Java projects sourced from GitHub repositories. Those datasets were filled with several well-known metrics (*e.g.*, cyclomatic complexity, number of lines of code, or maximum nesting) assessed by other authors.

Guggulothu and **Abdul Moiz** [Gug19, GAM19] detected code smells like *Shotgun Surgery*, *Message Chaining*, *Long Method*, and *Feature Envy*, using supervised machine learning (ML) classification techniques. By leveraging known data, the ML approach classifies new instances into binary categories, indicating whether a method is prone to have code smells. The datasets used in both studies consisted of methods from several Java systems. They computed specific object-oriented metrics, such as complexity and coupling, to serve as features or attributes in the datasets used on their projects. Then, they applied different classification algorithms to the trained datasets to detect the selected smells.

2.2.4 Slicing Techniques

Mark Wiser proposed the slicing technique with the original goal of using it in the debugging process. This methodology divides a program into different slices representing the statements that may affect a specific set of variables. Then, with the slicing approach, we can create other slices that usually capture subsets of code with different target behaviors [Wei81].

AlAbwaini et al. [AAJ⁺18] proposed an approach that uses precisely the slicing technique to identify “live” and “dead” code. “Dead” code consists of fragments of source code that are not executed and, therefore, do not influence the software system. The slicing algorithm used takes into account the goals of the different pieces of the analyzed code that should be divided into slices. For instance, if a function has several mathematical operations, each should be a different slice with well-defined functionalities. Despite considering it a good approach to find dead code, they stated that it does not work correctly in very complex software systems [AAJ⁺18].

2.2.5 Discussion and Open Issues

As described in Section 2.2, (p. 15), a code smell is a surface indicator of a deeper problem in the software system that decreases its quality. Then, the existence of a code smell does not require its correction through refactoring methods since it does not cause errors in its execution or results. The same happens the other way around. Even if the code was implemented using the best programming practices, developers might want to develop it differently and not correct it. Thus, it is mainly up to the developers to choose whether they wish to correct the code smells that their systems may have or not [Fow18, SS18].

Some tools and approaches already aim to detect code smells and help software developers improve their software systems if they wish to do so. However, these tools have some issues that must be taken into account and that represent challenges in identifying code smells:

Discontinuous Code Smells Detection. We analyzed several tools and approaches that aimed to detect possible code smells on the source code. However, most do not allow developers to continuously inspect their systems while programming them. With discontinuous and slow detection of possible code smells, these tools cannot shorten the feedback loop, which may increase developers’ time to finish their software programs or reach a better programming solution with more code quality. This happens because they must execute the smells detection plugins or methodologies

to receive valuable guidance. With fast and continuous feedback, these tools could identify possible code smells quickly. Besides, they would be capable of reducing the feedback, support, and guidance loop.

Identification of False-Positives. Some researchers believe that code smells detection tools can identify many false-positives [SS18]. They assume that when a tool only uses quality indicators calculated directly from the source code or its AST, it may present wrong results that can misguide the developers to create worse code. Nucci *et al.* [DPT⁺18] and Haendler *et al.* [HSS17] believe that this problem can be solved using prediction and classification algorithms that use a wide range of data to detect, predict, and classify possible code smells. As it was verified through the analysis of the state-of-the-art on code smells, few or no tools predict or classify code smells, which can indicate that they may be incorrectly detecting them, misleading the developers.

Lack of Impact Analysis Mechanisms. Several tools identify different types of code smells. However, when presenting the detected smells, most do not reveal their real impact on the code quality. It would be good for those tools to help developers understand that a specific smell impacts the code more negatively than another smell. Probably, that way, developers would be more informed when trying to mitigate them and would try to reduce a smell that could solve a set of other smaller smells, reducing the time and effort to improve their software quality.

Lack of Visualization Techniques. There is some evidence that visual and graphical elements help understand the information and data that some tools try to show to their users [Bie89, HCL05]. Therefore, these techniques can help developers to understand their software systems quickly. Most code smells detection tools do not provide visual insights about the software systems under analysis. If these tools provided visual information about the software programs, developers could verify which parts of the code should be changed and why there were identified code smells on it more easily and attractively.

Table 2.2, (p. 21) summarizes the literature review results on the approaches and tools used to detect code smells.

Table 2.2: Comparison of code smells detection approaches and tools.

	<i>Target</i>	<i>Method</i>	<i>Code Smells</i>	<i>Visual</i>	<i>Real-Time</i>	<i>Plugin</i>
[UOII17]	Java	Heuristics	Feature Envy	✗	✗	✓

Continued on next page

Table 2.2: Comparison of code smells detection approaches and tools. (Continued)

	<i>Target</i>	<i>Method</i>	<i>Code Smells</i>	<i>Visual</i>	<i>Real-Time</i>	<i>Plugin</i>
[AL20]	JavaScript	Heuristics	Primitive Property Assignment, Argument Count Mismatch, Duplicate Declaration, Undeclared Variables, etc	-	-	✓
[LTOT08, LT11]	Erlang	Heuristics	Code Clone	✗	✗	✓
[MGV10]	Java	Heuristics	God Class, Data Class, Feature Envy, Code Duplication	✓	✓	✓
[NNN ⁺ 12]	PHP, HTML, CSS, JavaScript	Source Code	Mixed Code, Duplicated Code, Syntax Errors, etc	✗	✗	-
[PDNP ⁺ 17]	Java Android Code	Source Code	Durable Wake-Lock, Inefficient Data Structure, Internal Setter, Leaking Thread, Member Ignoring Method	✓	✗	✓
[SMR ⁺ 20]	Java	Source Code	Test smells	✓	✗	✓
[WGS ⁺ 21]	Java	Source Code	Conditional Test Logic, Duplicate Assert, Test Maverick, Magic Number Test, Redundant Assertion	-	✗	✓

Continued on next page

Table 2.2: Comparison of code smells detection approaches and tools. (Continued)

	<i>Target</i>	<i>Method</i>	<i>Code Smells</i>		<i>Visual</i>	<i>Real-Time</i>	<i>Plugin</i>
[SE11]	-	Source Code	Lazy Redundant Smell, Environ- mental Smell, Population- Based Pipe Smell		✓	✓	✓
[FMS ⁺ 17]	Feature Lan- guages	Source Code	Code Clones		-	-	✓
[GGFN18]	Java Android Code	Source Code	Security Smells	Code	✓	✓	✓
[DPT ⁺ 18]	-	Machine Learning	Data Large Feature Long Method	Class, Class, Envy,	-	-	✗
[Pal15]	-	Latent Semantic Indexing	Long Method		-	-	✗
[LJX ⁺ 21]	Java	Deep Learn- ing	Feature Long Large Class, etc	Envy, Method,	✗	✗	✗
[DYD19]	Java	Deep Learn- ing	Brain Brain Method	Class,	✗	✗	✗
[Gug19, GAM19]	Java	Machine Learning	Shotgun Surgery, Message Chain- ing, Long method, Feature Envy		✗	✗	✗
	-	Slicing	Dead Code		✗	-	✗
[AAJ ⁺ 18]							

2.3 Refactoring

Increasingly, software systems are more complex and challenging to read, adapt, and maintain. These problems increase the effort and time needed to implement new features or change others

already implemented [MS06, KHFG16]. When developers need to restructure their systems, changing their internal structure without modifying the previously implemented behaviors, they are refactoring their software [Fow18, MT04, Opd92]. Refactoring helps improve software quality by making it more clean, well-structured, and self-explanatory [Fow18, MHB08, CAGA11, ADA18].

To implement a specific refactoring method, it is necessary to go through a set of actions. Firstly, developers need to identify refactoring opportunities. Then, they have to decide which are the best refactoring methods that should be applied, taking into account their programming context and main objectives. Third, they have to ensure that after applying the refactorings, the behaviors previously implemented are preserved. Finally, they have to access the quality of the final system [MT04, YSC⁺16].

As this research project mainly focuses on providing feedback and support to software developers through refactoring suggestions, we identified three essential activities to verify refactoring opportunities: *prediction*, *detection*, and *recommendation* [Bar20]:

Prediction. In this activity, developers try to find future refactoring opportunities by analyzing the evolution of their code, verifying the existence of code smells, and measuring specific code quality metrics.

Detection. To detect refactoring opportunities, developers have to analyze the software under maintenance. This can be done by detecting code smells or analyzing metrics that assess the values of particular quality attributes of a program. Then, the pieces of code developed using bad programming practices or with a low-quality degree should be refactored.

Recommendation. This step is related to visualizing all the refactoring opportunities found to solve a specific problem, allowing the developers to pick the best suggestion and action for their software systems.

Different refactoring methods aim to solve specific code smells [YSC⁺16]. *Martin Fowler* enumerated more than 50 refactoring techniques [Fow18]. Based on his catalog, in Table 2.3, (p. 24), we list the refactorings that aim to reduce specific code smells.

Table 2.3: Code smells and their respective refactoring solutions [Fow18].

<i>Code Smell</i>	<i>Common Refactorings</i>
Alternative Classes with Different Interfaces	Change Method Declaration, Move Method, Extract Superclass
Comments	Extract Method, Change Method Declaration, Introduce Assertion

Continued on next page

Table 2.3: Code smells and their respective refactoring solutions [Fow18]. (Continued)

<i>Code Smell</i>	<i>Common Refactorings</i>
Data Class	Encapsulate Record, Remove Setting Method, Move Method, Extract Method, Split Phase
Data Clumps	Extract Class, Introduce Parameter Object, Preserve Whole Object
Divergent Change	Split Phase, Move Method, Extract Method, Extract Class
Duplicated Code	Extract Method, Slide Statements, Pull Up Method
Feature Envy	Move Method, Extract Method
Insider Trading	Move Method, Move Field, Hide Delegate, Replace Subclass with Delegate, Replace Superclass with Delegate
Large Class	Extract Class, Extract Superclass, Replace Type Code with Subclasses
Lazy Element	Inline Method, Inline Class, Collapse Hierarchy
Long Method	Extract Method, Replace Temp with Query, Introduce Parameter Object, Preserve Whole Object, Replace Method with Command, Decompose Conditional, Replace Conditional with Polymorphism, Split Loop
Long Parameter List	Replace Parameter with Query, Preserve Whole Object, Introduce Parameter Object, Remove Flag Argument, Combine Methods into Class
Loops	Replace Loop with Pipeline
Message Chains	Hide Delegate, Extract Method, Move Method
Middle Man	Remove Middle Man, Inline Method, Replace Superclass with Delegate, Replace Subclass with Delegate
Mutable Data	Encapsulate Variable, Split Variable, Slide Statements, Extract Method, Separate Query from Modifier, Remove Setting Method, Replace Derived Variable with Query, Combine Methods into Class, Combine Methods into Transform, Change Reference to Value
Mysterious Name	Change Method Declaration, Rename Variable, Rename Field
Primitive Obsession	Replace Primitive with Object, Replace Type Code with Subclasses, Replace Conditional with Polymorphism, Extract Class

Continued on next page

Table 2.3: Code smells and their respective refactoring solutions [Fow18]. (Continued)

<i>Code Smell</i>	<i>Common Refactorings</i>
Refused Bequest	Push Down Method, Push Down Field, Replace Subclass with Delegate, Replace Superclass with Delegate
Repeated Switches	Replace Conditional with Polymorphism
Shotgun Surgery	Move Method, Move Field, Combine Methods into Class, Combine Methods into Transform, Split Phase, Inline Method, Inline Class
Speculative Generality	Collapse Hierarchy, Inline Method, Inline Class, Change Method Declaration, Remove Dead Code
Temporary Field	Extract Class, Move Method, Introduce Special Case

Some approaches and tools aim to identify possible refactoring opportunities. We divided the literature found on this subject by the main methodologies used. They are based on *source code*, *code metrics and heuristics*, *change history*, *artificial intelligence*, *game theory*, and *user direct action*.

2.3.1 Source Code Analysis

As it happens with the detection of code smells (*cf.* Section 2.2.2, p. 16), we can also identify possible refactoring opportunities by inspecting the source code of specific software projects and, then, pointing out the blocks of code that should be refactored [Fow18]. In this subject, we found several approaches and tools.

JS Refactor. Visual Studio Code automatic refactoring plugin for JavaScript, Vue, or HTML source code. This extension checks for possible refactorings on the code and allows it to execute them automatically. It supports refactorings like Extract Method, Extract Variable, Inline Variable, or Rename Variable [Fow18]. Besides, it also allows programmers to implement several utilities such as *Convert To Arrow Function*, *Convert To Function Declaration*, *Convert To Function Expression*, among others [Steb, Stea].

glean. VSCode extension that allows developers to refactor React, JavaScript, and TypeScript systems by reviewing their source code. When this tool is activated in a React project, it suggests some refactorings like extracting JSX code into a new component, converting class components to functional components and vice versa, wrapping JSX code with conditional, and moving JSX code between files. In a JavaScript or Typescript system, it only recommends extracting code between files [Wixa, Wixb].

Elegant PHP. Visual Studio Code refactoring tool for PHP source code that aims to improve the development workflow, increasing the programming experience. With this extension, a programmer can access refactorings like *Extract Method*, *Extract Variable*, and *Inline Variable* [Kos].

Super Sharp and **Vscode-Fsharp-Refactor**. VSCode plugins that allow refactoring C# and F# programs by inspecting their code. With these tools, developers have access to move and extract refactoring techniques. SuperSharp only supports the move classes, enumerations, interfaces, or structures refactorings [Cra]. Vscode-Fsharp-Refactor, on the other hand, only analyzes extract expressions, lambdas, and strings refactoring methods [Man].

LAMBDAFICATOR. Live Eclipse refactoring tool for Java lambda expressions that automates two refactoring methods. The first one, named *AnonymousToLambda*, turns anonymous inner classes into lambda expressions. The second one, called *ForLoopToFunctional*, converts collection loops into lambda expressions. While programming, developers receive several refactoring suggestions through the hint mode of the Eclipse IDE. After accepting the most suitable refactorings, the tool applies the refactoring techniques to all the source code files or generates a quick preview that lists all the modifications that will be executed. In this way, developers know about the changes that will be applied and can refactor their software systems in a few seconds [FGLD13].

HARP. It is a PyCharm plugin that facilitates comprehensive analysis spanning computation graphs and the Python code that hosts them. It accomplishes this through a collection of innovative techniques that enable a holistic approach to analysis. One of HARP's notable techniques is analytics-conscious speculative analysis, which navigates the complexities of Python. By adopting this approach, HARP can proactively anticipate and overcome potential issues, ensuring a smoother analysis process. Furthermore, HARP utilizes a unified representation that enhances computation graphs. This augmented representation captures all knowledge relevant to the holistic analysis, enabling a comprehensive understanding of the system under analysis [ZZS20].

refactorix. VSCode extension for JavaScript and TypeScript projects that search for Split and Extract Variable refactoring opportunities on the source code [Fow18]. It aims to help developers change and improve their systems faster. The refactoring methods are accessible through a menu they need to call when they intend to refactor their software systems [Oet].

AntiCopyPaster. Java IntelliJ plugin that tracks pasting of code fragments within the IDE and automatically suggests the *Extract Method* refactorings to prevent duplicated code. Unlike existing approaches, AntiCopyPaster integrates seamlessly with the developer's workflow and employs a classification model to determine whether a code fragment should be extracted. When a developer copies and pastes code, the plugin searches for duplicates in the currently opened file, briefly waits for potential code edits and makes a refactoring suggestion based on various features (cf. Figure 2.6, p. 28) [AIK⁺23].

ChatGPT. ChatGPT is a variant of the GPT language model developed by OpenAI, designed for generating human-like text responses conversationally. It is specifically fine-tuned and optimized for tasks involving natural language understanding and generation, making it well-suited for chatbots, virtual assistants, and other conversational AI applications [DSZ⁺23]. ChatGPT can also help its users refactor their code. It would only be necessary for users to provide the code block. Then, ChatGPT would offer some general tips and references to help understand and perform code refactoring in different programming languages. Despite these facts, we should note that ChatGPT does not allow to import projects as input. In other words, all the suggestions it would provide

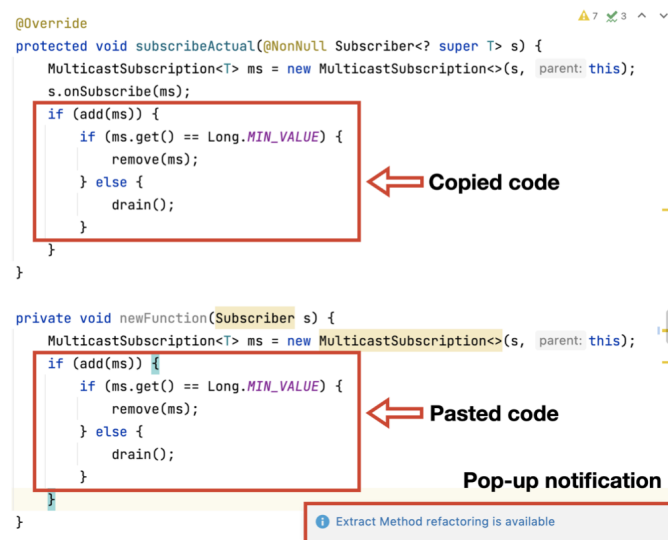


Figure 2.6: AntiCopyPaster identifying *Extract Method* opportunities (Figure 2 of [AIK⁺23]).

would be based solely on the piece of code provided. Besides, it would only work “on-demand” and not in real-time while its user was programming its code [Aby]. Also based on ChatGPT, *JetBrains research team* created an approach to apply the *Extract Function* refactoring [Fow18] on VSCode [Resc].

GitHub Copilot. It is an AI-powered coding assistant developed by GitHub in collaboration with OpenAI. It is designed to help developers write code more efficiently and effectively [NN22]. While Copilot offers a wide range of features and support for various programming tasks, it includes refactoring assistance as one of its capabilities. When it comes to refactoring, GitHub Copilot can mainly: (i) identify and suggest refactoring opportunities, (ii) automate refactoring tasks, and (iii) show examples and best practices. Besides, it supports various programming languages and integrates directly with popular code editors like Visual Studio Code. It is designed to assist developers during the entire development process, from writing code to testing and debugging [Nub].

Dipongkor et al. [DAN18] proposed an approach that analyzes the frequency of calls to different dependencies in each envied class. Then, considering that, it suggests the most frequently called envied class as a possible *Move Method* refactoring candidate.

Kimura et al. [KHIK12] proposed a technique based on method traces to identify *Move Method* refactoring opportunities. Their approach executes a given program and analyzes the trace of each method. Then, the approach counts the invocation of each method per execution phase. After, it creates a visual representation of the phase count data in which the refactoring candidates are displayed. Figure 2.7, (p. 29) represents an example of the phase count view. As can be seen, there are four methods in a class. The first three methods have similar patterns. However, the method “detectClone” differs from the others and is classified as irregular. Therefore, it is pointed out as *Move Method* candidate.

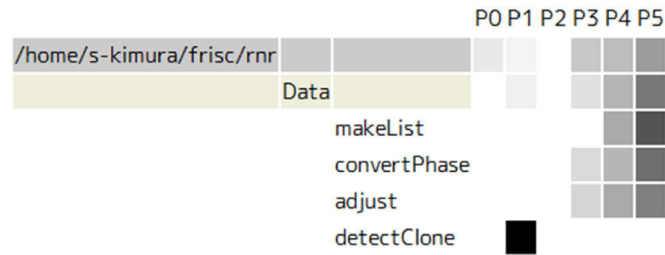


Figure 2.7: An example of a phase count view (Figure 4 of [KHIK12]).

Zhang et al. [ZXX⁺22] developed the first automatic refactoring tool that specializes in detecting and refactoring non-idiomatic code, commonly known as anti-idiom code smells. Their approach targets nine specific types of anti-idiom code smells, including *Chain Comparison*, *Truth Value Test*, and *Loop Else*. Their primary goal was to transform these anti-idiom code smells into idiomatic code that accomplishes the same functionality. To achieve this, they performed in-depth analysis of the abstract syntax tree (AST) transformations required to convert a piece of anti-idiom code into its idiomatic counterpart. They have identified four fundamental atomic AST-rewriting operations applicable across all idioms. These atomic operations were building blocks for the subsequent refactoring steps associated with each Pythonic idiom.

Brown et al. [BJB⁺20] introduced a refactoring technique that converts a C++ for loop into a GrPPI pipeline pattern. This pattern consists of multiple stages executed concurrently to process a sequence of data items indexed by the loop. The technique supports sequential stages, which process one data item at a time, and farm stages, which process multiple items in parallel. When applying the refactoring to a singular farm pattern, the code transforms a pipeline with a single farmed stage. The refactoring procedure involves creating a GrPPI pipeline object and inserting a source object responsible for returning consecutive values for the variable x as *optional* items. When no more data items exist, the source object returns an empty value. The pipeline stages within the original loop body are converted into a sequence of lambda expressions that follow the source object. These lambda expressions represent the transformed stages of the pipeline, allowing for concurrent processing of the data items. The refactoring technique effectively enables the adoption of the GrPPI pipeline pattern in the C++ codebase, enhancing its efficiency and potential for parallelization. Figure 2.1, (p. 29) and Figure 2.2, (p. 30) show an example of a for loop before and after being refactored by using the methodology presented by Brown et al. [BJB⁺20].

```

1 for (int i = 0; i < NUM_ELEM; i++) {
2     #pragma grppi seq stage
3     xs[i] = s1(xs[i]);
4     #pragma grppi farm stage 4
5     xs[i] = s2(xs[i]);
6 }

```

Listing 2.1: For loop before being refactored

```

1 parallel_execution_native pipe { };
2 pipeline(pipe,
3     [i=0,max=NUM_ELEM]() mutable -> std::optional<int> { // source
4         if (i < max) return i++;
5         else return {};
6     },
7     [&](int i) { // stage 1
8         xs[i] = s1(xs[i]);
9         return i;
10    },
11    farm(4, [&](int i) { // stage 2
12        xs[i] = s2(xs[i]);
13        return;
14    }));

```

Listing 2.2: For loop after being refactored

2.3.2 Code Metrics and Heuristics

We can also identify possible refactoring opportunities by measuring specific code metrics that assess particular code properties or using heuristics [LTOT08, Hal77a]. In this category, we found several approaches and tools, which will be described below.

ExtC. Eclipse extension that provides information about the cohesion of Java classes to help developers understand the structure of their programs. The cohesion is calculated using Eclipse Metrics2 plugin⁴. It also applies clustering algorithms using the class members to group them according to their affinity. With this tool, developers can know specific classes that should be extracted to create a simpler, better-design code with highly cohesive components. Therefore, ExtC suggests Extract Class refactoring opportunities. These recommendations are provided visually by a GUI (cf. Figure 2.8, p. 31) [CAGA11].

ARIES. Eclipse plugin that focus on detecting and applying *Extract Class* refactorings to reduce *Blob* smells [BDLM⁺12]. First, it parses each class to build a $n \times n$ matrix, where n is the number of methods of the class being refactored. A cell ij on the matrix represents the possibility that method i and method j should be declared and implemented in the same class. This likelihood is assessed by measuring their coupling degree, which is obtained through a weighted average of the Structural Similarity between Methods (SSM) [GS06], the Call-based Dependence between Methods (CDM) [BDO11], and the Conceptual Similarity between Methods (CSM) [MPF08].

Couto et al. [CRT18] proposed an semi-automatic approach that recommends *Move Method* refactorings. To do so, it relies on assessing the metrics of the Quality Model for Object Oriented Design (QMOOD) [BD02a, RC15, BD02b]. QMOOD links 11 object-oriented properties to the appropriated quality metric. Then, it identifies six quality attributes and the proper equations to measure them. Table 2.4, (p. 31) and Table 2.5, (p. 31) synthesize each metric and quality attribute.

⁴Eclipse Metrics2 Plugin, <http://metrics2.sourceforge.net>

Table 2.5: Measure of each quality attribute (Table 6 of [BD02a]). (Continued)

<i>Quality Attribute</i>	<i>Equation</i>
Flexibility	$0.25 \times Encapsulation - 0.25 \times Coupling + 0.50 \times Composition + 0.50 \times Polymorphism$
Understandability	$-0.33 \times Abstraction + 0.33 \times Encapsulation - 0.33 \times Coupling + 0.33 \times Cohesion - 0.33 \times Polymorphism - 0.33 \times Complexity - 0.33 \times Size$
Functionality	$0.12 \times Cohesion + 0.22 \times Polymorphism + 0.22 \times Messaging + 0.22 \times Size + 0.22 \times Hierarchies$
Extendability	$0.50 \times Abstraction - 0.50 \times Coupling + 0.50 \times Inheritance + 0.50 \times Polymorphism$
Effectiveness	$0.20 \times Abstraction + 0.20 \times Encapsulation + 0.20 \times Composition + 0.20 \times Inheritance + 0.20 \times Polymorphism$

Alzahrani [Alz20] presented a new metric that measures the cohesion of a specific class based on similarities between pairs of methods in terms of clients' usage. By computing this similarity, he could identify methods with low client cohesion that should be grouped and extracted to a new class by performing the *Extract Class* refactoring.

Meananeatra [Mea12] introduced a methodology that creates an optimum sequence of refactoring techniques to solve Long Method smells. He used a four-criteria method to select the optimal solution, composed by (i) the number of code smells removed, (ii) the maintainability degree of the solution, (iii) the size of the created sequence, and (iv) the number of elements of the source code changed. This approach included the *Replace Temp with Query*, *Introduce Parameter Object*, *Preserve the whole Object*, *Extract Method*, *Decomposed Conditional*, and *Replace Method with Method Object* refactoring methods.

Salgado [Sal20] created an extension for Visual Studio Code that immediately and continuously assesses code quality metrics as heuristics to find extract-related refactoring sequences opportunities. Therefore, this tool supports metrics like lack of cohesion, cyclomatic complexity, or dependency between methods to detect refactorings such as *Extract Variable*, *Extract Method*, and *Extract Class*. To measure each metric, this approach uses the AST of the software under analysis and some mathematical formulas.

Napoli et al. [NPT13] proposed an approach based on measuring the modularity of code to suggest *Move Method* refactorings. According to their strategy, modularity is assessed through structural and object-oriented metrics. As structural metrics, they used the *Fan-in* and *Fan-out* counters defined by Henry and Kafura [HK81]. Then, Napoli et al. applied the *Jaccard similarity coefficient* [TC09a] to them. As object-oriented metrics, they measured the lack of cohesion and coupling between objects. Then, considering the values of both types of metrics, they pointed out possible *Move Method* opportunities.

Charalampidou et al. [CAC⁺17] proposed an approach named SRP-based *Extract Method* Identification. This approach identifies fragments of code that collaborate between them by assessing their cohesion. The authors believe that by extracting code to a new method, they can increase the cohesion of the new code.

Pappalardo and Tramontana [PT13] identify and suggest *Extract Class* opportunities. Their approach was based on the strength of the interactions of each method. To assess this strength, they measure how well the methods of a class are coupled to that exact class. They counted the invocations of each method, the size of the parameters used on each invocation, and the attributes accessed through it.

2.3.3 Change History Analysis

Another way to identify refactoring opportunities is by analyzing the evolution of software systems and the changes they suffer over time [CFYH18]. On this subject, we found two different real-time refactoring tools.

Kobold. Live refactoring tool for client-side web applications. This tool captures the user actions and the change history to detect code smells and suggest proper refactoring techniques in real-time. To do it, developers have to create an account on Kobold, and then they can access all the code smells captured and the refactoring opportunities identified. With this tool, programmers can create several web application versions considering the recommended refactoring methods. Then, they can verify the state of their systems after implementing different possible refactoring methods [GGGR18].

Refactoring-Bot. Autonomous bot that performs Java refactorings to address common code smells, including *incorrect order of language modifiers*, *missing override annotations*, *commented-out code*, and *unused method parameters*. To maintain uninterrupted workflow for developers, the bot operates autonomously, presenting its suggested changes via pull requests for asynchronous review. This allows developers to conveniently assess the modifications using familiar tools provided by the version control platform at their preferred time [WB19].

BeneFactor. Eclipse live refactoring tool that suggests automated refactorings to programmers trying to manually refactor their code. It has two different components. The first one is the live detection module that identifies which manual actions were, in fact, refactorings. The second one is the code module, which recommends possible restructuring actions to automatically finish the manual refactoring that the developers already started to implement. To detect if a developer is refactoring his source code, the tool uses a code-changed based approach, verifying all the changes performed in the code AST. This approach considers refactoring activities like copying and pasting, variable or method renaming, selection of code, etc. Then, when BeneFactor detects these actions, it tries to finish them correctly without reversing the changes already implemented by the programmer. It uses the Eclipse Language Tool Kit to revert the source code and recover any partial refactoring that has already started. After, it tries to finish the respective refactoring action [GMH11].

RefDistiller. As said in (cf. Section 2.1.3, p. 12), it is a static analysis tool that inspects code to find possible refactoring anomalies. It provides feedback to developers on each anomaly, specifying

its location and possible causes. This tool focuses on two types of refactoring anomalies: (i) missing and (ii) extra edits. RefDistiller has a Missing Edit Anomaly Detector (MEDETECTOR) that searches for edits that the developers may have forgotten when refactoring their code. It has an Extra Edit Anomaly Detector (EEDETECTOR) that searches for edit actions beyond those required to perform a specific refactor [ASK14, ASM⁺18].

IntelliMerge. IntelliMerge is an advanced merging algorithm designed specifically for Java object-oriented programs, incorporating refactoring awareness and utilizing a graph-based approach. Its primary purpose is to handle three-way merge scenarios within Git projects. A typical merge scenario involves two branches, namely the left and right versions, that need to be merged together. Additionally, there is a base version, which represents the nearest common ancestor in the commit history. IntelliMerge takes this input and performs a comprehensive merge operation, producing a merged version and identifying conflicts that cannot be automatically resolved during the merging process. IntelliMerge executes its merging process through four sequential steps: code-to-graph, matching, merging, and graph-to-code. It supports refactorings like *Rename*, *Pull Up*, *Push Down*, or *Move Method* [SZZ⁺19].

RMiner. Technique that aims to detect refactoring operations applied to a Java project by analyzing the project's commit history in a git-based version control repository. This technique takes two revisions, a commit and its parent, as input and generates a comprehensive list of the refactoring operations applied between them. RMiner stands out for its highly accurate, efficient, and scalable refactoring detection capabilities, eliminating the need to build each commit individually. At the core of RMiner lies an AST-based statement-matching algorithm that does not rely on user-specified thresholds, making it resilient to noise introduced by statement restructuring during refactoring operations. The algorithm leverages two novel techniques: abstraction and argumentization. Abstraction addresses changes in the AST type of statements caused by refactoring operations, ensuring accurate matching. Argumentization, on the other hand, handles changes in sub-expressions within statements resulting from parameterization. By effectively matching the AST statements, RMiner enables the design of powerful detection rules for 15 representative refactoring types. These types include *Move Class*, *Rename Class*, *Extract Superclass/Interface*, *Pull Up Method*, *Push Down Method*, or *Rename Method* [TME⁺18].

RefactoringMiner 2.0. Eclipse tool that consists of an enhanced version of its predecessor, version 1.0, that introduces support for detecting low-level or submethod-level refactorings within the body of a method. This new capability allows RefactoringMiner to identify refactorings such as *Rename* or *Extract Variable*, which occur at a finer granularity within the code. RefactoringMiner 2.0 takes two revisions of a Java project as input to detect the refactorings. These revisions represent a commit and its parent in the directed acyclic graph that models the commit history in git-based version control repositories. By comparing these two revisions, the tool determines their refactoring operations. It offers support for detecting a comprehensive set of 40 refactoring types across six different kinds of code elements. The refactoring types include *Extract Method*, *Extract Class*, *Move Class*, or *Inline Variable*. The supported code elements encompass packages, type declarations, methods, fields, local variables/parameters, and type references [TKD22, TME⁺18].

RefactoringInsight. IntelliJ plugin based on *RefactoringMiner* [TKD22, TME⁺18] and *Kotlin-RMiner* [Resb] that analyzes code changes in Java and Kotlin software. Then, it presents the refactorings found through auto-folded suggestions complemented by their description. This plugin not only allows to view the changing differences with auto-folded refactorings but also to inspect refactorings in pull requests or in any specific commit and to explore the refactoring history of the code's methods and classes [KKS⁺21].

PyRef. Tool designed to automatically identify method-level refactoring operations in Python projects. Leveraging the capabilities of *RefactoringMiner 2.0* [TKD22], PyRef streamlines the process by taking a Git repository of a Python project as input and generating a comprehensive list of refactoring operations performed within the project. It specifically focuses on detecting nine types of method-level refactoring operations, namely *Rename Method*, *Add Parameter*, *Remove Parameter*, *Change/Rename Parameter*, *Extract Method*, *Inline Method*, *Move Method*, *Pull Up Method*, and *Push Down Method*. The detection process follows five distinct steps: (i) extracts and analyzes the code changes within the Python project, (ii) models the code elements, such as methods and parameters, to facilitate further analysis, (iii) matches the modeled code elements to identify potential refactoring operations, (iv) applies a set of refactoring heuristics to validate and confirm the detected operations, and (v) sorts the detected refactoring operations to present them in an organized manner [ALT⁺21].

RAID. Tool designed to enhance the process of reviewing code changes by instrumenting textual diffs, particularly those provided by GitHub, with information about refactorings. In typical textual diffs, refactorings are represented as a sequence of removed lines (-) on the left side and a sequence of added lines (+) on the right side. Reviewers are often required to infer whether these differences indicate a refactoring operation, which can be cognitively demanding. To address this issue, RAID seamlessly preprocesses the default diff provided by GitHub and generates an improved diff visualization that explicitly includes refactoring information. An example of this enhanced diff visualization is shown in Figure 2.9, (p. 36), where RAID adds an “R” button at the end of line four, indicating that the code has undergone refactoring. This button is included in both the source and destination of the refactoring, enabling easy two-way navigation. When the developers click on the button, RAID displays a floating window. This window provides detailed information about the detected refactoring, including the refactoring name, a short description, and the source and target file [BV21].

RefBot. Intelligent refactoring bot designed explicitly as a GitHub app, providing seamless integration into any project repository on GitHub. It offers customizable monitoring of repository quality, either through repeated pull requests or automatic execution triggered by significant decreases in quality analysis. During the analysis of pull request changes, RefBot examines the modified files to identify potential refactoring opportunities based on a predefined set of quality attributes. It aims to find the optimal sequence of refactorings to address any quality issues in the code. These recommended refactorings are then presented to the developer through an automatically generated pull request. The developer can review the recommendations and their potential impacts without interrupting the development pipeline. RefBot provides a comprehensive and detailed

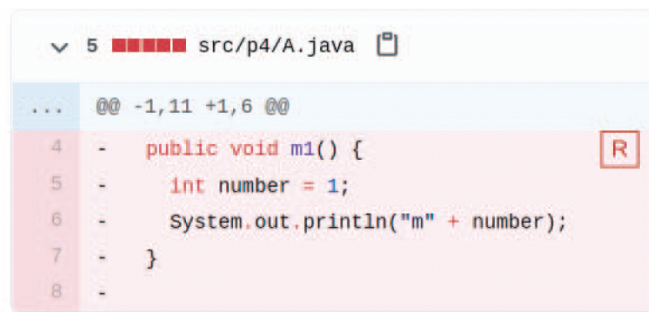


Figure 2.9: Example of the visual interface provided by RAID (Figure 2 of [BV21]).

report that allows the developer to assess the proposed refactorings thoroughly. Within this report, the developer can selectively choose which code changes to keep or ignore, aligning them with their specific requirements and goals. This tool supports refactorings like *Move Method*, *Move Field*, *Extract Class*, or *Encapsulate Field* [AOKC19].

Yoshida *et al.* [YNCI19] stated that developers tend to code without gasping the existence of code clones on their software. To mitigate this problem, they proposed a system that monitors the coding actions of developers to detect *Extract Method* candidates to reduce code clones. Basically, this system recommends *Extract Methods* to developers depending on their current activity.

Wang *et al.* [WHTK15] developed an innovative High-Performance Computing (HPC) refactoring tool designed to enhance the performance portability of HPC applications. Their approach introduces a unique mechanism that combines code refactoring with an undo feature, enabling the reversal of platform-specific optimizations to create a tunable application code. Additionally, the tool incorporates a redo mechanism that applies auto-tuning techniques to re-optimize the code. The undo mechanism plays a crucial role in the process as it allows for the removal of platform-specific optimizations that have been applied to the code. This step is necessary because auto-tuning tools typically require un-optimized code as their input. The code becomes adaptable and ready for further enhancements by undoing these optimizations.

2.3.4 Artificial Intelligence Techniques

We can also use Artificial intelligence to identify refactoring opportunities on the source code, as it happens with detecting code smells (*cf.* Section 2.2.3, p. 19). In this topic, we found some tools and approaches related to it (*e.g.*, ChatGPT already mentioned in Section 2.3.1, (p. 26)).

Code-Imp. Java refactoring that applies search-based techniques to detect refactoring opportunities. It supports 14 different refactoring actions divided into method-level, field-level, and class-level. Code-Imp uses the *Hill-Climbing algorithm* to evaluate the possible refactoring candidates. To assess each refactoring, it analyzes 28 code quality metrics with whom it calculates the fitness function. Then, the refactorings are only considered if they improve one or more metrics without deteriorating another one. This tool implements each refactoring directly on the AST, and after finishing it, it converts the AST into the code again [MC12, ÓTH⁺12].

COSP. Refactoring tool capable of identifying refactoring opportunities like God or Complex Classes by searching for problematic classes that need to be refactored, using the *Random Forest algorithm* associated with code quality metrics to predict possible code flaws. COSP uses a predefined confidence value to reject possible false-positives, which helps it alert software developers when a threshold is passed over. In the future, its developers aim to implement deep learning algorithms to execute refactoring actions by learning more about the changes made on the code [PBTP18, Pan19].

CREC. Refactoring tool that recommends possible refactorings to remove code clones. This tool uses 34 different features to train a *classifier* that aims to find code clones, suggesting *Extract Method* refactorings. Then, the classifier categorizes the features as *R-clones* or *NR-clones*. *R-clones* represent source code that should be refactored, and *NR-clones* represent pieces of code similar to others that are not code clones. For each feature classified as *R-clone*, the tool suggests that developers refactor the code [YGM⁺18].

MultiRefactor. Refactoring tool that combines search optimization algorithms with a multi-objective genetic algorithm. It takes Java code and any .jar libraries needed to compile and execute that code as input. This tool also considers different metrics, like aggregation, class design size, or coupling, to help find the refactorings. Besides, the refactorings to be detected on the code can also be configured. It detects several refactoring techniques, such as *Move Field Up*, *Remove Class*, or *Make Method Final* [MGM19].

Kurbatova et al. [KVGB20] created an approach named **code2vec**, which recommends *Move Method* refactorings to solve *Feature Envy* smells [Fow18]. It is based on capturing the syntactic structure and semantic information of a block of code to train a Support Vector Machine (SVM) classifier [MAB⁺12a, MAB⁺12b].

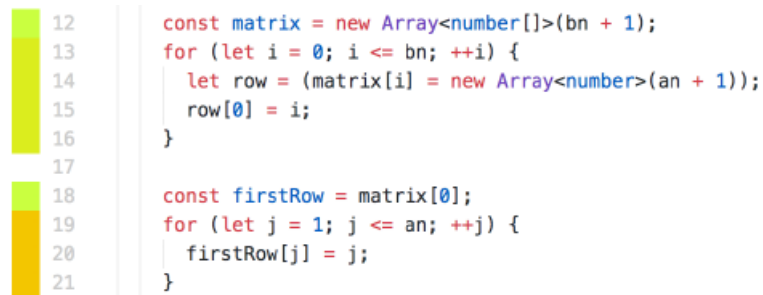
Tarwani and Chug [TC16] used a Greedy algorithm to assess the optimal refactoring sequences to be applied to a software program. Each refactoring results in a different maintainability degree, and the Greedy algorithm tries to find the chain of refactoring methods that will reach the highest maintainability possible. First, this approach verifies if a single refactoring can get the highest maintainability value and then tries to calculate a possible sequence to maximize that value. It includes refactorings like *Extract Method*, *Move Method*, *Extract Class*, *Extract Parameter*, to solve *God Class*, *Long Method*, *Feature Envy*, and *Long Parameter List* code smells.

Koc et al. [KECK12] developed a web-service that implements an Artificial Bee Colony optimization algorithm to automatically refactor the bytecode of compiled Java code. Their Bee Colony searches on a quality model that resulted from aggregating 24 object-oriented software metrics. Their web-service supported 20 refactoring actions, such as *Move Method*, *Introduce Factory*, and *Make Class Abstract*.

Chug and Tarwani [CT17] also suggested using the *A* algorithm* to identify the optimal refactoring sequences considering the number of detected code smells as the main decision criteria. To measure it, the approach calculates the Quality Decline Factor (QDF) of each class since the highest values of this metric correspond to classes with a large number of detected smells. After sorting the classes by their QDF, the classes with the highest values are selected to verify the most suitable refactoring sequence to create better source code. Then, they used the *A** algorithm,

considering the current solution and nine metrics that aim to maximize the maintainability degree. This approach supports several refactoring techniques, such as *Extract Method*, *Extract Class*, or *Encapsulated Field*, to possibly mitigate smells like *God Class*, *Dead Code*, among others.

Barbosa [Bar20] developed a tool for Visual Studio Code that used OPTICS clustering algorithm to cluster coupled lines of code in a method. Each cluster represented an *Extract Method* refactoring candidate. Besides, this tool also enabled the visualization of the clusters highlighted using different colors from a range of green and red, representing their severity. With it, developers can verify the code that should be refactored and the similarities between the clusters identified (cf. Figure 2.10).



```

12  const matrix = new Array<number[]>(bn + 1);
13  for (let i = 0; i <= bn; ++i) {
14      let row = (matrix[i] = new Array<number>(an + 1));
15      row[0] = i;
16  }
17
18  const firstRow = matrix[0];
19  for (let j = 1; j <= an; ++j) {
20      firstRow[j] = j;
21  }

```

Figure 2.10: Visualization of different refactoring candidates (Figure 4.5 of [Bar20]).

Xu et al. [XGLX17] presented a mixed approach that combines code quality metrics and a probabilistic model to identify *Extract Method* candidates. First, it assesses metrics like the number of lines of code, cyclomatic complexity, cohesion, or coupling to extract them into vectors. Then, it builds a log-linear probabilistic model based on a real-world dataset. Finally, after training the model, they apply it to assign each refactoring candidate a score and rank them according to those scores. That way, they maximize the likelihood of the collected dataset, helping to detect code that should be extracted to new methods accurately.

2.3.5 Game Theory

Game theory is a ramification of mathematics that studies possible scenarios where players should decide between several choices, which are the ones that maximize their results [Mye97]. This topic can be encompassed with identifying refactoring opportunities since it can help decide the most suitable refactorings to a specific programming context that aims to maximize quality, readability, or maintainability.

Vimala et al. [VNB⁺12] and **Bavota et al.** [BOD⁺10] proposed the use of game theory to identify the best refactorings to be applied in a software system, to increase its maintainability and quality. The authors gave weights to several properties of a refactoring method and different aspects of the source code, such as its number of classes or methods. After providing these weights, they applied game theory to them to know the most advantageous refactorings with which developers can reach better programming solutions with more quality.

2.3.6 User Direct Action

Some tools and approaches allow developers to directly manipulate their source code to refactor it without receiving any suggestions or guidance. With this refactoring approach, programmers do not need to know in advance the names or objectives of each refactoring, facilitating the execution of specific refactoring actions.

DNDRefactoring. Eclipse tool with which developers can refactor Java software systems only with drag-and-drop gestures (*cf.* Figure 2.11). With this tool, programmers drag and drop a piece of code on the desired target, using refactoring methods through simple movements. Therefore, it eliminates the need for an interface, menu, or dialog box to manipulate the source code automatically. Besides, it mitigates the problem of having the same refactoring with different names in distinct IDEs (*i.e.*, Fowler listed the Extract Method refactoring in his catalog [Fow18]. In contrast, NetBeans IDE listed the same refactoring as Introduce Method). Despite this, DNDRefactoring only supports 12 of the 23 types of Java refactoring methods included in Eclipse [LCJ13].

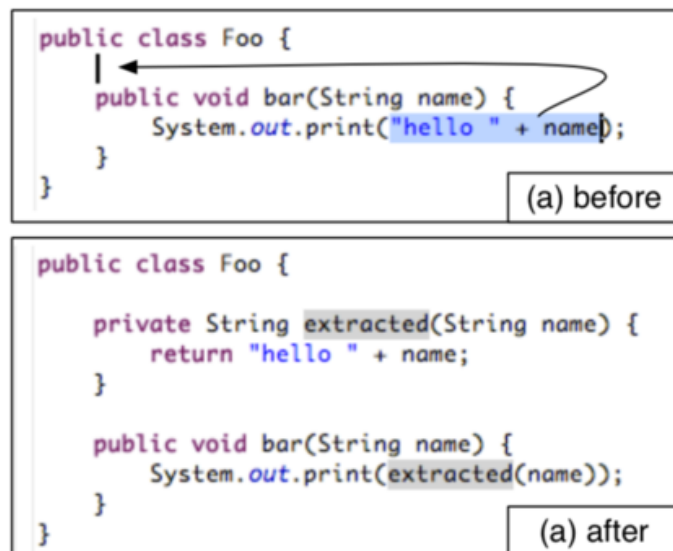


Figure 2.11: Drag-and-drop of an Extract Method refactoring (Figure 1 of [LCJ13]).

TOAD. IntelliJ tool that suggests *Extract Method* refactoring alternatives for Pharo code. This tool searches for possible code selections that should be refactored through the *Extract Method* refactoring by (i) analyzing the original code selected by the developer and (ii) searching for new code sections that are syntactically correct and meet the preconditions of IntelliJ to perform that *Extract Method*. Then, it suggests them by the similarity to the current code selection [SA19].

Imazato et al. [IHK17] proposed an approach that identifies *Extract Method* refactorings by learning information on features of *Extract Methods* performed in the past. As input, it takes the development history of several software systems. Then, it identifies the conducted refactorings and creates a model by learning them. Lastly, using the model and the source code of the new software under analysis predicts a new set of *Extract Method* candidates.

2.3.7 Slicing Techniques

As seen in (cf. Section 2.2.4, p. 20), we can also use slicing techniques to detect code smells and identify refactoring opportunities.

Rongviriyapanish and **Karunlanchakorn** [RKM15] proposed an algorithm to identify temporary variables that should be replaced with a query in a long method. This algorithm determines the statements that should be extracted and forms a query method through the slicing technique. They analyzed the relationship between statements by combining forward, backward, and chop slicing. They used forward slicing to explore the impact of each statement. Backward slicing was to check the statements impacted by a selected statement. Finally, chop slicing combines the output of the previous slicing techniques.

2.3.8 Combined Techniques

As verified in the previous sections, most refactoring tools focused mainly on identifying refactoring using only one type of technique. However, we found one case in which its authors used different or combined methodologies to detect multiple refactorings.

JDeodorant. Eclipse plugin that identifies and removes code smells by applying specific refactoring actions. This tool can have several code smells, such as *Feature Envy*, *Long Method*, or *God Class*. Besides, the tool can automatically execute *Extract Class*, *Extract Method*, or *Move Method* refactoring after developers accept the refactoring suggestion. Figure 2.12, (p. 40) shows how the tool provides feedback and support to programmers about the existence of *Feature Envy* code smells and how they have access to the possible refactoring methods [Nik, FTC07, TC09b, FTSC11, TC09a]. This plugin was also replicated to work on IntelliJ under the name of *IntelliJDeodorant* [Resa].

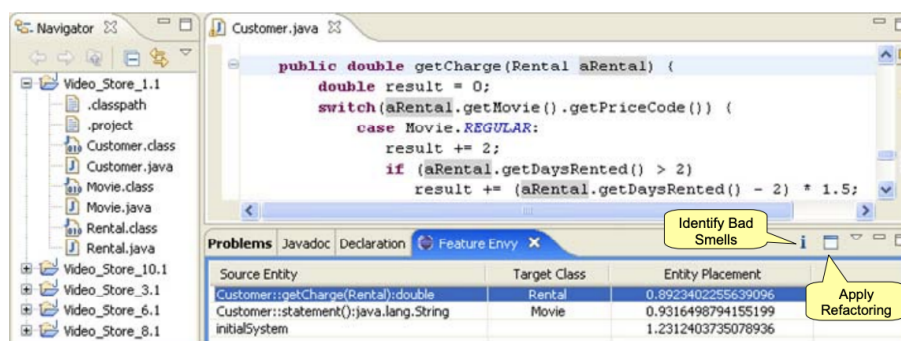


Figure 2.12: Identification of Feature Envy code smells using JDeodorant (Figure 1 of [FTC07]).

On this tool, **Tsantalis** and **Chatzigeorgiou** [TC09b] identified, suggested, and applied *Extract Method* candidates using a slice approach. It is based on combining the slices by applying a block-based slicing methodology to create a refactoring candidate. Each *Extract Method* suggested by their approach adheres to three principles: (i) the extracted code should compute a variable declared in the original method, (ii) the behaviour of the software should be preserved after refactoring the code, and (iii) the extracted code should not be excessively duplicated in the original code.

Fokaefs et al. [FTSC11] presented the approach included in JDeodorant to identify *Extract Class* candidates. This approach is based on applying a hierarchical clustering algorithm on a class through the Jaccard distance as the distance metric. This distance is calculated by comparing the similarity of the entity sets of the fields and methods of the class under analysis. The entity set of a field or method contains all the members of that class that use or are used by the element in question.

Fokaefs et al. [FTC07] and **Tsantalis** and **Chatzigeorgiou** [TC09a] also suggested a methodology to identify *Move Method* refactoring opportunities, using the distance between entities. The distance between entities is calculated using code metrics to measure the degree of relationship between entity sets. The detection of *Move Method* opportunities considers the number of entities accessed by a specific method from a target class and the distance value between that method and the target classes. The execution of the refactoring technique is done semi-automatically since it needs some input from programmers to solve future problems that may occur with Model-View-Controller based software systems.

2.3.9 Discussion and Open Issues

Refactoring is a technique for restructuring an existing software system by changing its code without modifying its behaviors. Usually, when developers refactor their programs, they expect to accomplish better solutions with more quality. To do so, executing a sequence of refactoring methods is essential to achieve better code. Besides, to successfully perform refactoring and achieve the desired goals, developers and the refactoring suggestion tools should consider the current code and the final version of it after executing a refactoring technique [Fow18].

To learn more about refactoring and the existing tools or approaches, we analyzed the state-of-the-art refactoring detection and suggestion techniques. From this study, we verified that some tools were already developed for specific IDEs that aim to help developers change their systems to achieve self-explanatory and well-designed programs with high quality [Bar20, Sal20, ÓTH⁺12].

Furthermore, we also found some essential points. First, we checked that most tools, mainly those that try to calculate the best refactoring techniques to be applied, measure their impact using code quality metrics. However, in each suggestion, even if it recommends the best refactoring possible to a particular programming context, the last word is from the developer who validates the suggestion and decides whether to accept it. Then, we think this kind of recommender should provide relevant information on each refactoring proposal, which does not happen in some cases where the recommendation only contains the name of the refactoring that should be implemented.

Thus, we enumerated the main gaps in the current state-of-the-art regarding the detection and suggestion of refactoring opportunities:

Discontinuous Identification of Refactoring Opportunities. Most refactoring tools do not inspect software systems continuously while developers are programming to detect possible refactoring opportunities. With the constant need to execute a specific tool to receive feedback, support, and guidance on their systems, developers increase their total development time. We believe that

refactoring tools that continuously try to identify refactoring opportunities while developers create their systems would be able to detect and suggest quick refactoring actions capable of improving the quality of software systems. Therefore, programmers would be able to reach better programming solutions faster.

Lack of Valuable Information. Some refactoring recommenders do not provide relevant information about the refactorings they are suggesting. Despite having some research works where there is a dissociation of the refactoring from its name, giving more relevance to the action that will be performed, most of the tools give little or no additional complementary information about each refactoring. For instance, they should include some data about the evolution of specific code quality metrics so that the developers can know which aspects will or will not improve when they implement one of the suggested refactorings or refactoring sequences.

Tools Hidden on IDEs. Sometimes, to receive refactoring suggestions, developers have to activate some options on a specific menu. However, they may not know that a particular tool exists or how it should be properly activated. Even when they are aware of its existence, they may not remember its existence while programming. Programmers continue to refactor their code manually, which reduces the value and relevance of refactoring recommendation tools.

Lack of Prediction. To identify possible refactoring opportunities, more and more smart algorithms are being used, namely, to calculate the best refactoring methods to be applied, taking into account the current programming context of a developer. Still, these approaches do not predict future refactoring methods that would help developers increase awareness of their software system.

Lack of Impact Sorting Mechanisms. Several refactoring tools point out several refactorings at a time. However, most tools do not suggest the refactoring candidates ordered on their impact on the code quality. When applying a specific refactoring, developers want to improve not only the structure of their systems but also their readability, adaptability, and maintainability. So, it would be good to have tools that help them make better refactoring decisions based on their impact on the code quality.

Lack of Visualization Techniques. The use of visual forms to represent software systems is recommended since they can help developers understand them quickly and easily. However, most refactoring tools we analyzed do not use software visualization techniques to represent the programs under analysis. We believe that if these tools could provide visual data about the systems and their code, developers could know in a user-friendly way which blocks of code should be refactored.

Table 2.6, (p. 43) summarizes the results found in the literature review on the methodologies and tools used to refactor software.

Table 2.6: Comparison between refactoring approaches and tools.

	<i>Target</i>	<i>Refactorings</i>	<i>Method</i>	<i>Visual</i>	<i>Real-Time</i>	<i>Plugin</i>
[Steb, Stea]	JavaScript, Vue, HTML	Extract Variable, Inline Variable, Rename Variable	Source Code	✗	✓	✓
[Wixa, Wixb]	React, JavaScript, Type-Script	Extract Code, Convert Components, Move Code, etc	Source Code	✗	✓	✓
[Kos]	PHP	Extract Method, Extract Variable, etc	Source Code	✗	-	✓
[Cra]	C#	Move-based Refactorings	Source Code	✗	-	✓
[Man]	F#	Extract-based Refactorings	Source Code	✗	-	✓
[FGLD13]	Java	Lambda-based Refactorings	Source Code	✗	✓	✓
[ZZS20]	Python	Python Specific Refactorings	Source Code	✓	✗	✓
[Oet]	TypeScript, JavaScript	Split Variable, Extract Variable	Source Code	✗	✓	✓
[AIK ⁺ 23]	Java	Split Variable, Extract Method	Source Code	✓	✓	✓
[DSZ ⁺ 23, Aby]	Various programming languages	Several refactorings but not moving-related techniques	Source Code, Artificial Intelligence	✓	✗	✗
[Resc]	Various programming languages	Extract Function	Source Code, ChatGPT	✓	✗	✗

Continued on next page

Table 2.6: Comparison between refactoring approaches and tools. (Continued)

	<i>Target</i>	<i>Refactorings</i>	<i>Method</i>	<i>Visual</i>	<i>Real-Time</i>	<i>Plugin</i>
[NN22, Nub]	Various programming languages	Several refactorings but not moving-related techniques	Source Code	✓	✓	✓
[DAN18]	-	Move Method	Source Code	-	-	-
[KHIK12]	-	Move Method	Source Code	✓	-	✓
[ZXX ⁺ 22]	Python	Python Idiomatic Refactorings	Source Code	-	-	✓
[BJB ⁺ 20]	C++	Lambda Refactorings	Source Code	-	-	-
[CAGA11]	Java	Extract Class	Metrics	✓	✗	✓
[BDLM ⁺ 12]	-	Extract Class	Metrics	-	-	✓
[CRT18]	-	Move Method	Metrics	-	-	-
[Alz20]	-	Extract Class	Metrics	-	-	-
[Mea12]	-	Extract Method, Introduce Parameter Object, etc	Heuristics	✗	✗	✗
[Sal20]	TypeScript, JavaScript	Extract Variable, Extract Class, Extract Method	Heuristics	✓	✓	✓
[NPT13]	-	Move Method	Metrics	-	-	-
[CAC ⁺ 17]	-	Extract Method	Metrics	-	-	-
[PT13]	-	Extract Class	Metrics	-	-	-
[GGGR18]	-	-	Change History	-	✓	✓

Continued on next page

Table 2.6: Comparison between refactoring approaches and tools. (Continued)

	<i>Target</i>	<i>Refactorings</i>	<i>Method</i>	<i>Visual</i>	<i>Real-Time</i>	<i>Plugin</i>
[WB19]	Java	Incorrect Order of Language Modifiers, Missing Override Annotations, Commented-out Code, Unused Method Parameters	Change History	-	✗	✓
[GMH11]	Java	Rename Method, Rename Variable, Extract Method	Change History	✗	✓	✓
[ASK14, ASM ⁺ 18]	-	Miss Edit, Extra Edit	Change History	-	-	-
[SZZ ⁺ 19]	Java	Rename, Pull Up, Push Down, Move Method, etc	Merging Versions	✗	✗	✓
[TME ⁺ 18]	Java	Change Package, Move Class, Rename Class, Extract Superclass, Pull Up Method, Push Down Method, Rename Method, Move Method, Move Field, etc	Change History	-	-	-
[TKD22, TME ⁺ 18]	Java	Extract Method, Extract Class, Move Class, Inline Variable, Extract Variable	Revision/Change History	✓	✗	✓

Continued on next page

Table 2.6: Comparison between refactoring approaches and tools. (Continued)

	<i>Target</i>	<i>Refactorings</i>	<i>Method</i>	<i>Visual</i>	<i>Real-Time</i>	<i>Plugin</i>
[KKS ⁺ 21]	Java, Kotlin	Extract Method, Extract Class, Move Class, Inline Variable, Extract Variable	Revision/Change History	✓	✗	✓
[ALT ⁺ 21]	Python	Rename Method, Add Parameter, Remove Parameter, Rename Parameter, Extract Method, Inline Method, Move Method, Pull Up Method, Push Down Method, etc	Revision/Change History	-	-	-
[BV21]	-	-	Revision/Change History	✓	-	✓
[AOKC19]	Java	Move Field, Move Method, Encapsulate Field, Extract Class, Pull Up Field, Pull Up Method, Push Down Field, Push Down Method	Revision/Change History	✓	✗	✓
[MGM19]	Java	Move Field Up, Remove Class, Make Method Final, etc	Optimization Algorithm, Genetic Algorithm	-	-	✓
[WHTK15]	-	Change History	-	-	✓	

Continued on next page

Table 2.6: Comparison between refactoring approaches and tools. (Continued)

	<i>Target</i>	<i>Refactorings</i>	<i>Method</i>	<i>Visual</i>	<i>Real-Time</i>	<i>Plugin</i>
[MC12, ÓTH ⁺ 12]	Java	Push Up Method, Push Down Method, Collapse Hierarchy, etc	Hill-Climbing	✗	✓	✓
[PBTP18, Pan19]	-	Solve God Class Smells	Random Forest Algorithm	✗	✗	✓
[YGM ⁺ 18]	-	Solve Code Clones	Classifier-based approach	✗	✗	✓
[YNCI19]	-	Extract Method	Change History	-	-	-
[KVGB20]	-	Move Method	Support Vector Machine	-	-	-
[TC16]	-	Extract Method, Move Method, Extract Class, Extract Parameter	Greedy Algorithm	✗	✗	✗
[KECK12]	Java	Move Method, Introduce Factory, Make Class Abstract, etc	Artificial Bee Colony Algorithm	-	-	-
[CT17]	-	Extract Method, Extract Class, Replace Type Code with State, etc	A* Algorithm	✗	✗	✗
[Bar20]	TypeScript, JavaScript	Extract Method	OPTICS Algorithm	✓	✓	✓
[XGLX17]	-	Extract Method	Metrics, Probabilistic Model	-	-	-
[VNB ⁺ 12]	-	-	Game Theory, Metrics	✗	✗	✗

Continued on next page

Table 2.6: Comparison between refactoring approaches and tools. (Continued)

	<i>Target</i>	<i>Refactorings</i>	<i>Method</i>	<i>Visual</i>	<i>Real-Time</i>	<i>Plugin</i>
[BOD ⁺ 10]	-	-	Game Theory, Metrics	✗	✗	✗
[LCJ13]	Java	Extract-based and Move-based Refactorings	Direct manipulation of code	✗	✗	✓
[SA19]	Pharo	Extract Method	Direct manipulation	✓	-	✓
[IHHK17]	-	Extract Method	Prediction Model	-	-	-
[RKM15]	-	Replace with Query	Slicing Technique	-	-	-
[FTC07, FTSC11, TC09b, Nik, TCC08, TC09a, TCC18, Resa]	Java	Move Method, Extract Method, Extract Class, Extract Clone, etc	Combined Techniques	✓	✗	✓

2.4 Code Quality Metrics

Code quality metrics are standard measures of specific properties of a software system that try to evaluate their quality [Hal77a, CK94, LMD06, BBM96, RMT09, Mar04]. Besides, these values help software developers monitor and detect design problems [Fer19, Mar04]. There is a large set of code quality metrics. Next, we define some of them:

Halstead Complexity Metrics. In 1977, *Maurice Halstead* introduced metrics capable of measuring the complexity of a software system. He stated that code quality metrics should reflect an implementation independently of its execution and results. These metrics include the volume and length of a specific implementation, the difficulty and effort to understand a specific block of code, and the estimated rate of bugs that can be delivered by a program [Hal77a].

Cohesion. This metric was identified by *Larry Constantine* in the *late 1960s* [YC79], and it measures the strength between the methods and the data of a specific class. In other words, it measures whether or not particular elements of a class belong together. Typically, a module can

have “*high cohesion*” or “*low cohesion*”. Developers should develop cohesive code that translates into high robustness, comprehension, reliability, and adaptability levels. Then, systems with low cohesion are hard to understand, change, and maintain.

Cyclomatic Complexity. This metric was described in 1976 by *Thomas J. McCabe* [McC76] and represents the number of independent paths in a specific method. This metric is important since it allows developers to limit their software’s complexity while programming. It also allows the analysis of the structure of their programs and how many tests they should do to cover all possible paths of their source code.

Maintainability Degree. This is one of the most complex and important metrics. It is measured considering the volume of a method, its number of lines of code, its cyclomatic complexity, and its number of comments. Then, if these metrics have critical values, the maintainability index will also have a bad value. Therefore, developers will have more difficulty understanding, changing, or adapting their methods [Mac].

We can use code quality metrics to compute quality attributes that can help predict or detect code smell occurrences and identify possible refactoring opportunities [BD02a]. To learn more about those approaches, we studied the current literature on the topic. Then, we were able to divide them into distinct categories based on the analysis of *source code*, *abstract syntax tree*, *change history*, and *models*.

2.4.1 Assess Quality Attributes

We can directly measure specific code quality metrics using the systems’ source code under analysis. In this case, the metrics are calculated by assessing particular quality attributes of the code. Some of the tools and approaches that use this methodology are described below.

CodeMR. IntelliJ plugin for Java, Kotlin, and Scala software systems. This tool evaluates code quality metrics and represents them in a proper interface. These metrics are divided into four categories like the ones enumerated in Table 2.7, (p. 50) [Cod].

CoreMetrics. Visual Studio Code and .Net Core extension that assesses several code metrics. This tool only works on the Windows operating system since it uses the Roslyn compiler to determine some metrics. With this tool, developers can verify the problematic pieces of code that need to be changed. It measures metrics like cyclomatic complexity, class coupling, inheritance depth, lines of code, and maintainability degree. It also enables developers to customize the respective thresholds for all the supported metrics [Kon].

SonarQube. Open-source platform for static code analysis that helps developers identify bugs, code smells, and security vulnerabilities. It evaluates code quality across multiple dimensions, such as maintainability, reliability, test coverage, cyclomatic complexity, and security, supporting more than 25 programming languages [MBM⁺19, CP13].

Tech Debt Tracker. Visual Studio Code extension for TypeScript and JavaScript systems that aims to help developers assess the quality of the functions and methods implemented. The

tool analyzes code quality metrics like method length, comprehension, nesting depth, cyclomatic complexity, number of parameters, and density of comments. Then, this plugin compares the calculated metrics with pre-defined thresholds. Using these results, the tool applies an algorithm that classifies the code with grades from *A* (least likely to be fixed or refactored soon) to *E* (most likely to be fixed or refactored soon) (cf. Figure 2.13, p. 51) [Stec].

Code City. Visualization tool that presents the source code as a city through the city metaphor, using software metrics to create visual forms, where buildings represent software classes and the districts are their packages. It assesses very common metrics like the number of packages, the number of classes of each package, the number of methods of each class, and its number of properties. With these metrics, Code City knows how many buildings it has to represent or the correct height and width of each building. Then, this tool helps developers understand their systems in a user-friendly way, where they can find parts of their code that need to be changed to improve their quality [WLR11]. This tool will be further analyzed in Section 2.5.3, (p. 58).

Table 2.7: Metrics supported by CodeMR divided into four categories [Cod].

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

Arshad and Tjortjis [AT16] suggested an approach to maintain C# software systems based on code quality metrics and *K-means data mining algorithm*. They measured different metrics, such as the number of methods per class, depth of inheritance, or the number of children. Then, they used a customized implementation of the K-means clustering algorithm on the metrics extracted from the C# programs. This technique, when applied in a software engineering context, grouping the data according to the similarity of attributes, helped find some programming patterns or outlier cases that might need further attention from developers.

Ludwig et al. [LXW18] proposed an open-source plugin that measures, groups, and visualizes several code quality metrics. Their main goal was to assess the reliability and maintainability of software based on architectural metrics such as the propagation cost and complexity metrics like the cyclomatic complexity and comment metrics.

Müller and Fritz [MF16] introduced an innovative and unique approach that deviated from previous methods. Their process automatically identified code quality issues while developers changed their code using biometrics. Applying biometrics such as heart rate variability on machine

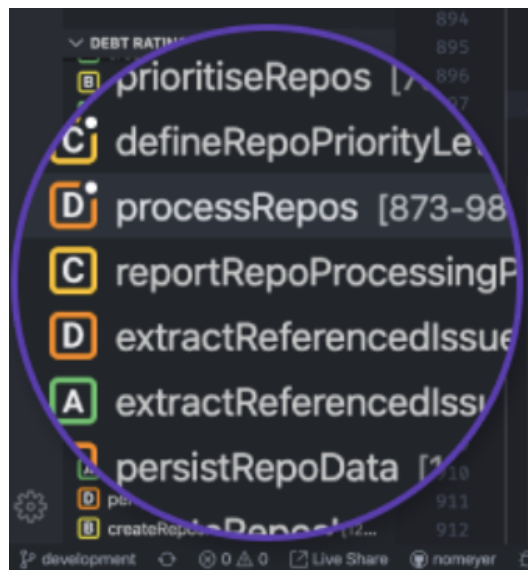


Figure 2.13: Tech Debt Tracker results. [Stec]

learning predictors successfully determined developers' difficulty when working on specific code sections. Additionally, this approach enabled the early detection and resolution of code quality concerns even before the changes were committed to the code repository. This work analyzed the evolution of metrics like cyclomatic complexity, size metrics (*e.g.*, number of lines of code), or the number of distinct modules or components directly dependent on a particular module or component within a software system.

Alqmase *et al.* [AAG19] introduced a framework that uses the expectation maximization algorithm to generate clusters based on a simplified set of metrics, including lines of code, lack of cohesion, and coupling between objects. The boundaries of these clusters are determined through source code analysis and manipulation, employing statistical measures such as mean and standard deviation.

2.4.2 Abstract Syntax Tree

This approach to calculate particular code quality metrics is similar to the one presented above. In this case, each metric is measured by verifying the different nodes of the abstract syntax tree representing the source code under analysis. Next, we summarize some tools and techniques created to measure code quality metrics based on ASTs.

CodeMetrics. Real-time Visual Studio Code extension that calculates the cyclomatic complexity of TypeScript, JavaScript, or Lua systems. This tool measures these metrics by (i) creating the AST of the source code, (ii) analyzing the number of possible paths of each AST, (iii) summing the cyclomatic complexity calculated in each node to have the value per method. The assessed values are directly presented to developers on the file editor. They can click on the code lens to collect more information about the complexity of a specific method. This tool also enables developers to customize, in advance, the maximum complexity that each method should have [Tam].

LiveMR. Extension developed for Visual Studio Code, for JavaScript and TypeScript systems, that analyzes and gives feedback on quality metrics in real-time. With this tool, programmers are always aware of the impact of their programming actions on the quality of their software systems. To provide more valuable information to developers, LiveMR also evaluates each version saved on the project's Git repository. Then, developers know the state of their systems in previous versions, and they can compare their current software to prior versions. It supports 19 distinct metrics from the number of lines of code or number of methods to the Halstead metrics [Hal77a] or the maintainability degree. Besides, the evolution of each metric is visualized through bar charts on the side menu of the IDE [Fer19].

2.4.3 Software Revisions Analysis

We can measure several quality metrics considering different versions of a software system.

An example is **SQUANER**, a code quality metrics analyzer that monitors the evolution of quality attributes in object-oriented systems. This tool connects to SVN repositories and, for each commit, measures several metrics that help evaluate the quality of that version. It analyzes metrics like the number of classes, properties of each class, and code lines in each method or class. Besides, it also assesses other essential quality aspects, such as reusability, flexibility, comprehension, or effectiveness index, and even structural design properties like coupling, polymorphism, or encapsulation degree. It then provides information about quantified values and gives relevant instructions about how developers should improve their software programs. Therefore, its main objective is to help programmers reduce the costs and efforts needed to understand and maintain their systems [HKA10].

Another example is the research conducted by **Liu et al.** [LLG⁺17]. Their study put forth the Historical Versions of Sequence Metrics (HVSM) concept to predict software defects in continuous versions using a recurrent neural network. Their approach aimed to enhance defect prediction capabilities by leveraging historical data and sequence-based metrics.

Lakshmi and **Maheswari** [LM16] conducted defect prediction by utilizing object-oriented metrics and software version history. Their approach involved ranking the most impactful attributes from the dataset to enhance prediction accuracy. Upon completion of the prediction process, the modules were categorized based on severity, allowing for the estimation of the cost associated with resolving the identified issues.

2.4.4 Models Definition

We can also use models representing our software systems to assess specific code quality metrics. **iPlasma**, a tool for Java and C++ programs, uses quality metrics and model extractors to evaluate the quality of software systems such as Eclipse or Mozilla. This tool creates a model of the system under analysis to extract valuable and relevant data about the program, such as its functions, methods, variables, parameters, or inheritance relationships. After that, it analyzes the developed model using four distinct categories of quality metrics: *size metrics*, *complexity metrics*,

cohesion metrics, and *coupling metrics*. It also inspects the model structurally to check possible deviations from good design practices based on detecting code duplication [MMM⁺05].

2.4.5 Discussion and Open Issues

Code quality metrics measure some quality attributes of software systems [LMD06]. Several metrics help developers know if their programs have quality and whether they are easy to read, understand, adapt, and maintain. The most common metrics to appear in different approaches are the number of lines of code, the cyclomatic complexity, and the cohesion degree.

We verified that various tools and approaches aim to calculate and analyze code quality metrics. However, they have some limitations that should be explored more to accomplish the full potential of this topic and help developers produce software of higher quality. Next, we describe all the problems that we found with the current state-of-the-art on this concept:

Discontinuous Analysis of Code Quality Metrics. Most code quality metrics tools and approaches do not analyze the respective metrics while developers implement their systems, as it happens with the code smells detection (*cf.* Section 2.2, p. 15) and refactoring recommendation tools (*cf.* Section 2.3, p. 23). Then, developers do not receive feedback about specific metrics and their evolution in real and programming-time. Developers cannot completely know about their code quality without quick and continuous feedback. So, programmers would benefit from code quality metrics tools based on the regular analysis of metrics to improve their software systems in a faster way.

Lack of Information about each Metric. One aspect we considered problematic in the quality metrics analysis tools is the lack of information about each supported metric. Without information about which metrics are supported or what elements they measure, developers may doubt the aspects they evaluate and their relevance. If developers have to read the documentation that supports these tools, they will increase the time needed to implement quality software systems even more. Besides, this documentation is sometimes quite small, lacking detail.

In this way, we thought it would be valuable for tools that analyze quality metrics to provide data to their users about the metrics they analyze in textual or visual forms to help them fully understand and use the tools. Thus, they could create good programs, with better code, in a reduced time.

Table 2.8, (p. 53) presents the tools and techniques we found on code quality metrics.

Table 2.8: Comparison between code quality metrics approaches and tools.

	<i>Target</i>	<i>Metrics</i>	<i>Real-Time</i>	<i>Plugin</i>
[Cod]	Java	Class Coupling, Lack of Cohesion, Cyclomatic Complexity, etc	-	✓

Continued on next page

Table 2.8: Comparison between code quality metrics approaches and tools. (Continued)

	<i>Target</i>	<i>Metrics</i>	<i>Real-Time</i>	<i>Plugin</i>
[Kon]	-	Cyclomatic Complexity, Class Coupling, Lines of Code, Maintainability, etc	-	✓
[MBM ⁺ 19, CP13]	More than 25 programming languages	Reliability, Test Coverage, Cyclomatic Complexity, etc	✗	✓
[Stec]	TypeScript, JavaScript	Function Length, Cyclomatic Complexity, Argument Count, Nesting Depth, Comment Density, etc	-	✓
[WLR11]	Java	Lines of Code, Number of Classes, Number of Methods per Class, etc	✓	✓
[AT16]	C#	Methods per Class, Depth of Inheritance, Lack of Cohesion, Number of Public Methods, Response For a Class, etc	-	✗
[LXW18]	-	Architectural Metrics, Complexity Metrics	✗	✓
[MF16]	Online Code	✓	✗	
[AAG19]	-	Lines of Code, Lack of Cohesion, Coupling Between Objects	✗	✓
[Tam]	TypeScript, JavaScript, Lua	Cyclomatic Complexity	✓	✓
[Fer19]	TypeScript, JavaScript	Cyclomatic Complexity, Halstead Metrics, Maintainability Degree, etc	✓	✓
[HKA10]	-	-	✗	-
[LLG ⁺ 17]	-	Sequence Metrics	✗	✗
[LM16]	-	Object-Oriented Metrics	✗	✗
[MMM ⁺ 05]	Java, C++	Lines of Code, Cyclomatic Complexity, Tight Class Cohesion, etc	-	-

2.5 Software Visualization

Software systems are increasingly complex, gathering different information and features [CZ11]. Maintaining a system is one of the most costly and challenging tasks in the software development cycle [BGE95, KHFG16]. Therefore, it became necessary to look for an approach that facilitates software comprehension to help change and maintain it more easily.

It is well-known that humans learn better from graphical elements than from numerical or textual data [Bie89, FLC⁺02, HCL05]. Using visualization techniques to represent software systems can help their developers by reducing the total development time and cost, increasing their success in achieving good results [SWM00].

Software visualization dates back to *before 1960* when engineers represented their systems using diagrams. In 1959, *Haibt* [Hai59] researched further on this topic and developed a program that created flowcharts that showed Fortran or Assembly software systems. Then, three years later, *Knuth* [Knu63] developed a similar approach where he integrated some documentation about the source code to help generate the correct flowcharts. Years passed, and in 1980, the main research about software visualization changed. It began focusing on the implemented behaviors of software systems and not only on their code or architecture. Thus, in 1981, *Baecker* [Bae98] released the movie “*Sorting Out Sorting*”⁵ that provided visual information to software developers about some sorting algorithms like Quick-Sort, Bubble-Sort or Shell-Sort, to help them understand these algorithms.

Since then, several research projects aimed to study the concept of software visualization with other topics such as 3D visualization (*cf.* Figure 2.14) [Rei95, YM98a]. Moreover, software visualization started to be connected to the implementation step in the software development stage, assisting the refactoring recommendations provided by several tools or IDEs [MHB10].

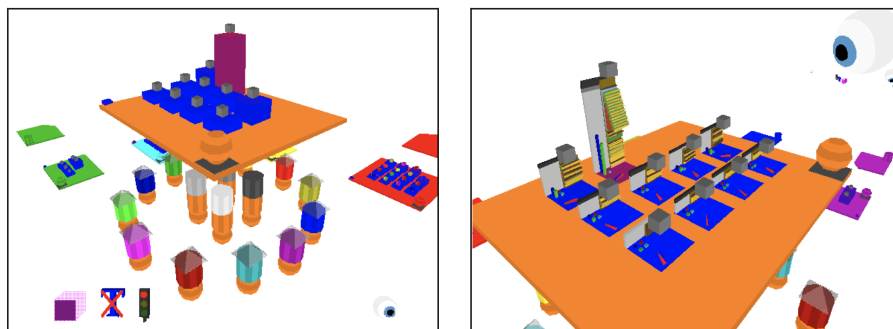


Figure 2.14: 3D visualization of a software system (Figures 2 and 3 of [YM98a]).

Therefore, using visual representations, software systems can tell several stories using a more exciting and appealing approach, helping those analyzing them [FLC⁺02, HCL05].

Several tools and approaches use visual metaphors to visually represent software systems. Visual metaphors in software development allow developers to understand more easily their programs [AD07, YM98b]. Table 2.9, (p. 56) summarizes some visualization techniques and

⁵Sorting Out Sorting, <https://www.imdb.com/title/tt0210301/>

metaphors. One of the most popular metaphors for visualizing software systems is the city metaphor [KM00].

Table 2.9: Summary of possible visualization methods and metaphors (Table 1 of [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
		Metrics	4.3	3D Clustered Graph	3D clustered graph
				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

Next, we describe some of the most relevant methodologies and tools, considering one of our research project's goals: to provide feedback, support, and guidance to developers about their software programs to improve them.

2.5.1 Code-Line-Level Visualization

Code-Line-Level tools aim to represent software systems at the source code level. The approaches using this visualization methodology are usually very low-level. However, they still can provide a high degree of abstraction that regular code editors do not have [Bar20].

SeeSoft. Code-line-level visualization tool that represented lines of code through colored squared pixels. Each line is illustrated by a pixel, and its length is proportional to the length of the line of code. The colors assigned to the pixels depend on the structure of the code implemented in that line. To represent, at the same time, several source code files, the tool puts the different pixels of each file side-by-side [ESS92].

Sv3D. Code-line-level 3D visualization tool, inspired in SeeSoft [ESS92], that illustrates the different lines of code in 3D space (*cf.* Figure 2.15, p. 57). The tool has more space to display more information, using visual parameters like x-position, y-position, height, depth, and color. However,

this advantage can also be a disadvantage since more data is being displayed simultaneously to developers, which can overload them with information [MFM03, CZ11].

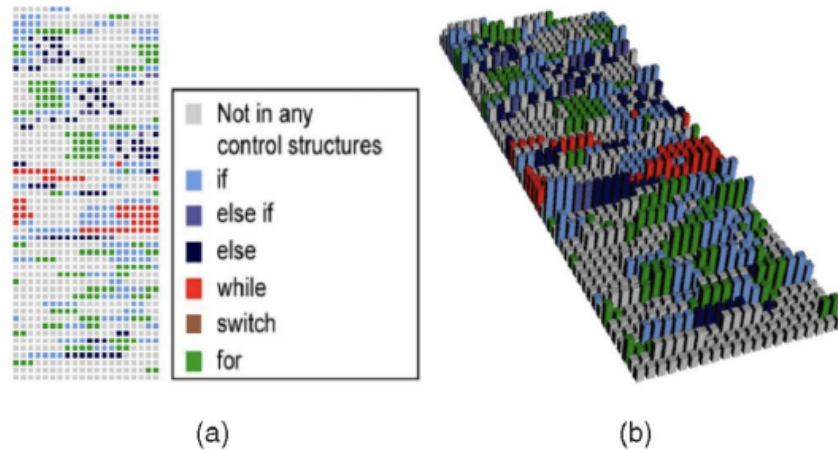


Figure 2.15: Comparison between SeeSoft and Sv3D: (a) SeeSoft, (b) Sv3D (Figure 1 of [CZ11]).

CTRAS. Crowdsource testing approach that leverages code clones to enhance the information provided in bug descriptions and enhance the effectiveness of reviewing test reports through a comprehensive and understandable report [HFJ⁺19].

SAMOA. Visualization tool designed to aid in software maintenance activities. It visually represents source code, third-party libraries, and historical data using pie charts. The tool facilitates detailed analysis of the structural and evolutionary aspects of systems. It is implemented as a web application and can be accessed on a standard computer screen [ML13].

2.5.2 Class-Level Visualization

Class-Level methodologies represent the inner functionalities of a class, which can help developers understand each class position and responsibilities in the overall system [Bar20].

Class Blueprint. Class-level visualization methodology that helps developers know how their classes work and what their responsibilities on the overall system are. In particular, this technique displays the structure of the classes under analysis, the flow between methods, and the attributes accessed by them (*cf.* Figure 2.16, p. 58). The main objective of Class Blueprint is to create distinct patterns on the classes analyzed to help developers understand, quickly and in a user-friendly way, what happens inside a class and how everything relates [LD01, DL05].

Edge Evolution Film Strip. Class-level visualization that showcases the evolution of the relationship between two modules of the software. Each module is represented by a graph, where the vertices correspond to the module's classes, and the edges correspond to the relationships between those classes. The relationship between two modules is represented by existing connections between the graphs corresponding to those modules. The versions of the relationships between the modules are arranged vertically on strips. This technique is used in tools like **Softwareonaut**. It visually

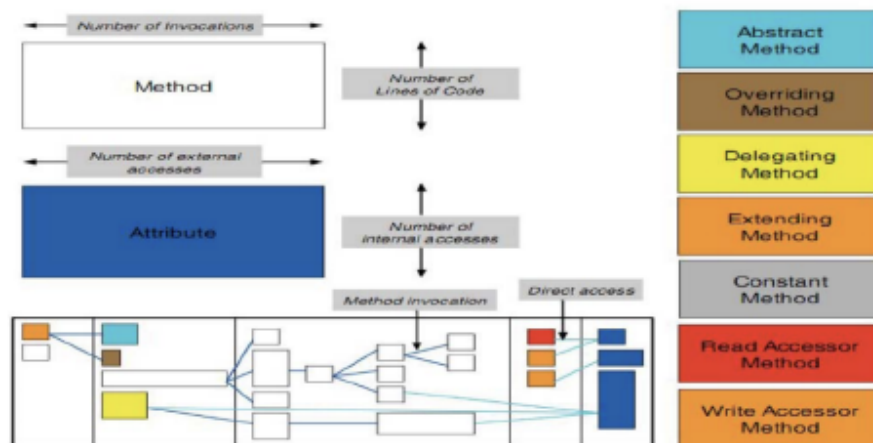


Figure 2.16: An example of the Class Blueprint technique (Figure 2 of [CZ11]).

represents large-scale software, offering three perspectives to visualize them: (i) exploration, (ii) detail, and (iii) map. [LL07].

2.5.3 Architecture-Level Visualization

Caserta [CZ11] stated that graphically representing the architecture of software systems is one of the most important ways to translate them. We can represent the different architectures using 2D or 3D methodologies or metaphors.

City Metaphor. Metaphor that represents a software system as a city that contains different districts and buildings with distinct characteristics. With this visual strategy, developers can understand and analyze their systems better and more intuitively. With the city metaphor, software packages are represented in the city as its districts. Then, each district has buildings that illustrate the classes of each package. Each building has its appearance, with specific physical properties like weight, height, and color. These characteristics represent different metrics of each class, such as the number of properties, the number of methods, etc. Figure 2.17, (p. 59) presents all the aspects mentioned above of city metaphor included on the tool Code City [WLR11, Ama18].

Code City. Time-dependent software visualization tool allows developers to see how their software systems evolve over the different development iterations in real-time, and which parts of the systems need to be improved or refactored. As mentioned, this tool uses the city metaphor, where each package represents a district that includes buildings that are the classes inside a specific package. Each building has a different appearance. Its color is related to the number of lines of code of the class represented by that building. Its length and width are measured using the number of properties of the class, and its height is calculated by the number of methods defined inside the class [WLR11].

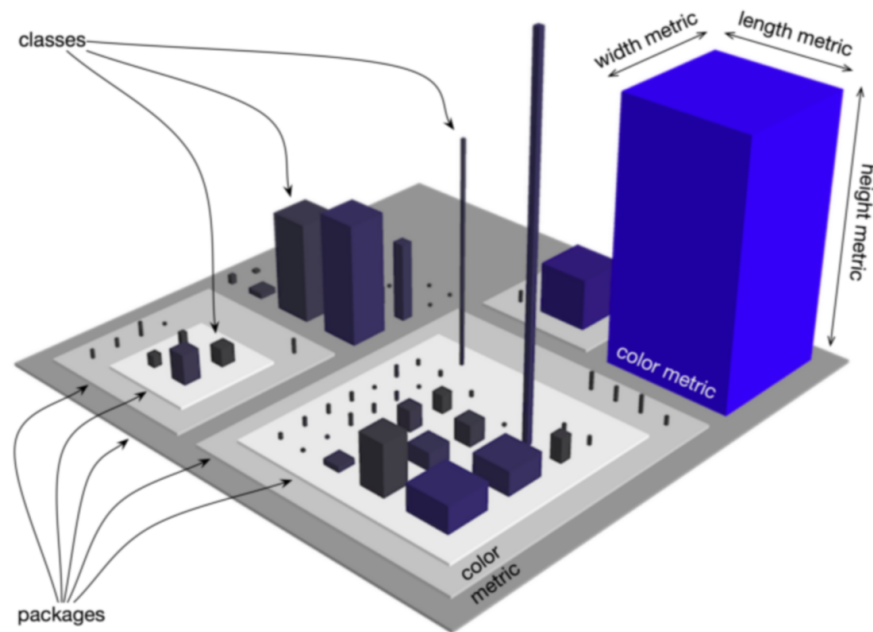


Figure 2.17: Code city applying the city metaphor (Figure 1 of [WLR11]).

VR City. Similar to Code City [WLR11], VR City enables users to observe and interact with a visual depiction of source code. The key distinction is that VR City presents the visual representation of the code within a virtual reality environment [VNP17].

Box Tree. Technique that represents software using boxes arranged in a tree format. This strategy allows for understanding the hierarchy between software entities by utilizing the adjacency of boxes corresponding to lower-level entities with boxes representing higher-level entities. The dimensions of the data are represented through variations in color, height, depth, and size of the boxes. Tools like **Bloom** use this architecture-level representation. **Bloom** helps understanding software information [Rei01].

Nested View. Visualization methodology that displays the hierarchy and relationship between software components (e.g., packages, classes, inheritance, interfaces, and their implementations). This technique consists of nested rectangles. However, it uses edges to represent the semantics of the relationship between these rectangles. This technique is used on **Creole**, an integration of Shrimp with the Eclipse IDE. With this tool, we can visualize the relationship between packages due to method calls and attribute access [PSM09].

Semantic Dependency Matrix. uses matrices to display perspectives on invocation dependencies between software modules. This technique allows for exploring details of the methods and classes involved in the dependency. Classes correspond to columns, and methods (grouped by classes) correspond to rows. Dependencies are represented by colors corresponding to the characteristics of the dependencies. This technique is also used on **Softwareonaut** [LL07].

Panas et al. [PQV07] also used the city metaphor to represent the architecture of software systems as a way of helping developers to inspect their programs. Figure 2.1, (p. 11) presents this

approach.

Geon. Visualization method that focuses on the relationship between software entities. This tool uses geon diagrams to visualize UML diagrams by mapping 2D data to 3D. Geons are 3D regular solids, like cylinders or cubes, with complementary data about how they are connected between them (cf. Figure 2.18) [IW00].

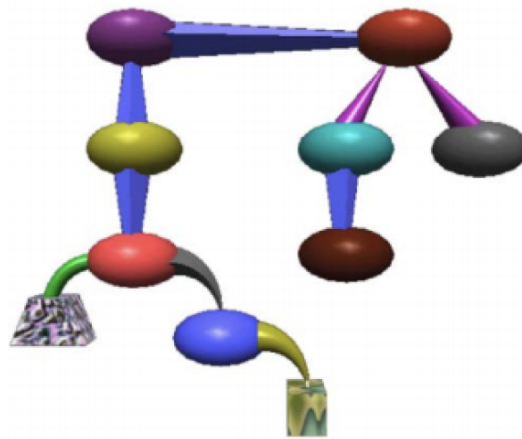


Figure 2.18: An example of a geon isualization methodology (Figure 10 of [CZ11]).

2.5.4 Other Approaches

There is a catalog of lightweight visualization, that supports the inspection of code smells, named **NosePrints**. Parnin *et al.* [PGN08] consider that classic visualization techniques are not perceptible enough and that there is a small opening on the topic of software visualization in pointing out possible flaws in software systems. Then, they proposed **NosePrints**, a visual catalog that uses software metrics to help detect code smells. Figure 2.19, (p. 60) summarizes the code smells found in a software system. This tool can be included in each category mentioned above since it analyzes the code, classes, and overall architecture.



Figure 2.19: Health screen of different classes and possible code smells (Figure 3 of [PGN08]).

This catalog enumerates the visual representation of each code smell. It also provides valuable textual information about the smell, its context, and its visual form. For instance, a *Message Chain* smell can be represented as it is seen in Figure 2.20, (p. 61). In its visual representation, we can see four different aspects. The first one is the dots on the left that indicate the length of the chain, or in other words, the number of affected variables or methods called by the chain. The second one is the

number in parentheses, (n) , that indicates how often the chain happens on the source code. Then, the dot-separated characters represent the chain under analysis. Finally, the squares on the right represent the spread that symbolizes the project and the classes in which the *Message Chain* smells appear.



Figure 2.20: Visual representation of a Message Chain smell (Figure 5 of [PGN08]).

2.5.5 Discussion and Open Issues

With large and complex software systems comes the difficulty in understanding their source code, architecture, or interactions [CZ11]. Software visualization aims to help mitigate this problem by creating clearer and human-comprehensible visual representations of software programs. However, like any topic of this research project, software visualization has some issues, too. Its main problems are related to the graphical forms used to represent the software programs and their effectiveness in illustrating the data and information of a system. Gracanin *et al.* [GME05] stated some key points regarding software visualization:

Scope. When we create visual elements on a software visualization approach, we must consider that complex abstractions can create extra comprehension problems, reducing the positive effects that software visualization techniques should bring to us. Then, we should define a scope in which our visual elements should lay on.

Representation. To create the possible visual representations, we should consider the defined scope and the abstracted data's type and level of detail. Therefore, the visual representation should be adapted to the information that we want to represent.

Visual Metaphor. The metaphors help map specific software characteristics into metaphorical objects to increase the expressiveness of the visual representations. We should verify the consistency between the software artifacts and the respective metaphor to pick the right visual metaphor to illustrate our systems. Besides, we should choose metaphors with enough detail to represent the intended software.

Automation. Software visualization tools should work in symbiosis with the development flow to update the visual representations automatically when something changes on software.

Navigation and Interaction. Visual representations should allow their users to navigate and interact directly with them.

Abstractness. Software visualization tools should be able to represent different levels of granularity and abstraction.

Table 2.10, (p. 62) summarizes the tools and approaches found in the concept of software visualization.

Table 2.10: Comparison between software visualization approaches and tools.

	<i>Representation</i>	<i>Quality Metrics</i>	<i>Refactorings</i>	<i>Real-Time</i>	<i>Plugin</i>
[ESS92]	Code-Line-Level	✓	-	-	✓
[MFM03, CZ11]	Code-Line-Level	✓	-	-	✓
[HFJ ⁺ 19]	Code-Line-Level	-	-	-	-
[ML13]	Code-Line-Level	-	-	-	-
[LD01, DL05]	Class-Level	-	-	-	✓
[LL07]	Class-Level and Architecture-Level	-	-	-	✓
[WLR11]	Architecture-Level	✓	✓	✓	✓
[VNP17]	Architecture-Level	✓	✓	✓	✓
[Rei01]	Architecture-Level	-	-	✓	✓
[PSM09]	Architecture-Level	-	-	✓	✓
[PQV07]	Architecture-Level	✓	✓	✓	✓
[IW00]	Architecture-Level	✓	-	-	✓
[PGN08]	Other	✓	-	-	✓

2.6 Liveness

Liveness enables developers to continuously and immediately observe the behavior of their software, reducing the edit-compile-run cycle and overall development time [Tan90, Gol04]. It applies throughout the software development cycle [SBF14], particularly in development and maintenance, by providing instant feedback [Lou18, Ama18].

Tanimoto introduced a liveness hierarchy in 1990 [Tan90], defining four levels (first four level of Figure 2.21, (p. 63)). At level one, visual representations like flowcharts convey information. Level two allows developers to execute code and review results afterward. Level three shows outputs immediately after execution. Level four continuously runs code in the background, updating results in real-time.

In 2013, *Tanimoto* presented a new hierarchy with the four older levels plus two new ones, as shown in Figure 2.21 [Tan13]. At level five, named “*Tactically Predictive*”, the computer not only executes the software while developers are programming but also foresees multiple following

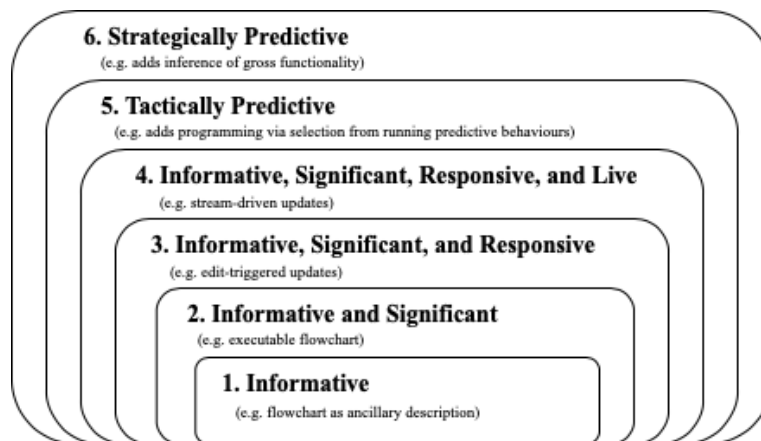


Figure 2.21: Liveness hierarchy proposed by Tanimoto (Figure 2 of [Fer19]).

actions, always staying a step ahead of them. In the last level, called “*Strategically Predictive*”, the computer predicts the next steps of the developers and strategically selects the best ones. Liveness improves software development by enhancing:

Immediacy. Sometimes, understanding or predicting specific behaviors of the software systems is hard. Instant or immediate feedback on the outputs of a program can reduce the struggle to do it by simplifying that process.

Exploration. When programmers develop their software systems, they must iterate and adapt them to their requirements. Then, software development relies on the trial-and-error process directly related to the immediacy characteristic of liveness. The faster developers iterate on their systems, the faster they can converge to the best solution possible.

Stability. With instant and continuous feedback on how software systems work and evolve, developers have more control over their outputs. Therefore, they can develop a stable system faster.

A key subset of liveness is *live programming*, where developers see code execution results in real-time, eliminating the traditional compile-run-debug cycle [McD13a, RRL⁺18]. Live programming tools offer instant execution, interactive debugging, and visualizations, enhancing learning, algorithm exploration, and rapid prototyping [SHP19, Pax02].

By reviewing the current literature on the topic of liveness, we found out that this subject can be included in *development environments*, *programming languages*, and *development tools*.

2.6.1 Live Programming Environments

Some programming development environments use specific live mechanisms. Each interacts in different ways with their users, and they have other characteristics regarding their main goals. These live environments are described below.

Circa. Live environment and language based on backpropagation algorithms allow programmers to develop live software systems. With Circa, developers can click on a specific part of their systems’

outputs and immediately inspect the sequence of code that created that result. When they click on the results, Circa checks the call on the code that created that result. In this way, developers can change their source code faster. As seen in Figure 2.22, (p. 64), a user clicked on the smaller tree, and Circa instantly showed him the calls executed to create that specific tree [Fis13].

Circa allows developers to visualize source code in its textual form and through diagrams representing the sequence of calls that generated a specific output. Besides, with it, programmers can manipulate the code directly on the diagrams, and the changes will immediately be reproduced on the source code itself [Fis13].

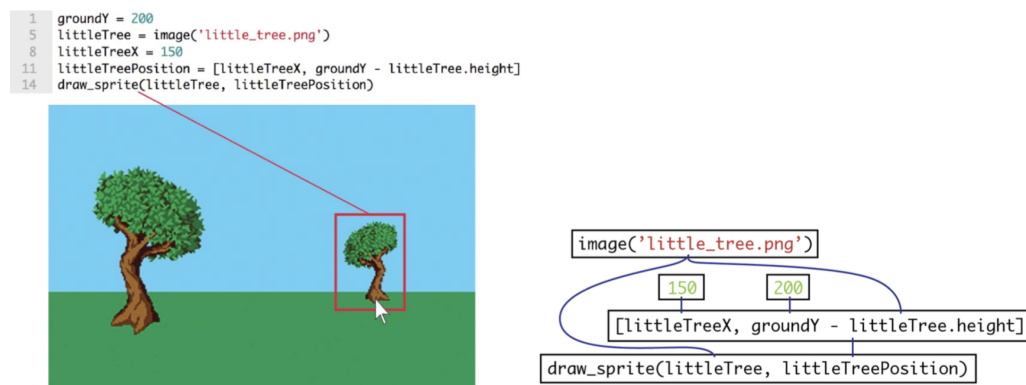


Figure 2.22: Example of a program being executed using Circa (Figures 3 and 4 of [Fis13]).

Euclase. Live environment where its users can automatically verify the results of their software programs. It allows the implementation of interactive Web Applications by combining constraints and finite state machines to establish the different behaviors of the code. This environment was idealized to help programmers develop their systems with the fewest lines of code possible. It also considers performance issues since it tries to make each modification or analysis of the outputs as quick as possible. However, when developers implement many changes in a program simultaneously, their results are only presented to them after the last change made [OMB13].

PhysicalBits. Web-based programming environment designed to seamlessly interact with a robot through a serial port connection. When users modify their programs, PhysicalBits automatically compiles and verifies them before transmitting them to the virtual machine. This process is completed in less than 100 milliseconds, allowing the robot to promptly execute the updated program [MTZ20].

Moon. Live environment that supports a homonym programming language and provides live feedback to programmers about their software systems. This environment instantly provides visual support for the programs under analysis while being developed (*cf.* Figure 2.23, p. 65). This support provided to developers relates to specific parts of the code that the software did not compile or execute. It is represented by colors, where green represents code that was correctly compiled, yellow is code that was changed, and red shows possible errors [LL13].

⁶An open source implementation based on Smalltalk, <https://www.pharo.org>

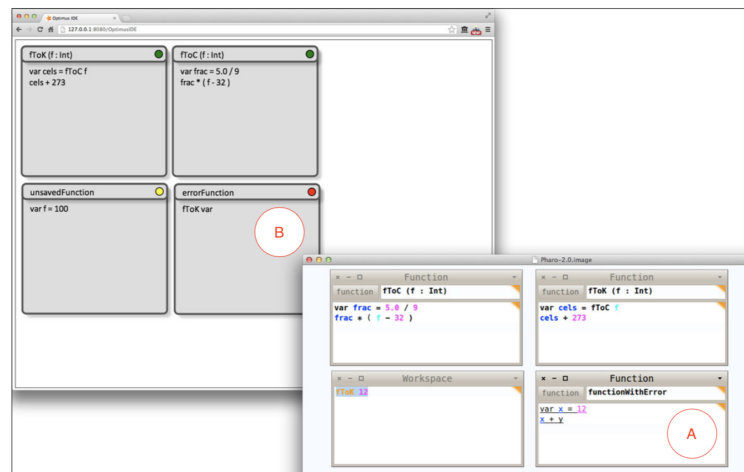


Figure 2.23: Moon development environment represented in two different ways: (A) direct integration using PHARO⁶, and (B) web-based IDE, written in JavaScript and HTML5 (Figure 1 of [LL13]).

Kanon. Live environment developed in JavaScript for data structures, providing developers with two panels. In the first one, they can edit their source code; in the second, they can check it through visual elements. The visualization panel presents the data structures using simple shapes and forms like ovals interconnected with arrows. When the developers change the code, the tool is executed again, and the visualization panel is automatically updated. Each modification is shown to the programmers by either using animations on the program's visual representation or by presenting a summary of the changes implemented [OMIA17].

Roel [Wuy02] believes there is a gap between the design and the implementation of software systems. This discrepancy may be due to different causes like the drift between the design and the code developed, problems with the documentation, or the lack of iterative development support. To face this problem and focus mainly on the idea that this gap may be caused by the lack of liveness in the development cycle, he created a framework that integrates a declarative framework in Smalltalk's development environment, and that is based on SOUL (Smalltalk Open Unification Language), a language that he developed. This framework automatically verifies any changes made to the source code and calculates several queries the program must comply with whenever a method is changed. If the system fails, the framework launches different warnings to alert developers that their programming actions are not by its design (*cf.* Figure 2.24, p. 66).

Improv. Programming environment and language created for describing high-level robot motions. It includes real-time visualization features that allow users to see the resulting movements on either physical or simulated robots [NBGL18].

Shiranui. Programming environment designed for live unit testing. With it, developers can actively examine and evaluate the behavior of their functions and quickly convert them into persistent test cases. This seamless transition from live exploration to comprehensive testing enhances the development and quality assurance processes [IMA15].

RMPATCH. Flexible and generic dynamic patch architecture. It can be customized to meet

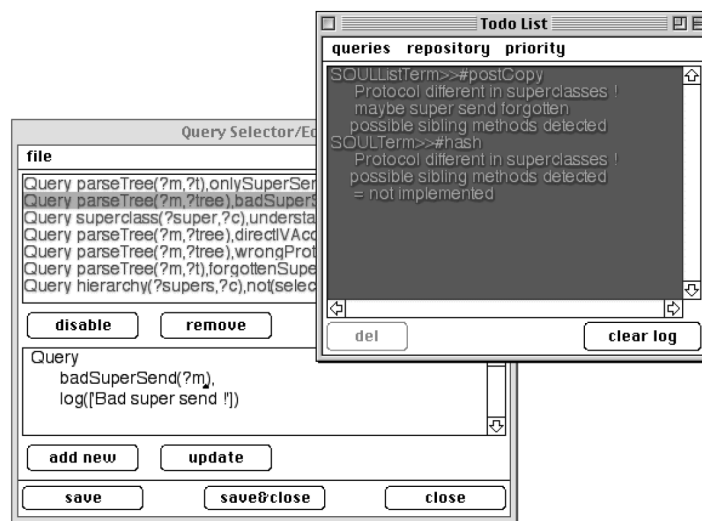


Figure 2.24: Warnings launched when there is a gap between design and implementation (Figure 5.3 of [Wuy02]).

specific domain-specific state migration requirements. This architecture enables the smooth integration of model deltas into running systems, bridging the gap between textual Domain-Specific Languages (DSLs) and executing programs. By combining textual DSL code analysis with RM-PATCH, updating running systems based on changes in the DSL, code becomes a seamless process, offering improved flexibility and adaptability in dynamic environments [RS19].

Focused Live Programming with Loop Seeds. Programming paradigm where developers can provide hypothetical values to initiate a loop iteration. This programming environment responds by emphasizing the visualization of this hypothetical loop iteration in real-time. This approach allows developers to input specific values they envision for the loop iteration, enabling focused exploration and fine-tuning of their code. The programming environment highlights and provides visualizations that concentrate on the desired loop iteration based on the provided seed values [Ler20a].

Hazel. Experimental live lab notebook programming environment specifically designed for data science tasks. The authors aim to collaboratively design both the Hazel language and its accompanying editor, exploring concepts like edit-time semantic conflict resolution mechanisms and mechanisms enabling library providers to install library-specific editor services [OVH⁺17].

TrueGrid. Novel environment that bridges the gap between traditional code and spreadsheet-like grids within the IDE. It combines ordinary code and spreadsheet cells, providing a unique programming experience. With TrueGrid, users can program spreadsheet cells using a comprehensive programming language, leveraging the advantages of source code. Developers can take advantage of the grid functionality offered by TrueGrid for interactive exploration and testing of their code. The grid interface allows visual manipulation and data analysis, facilitating a more intuitive and iterative coding process [HvdS16].

Flowboard. Platform that adopts the Flow-Based Programming (FBP) paradigm as an alternative to traditional imperative programming. It revolutionizes the learning experience by enabling

learners to construct processing nodes and connect them in a graph-like structure, facilitating the flow of signals and data. This platform includes a visual flow-based editor designed for the iPad, integrated with an Arduino board and two breadboards. This setup allows learners to connect their visual graphs to input and output electronics, enhancing the hands-on aspect of learning. A key advantage of Flowboard is its immediate feedback loop [BVZ⁺19].

DeployGround. Comprehensive framework designed to simplify the creation of web-based tutorials focusing on learning APIs, playgrounds, and application development. It streamlines the process of understanding and utilizing various APIs by providing an interactive and intuitive learning experience. The framework includes powerful features for designing interactive playgrounds and deploying applications seamlessly. By using DeployGround, tutorial creators can enhance the learning process by providing a dynamic environment where learners can actively experiment and engage with the APIs in real-time [KG18].

MIT App Inventor. Programming environment aimed at simplifying the creation of mobile apps for Android devices, especially for individuals with limited or no programming experience. It utilizes a WYSIWYG editor, allowing users to construct mobile apps by arranging pre-defined components within a web browser. One notable feature of App Inventor is its emphasis on live programming, making the platform accessible to novice developers. With live programming, developers can interact with the evolving state of their program as they build it. Any changes made within the web browser are immediately reflected in the running app on the connected device, enabling instant feedback and real-time testing [STA⁺14].

Wu et al. [WQB⁺11] presented a prototype live learning environment built and deployed on the Google Cloud platform. The primary goal of this environment is to evaluate and analyze the learning outcomes of suggested learning objects and learning models in real-time.

Zhao et al. [ZSF⁺22] introduced a solution that leverages live programming techniques to improve neural network architecture editing. They introduced an always-on visualization feature that dynamically reflects the tensor states in real-time as the user modifies the program. Additionally, the solution executes the program continuously, providing real-time warnings to prevent crashes. This proactive measure enhances the overall editing experience and stability of the program.

Martin and Colton [MC19] introduced a versatile programming environment designed to assist game developers during the implementation of their game projects. This environment offers live feedback, providing valuable guidance and support throughout game development.

Ferdowsifard et al. [FOP⁺20a] proposed a comprehensive IDE that includes a unique feature: real-time display of input/output examples during code execution. The authors aimed to utilize live programming to offer novice developers an innovative way to interact with and understand programming concepts while providing a valuable development tool for experienced programmers. The IDE serves as a versatile platform that caters to the needs of developers at various skill levels.

Beckmann et al. [BKR⁺21] introduced an innovative live game development environment aimed at facilitating game prototyping through a holistic live programming workflow. This environment prioritizes two fundamental aspects for developers: gaining insights into code modifications' visual effects and establishing a clear connection between the game's behavior and its underlying code.

McDirmid [McD13b] proposed a groundbreaking system that empowers developers to seamlessly navigate both code and program execution. By leveraging this system, developers can effortlessly transition from a method call to its definition and conveniently inspect expression values within the editor. This eliminates the laborious task of manually scouring program execution to locate relevant points for analysis. Instead, the authors devised a dedicated pane alongside the editor, which provides a comprehensive overview of program execution via tracing. These trace elements are directly interconnected with execution points, enabling efficient diagnosis and probing of identified issues or problems within the trace.

Skinner et al. [STPWR20] introduced a state-of-the-art live programming and simulation environment tailored for hardware design. This language-independent approach focuses on providing fast feedback to the user through incremental compilation, hot binary reloading, and check-pointing techniques. By utilizing incremental compilation, the environment optimizes the compilation process by recompiling only the modified portions of the code. This approach saves valuable time and resources, enabling users to receive immediate feedback on their changes and promoting an efficient development process.

2.6.2 Live Programming Languages

Besides live development environments, there are also some live programming languages. Some of them coincide with the programming environments mentioned above.

Circa [Fis13], **Moon** [LL13], **Improv** [NBGL18], **Hazel** [OVH⁺17] besides being live development environments are also live languages. **Circa** is an intuitive language similar to other programming languages such as Java, JavaScript, or C#. Combined with the **Circa** environment, developers can create their software systems more easily since they can quickly edit and change their code through visual forms that represent it (cf. Figure 2.22, p. 64) [Fis13]. The language **Moon**, when combined with its development environment, provides immediate feedback and support to programmers about their systems. Therefore, it allows them to verify their results right after implementing specific features or modifications [LL13]. **Improv** is a live programming language developed to shorten the time between programming a physical or emulated robot and seeing it moving depending on the code developed [NBGL18]. **Hazel** is a programming language that helps on specific data science tasks in real-time [OVH⁺17].

In addition to these languages, **Hermans et al.** [HJR⁺16] concluded that spreadsheets could also be considered as a way of programming. Since their users can change the sheets and automatically see the new results, the spreadsheets can also be seen as a live programming language. They are considered the most successful end-user programming language because they allow testing, reverse engineering, detecting code smells, and identifying refactoring opportunities, as with the other programming languages. However, this language has some problems like a long life span, too many users simultaneously, or lack of documentation.

SuperGlue. Live reactive language based on data-flow programming models to develop programs using components connected by signals and to create data-flow graphs. These data-flow models are supported by five distinct aspects [McD07]:

Signals. Values of the data-flow model, which simplify the communication between components.

Classes. Components of the data-flow model that encapsulate the different behaviors.

Connections. Components that associate two signals to maintain their values equal.

Extensions. Components that represent cause signals to extend classes.

Conditions. Gates created when connections and extensions are included in the data-flow model.

When there is a valid change on the source code, SuperGlue includes it in the program already being executed, as it can be seen in Figure 2.25, (p. 69), where PacMan is being developed iteratively [McD07].

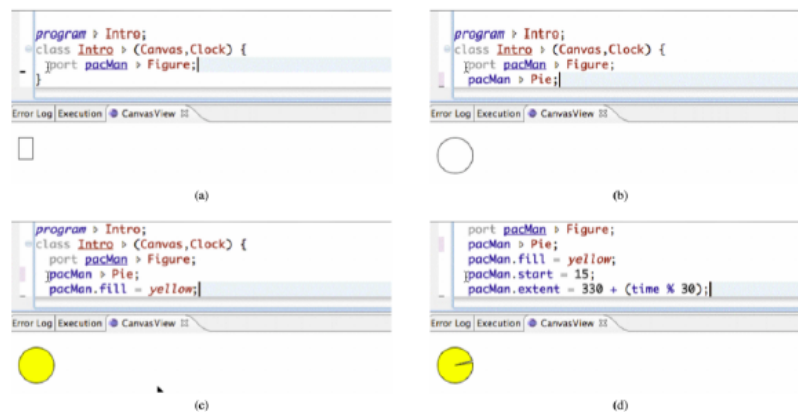


Figure 2.25: PacMan being developed on SuperGlue (Figure 1 of [McD07]).

TouchDevelop2. Live language that aims to develop tasks where the implementation cycle gets boring since they require several development iterations. This language is continuously recompiled and re-executed when developers change their software systems. It allows developers to interact bi-directionally between the source code and the UI elements. When selecting a particular UI element, they automatically choose their corresponding code, and vice-versa. Besides, they can also edit the values of specific variables by editing their respective UI components [BFdH⁺13].

Niephaus et al. [NRE⁺20] introduced a language-agnostic approach for Visual Studio Code for developing an Example-based Live Programming (ELP) system using two key technologies. This system extends ELP support to programming languages compatible with the GraalVM and seamlessly integrates with Visual Studio Code. It demonstrates how such an approach enables using ELP in polyglot programming scenarios.

DeLine [DeL21] showcased a language compiler for Visual Studio Code that employs a flexible set of “recipes” to execute various steps in the user’s data science workflow. These recipes combine the expressiveness of written notation with the user-friendly nature of a graphical user interface

(GUI). By incorporating live programming, the approach provides instant feedback to users during program edits or interactions with the GUI, creating a seamless and responsive experience.

Besides the languages mentioned above, there are also mainstream languages that include the concept of liveness, like **Swift**, **Java**, or **Smalltalk**. With Swift, a live programming language created by Apple, programmers can develop their systems and immediately verify their outputs while implementing them with their interactive playgrounds [KRB18]. Through the Java Platform Debugger Architecture, developers can change some parts of Java programs while they are executed [Ora14]. Besides, Smalltalk is an object-oriented programming language created to improve the “human-computer” interaction. With this language, developers can modify their objects while their systems are running [Kay93, RG09].

2.6.3 Live Tools and Approaches

Live tools are used as extensions on several well-known IDEs such as Eclipse or IntelliJ. Through our literature review, we found some of them.

LiveMTBD. Eclipse plugin that helps developers to transform their models in a live way. The tool asks for a concrete template, and then the developers apply their transformations directly on the provided template rather than writing each transformation rule manually. After, these modifications are recorded while being programmed, and they are summarized in a pattern. With LiveMTBD, developers can access a live demo tool that records all the changes applied to a model. Then, when they need to verify a specific modification and all the actions executed, they can check the recording. Since the transformation patterns are stored in a centralized repository chosen by the developers, other programmers can access those recordings. Thus, LiveMTBD can also be considered a collaborative tool [SGW⁺11].

Quokka. Live tool that allows developers to visualize Kaluta emulations in real-time, benefiting QA teams since it helps to understand the behaviors and actions of large software systems and identify possible performance and scalability problems. Kaluta is an emulator for large-scale enterprise software that only uses a single physical host. With Quokka, its users have access to an interface that shows the models and interactions being executed by the engine, helping them check possible improvements that can be performed on the software. In this interface, the external systems are represented by circles in the center of the dynamic view, and rectangles show the endpoints of each interaction. The open communication channels are visualized using small circles in the middle of the correspondent endpoint. Then, each message being transmitted is presented by directed triangles positioned depending on the transmission. All of these visual characteristics can be analyzed in Figure 2.26, (p. 71) [HSHV12].

SEEDE. Tool integrated within the Code Bubbles development environment that transforms the coding experience by enabling continuous execution while programmers edit their code. SEEDE provides real-time visibility into intermediate and final execution results, including variables, control and data flow, output, and graphics. These updates are seamlessly synchronized with the user’s edits, enhancing the development process [RXH18].

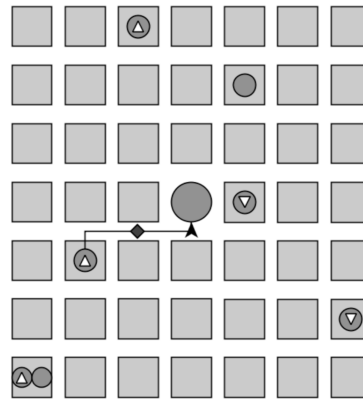


Figure 2.26: Quokka interface showing the interactions between endpoints (Figure 2 of [HSHV12]).

StriSynth. Tool that empowers end-users to perform data transformations using examples. By leveraging the provided examples, StriSynth automatically and continuously generates scripts for complex file manipulations, simplifying the manipulation and transformation of data. While initially focusing on file manipulations, StriSynth also provides a versatile framework for general string transformations. It builds upon and extends the functionality of Flash Fill, a popular Microsoft Excel extension for string transformations [GMNP15].

VERSABOX. Tool that implements projection boxes specifically designed for the Python programming language. It offers developers a user-friendly interface to customize projection boxes in real-time while coding. This dynamic capability empowers developers to effectively manage information overload by leveraging the versatility of projection boxes as a powerful tool [Ler20b].

SNIPPY. Tool developed to integrate the *Small-Step Live Programming by Example* paradigm, a novel programming approach that enables the local application of Programming by Example. This paradigm empowers programmers to modify the runtime values shown in the live visualization, allowing them to generate local program snippets. *Small-Step Live Programming by Example* offers a unique programming approach that combines live visualization and local code generation. By interactively adjusting the runtime values, programmers can observe the immediate impact on the program and generate code snippets tailored to their specific needs [FOP⁺20b].

NaturalMash. Live mashup plugin that helps developers update the output of mashups as their source is being changed. A mashup is a simple web application. NaturalMash is based on the natural programming language in “*What You See Is What You Get*” (WYSIWYG). This tool is composed of an intuitive environment with four main elements: (i) visual field, which shows the modifiable output perspective, (ii) text field, where the mashup language can be edited, (iii) component dock, that presents a list of mashup components being modified, and (iv) stack, with the created mashups [AP12, AP13].

Mattis et al. [MRH17] proposed an approach that collects multiple individual changes simultaneously and makes them visible to developers together. It can test a set of changes before extending them to the application being executed. Thus, it enables a live programming workflow that fully controls the change granularity, has immediate and continuous feedback on the tests executed, and

delays their effect on the running code.

Zhang and Hu [ZH22] presented a comprehensive framework that facilitates bidirectional live programming for incomplete programs. By extending the output value structure, introducing hole binding, and providing formal definitions for well-behaved bidirectional evaluators, this framework empowers developers to seamlessly work with partially completed code. The key component of this framework is the extension of the output value structure, which enables the representation and manipulation of partial results. This enhancement allows developers to work more effectively with incomplete programs by capturing intermediate states and gaining valuable insights for informed decision-making during development.

Kuhn [Kuh13] proposed a method for automatically extracting scripted tests from recordings of live programming sessions. This approach enables extracting scripted test cases using clustering techniques by monitoring the development environment and recording program input, user-issued function calls, and program output during exploratory testing sessions. By leveraging exploratory testing sessions, developers can transform them into valuable and reusable test assets, promoting efficiency and ensuring comprehensive test coverage based on real-world scenarios and user interactions.

Schütz and Wimmer [SW19] introduced an application that allows developers to make runtime modifications to various aspects of their software, such as the scene, render engine, shader code, and input handling. This functionality is accessible through standard desktop IDEs via a desktop mirror. With this application, developers can seamlessly modify and customize these elements in real-time during the runtime of their application.

Rein *et al.* [RLR⁺19] demonstrated the ability of Context-Oriented Programming (COP) to capture system-wide instrumentation in a modular manner, focusing on the essential components of a system. The authors implemented instrumentation for an example-based live programming tool as a use case for COP, highlighting the compatibility between the features of COP and the requirements of example-based live programming.

Biegel *et al.* [BLD15] introduced a new live tool for IntelliJ IDE for Java programs. This tool was developed as a web application that can be executed on a computer or tablet. Its users can modify the state of an object or call specific methods and verify the evolution of that system while changing them directly on the application. It is linked to the IDE, where the source code is allocated, and then it visually represents the different objects, methods, and properties. When someone updates one instance, the tool automatically changes the source code to the new state of the corresponding object. Furthermore, a developer can create a new object on the tool, which will then be propagated to the respective source code.

Google Chrome or other web browsers can also be classified as a live programming tool. When integrated with their web development tools, they allow reloading web resources like HTML and JavaScript without refreshing the web pages [Chr17]. Furthermore, some frameworks based on JavaScript like React.js and Vue.js also provide live feedback to their developers, while their source code is implemented without compiling or reloading it [Rea17, Vue14].

2.6.4 Discussion and Open Issues

Liveness is a concept that allows developers to have the results of their software systems right away while implementing them [CZ11, Tan13]. Moreover, once we shorten the time between programming and verifying the results obtained from it, we receive quick feedback and support about our systems, which helps us reduce the time and cost needed to change specific behaviors when necessary. This happens since complex systems are usually hard to understand and, consequently, to predict. Therefore, when developers have programs with a high level of liveness, they can mitigate these main problems since their programming activities are seen as cause-effect actions. Each effect is produced immediately after implementing the feature or behavior that caused it. Then, the effort needed to understand and evolve the software system will be reduced [ARC⁺19, Tan13].

Although several tools, development environments, and programming languages use the concept of liveness, other topics should be more widely explored to achieve its full potential and main objectives. Next, we summarize all the issues and major challenges that should be addressed to achieve most of the liveness capabilities:

Low Levels of Liveness. The majority of the analyzed tools support low levels of liveness. They only work with level-2 or level-3 of liveness, which are only informative and responsive. So, they do not predict future actions as it happens on the level-5 or level-6. Equally, in general, research projects that focus on level-4 are rare. Therefore, we think that research projects supporting this concept should aim to include higher levels, like level-5 and level-6, to fulfill all the capabilities of liveness.

Latency. One of the challenges of liveness is latency. With liveness, developers should be coordinated with their development environments, and the results presented to them should be automatic and instantaneous. The development environments may need to dedicate some computational power to execute a program and present its outputs continuously, which can cause some latency problems. Live tools should consider that some powerful mechanisms or algorithms may increase the feedback loop and reduce the efficiency and practicality of the tools since they may need more computational power than normal [Tan90].

Lack of Support. As was verified with the analysis of the state-of-the-art on liveness, some traditional programming languages or IDEs are trying to include higher levels of liveness. However, so far, the efforts have been insufficient because they do not include any prediction mechanisms that allow higher levels of liveness (like level-5 or level-6). Thus, we believe that there should be more research on this topic to revise the live tools, languages, and development environments currently available to increase their levels of liveness.

Intermediate States. While developers are programming, they can make some mistakes in the source code that make it invalid or unsafe. Most current live environments or tools do not analyze the intermediate states of software programs to verify their validity, which can generate future problems when executing the systems. To reduce this issue, the development environments and

tools should implement mechanisms that verify if the consistency and validity of software systems are maintained over their development. This can be achieved, for instance, by using formal methods or tests.

Unsuitable Visual Representations. Visual representations can increase human comprehension of a specific concept. However, visual techniques sometimes hide important details through abstractions that can be useful to users (in this case, developers). Besides, visual forms can become obsolete when a system is evolving since they are unreliable to the new software [ARC⁺19]. We address this problem in more detail on the analysis of the state-of-the-art on software visualization in Section 2.5, (p. 55).

Table 2.11, (p. 74), Table 2.12, (p. 75), and Table 2.13, (p. 76) summarize the current live environments, languages, and tools that we found out in our literature review about liveness.

Table 2.11: Comparison between live environments.

	<i>Target</i>	<i>Main Features</i>
[Fis13]	Circa	Inspection and code manipulation
[OMB13]	Visual Languages	Help inexperienced programmers
[MTZ20]	Robots	Help programming robots
[LL13]	Moon	Visualization of entities and states
[OMIA17]	Data structures	Connected parallel panels, visual representation and summary of changes
[Wuy02]	SOUL	Warning when there is a gap between design and implementation of software systems
[NBGL18]	Robots	Help programming robot
[IMA15]	Unit Tests	Convert code behaviors into unit test cases
[RS19]	Domain-Specific Languages	Integrate model deltas into the running system
[Ler20a]	Loop Iteration	Initiate a loop iteration with loop seeds
[OVH ⁺ 17]	Data Science Tasks	Help making data science tasks easier
[HvdS16]	Software Development	Improve programming experience by juxtaposing ordinary code and spreadsheet cells
[BVZ ⁺ 19]	Flow-Based Programming	Construct processing nodes and connect them in a graph-like structure

Continued on next page

Table 2.11: Comparison between live environments. (Continued)

	<i>Target</i>	<i>Main Features</i>
[KG18]	APIs	Streamline the process of understanding and utilizing various APIs
[STA ⁺ 14]	Android App Development	Help creating mobile apps for Android devices
[WQB ⁺ 11]	Google Cloud Platform	Assist in the evaluation and analysis of learning outcomes for suggested learning objects and learning models
[ZSF ⁺ 22]	Neural Network Architectures	Provide real-time warnings to help preventing program crashes
[MC19]	Game Development	Assist game developers while creating their games
[FOP ⁺ 20a]	Software Development	Help developers interact and understand their code and programming actions
[BKR ⁺ 21]	Game Development	Connects the developers with their games and their underlying code
[McD13b]	Software Development	Help developers navigate code and its execution
[STPWR20]	Hardware Design	Help users receive quick feedback on their code changes

Table 2.12: Comparison between live languages.

	<i>Own IDE</i>	<i>IDE(s) Name</i>
[Fis13]	✓	Circa
[LL13]	✓	Moon
[NBGL18]	✓	Improv
[OVH ⁺ 17]	✓	Hazel
[HJR ⁺ 16]	✗	Microsoft Excel, Google Sheets, etc
[McD07]	-	-
[BFdH ⁺ 13]	-	-

Continued on next page

Table 2.12: Comparison between live languages. (Continued)

	<i>Own IDE</i>	<i>IDE(s) Name</i>
[NRE ⁺ 20]	✗	Visual Studio Code
[DeL21]	✗	Visual Studio Code
[DeL21]	✗	Visual Studio Code

Table 2.13: Comparison between live programming tools.

	<i>Target</i>	<i>Refactoring</i>	<i>Quality Metrics</i>	<i>Visual</i>	<i>Plugin</i>
[SGW ⁺ 11]	-	✗	✗	✗	✓
[HSHV12]	Kaluta emulations	✗	✗	✓	✗
[RXH18]	Software Development	✗	✗	✓	✓
[GMNP15]	Data Transformations	✗	✗	✓	-
[Ler20b]	Python Projection Boxes	✗	✗	✓	✓
[FOP ⁺ 20b]	Small-Step Live Programming by Example Paradigm	✗	✗	✓	-
[AP12, AP13]	Mashups	✗	✗	✓	✓
[MRH17]	-	-	✓	✓	
[ZH22]	Incomplete Programs	✗	✗	✓	-
[Kuh13]	Scripted Tests	✗	✗	✓	-
[SW19]	Scene, Render Engine, Shader Code	✗	✗	✓	✓
[RLR ⁺ 19]	Context-Oriented Programming	✗	✗	✓	✓
[BLD15]	Java	✓	✗	✓	✓
[Chr17]	Web programming languages	✗	✗	✓	✓

Continued on next page

Table 2.13: Comparison between live programming tools. (Continued)

	<i>Target</i>	<i>Refactoring</i>	<i>Quality Metrics</i>	<i>Visual</i>	<i>Plugin</i>
[Rea17, Vue14]	JavaScript	✗	✗	✓	✓

2.7 Summary

In this chapter, we described some approaches and tools related to the main topics of our research project. With the analysis of their current state-of-the-art, we aimed to know in more detail the existing methods, techniques, and tools and their main limitations.

To summarize the results of our literature review, we created Table 2.2, (p. 21) to Table 2.13, (p. 76) in which we synthesize some of the features that we consider essential to take into account when we analyze approaches or tools within the scope of our research project.

We discovered that most software inspection tools analyze software quality and review code to search for possible implementation problems. Also, these tools use visual methodologies to provide feedback and support to developers about their analysis (*cf.* Table 2.1, p. 14).

On detecting code smells (*cf.* Table 2.2, p. 21), we verified that a large range of approaches aimed to find possible code flaws in software. Most of them use specific and well-defined methodologies to address their main objectives. Although, most of them do not use any software visualization technique to help represent the systems under analysis or even to identify the smelly blocks of code. Also, they do not focus on the smells' impact on the code as a way to create awareness among developers on their code quality. Besides, they only focus on the ASTs of the source code, and they do not consider the objectives of each block of code, which can make them find false-positives smells. According to Nucci *et al.* [DPT⁺18], Machine Learning – when applied to detecting code smells – can mitigate this problem. However, most of them do not use any of these algorithms. Furthermore, just one of the approaches presents a high level of liveness, indicating that the scientific community should further analyze this subject combined with detecting code smells.

About the refactoring tools and approaches that already exist (*cf.* Table 2.6, p. 43), we observed that most of the verified methodologies could only be used by a small set of programming languages. However, they support many refactoring techniques within the same category. Also, as with the smells detection tools and approaches, most of the refactoring techniques we explored also do not provide visual insights about the software systems under analysis and the specific impact caused by each identified refactoring on code quality. Besides, they also do not implement a high level of liveness, which slows the feedback and support provided to developers about their software systems.

As it happened with the two subjects mentioned above, most of the code quality metrics tools (*cf.* Table 2.8, p. 53) can only be supported by a small set of programming languages or similar programming languages with similar characteristics and paradigms. Nevertheless, most of the tools support a wide range of metrics, which enrich the feedback provided to developers on the quality of their programs. Valuable data about the supported metrics should complement this feedback because some programmers do not know what they mean. Also, they only present textual

information to developers about the evaluated metrics. Finally, we verified that these tools have similar disadvantages to code smells and refactoring tools. However, one thing that does not occur in this case is the extreme lack of liveness since some analyzed tools incorporate specific live characteristics.

On software visualization (*cf.* Table 2.10, p. 62), we concluded that this topic is a complementary source of information that enriches other concepts like the detection of code smells or refactoring opportunities and the analysis of code quality metrics, by providing developers with visual forms that represent their programs, behaviors, and features in a user-friendly way.

Then, the research on liveness showed the potential and impact of this subject on the software development cycle and feedback loop. On this topic, we analyzed several live environments (*cf.* Table 2.11, p. 74), languages (*cf.* Table 2.12, p. 75), and tools (*cf.* Table 2.13, p. 76). With them, we verified that live environments like Circa [Fis13] or Moon [LL13] were developed to be used by the homonym live programming languages. Most of the live tools were also developed to be used by specific programming languages or programming contexts. The majority of them were created as a plugin for a particular IDE. However, these tools aim to reduce the total development time and not to improve the quality of software systems since they do not analyze code quality metrics or try to detect refactoring opportunities. Furthermore, the live environments and tools use different visual forms to represent the systems being analyzed to improve the programming experience.

Therefore, with the analysis of current state-of-the-art on the topics of software inspection, code smells, refactoring, code quality metrics, software visualization, and liveness, we were able to verify the contributions of the different approaches and tools related to these subjects and describe some limitations that we aim to address on our research project. In the next chapter, Chapter 3, (p. 79), we discuss the main open issues we point out from this literature review. From those, and considering our main guidelines, we could state a research hypothesis and list several research questions.

Chapter 3

Research Problem

3.1	Open Issues	80
3.2	Guidelines	81
3.3	Thesis Statement	82
3.4	Research Questions	82
3.5	Research Scope	83
3.6	Methodology	84
3.7	Previous Research	88
3.8	Main Contributions	88
3.9	Summary	89

This chapter addresses the issues identified in our literature review (*cf.* Chapter 2, p. 9). In Section 3.1, (p. 80), we outline the problems we have mainly focused on in our research. Next, in Section 3.2, (p. 81), we detail our main guidelines, and in Section 3.3, (p. 82), we present the hypothesis that forms the foundation of our project. Building on this hypothesis and directives, in Section 3.4, (p. 82), we enumerate the research questions our project seeks to answer.

Additionally, we define the scope of our research project in (*cf.* Section 3.5, p. 83) and provide an overview of our research and validation proposals in (*cf.* Section 3.6, p. 84).

Finally, we discuss the outcomes achieved in prior work related to some of the topics this research project addresses in Section 3.7, (p. 88). Subsequently, Section 3.8, (p. 88) highlights the contributions this work aims to deliver.

3.1 Open Issues

In Chapter 2, (p. 9), we have presented several approaches and tools related to the main topics of our research project. However, none of those solutions included all the aspects of our work that we consider essential to achieve the desired goals (*cf.* Section 1.4, p. 5).

We also verified the lack of support these tools provide to software developers. So, we believed that there was a need for research projects like ours, which create a *Live Refactoring Environment* more human-centered, that considered other subjects like code quality metrics and software visualization to help developers achieve better programming results.

The following is a list containing the main issues found in our literature review that we pretend to mitigate with our approach:

Lack of presence on the IDEs. There are many refactoring tools for several IDEs. However, most software developers do not know that their programming environments have refactoring recommenders or code quality metrics analyzers. These functionalities are usually executed on the background of the IDEs or are hidden in a menu that we need to run to start them. This lack of presence and availability on the different IDEs reduces the efficiency and usability of these tools since developers can lose their train of thought when they have to activate them to verify their results, breaking their “*programming-flow*” [HMC⁺18, FMS⁺17]. Also, many of these tools require previous configurations, which increase the total time needed to provide feedback, support, and guidance to programmers [Bar20]. So, refactoring or software analysis tools need to be more accessible to developers on their IDEs to allow them to perform the required actions in a better, quicker, and not intrusive way.

Lack of feedback on the severity of each code smell. Even though several tools detect code smells and identify refactoring opportunities to mitigate these smells, most do not focus on classifying the severity of each code smell for code quality. In this way, developers cannot know in advance that a specific refactoring will have a more significant impact on the quality of their solution than another. Thus, each refactoring suggested should be complemented with feedback on its effects on the code to assist and guide developers toward better refactoring decisions and, therefore, better code.

Low flexibility in the workflow. This problem is closely related to the first issue described. Several tools already recommend refactorings, detect code smells, or analyze code quality metrics. However, even though these tools are available and accessible to developers, most do not allow them to examine a software system immediately and continuously while programmers implement their systems. This fact increases the time needed to support and guide developers towards better programming solutions. We would reduce the respective feedback and refactoring loop by introducing higher levels of liveness in a tool or environment capable of analyzing metrics, detecting code smells, and suggesting refactoring techniques. Then, the guidance provided to developers would be faster and more fluid, allowing them to implement the best programming practices quickly.

Lack of visualization techniques. Current tools and approaches are very effective at measuring code quality metrics, detecting code smells, and identifying refactoring opportunities. However, there is a lack of visual feedback, support, and guidance to developers about their systems. In this way, sometimes developers are not aware in a user-friendly way of the impact of their programming actions and decisions on their software, namely on their quality [Sal20]. A live refactoring environment should provide visual insights to developers about their software systems to help them understand everything happening with their systems and how they evolve with time and with the refactorings implemented.

3.2 Guidelines

To reduce all the issues described above, we considered four essential guidelines related to software development that make our proposed solution (described in Section 3.6, (p. 84)) relevant to software developers.

The first guideline we considered is that software developers benefit from feedback and guidance about their software systems as they implement them [Tyn17]. Most of the time, developers must fulfill specific deadlines to finish a set of features. With short deadlines, programmers tend to implement their systems with poor structure and quality, going against the code quality standards. However, in this research project, we considered that most programmers want to develop the intended features with high quality so that they would have no difficulty changing their systems when needed. Therefore, we contemplated that they would benefit from receiving feedback and support about their programs through refactoring recommendations that aim to reduce specific code smells that degrade the quality of their source code.

In the second guideline, we assumed that developers would benefit from receiving visual refactoring suggestions instead of textually pointing them out. As mentioned in Section 2.5, (p. 55), humans usually learn better from visual components than from textual elements [Bie89, FLC⁺02, HCL05]. Then, we believe that developers would prefer to receive refactoring feedback through visual techniques. We also thought that through visually identifying and suggesting refactorings, developers would be more aware of the possible code flaws that decrease their code quality, and, therefore, they would make better decisions to restructure their code and improve specific code quality metrics.

As a third guideline, we assumed that developers would benefit from receiving complementary support for each refactoring suggestion. Many refactoring tools recommend a set of refactoring opportunities without sorting them by a heuristic. In our case, we presumed that developers would like to receive refactoring recommendations complemented by any information about their impact on the code quality. One of the main goals of refactoring code is to mitigate specific code smells to make it more self-explanatory, readable, adaptable, and maintainable. In other words, to improve its quality. Thus, developers would benefit from improved refactoring feedback (*e.g.*, visual feedback) that would help them make more informed refactoring decisions based on their impact on code quality.

Finally, as a fourth guideline, we considered that developers would benefit from receiving refactoring suggestions, in real-time, while programming their software. This is because, as mentioned in Section 2.6, (p. 62), we thought that the sooner a developer realizes the results of his programming actions, the sooner he can change them so that they are by his standards. This also refers to the quality of code each developer wants for their system. Of course, as already mentioned, we believe developers would like to create the best possible solution with the highest quality. That is why we thought that by creating a live solution that presents refactoring opportunities in real-time, developers would be able to converge to better solutions in less time, in earlier development phases, and at times when their code is not complex or with too many lines of code to make it harder to refactor.

3.3 Thesis Statement

The preceding chapters and sections have discussed the relevance of implementing a *Live Refactoring Environment*. We believe that a *Live Refactoring Environment* helps reduce the feedback loop and provides quicker support and guidance to developers about their programs. We also assume that with faster instructions in the form of refactoring opportunities found on the source code, developers can quickly change their software systems, increasing their readability, adaptability, and maintainability. Therefore, this research project was supported by the following statement:

“A development environment that continuously inspects source code, detects and visualizes code smells, and recommends refactoring techniques, in real-time, helps software developers to achieve code with more quality and faster.”

With this statement, we hypothesized that continuous and live analysis of software systems would positively impact the software development cycle since it would allow developers to know the results of their programming actions while implementing them. With it, we could improve the feedback and support provided to programmers, using an environment that inspects the source code, detects and visualizes code smells, and recommends refactoring methods.

We believed that with constant exposure to the code smells and possible refactoring methods, while programming, developers could be more aware of the state and quality of their systems. Besides, with this constant awareness about their software, developers could make better programming decisions in earlier development stages. Thus, they could create good code with high adaptability and maintainability degrees, which is easy to understand and, therefore, of good quality.

3.4 Research Questions

As mentioned in Section 1.4, (p. 5), this project aimed to improve and speed-up the feedback, support, and guidance provided to software developers about their systems by visually recommending refactorings that could be applied to the code automatically. To achieve this goal, we needed

to combine concepts like liveness, analysis of code quality metrics, detection of code smells, and visual identification of refactoring opportunities. In this way, and to be by the hypothesis stated below in Section 3.3, (p. 82), we identified four main research questions:

RQ1 *“Is it possible to inform developers, using a color-coded system, about possible refactoring opportunities in their code?”*

With this research question, we aimed to verify if we could detect and visualize the code smells present in the source code using a color-coded approach while developers are programming. Furthermore, we also aimed to check if this approach would help developers know which code blocks indicate deeper problems in their systems.

RQ2 *“What is the impact of real-time refactoring identification on the developers’ awareness of their code quality?”*

The main objective of this research question was to measure the impact of a live code smell detection approach on the awareness of code quality. Basically, we aimed to verify *(i)* if code smells deteriorate the code quality, and *(ii)* if by detecting them, we could help developers to know that their systems have a lower quality degree that makes them hard to read, understand, adapt, and maintain.

RQ3 *“What is the impact of a live refactoring environment on their code quality?”*

With this research question, we aimed to study the impact of the constant exposure to refactoring recommendations on the quality of programming solutions. To measure these parameters, we used several code quality metrics that provided valuable information about them. With a positive impact on these parameters, we expected to verify a positive effect on some quality attributes of the software systems and conclude that developers could reach good programming solutions when they used a live refactoring environment.

RQ4 *“What is the impact of a live refactoring environment on the time required to achieve high-quality programming solutions?”*

This last research question aimed to study the impact of liveness in a refactoring recommender, where developers would know in advance the refactoring techniques that they could apply to their code. With the guidelines pointed out in Section 3.2, (p. 81), we believed that using a higher level of liveness in a refactoring recommendation approach would help speeding-up the convergence to a good programming solution with better code.

3.5 Research Scope

As in any research project, we needed to define the boundaries and scope of this work to know what we should do and to what extent we could do it.

This project aimed to help software developers quickly produce better code with higher quality. Thus, our contributions fall under software engineering, namely in the software development cycle and the development and maintenance phases themselves [SBF14].

We also had to consider that since we were studying and trying to combine different concepts, we had to know the limits of each topic and the advances already made by the scientific community so that the results obtained were as innovative as possible.

Even though we were only mainly focusing on software engineering, we also had to analyze concepts outside our main scope. This happened because we tried to use artificial intelligence algorithms to detect the best refactoring sequences that should be applied in a specific programming context.

Finally, since the refactoring catalog presented by *Martin Fowler* is quite extensive, we decided that our approach would only support some of them. Thus, our refactoring environment only identifies, suggests, and applies some of the most frequent refactoring types, which are easy to understand when recommended and executed (*e.g.*, *Extract Method*, *Extract Class*, *Move Method*, or *Replace Inheritance with Delegation* [Fow18]).

3.6 Methodology

Before starting to implement and validate our proposed approach, we studied the existing research methodologies to use them as a basis for developing our solution.

3.6.1 Research Methods

We planned to perform several experiments to validate our hypothesis and proposed solution. However, in computer science, there is no well-defined taxonomy of possible methods that can be applied to verify the validity of new solutions. To fill this gap, in 1998, *Zelkowitz* and *Wallace* [ZW98] proposed a taxonomy that groups research methods into four distinct categories, that despite being several years old, they are still highly used [Eas07, TP07, Thi14].

Scientific Method. Scientists describe a theory to justify a specific phenomenon. They propose a hypothesis, and later, they test different variations of their hypothesis. Then, they collect data to validate or refute the claims they stated in their hypothesis.

Engineering Method. Engineers implement and test a possible solution to a hypothesis that tries to solve a particular problem. With the data collected, they try to improve their solution until it does not require more improvements.

Empirical Method. Researchers propose a statistical methodology to validate a hypothesis. However, the hypothesis might not be formally described as it happens in the scientific method.

Analytical Method. Researchers develop a formal theory/hypothesis. Then, the results collected from that theory can be compared with empirical observations to validate it.

Besides the taxonomy described below, *Zelkowitz and Wallace* [ZW98] also identified twelve types of experimental approaches, for software engineering, grouped into three main categories:

Observational Methodologies. In an observational approach, researchers collect data while implementing their project. They have not fully controlled the development of their solution other than using the technologies they studied. This category groups methods like project monitoring, case study, assertion, and field study.

Historical Methodologies. In a historical approach, researchers collect data from different finished projects. They only need to analyze what they have collected. This category groups methods like legacy data, study of lessons learned, and static analysis.

Controlled Methodologies. In a controlled approach, researchers have multiple instances of the statistical validity of the data collected. This category includes replicated experiments, synthetic environment experiments, dynamic analysis, and simulation.

Besides, additional research approaches are also related to studying different software architectures. In 2001, *Shaw* [Sha01] proposed a classification that divides the research methods into five different types of products:

Qualitative or Descriptive Model. In this method, researchers organize and report relevant observations about the world. They create and defend generalizations from real examples and structure a problem, formulating the right questions about it. Additionally, they do a meticulous analysis of a software system or its development process.

Technique. In this method, researchers propose new procedures or development approaches to fulfill particular tasks. They develop a new technique to choose among different options.

System. In this method, researchers use the development of a software system as both a source of insight and a supporter of results.

Empirical Predictive Model. In this method, researchers develop predictive models from data they previously collected.

Analytic Model. In this method, researchers develop structural models that allow them to execute a formal analysis.

The research methods that should be used on a specific project are strongly related to the study being done. Therefore, we have to choose the correct combination of research methods to fulfill the main goals of our research project.

In this way, we decided that our research project would follow the *Engineering Research Method* [ZW98]. Thus, with it, we expected to develop a solution and validate it. We also complemented this methodology with the *historical method* while doing the literature review on the project's main topics. We also validated the proposed approach using the *observational* and *controlled methods*.

3.6.2 Proposed Development Approach

To implement a *Live Refactoring Environment* as we hypothesized (cf. Section 3.3, p. 82), we divided its development into four distinct phases. Each phase focused on the main concepts of our research project:

1. **Analysis of Code Quality Metrics:** In the first development phase, we analyzed a set of relevant quality metrics that were later used to detect code smells and identify refactoring opportunities. These metrics were measured through the *abstract syntax tree* (e.g., cyclomatic complexity) or using specific *mathematical formulas* (e.g., maintainability index). To improve this analysis, we considered the approaches used by other tools with the same goals, such as Eclipse Metrics2 [RTSB17], CodeMR [Cod] or refactorix [Oet].
2. **Detection of Code Smells:** In the second phase, we analyzed each previously measured metric to detect code smells like *Long Class*, or *Long Method* [Fow18]. To facilitate the detection of specific smells, we created a catalog that relates different quality metrics with particular code smells (cf. Table 4.5, p. 121). We also tried to complement this detection methodology by directly analyzing specific aspects of the source code. Besides, we also tried to use correlation techniques (cf. Section 4.2.4, p. 110).
3. **Identification of Refactoring Opportunities:** In the third phase, we used the previously detected code smells to identify and suggest different refactorings to developers. For these suggestions, we also created a catalog that associates the different code smells and code quality metrics with the possible refactoring methods (cf. Table 4.5, p. 121).
4. **Implementation of Live Mechanisms:** In the fourth phase, we augmented the liveness level (up to level 4 (cf. Section 2.6, p. 62)) [Tan13] to the entire process that we previously summarized. With the introduction of this subject, the proposed tool reduces the “*edit-compile-run*” loop that usually occurs in different tools, software systems, or IDEs. Then, the support provided to developers, through refactoring suggestions, is faster and more fluid.

In this way, we created a *Live Refactoring Environment* that can help any developer implement better programming practices faster and more informedly.

3.6.3 Validation Approach

One of the contributions of this research project is the empirical experiments that we have used to evaluate the proposed solution, answer our research questions, and validate the hypothesis supporting it. We used academic case studies, with different programming tasks performed by programming students, to increase the quality of their software systems. By improving their code quality, the participants also enhanced several aspects, like their systems’ readability, adaptability, and maintainability.

The participants of these experiments were divided at least into two distinct groups. One that had access to the developed live environment and others that did not use any auxiliary tool to perform the different tasks.

Besides, we also executed an automated experiment that crawls several GitHub projects and analyzes and refactors them to check if our *Live Refactoring Environment* can restructure them to improve their code quality.

From the experiments with human participants, we have collected different data, namely:

- **Usability:** We have measured the usability of the proposed tool to verify if it was, in fact, easy to understand, accessible, and appealing to its users. We have also used this metric to help us validate our first research question. Since this question aimed to check the non-intrusive, user-friendly ways to present the detected code smells to developers in real-time, we thought that measuring the usability of each detection could help us verify and validate the respective question.

To measure this value, we applied usability questionnaires at the end of each experiment. This questionnaire was based on the System's Usability Scale (SUS). It had some complementary questions to estimate if the participants correctly understood the tool, its overall purpose, and if it was easy to understand the detected smells and the visual methodologies used to present them. Furthermore, we believed that by measuring the number of times participants used the refactoring tool, we would have more data about the usability of our approach.

- **Awareness:** We have also evaluated the impact on developers' awareness when they had access to the code smells detected, in real-time, by our approach. To measure it, we calculated how often the participants used the environment to refactor their source code and how many times they accepted the suggestions it gave without modifying the final results after applying the recommended refactoring.

We thought that if developers had more awareness of the code smells detected, they would also be more conscious of the quality of their software systems. Therefore, they would want to refactor their systems to improve them.

- **Code Quality:** This validation metric was related to the final quality of software systems. To measure it, we had to verify if the refactorings applied to the source code led to better programming solutions. We have measured different code quality metrics supported by our approach and compared the results before and after each refactoring, using or not our environment. With the collected values, we had enough data to get an overview of the final quality of each source code, allowing us to check if a live refactoring environment like ours impacted the quality of software solutions.
- **Time Performance:** We also aimed to validate the time performance of the proposed environment. Since our solution's major goal was to provide feedback and support as fast as possible while developers were programming, we thought verifying if the idealized tool achieved the

desired live requirements would be essential. To obtain data that would help us measure our extension's time performance, we had to compare the time needed to fulfill each task of the experiments in the two scenarios possible, with and without the developed tool. Besides, we also checked if, by using our approach, the experiment participants would converge to better code, faster.

With the automatized experiment, we only focused on collecting code quality data. In this case, we assessed several code quality metrics over each refactoring applied by the approach. Then, we compared their values before and after each refactoring applied to the code. With them, as with the remaining experiments, we aimed to verify if a refactoring environment like ours could improve code quality.

Then, with all these essential data, we have validated the results using hypothesis tests [KK15]. Through this validation method, we aimed to answer our research questions (*cf.* Section 3.4, p. 82) and validate our research hypothesis (*cf.* Section 3.3, p. 82).

3.7 Previous Research

Previously, we have implemented a *JavaScript* and *TypeScript* plugin for *Visual Studio Code* named *LiveMR* [Fer19, FRFA20]. This plugin analyzed several code quality metrics (*e.g.*, the number of lines of code, Halstead metrics, cyclomatic complexity, or maintainability index), while developers were programming their systems. To measure them, we used the *code's AST* and *specific mathematical formulas* [Hal77b]. Then, we provided information about these values through bar charts on the IDE's side menu. Besides, we also created a simple mechanism that identified *Extract Variable* and *Split Variable* refactoring opportunities, using the *code's AST* [Fer19].

We were also involved in other projects where there were implemented mechanisms to live recommend extract-related refactorings to solve possible code smells like *Long Methods* [Bar20, Sal20]. These techniques were addressed and used in this research project as a basis to detect and suggest other refactoring methods.

3.8 Main Contributions

Our research project was divided into four significant contributions related to its main goals:

State-of-the-Art. The first contribution that was drawn from this research project was the detailed literature review done on topics like *Software Inspection*, *Code Smells*, *Refactoring*, *Code Quality Metrics*, *Live Programming*, and *Software Visualization* (*cf.* Chapter 2, p. 9). Research about code quality metrics, code smells, and refactoring was a fundamental starting point for this project since these three topics were very present in this work. The research related to liveness was also crucial since implementing this concept in our solution improved its fluidity. Last but not least, we also did some research on software inspection and software visualization. We analyzed the first subject because it includes the concepts of code quality metrics, code smells, and refactoring. And with

the second one, we aimed to know more about the techniques currently used to present data to programmers.

Live Refactoring Environment. The second contribution of this research project was the development of a *live refactoring environment* (cf. Chapter 4, p. 91). With this idealized solution, software developers could have access to several refactoring suggestions that aimed to improve the quality of their code. This refactoring environment was created as a *Java plugin* for *IntelliJ IDE*. It was focused on analyzing specific quality attributes to detect particular code smells. By identifying different smells, this environment selected possible refactoring methods capable of reducing the respective smells, in real-time.

Empirical Validation. The third contribution of this research project consisted of the empirical experiments that helped us evaluate our proposed solution, answer the research questions, and validate the hypothesis that supported it (cf. Chapter 5, p. 123). We used academic case studies performed by programming students and an automated version of our approach to crawl, inspect, and refactor GitHub projects. The results collected from the experiments were analyzed using hypothesis tests [KK15]. In this way, we were able to gain more confidence in our approach and to check that it was indeed necessary and that it helped developers improve their software systems.

Scientific Publications. The fourth contribution collected from this research project was the set of publications that resulted from different stages of our work (cf. Appendix A, p. 225). These publications are mainly about the problem we aimed to solve, our proposed solution, and how we implemented and tested it.

3.9 Summary

Building on existing tools designed to support high-quality software development (cf. Chapter 2, p. 9), we identified key issues (cf. Section 3.1, p. 80). We also established our hypothesis (cf. Section 3.3, p. 82), guided by our main principles (cf. Section 3.2, p. 81). We declared that a real-time development environment capable of continuously inspecting code, visually identifying code smells, and suggesting refactorings could enable developers to produce higher-quality code more efficiently.

As is common in research projects [Kum14], our hypothesis was supported by research questions aimed at guiding our investigation (cf. Section 3.4, p. 82). Based on these questions, we defined our research scope (cf. Section 3.5, p. 83) and adopted a methodology suitable for addressing the problem (cf. Section 3.6, p. 84). We also reviewed prior related work (cf. Section 3.7, p. 88) and summarized the outcomes achieved through this project (cf. Section 3.8, p. 88).

This analysis led to the development of our research solution. The next chapter, Chapter 4, (p. 91), details the solution's components, the development process, key decisions, and its limitations.

Chapter 4

Live Refactoring Environment

4.1	Development Environment	92
4.2	Refactoring Detector and Applier	93
4.3	Refactoring Visualizer	116
4.4	Liveness Features	119
4.5	Limitations	120
4.6	Summary	120

Large and complex software systems are often difficult to read, adapt, and maintain. Refactoring such code is a costly and time-intensive process, a challenge further compounded when performed manually by developers [MHB10, CAGA11].

Although development environments offer tools to provide feedback and guidance on improving code, these tools are typically used in later stages of the development cycle [Tyn17]. As a result, developers may not benefit from such feedback until they have finalized a solution, which can lead to suboptimal code quality.

We hypothesized that offering faster refactoring guidance during the early stages of development would empower developers to more efficiently modify, adapt, and maintain their code, ultimately improving its quality. By continuously inspecting software and assisting developers in visually identifying, suggesting, and applying refactorings, we aimed to positively impact the software development process and enhance code quality (*cf.* Chapter 3, p. 79).

To address this hypothesis, we proposed a *Live Refactoring Environment*, a tool designed to identify, suggest, and apply refactorings in real time. The workflow of this solution is illustrated in Figure 4.1, (p. 92).

Unlike traditional refactoring tools, this solution focuses not only on identifying refactoring opportunities but also on enhancing the overall developer experience.

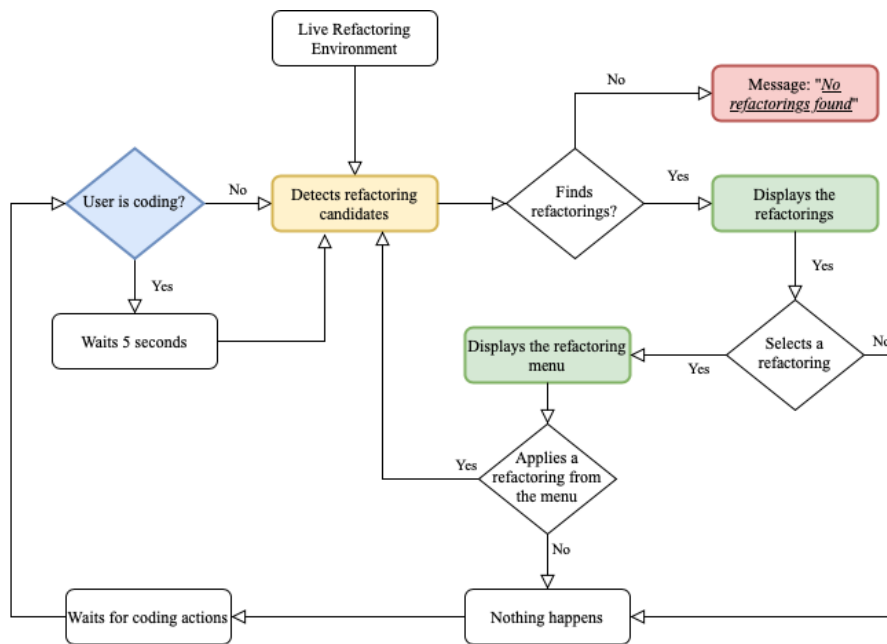


Figure 4.1: Workflow synthesizing the developed Live Refactoring Environment.

In the following sections, we discuss the environment chosen for the tool’s development (*cf.* Section 4.1, p. 92), describe its main characteristics (*cf.* Section 4.2, p. 93), and outline its limitations (*cf.* Section 4.5, p. 120). We also explain the rationale behind key implementation decisions.

4.1 Development Environment

Our approach involves creating a Live Refactoring Environment for IntelliJ IDE as a Java plugin (for Java 8 or higher versions). We evaluated several IDEs for their ability to access and manipulate the Abstract Syntax Tree (AST) of code, provide visual suggestions for identified refactorings, and apply them automatically. After considering the popularity of various IDEs, we determined that Visual Studio Code and IntelliJ IDE are the most widely used options [JRe, So21].

While it would have been easier to develop a plugin for Visual Studio Code that targets TypeScript and JavaScript software, we decided to use IntelliJ IDE instead because it has all the necessary APIs to implement our approach. There are also many public Java projects that could be used to validate our refactoring tools. Additionally, our literature review showed that many existing refactoring tools support Java code, and numerous large and complex Java software projects have been used to test and validate the approaches of other researchers [TCC18, FTC07, AI 12, KECK12].

Then, we started developing our refactoring environment using IntelliJ Community Edition 2021. However, at the end of 2022, when we began testing our approach in a larger experiment (*cf.* Section 5.2, p. 143), we verified that our environment did not work correctly when using it in newer IntelliJ IDE versions. This happened because of several features that changed between versions of IntelliJ that did not allow the creation of a start action with several threads from the main menu of

IntelliJ. Thus, we had to change our tool to work with newer versions of IntelliJ, and from then on, we continued developing and maintaining the tool using IntelliJ Community Edition 2022. The main differences between tools are addressed in Section B, (p. 233).

4.2 Refactoring Detector and Applier

To establish a refactoring process, we must first identify potential opportunities for refactoring within the codebase and present them to developers for review. If the suggested refactorings are approved, we must apply them to the code without disrupting the previously implemented functionalities.

In the following sections, we will outline the features of our refactoring environment and describe the approaches we use to identify and implement various types of refactorings. The techniques that shorten the refactoring process and clearly and intuitively suggest the refactorings to developers are described in Section 4.4, (p. 119) and Section 4.3, (p. 116), respectively.

4.2.1 Architecture

Figure 4.2, (p. 94) synthesizes the architecture of our solution by presenting a high-level diagram with all the main layers, packages, and relationships between the different components composing our environment.

Our system was built using a three-layered architecture composed of *logic*, *core*, and *presentation*. These layers can be described as follows:

Logic. This layer is responsible for evaluating various metrics that measure the quality of the code, such as complexity, maintainability, or cohesion. This layer uses these metrics to identify areas of the code that could benefit from refactoring. Besides, in this layer, the refactoring suggestions are produced based on the source code and coding events, such as changes or modifications made to the code.

Core. This layer contains the core components shared by the other layers. The package *Core* includes all the information needed to detect some of the refactorings supported by our environment. For instance, to identify *Extract Method* candidates, we need the class *Fragment*, which represents a block of code that can be extracted. Also, to identify a refactoring candidate independently of its type, we need to know its range, severity, and representing color (cf. Section 4.3, p. 116). For that, we need the classes *Range*, *Severity*, and *Gutter*, respectively.

Presentation. This layer handles the client-side of our refactoring environment. Its main goal is to (i) get the configuration input from the users and (ii) provide the correct output as refactoring suggestions to them. Basically, it sets up the environment according to the thresholds provided by the users and displays the results (e.g., the existence or not of refactoring candidates on the code) back to them.

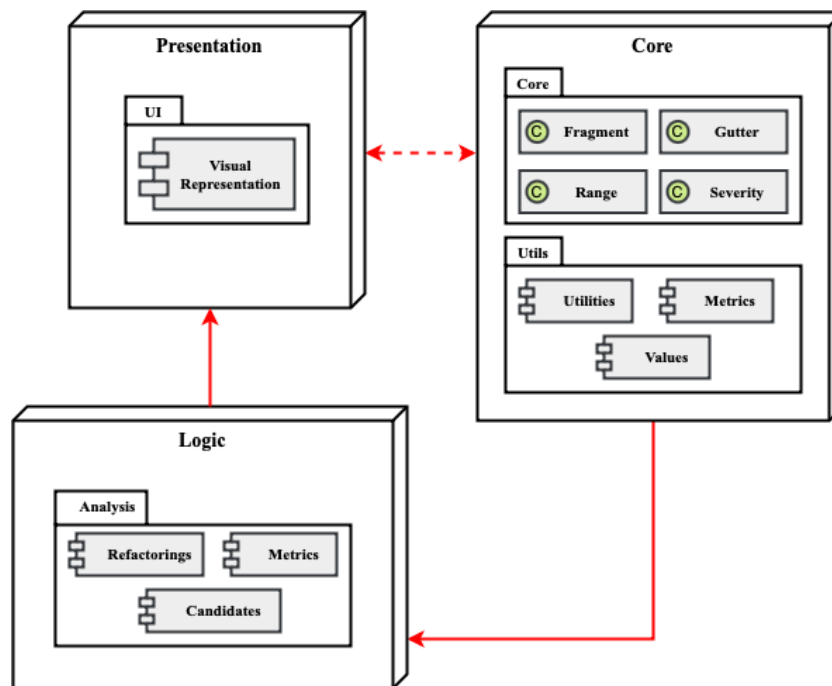


Figure 4.2: Architecture of the developed Live Refactoring Environment.

For each layer, we have different packages. For the *Logic Layer*, we have the package *Analysis*. For the *Core Layer*, we have the packages *Core* and *Utils*. Lastly, for the *Presentation Layer*, we have the package *UI*. They are described as follows:

Analysis. The source code inspection occurs in the *Analysis Package*. It scans the code as *PSI* objects¹, assesses several code quality metrics, identifies, and applies specific refactoring candidates. Therefore, we decomposed this package into three other sub-packages:

Refactorings. The detection of all the refactorings supported by our environment goes into this package. Its main function is to obtain all the needed data from the source code (through the *PSI* objects), so it can verify the different refactoring opportunities present in the code. This package also resorts to the metrics measured on the *Metrics* sub-package. Besides, each refactoring candidate corresponds to an object created inside the *Candidates* sub-package.

Metrics. This package measures the code quality metrics by method, class, or file. For that, it takes the *PSI* objects and assesses several quality aspects to measure metrics such as the cyclomatic complexity, Halstead Effort, or Maintainability Degree. Then, these metrics will be used to help identify and sort the refactoring candidates by their severity and urgency to be applied to the code (cf. Section 4.2.2, p. 95).

Candidates. We have an object related to each refactoring candidate with specific characteristics, fields, and methods.

¹PSI - Program Structure Interface by IntelliJ, <https://plugins.jetbrains.com/docs/intellij/psi.html>

For instance, an *Extract Method* candidate is represented by an object of type *ExtractMethodCandidate* characterized by the original method, the fragment of code to be extracted or the range of that fragment (cf. Section 4.2.3.1, p. 99). On the other hand, an *Extract Class* candidate is represented by an *ExtractClassCandidate*, which contains the original class or the methods to be extracted to a new class (cf. Section 4.2.3.2, p. 100).

Core. This package holds all the essential objects to help visually suggest the refactoring candidates. The object *Range* helps to identify the lines of code that represent a refactoring opportunity. The object *Severity* helps know the urgency of applying a specific refactoring. And, the object *Gutter* helps recommend each refactoring using colors (both objects *Severity* and *Gutter* will be further explained in Section 4.3, (p. 116)). Also, in this package, we have the object *Fragment*, whose only purpose is to identify possible code fragments that could be converted into possible *Extract Method* opportunities (cf. Section 4.2.3.1, p. 99).

UI. The *UI* package receives all the information needed from the *Logic Layer* to visually suggest each refactoring to the environment's users. To help transform this information into something visually appealing with high usability, it also resorts to the capabilities offered by the *Core Layer*, mainly to the objects *Severity* and *Gutter*. Besides, it also uses some functionalities from the *Utilities* sub-package from the package *Utils*. With all of them, the *UI* package is responsible for displaying the refactoring suggestions and all the changes caused by events such as coding or undoing an action and applying the refactoring actions when a suggestion is accepted.

Utils. This package comprises all utility methods and common thresholds between the remaining layers to help assess the code quality metrics and detect the refactoring opportunities. The sub-package *Utilities* includes all the methods that are commonly used by several other objects, mainly from the *Logic* and *Presentation Layer*. The sub-package *Metrics* contains all the functionalities responsible for helping assess specific code quality metrics. The sub-package *Values* includes all the thresholds used to identify the refactoring opportunities, which users can configure to change the scope of the environment.

4.2.2 Code Quality Metrics

We decided to measure a set of code quality metrics to help identify and sort each refactoring included in our approach and assess the code quality. Previous refactoring work shows that code quality metrics help detect refactoring opportunities [SVH10, CRT18, NPT13]. In our case, we assessed the following metrics:

Basic Metrics. As basic metrics, we are talking about metrics such as the number of lines of code, number of comments, number of methods, number of fields, and number of long methods, among others. Notice that the number of lines of code was measured by counting all lines and subtracting those that were either blank or contained only comments;

Cyclomatic Complexity. Metric that assesses the number of independent paths in a piece of code [McC76]. By independent paths, we are referring to switch or if conditions, loops, or binary expressions;

Cognitive Complexity. Metric that measures how difficult it is to read and understand a block of code. This metric ignores the structures allowing multiple readable statements, and it is incremented when flow-breaking structures are nested or when the linear flow of the code is broken [Cam18];

Halstead Length. Metric that assesses the total number of operators and operands in a block of code [Hal77a];

$$Length = operands + operators \quad (4.1)$$

Halstead Volume. Metric that assesses the size of a block of code [Hal77a];

$$Volume = Length * \log (uniqueOperands + uniqueOperators) \quad (4.2)$$

where *uniqueOperands* and *uniqueOperators* represent the number of distinct instances of each operand and operator of a block of code;

Halstead Difficulty. Metric that assesses the difficulty in reading or writing a block of code [Hal77a];

$$Difficulty = \frac{uniqueOperators}{2} * \frac{operands}{uniqueOperands} \quad (4.3)$$

Halstead Effort. Metric that assesses the effort to read or write block of code [Hal77a];

$$Effort = Volume * Difficulty \quad (4.4)$$

Lack of Cohesion. Metric that assesses the cohesion of a class. Higher values suggest a lower class cohesion and, therefore, the need for performing a refactoring [CK91, HS02].

$$LCOM = \frac{\frac{1}{v} \cdot a - m}{m - 1} \quad (4.5)$$

where **m** represents the number of methods in a class, **a** the number of methods in a class that access an instance variable, and **v** the number of instance variables;

Maintainability Index. Metric that holistic accesses the relative maintenance of a block of code by combining other code metrics [CALO94]. The formula used in this project was the one proposed in 2011 by Microsoft²:

²Maintainability Index by Microsoft, <https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=vs-2022>

$$Maintainability = \max(0, \frac{171 - 5.2 * \ln(HV) - 0.23 * CC - 16.2 * \ln(LOC) * 100}{171}) \quad (4.6)$$

where HV , CC , and LOC represent the Halstead volume, cyclomatic complexity, and number of lines of code of the block of code being analyzed.

Table 4.1, (p. 97) synthesizes all the metrics included in this research project and divides them by type. We have three types of groups: file, class, and method metrics. *File Metrics* represent the average measurement of each metric assessed for each class defined inside a file. *Class Metrics* represent the average measurement of each metric evaluated for each method of a class. *Method Metrics* represent the metrics evaluated for each method.

Table 4.1: Code quality metrics supported by our approach, grouped by type.

<i>Type of Metric</i>	<i>Code Quality Metrics</i>
File Metrics	Number of Lines of Code, Number of Comments, Number of Classes, Number of Methods, Average Number of Long Methods, Average Lack of Cohesion, Average Cyclomatic Complexity, Average Cognitive Complexity, Average Halstead Metrics, Average Maintainability index
Class Metrics	Number of Fields, Number of Public Fields, Number of Methods, Number of Long Methods, Class Lack of Cohesion, Average Cyclomatic Complexity, Average Cognitive Complexity, Average Halstead Metrics, Average Maintainability Index
Method Metrics	Number of Parameters, Number of Lines of Code, Number of Comments, Number of Statements, Method Lack of Cohesion, Cyclomatic Complexity, Cognitive Complexity, Halstead Length, Halstead Vocabulary, Halstead Volume, Halstead Difficulty, Halstead Effort, Halstead Level, Halstead Time, Halstead Bugs Belivered, Maintainability Index

4.2.3 Supported Refactorings

With our main goal in mind, we started studying and analyzing several refactorings to include in our approach. Our decision was made considering previous work on refactorings, the difficulty of implementing specific refactorings, and also the time that would take to identify specific cases. Therefore, we decided to focus our work on just seven of the possible refactorings. They are described as follows [Fow18]:

Extract Method. Methods with several lines of code can be hard to read, understand, and maintain [Mar08]. *Extract Method* refactoring consists of extracting a block of code from a method to a new one, replacing the old code with a call to the created method. Therefore, with it, we create a

cleaner and self-explanatory code. This refactoring can reduce code smells like *Long Method* or *Duplicated Code*.

Extract Class. We start by creating simple classes. However, with some iterations, we tend to complicate them. And, several times, we have classes that perform more work than the one we ever thought. Then, *Extract Class* helps reduce the work of a class by extracting related methods and fields to a new class to separate functionalities through different classes. This refactoring can reduce code smells like *Large Class*, *Feature Envy*, or *Data Clumps*.

Extract Variable. When coding, we may create complex expressions that are hard to understand. That is the main reason for applying a *Extract Variable* refactoring. With it, we place the result of the expression or a part of it in a new variable that can be more self-explanatory. This refactoring mostly reduces the *Long* or *Complex Expression* smells.

Move Method. We also tend to create methods inside a class that contains more data from other classes than its own. This increases the dependencies between classes and reduces cohesion inside the original class. Then, the *Move Method* refactoring helps move a method from its original class to another one that uses the method the most, creating a reference to it on the original one. This refactoring can reduce code smells like *Feature Envy* or *Large Class*.

Replace Inheritance With Delegation. Suppose a subclass is not an extension of a superclass or only uses a small portion of the methods of its superclass. In that case, we should apply the *Replace the Inheritance With Delegation* refactoring. With it, we create a field and declare it with the superclass object. Then, we delegate the methods to that object and eliminate the inheritance. This way, the superclass will be a helper of the subclass. This refactoring mostly reduces the *Inappropriate Inheritance* code smell.

Introduce Parameter Object. When creating a method, we tend to identify multiple parameters. However, good programming practices say that a method should not contain more than three or four parameters [Mar08]. Then, the *Introduce Parameter Object* refactoring helps replace multiple parameters with an object to make the code cleaner, with more quality. This refactoring reduces the *Long Parameter List* code smell.

String Comparison. When comparing strings, some developers, mainly the inexperienced ones, tend to use the operator “==”, and not the method *equals()* that several programming languages provide (Java is one of them). Moreover, IntelliJ also analyzes this refactoring. But not in real-time, with a visual suggester like ours.

We consider that from these refactorings, just three of them are easy and quick to identify on code. They are the *Extract Variable*, *Introduce Parameter Object*, and *String Comparison* refactorings. The rest of them can be classified as “more tricky” and sluggish to detect since their

identification includes several steps and involves several calls to complementary methods. Besides, most of them are widely supported by IntelliJ IDE³.

The following sections describe the steps taken to identify the previously described refactorings and all the decisions we had to make to implement our approach.

4.2.3.1 Extract Method

An *Extract Method* moves a specific block of code to a new method, replacing the original code with a call to this newly created method [Fow18].

The approach used to identify *Extract Method* refactorings was based on the work from Salgado [Sal20]. First, our system selects fragments of code suitable for extraction to a new method. This methodology operates in two phases: (i) extracts individual nodes across each method being analyzed (cf. Algorithm 1, p. 100), and (ii) combines the identified nodes into a possible refactoring candidate (cf. Algorithm 2, p. 101).

In **phase (i)**, we start by analyzing each method implemented in a file. For each method, we assess its code quality metrics, and we verify if some of them comply with specific thresholds. First, we confirm if the method is long. In other words, we check if its number of lines of code is higher than *MIN_LOC*, whose default value is 30.0. This was a value defined by us. We also verify if its total number of lines, its cyclomatic complexity, and Halstead Effort are higher than *MIN_METHOD_LENGTH*, *MAX_COMPLEXITY*, and *MAX_EFFORT*, whose default values are 60.0, 15.0, and 300.0. These thresholds were based on the work of Thomas McCabe Sr., Thomas McCabe Jr., and McCabe Software⁴. Then, to ensure that the method has enough nodes to be extracted, we verify if its number of statements is higher or equal than the double of *MIN_STATEMENTS*, whose default value is 3.0. By statement, we are referring to nodes, such as *variable declarations*, *conditions*, or *loops*. After verifying if a method complies with all these clauses, our system analyzes all its statements. To be a statement capable of being extracted, it cannot represent a return statement or contain any break, continue, or return statements as children.

In **phase (ii)**, our approach tries to combine consecutive extractable fragments (Line 18, (p. 101) of Algorithm 2, (p. 101)). We connect them to create more complex *Extract Method* candidates since we (as well as Salgado [Sal20]) believe that would cause a more positive impact on the code. After combining these nodes forming new *Extract Method* candidates, our approach filters them to remove trivial refactoring opportunities. We start by checking if each candidate is not entirely composed of trivial statements, such as variable statements (Line 30, (p. 101) of Algorithm 2, (p. 101)). Then, we verify if it does not contain more than a specific percentage of the original method's statements (represented by *MAX_ORIGIN_PERCENTAGE*, whose default value is 80%) and contains a minimum number of statements (represented by *MIN_STATEMENT*, whose default value is 3.0).

³IntelliJ Code Refactoring, <https://www.jetbrains.com/help/idea/refactoring-source-code.html>

⁴McCabe Software, <http://www.mccabe.com>

Algorithm 1 Identification possible extractable code fragments.

```

1: fragments  $\leftarrow \square$ 
2: for each method  $\in$  sourceFile do
3:   if method.length  $\geq$  MIN_METHOD_LENGTH and method.complexity  $\geq$ 
     MIN_COMPLEXITY and method.effort  $\geq$  MIN_EFFORT then
4:     if method.length  $\geq 2 \times$  MIN_STATEMENTS and method is not CONSTRUCTOR
       then
5:       for each statement  $\in$  method do
6:         if statement is not BREAK and statement is not RETURN and statement is
           not CONTINUE then
7:           range  $\leftarrow$  Range(statement.start, statement.END)
8:           ranges.push(range)
9:           fragment  $\leftarrow$  Fragment(statement, range, method)
10:          fragments.push(fragment)
11:        end if
12:      end for
13:    end if
14:  end if
15: end for

```

Notice that all the thresholds used in both **phase (i)** and **(ii)** can be configured by the tool's users.

Once we find the final set of *Extract Method* opportunities, we assess specific metrics measured in each extractable fragment to help us sort the candidates by their relevance. Based on the approach used by Meananeatra *et al.* [MRA11], we consider their number of statements, lines of code, cyclomatic complexity, and lack of cohesion. This method acts as a predictor of how extracting the nodes of a candidate would impact the file metrics. Our system sorts the candidates in descending order by their number of statements and lines of code, followed by their cyclomatic complexity and lack of cohesion. That way, the first candidates on the set will be the ones that would result in methods with a lower number of statements and lines of code, lower cyclomatic complexity, and lack of cohesion.

Finally, our tool takes the set of *Extract Method* refactorings and suggests them to its users using the visual methodology described in Section 4.3, (p. 116). Then, its execution is performed automatically through IntelliJ by using its built-in command *ExtractMethodProcessor*⁵ that is responsible for method extraction.

4.2.3.2 Extract Class

Extract Class refactoring takes a filtered class whose methods share more similarities and transforms it into a new class [Fow18].

⁵*ExtractMethodProcessor* by IntelliJ IDE, <https://github.com/JetBrains/intellij-community/blob/master/java/java-impl-refactorings/src/com/intellij/refactoring/extractMethod/ExtractMethodProcessor.java>

Algorithm 2 Identification of *Extract Method* candidates.

```

1: candidates  $\leftarrow$  []
2: firstCandidates  $\leftarrow$  []
3: statements  $\leftarrow$  []
4: ranges  $\leftarrow$  []
5: methods  $\leftarrow$  []
6: metrics  $\leftarrow$  FileMetrics(sourceFile)
7: for each fragment  $\in$  fragments do ▷ From Algorithm 1, (p. 100).
8:   statements.push(fragment.statement)
9:   ranges.push(fragment.range)
10:  methods.push(fragment.method)
11: end for
12: for i  $\leftarrow$  0 to ranges.length do
13:   candidate  $\leftarrow$  ExtractMethodCandidate(ranges[i], statements[i], methods[i])
14:   firstCandidates.push(candidate)
15: end for
16: for i  $\leftarrow$  0 to firstCandidates.length - 1 do
17:   for j  $\leftarrow$  0 to firstCandidates.length do
18:     if areConsecutive(firstCandidates[i], firstCandidates[j]) then
19:       range  $\leftarrow$  Range(firstCandidates[i].range.start, firstCandidates[j].range.end)
20:       nodesTemp  $\leftarrow$  []
21:       nodesTemp.addAll(firstCandidates[i].statements)
22:       nodesTemp.addAll(firstCandidates[j].statements)
23:       candidate  $\leftarrow$  ExtractMethodCandidate(range, nodesTemp, method)
24:       if candidate in firstCandidates then firstCandidates.push(candidate)
25:       end if
26:     end if
27:   end for
28: end for
29: for each candidate  $\in$  firstCandidates do
30:   if isNotCandidateOnlyVariables(candidate) then
31:     oldStatements  $\leftarrow$  candidate.lengthMethod
32:     if candidate.numStatements  $\geq$  MIN_STATEMENTS then
33:       if candidate.numStatements  $\leq$  MAX_ORIG_METHOD x oldStatements then
34:         candidates.push(candidate)
35:       end if
36:     end if
37:   end if
38: end for

```

Our approach is summarized in Algorithm 3, (p. 103). It starts by checking if the original class is an interface or enumeration. If not, our methodology assesses some class quality metrics. This step was based on the work of Lanza and Marinescu to detect *God Class* code smells [LMD06]. Considering this approach, we assessed the number of methods per class, its lack of cohesion, and foreign data. By foreign data, we are talking about the number of methods or fields from other classes that a specific class accesses. With these metrics, we aimed at reducing the number of

simple *Extract Class* candidates that would cause a lower impact on the code. We compare the values of the metrics with the thresholds *MIN_METHODS*, *MAX_LCOM*, and *FOREIGN_DATA*, which default values are 4.0, 0.33, and 2.0, respectively.

Then, after verifying these metrics, our approach is based on the work from Salgado [Sal20], which consists of an adapted methodology from Bavota *et al.* [BDO11]. In their method, they calculate three $M \times M$ matrices (with M representing the number of methods in the analyzed class). These matrices contain information on the Conceptual Similarity between Methods (*CSM*), Call-based Dependence between Methods (*CDM*), and Structural Similarity between Methods (*SSM*) metrics.

Bavota *et al.* [BDO11] approach also applied graph theory to further improve the refactoring suggestion. They used a *Ford-Fulkerson* algorithm followed by a *Max-Flow Min-Cut* algorithm to filter a subset of methods that should be extract [CLRS09]. However, as Salgado [Sal20] did, we believe that employing graph theory would increase the time needed for our system to present the *Extract Class* opportunities, stripping it of its live characteristics. Thus, we decided to entirely follow the work from Salgado [Sal20] from this step forward.

The data collected by Salgado [Sal20] on his experiment showed that even being a more straightforward approach, not so precise as the original one, this approach can identify suitable refactoring opportunities. Considering it, we used the weight matrix measured between all methods on the assessed class, iterated row by row, and checked its values against two thresholds. The lowest one, with a default value of 0.15, is aimed at filtering out spurious methods like getters and setters. The highest one, with a value of 0.30, is aimed at filtering out methods whose metrics are not similar enough to others and, therefore, should be extracted to a new class. Then, methods whose value on the weight matrix is on the interval $]0.15; 0.30]$ should be included in the set of methods to be extracted to a new class.

These steps are executed on the methods of Line 7, (p. 103) and Line 8, (p. 103) of Algorithm 3, (p. 103).

After having the set of method combinations representing a possible refactoring opportunity, our approach verifies if each candidate fulfills a set of restrictions. First, we check if each candidate contains more than a minimum number of methods to be extracted. This value is represented by *MIN_NUM_METHODS*, whose default value is 3.0. Then, it verifies if a candidate makes the original class too small and empty after the *Extract Class* is applied. For that, we check if the number of methods to extract is lower or equal to a specific percentage of the original class. This percentage is represented by *PERCENTAGE_ORIGINAL_METHODS*, whose default value is 70%. Notice that the tool's users can configure these thresholds. Finally, as the last step, we verify if the candidate can be applied to the code. The *isApplicable()* method (Line 14, (p. 103) of Algorithm 3, (p. 103)) is based on the *isApplicable()* method created by Fokaefs *et al.* [FTSC11] on IntelliJDeodorant⁶ on their *Extract Class* approach. This method verifies if each method of an *Extract Class* candidate is not a synchronized, override, or abstract method, does not contain any

⁶IntelliJDeodorant, <https://github.com/JetBrains-Research/IntelliJDeodorant/tree/master/src/main/java/org/jetbrains/research/intellijdeodorant>

super method invocation or any field access of enclosing classes, or is a read or write object. The candidate can be applied to the code if none of these clauses is true.

Algorithm 3 Identification of *Extract Class* candidates.

```

1: candidates  $\leftarrow$  []
2: for each class  $\in$  sourceFile do
3:   if class is not ENUM and class is not INTERFACE then
4:     methods  $\leftarrow$  class.getMethods()
5:     metrics  $\leftarrow$  ClassMetrics(class)
6:     if metrics.numMethods  $\geq$  NUM_METHODS and metrics.lcom  $>$  MAX_LCOM and
       metrics.foreignData  $\geq$  FOREIGN_DATA then
7:       methodsCombinations  $\leftarrow$  methodsCombinations(methods)
8:       methodsNames  $\leftarrow$  getMethodsToBeExtracted(class, metrics.weightMatrix)
9:       for each methodsCombined  $\in$  methodsCombinations do
10:        entities  $\leftarrow$  methodsCombined
11:        candidate  $\leftarrow$  ExtractClassCandidate(source file, class, entities)
12:        if candidate.numTargetMethods  $\leq$  class.numMethods  $\times$ 
           PERCENTAGE_ORIGINAL_METHODS then
13:          if candidate.numTargetMethods  $\geq$  MIN_NUM_METHODS then
14:            if isApplicable(candidate) then
15:              candidates.push(candidate)
16:            end if
17:          end if
18:        end if
19:      end for
20:    end if
21:  end if
22: end for

```

Lastly, with all the suitable *Extract Class* candidates, our approach sorts them in descending order of specific values. First, it compares the number of methods included in each candidate. If there is a tie, it compares their lack of cohesion, and if needed, it also compares their cyclomatic complexity. These metrics are an average of the values measured in each method.

Then, these refactorings are suggested through the visual methodology detailed in Section 4.3, (p. 116). To apply each *Extract Class* opportunity to the code, we resorted to the methodology used on IntelliJDeodorant⁷.

4.2.3.3 Extract Variable

An *Extract Variable* refactoring consists of assigning the result of a specific expression or its individual parts to separate variables with self-explanatory names [Fow18].

⁷Extract Class Executor by IntelliJDeodorant, <https://github.com/JetBrains-Research/IntelliJDeodorant/blob/master/src/main/java/org/jetbrains/research/intellijdeodorant/ide/refactoring/extractClass/ExtractClassRefactoring.java>

To find an *Extract Variable* refactoring, our system considers the non-void calls of each method that should be later extracted to a new variable. This refactoring improves the code readability at the expense of creating a new local variable.

This project follows the approach presented by Salgado [Sal20]. The main challenge of this approach is differentiating the non-void calls from the void ones. Since we choose to analyze file-by-file, we cannot access the invoked external methods and their returning values. That way, we had to resort to IntelliJ capabilities, specifically those offered by the PSI objects.

The PSI library allows filtering all call expressions inside each method. And, for each call expression, we have access to the called method, an instance of a *PsiMethod*. Then, for each *PsiMethod*, we have access to its returning type. If the returning type flag is of type void, that expression is classified as a non-void call that our system should further analyze. Algorithm 4, (p. 104) summarizes the detection of non-void call expressions.

Algorithm 4 Assessing all non-void calls from a method.

```

1: nonVoidCalls  $\leftarrow$  []
2: for each methodCallExpression  $\in$  method do
3:   calledMethod  $\leftarrow$  methodCallExpression.method
4:   if calledMethod.type.equals(PsiType.VOID) then
5:     nonVoidCalls.push(methodCallExpression)
6:   end if
7: end for

```

After identifying the non-void calls, our approach filters them to remove trivial *Extract Variable* candidates. By trivial candidates, we refer to expressions small enough to be easily understood by software developers. Thus, we filter out non-void call expressions whose length is smaller than the threshold *MIN_LENGTH* that the tool's users can configure. Its default value is 12.0 characters. Then, the calls that do not exceed that length are not identified as possible refactoring opportunities.

Algorithm 5, (p. 104) describes all the steps that our approach takes to detect *Extract Variable* candidates.

Algorithm 5 Identification of *Extract Variable* candidates.

```

1: candidates  $\leftarrow$  []
2: for each method  $\in$  sourceFile do
3:   for each nvc  $\in$  getNonVoidCalls(method) do ▷ From Algorithm 4, (p. 104).
4:     if nvc.length  $\geq$  MIN_LENGTH then
5:       range  $\leftarrow$  Range(nvc.start, nvc.end)
6:       candidate  $\leftarrow$  ExtractVariableCandidate(range, nvc, method)
7:       candidates.push(candidate)
8:     end if
9:   end for
10: end for

```

After identifying all the suitable *Extract Variable* candidates, our approach sorts them in descending order according to the length of their respective non-void call expressions. This sorting

mechanism can be further improved by measuring the frequency of each call. This way, we can also remove duplicate code inside a method.

As with the previously described refactorings included in our tool, this kind of refactoring is suggested through the visual methodology presented in Section 4.3, (p. 116). And its execution is performed directly using the built-in command for variable extraction provided by IntelliJ, named *IntroduceVariableHandler*⁸.

4.2.3.4 Move Method

Move Method is a refactoring that allows to move a method from its original class to another that uses it the most [Fow18].

Our approach for identifying *Move Method* refactorings is based on the work from Tsantalis and Chatzigeorgiou [TC09a] and Fokaefs *et al.* [FTC07], where they presented an approach based on the distance between methods and classes. Considering it, we implemented an approach that considers a method and compares the number of times other classes call it in comparison to its original class.

Algorithm 6, (p. 106) summarizes all the steps of our approach to detect possible methods that should be moved to other classes.

First, we verify if the original class is an interface or enumeration. If not, our approach starts by analyzing each method of that class. If a method is not a constructor, a getter, or a setter, we identify all the classes using it (Line 9, (p. 106) of Algorithm 6, (p. 106)). Its own class is always included in this list, even if it does not use the method. Then, for each identified class, we measure the number of calls related to that method. After that, we compare the number of code references assessed for each class. Suppose the number of code references of a class is not zero and it is higher than the number of references of the original class. In that case, that class is more suitable for implementing that specific method. For each more suitable class to have the method implemented inside it, we have an *Move Method* candidate.

Once we find the *Move Method* candidates, we sort them in descending order according to their number of external code references. Thus, candidates with more references to a specific class than another one are placed on the set of refactorings in an earlier position.

Then, our tool suggests each *Move Method* using the visual methodology described in Section 4.3, (p. 116). Its execution is performed based on the approach used for *Move Method* refactorings on IntelliJDeodorant⁹.

⁸*IntroduceVariableHandler* by IntelliJ IDE, <https://github.com/JetBrains/intellij-community/blob/master/java/java-impl-refactorings/src/com/intellij/refactoring/introduceVariable/IntroduceVariableHandler.java>

⁹*Move Method Executor* by IntelliJDeodorant, <https://github.com/JetBrains-Research/IntelliJDeodorant/tree/master/src/main/java/org/jetbrains/research/intellijdeodorant/ide/refactoring/moveMethod>

Algorithm 6 Identification of *Move Method* candidates.

```

1: candidates  $\leftarrow$  []
2: metrics  $\leftarrow$  FileMetrics(sourceFile)
3: for each class  $\in$  sourceFile.classes do
4:   if class is not INTERFACE or class is not ENUM then
5:     for each method  $\in$  class.methods do
6:       if method is not CONSTRUCTOR or method is not GETTER or method is not
         SETTER then
7:         methodMetrics  $\leftarrow$  metrics.get(method)
8:         codeReferences  $\leftarrow$   $\langle$  PSICLASS, INT  $\rangle$  []  $\triangleright$  codeReferences is represented by
           an HashMap with key PsiClass and integer values initiated with zero
9:         codeReferences  $\leftarrow$  classesIncludingMethod(method)  $\triangleright$  Gets all the classes
           invoking that method
10:        for each classReference  $\in$  codeReferences.keys do
11:          for each methodCallExpression  $\in$  classReference.methods do  $\triangleright$  To get
            all the method invocations inside the methods of a class
12:            if methodCallExpression.originalMethod == method then
13:              codeReferences.set(classReference) += 1
14:            end if
15:          end for
16:        end for
17:        ownReferences  $\leftarrow$  codeReferences.get(class)
18:        for each classReference  $\in$  codeReferences.keys do
19:          numReferences  $\leftarrow$  codeReferences.get(classReference)
20:          if numReferences  $\neq$  0 and numReferences > ownReferences and
            classReferences  $\neq$  class then
21:            candidate  $\leftarrow$  MoveMethod(method, class, classReference, numReferences)
22:            candidates.push(candidate)
23:          end if
24:        end for
25:      end if
26:    end for
27:  end if
28: end for

```

4.2.3.5 Replace Inheritance With Delegation

Replace Inheritance with Delegation consists of creating a field to hold an object of a superclass, delegating the relevant methods to that superclass object, and removing the inheritance relationship [Fow18].

To identify *Replace Inheritance with Delegation* refactorings, we based our method on the metrics and respective approach suggested by Lanza and Marinescu to detect *Feature Envy* smells [LMD06].

As seen in Algorithm 7, (p. 108), our methodology starts by analyzing each class defined inside a file. For each class, it checks if it is a subclass. If so, our approach checks if the number of protected or public fields and methods is greater or equal to specific thresholds defined as *NUM_PROTECTED*

and *NUM_PUBLIC*, whose default values is *1.0*. These values can be configured by the tool's users. If one of these conditions is confirmed, our approach gets the respective superclass and starts measuring the number of inherited and override methods.

To do so, it assesses all the methods of the analyzed class and compares them with the methods from its superclass. If a method was overridden, we would increment its counter. Then, we verify the constructors to count the number of super methods calls. For each super method call, we increment the counter of inherited methods. After that, we assess all method call expressions defined inside the class. If the original method from a call expression comes from the superclass, we increment the counter for inherited methods again.

As the last step of Algorithm 7, (p. 108), we verify if the ratio of override or inherited methods is smaller than *RATIO_OVERRIDE* or *RATIO_INHERITED*, which default values are both 33%. These values can also be configured by the users. Then, if the condition is true, the approach has found a possible *Replace Inheritance to Delegation* candidate.

After identifying all the *Replace Inheritance with Delegation* candidates, our approach sorts them in descending order according to their number of overridden and inherited methods.

As with other refactorings included in our tool, this kind of refactoring is suggested through the visual methodology described in Section 4.3, (p. 116). And its execution is also performed directly through the built-in command for replacing inheritance with delegation provided by IntelliJ, which is called *InheritanceToDelegationDialog*¹⁰.

4.2.3.6 Introduce Parameter Object

A method should not have more than three or four parameters. We can reduce the number of parameters of a method by applying the *Introduce Parameter Object* refactoring [Fow18].

To detect this kind of refactoring, our tool uses a straightforward approach. Our system considers all methods defined inside a file and verifies the number of parameters of each method. If a method has more parameters than the threshold *MAX_NUM_PARAMETERS*, whose default value is *4.0*, it is a possible refactoring candidate.

Algorithm 8, (p. 108) synthesizes the described approach. We can further improve this approach by verifying the relationship of the parameters that will be combined into a new object. This way, we would only introduce objects with related parameters.

After identifying all these refactoring candidates, our system sorts them in descending order of the number of parameters that would compose the new object. Finally, these opportunities are suggested through our visual methodology (cf. Section 4.3, p. 116), and they are applied to the code using the built-in command for introducing parameter objects provided by IntelliJ, named *IntroduceParameterObjectDialog*¹¹.

¹⁰*InheritanceToDelegationDialog* by IntelliJ IDE, <https://github.com/JetBrains/intellij-community/blob/master/java/java-impl-refactorings/src/com/intellij/refactoring/inheritanceToDelegation/InheritanceToDelegationDialog.java>

¹¹*IntroduceParameterObjectDialog* by IntelliJ IDE, <https://github.com/JetBrains/intellij-community/blob/master/java/java-impl-refactorings/src/com/intellij/refactoring/introduceParameter/IntroduceParameterDialog.java>

Algorithm 7 Identification of *Replace Inheritance with Delegation* candidates.

```

1: candidates  $\leftarrow \square$ 
2: for each class  $\in$  sourceFile do
3:   if class is INHERITED_CLASS then
4:     if class.numProtectedData  $\geq$  NUM_PROTECTED or class.numPublicData  $\geq$ 
       NUM_PUBLIC then
5:       superClass  $\leftarrow$  class.superClass
6:       numOverride  $\leftarrow$  0
7:       numInherit  $\leftarrow$  0
8:       for each methodSuperClass  $\in$  superClass.methods do
9:         for each method  $\in$  class.methods do
10:          if method is OVERRIDE then
11:            numOverride ++
12:          end if
13:          if method is CONSTRUCTOR and method has SUPER then
14:            numInherit ++
15:          end if
16:        end for
17:      end for
18:      for each methodCallExpression  $\in$  class.methodCalls do
19:        originalMethod  $\leftarrow$  methodCallExpression.originalMethod
20:        if originalMethod.class.equals(superClass) then
21:          numInherit ++
22:        end if
23:      end for
24:      ratioInherit  $\leftarrow$  numInherit / class.numMethods
25:      ratioOverridden  $\leftarrow$  numOverride / class.numMethods
26:      if ratioInherit  $<$  RATION_INHERITED or ratioOverridden  $<$ 
        RATION_OVERRIDE then
27:        counters  $\leftarrow$  [numInherit, numOverride]
28:        candidate  $\leftarrow$  InheritanceDelegationCandidate(class, superClass, counters)
29:        candidates.push(candidate)
30:      end if
31:    end if
32:  end if
33: end for

```

Algorithm 8 Identification of *Introduce Parameter Object* candidates.

```

1: candidates  $\leftarrow \square$ 
2: for each method  $\in$  sourceFile do
3:   if method.parameters.length  $\geq$  MAX_NUM_PARAMETERS then
4:     candidate  $\leftarrow$  IntroParamObject(method)
5:     candidates.push(candidate)
6:   end if
7: end for

```

4.2.3.7 String Comparison

As good programming practices, we should check if strings are equal by using the “*equals()*” method and not the “*==*” operator [Dua22].

Then, to identify *String Comparison* refactorings, our tool uses a straightforward and quick approach. It starts by filtering the binary expressions defined inside each method inside the opened file. Then, it verifies the operands of those expressions. If both operands are strings, it checks if the operator is of type “*==*”. If so, the expression is automatically identified as a refactoring candidate. Algorithm 9, (p. 109) summarizes the described approach.

Algorithm 9 Identification of *String Comparison* candidates.

```

1: candidates  $\leftarrow$  []
2: for each method  $\in$  sourceFile do
3:   for each binaryExpression  $\in$  method.binaryExpressions do
4:     if binaryExpression.leftOperand is STRING and binaryExpression.rightOperand
       is STRING then
5:       if binaryExpression.operator is “==” then
6:         candidate  $\leftarrow$  StringComparison(binaryExpression, method)
7:         candidates.push(candidate)
8:       end if
9:     end if
10:  end for
11: end for

```

After finding the *String Comparison* opportunities, our approach sorts the candidates in descending order of the length of the binary expression represented by the respective candidate. Then, they are recommended using the visual methodology detailed in Section 4.3, (p. 116), and they are executed automatically by replacing the operator with the “*equals()*” method using the PSI capabilities offered by IntelliJ. As described in Algorithm 10, (p. 109), we first need to get left and right operands from the binary expression (cf. Algorithm 3, p. 109). Then, we need to create a new expression now containing the “*equals()*” method and not the “*==*” operand (cf. Algorithm 6, p. 109). Finally, we must replace the old expression with the new one (cf. Algorithm 7, p. 109).

Algorithm 10 *String Comparison* refactoring applier.

```

1: procedure STRINGCOMPARISON(candidate)
2:   binaryExpression  $\leftarrow$  candidate.node  $\triangleright$  candidate is represents the refactoring candidate.
3:   lExpr  $\leftarrow$  binaryExpression.getLOperand
4:   rExpr  $\leftarrow$  binaryExpression.getROperand
5:   newText  $\leftarrow$  lExpr.text + “.equals()” + rExpr.text + “”
6:   equalsCall  $\leftarrow$  createExpressionFromText(newText, null)  $\triangleright$  createExpressionFromText is
     a method that can be accessed through the PsiTree.
7:   binaryExpression.replace(equalsCall)
8: end procedure

```

4.2.4 Artificial Intelligence

One of our initial goals was to include Artificial Intelligence techniques to identify the best refactoring methods. However, when we started implementing some AI algorithms, we verified that the code inspection and further refactoring detection and suggestion took longer than when only using code quality metrics combined with assessing the code's AST.

We know that probably, with AI, we can have better refactoring suggestions, and we can eliminate possible false positive cases [DPT⁺18, HSS17]. However, since the main focus of our project was to verify if a *Live Refactoring Environment* could create better code, faster, we had to balance its outcome in terms of code quality with the time needed to reach those solutions. Therefore, we decided to keep our approach only focused on code quality metrics and not include any AI algorithms.

Next, we describe the AI methodologies we tried to use to identify *Extract Method* (cf. Section 4.2.4.1, p. 110) and *Extract Class* (cf. Section 4.2.4.2, p. 112) refactorings.

4.2.4.1 Extract Method

We tried to use the approach defined by Barbosa [Bar20] to identify *Extract Method* opportunities. He focused his work on using a clustering algorithm to identify *Extract Method* refactorings.

First, he studied the pros and cons of three different unsupervised learning techniques: *K-Means* [Mac67], *DBSCAN* [EKSX96] and *OPTICS* [ABKS99]. Then, he concluded that *OPTICS* was the best one to create a live refactoring approach capable of identifying *Extract Method* candidates on the source code.

However, to use a clustering algorithm, he had to convert the code from each method into feature vectors, an n-dimensional representation of an object of interest. He classified an object of interest as each statement inside a method. Therefore, each cluster would have different statements, each occupying a single line of code. He built the feature vectors by assessing the variables of each statement and their structural dependencies. With these two types of features, he aimed to build clusters that represented the refactoring candidates, allowing a better division of responsibilities inside each method. For that, he used a binary representation, where a *1* states that a certain feature is present, while a *0* states the absence of it. Figure 4.1, (p. 110) and its respective binary representation in Table 4.2, (p. 111) summarized this part of Barbosa approach [Bar20].

```
1 String buildName(String firstName, String lastName) {  
2     if (lastName != "") {  
3         return firstName + " " + lastName;  
4     }  
5     return firstName;  
6 }
```

Listing 4.1: Example of a method in Java.

Table 4.2: Binary representation of the code example of Figure 4.1, (p. 110).

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

The three statements represented in Table 4.2, (p. 111) are the ones from lines 2, 3, and 5 of Figure 4.1, (p. 110). Each statement has a 1 in *firstName* or *lastName* if these variables are used on the respective statement, or 0 otherwise. The features related to the structural dependencies are referred to as the *level* features. These structural features are accumulative (e.g., if a complex statement is present inside the if-statement, then all the statements within it would inherit its *level-1* feature). This implies that all statements possess the *level-0* feature, which is not significant in terms of division during the application of the clustering algorithm. As a result, it can be discarded.

After creating its binary representation, Barbosa [Bar20] analyzed the best distance function to use on the OPTICS algorithm. A distance function helps measure how different (or similar) two statements are in their representation. In his solution, he decided to use the Hamming Distance¹². This distance function can be summarized as follows:

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

where A and B are the set of all the features of two statements with feature's length m .

Thus, with this distance function, he was able to get the number of features the two statements do not share (e.g., statements 1 and 2 in Figure 4.2, (p. 111) only do not share the feature *firstName*, so their distance would be of 1).

Having the correct data representation and the distance function, Barbosa [Bar20] only needed to determine the minimum number of points required to create a cluster. He defined this parameter as the minimum number of lines to include on the *Extract Method* candidate. This value is customizable in his solution.

Upon completing all necessary steps, he executed the OPTICS algorithm and obtained all refactoring candidates. However, some were deemed irrelevant. Then, he evaluated each cluster using the Silhouette Coefficient [Rou87], which considers the similarity between data points within a cluster. The clusters were classified on a scale of $[-1; 1]$, where -1 indicates the misclassification of a data point into another cluster, and $+1$ represents a perfect match. The overall classification of the clusters was obtained by averaging the silhouette of each point. Averages closer to $+1$ denote well-formed clusters and signify significant differences from other clusters.

We implemented this approach in our *Live Refactoring Environment* and tested it on a Java file with more than 345 lines of code. However, as expected, since it takes several steps and time

¹²Hamming Distance, https://en.wikipedia.org/wiki/Hamming_distance/.

to get a fittable set of *Extract Method* candidates (cf. Table 4.4, p. 116), we decided to go with our first choice, which was the one described in Section 4.2.3.1, (p. 99). Besides, we also aim to help developers be aware of possible code flaws, as well as let them be the ones who decide if a refactoring candidate is, in fact, relevant to be applied to their code. So, we also verified that our approach provided more *Extract Method* suggestions than the one from Barbosa [Bar20], even if some of them were very simple and low-impact refactoring candidates.

4.2.4.2 Extract Class

To identify *Extract Class* opportunities, we used the clustering approach presented by Fokaefs *et al.* [FTSC11]. They started by studying the pros and cons of several clustering algorithms, such as *K-Means* [Mac67], *DBSCAN* [EKSX96], or the *Hierarchical Agglomerative Algorithm* [ZMRA13]. Then, they decided to follow the *Hierarchical Agglomerative Algorithm* since it was the only that satisfied all of their criteria:

1. It requires no random initializations (unlike *K-Means* [Mac67]) and it always produces the same output;
2. It does not require pre-defined stop conditions (unlike *K-Means* [Mac67]);
3. It does not require pre-defined input (unlike *K-Means* [Mac67] and *DBSCAN* [EKSX96]).

The *Hierarchical Agglomerative Algorithm* starts by assigning each class property or method to an individual cluster. Then, after each iteration, it merges the two closest clusters into a new one. Lastly, it finishes when all entities (properties or methods) are contained in one cluster, representing the root of a hierarchy of clusters. This hierarchy of clusters can be represented by a dendrogram. Each leaf of the dendrogram tree represents an entity; the root is the final cluster, and the intermediate nodes are the actual clusters. The height of the tree is related to the distance in which two clusters were merged [ZMRA13].

Another relevant characteristic of this clustering algorithm is the *distance merging criteria*. These criteria define which clusters should be merged into a new one. The authors analyzed several methods that help determine the two closest clusters, such as the *complete linkage* (maximum distance between the members of two clusters), *average linkage* (average distance), or the *single linkage* (minimum distance). They verified that, according to Anquetil and Lethbridge [AL99], *complete linkage* favors less coupled clusters, and average linkage is somewhere in-between. Since Fokaefs *et al.* [FTSC11] approach aims at producing fairly cohesive classes with low coupling degree, they decided to follow a *single linkage* methodology to decrease the coupling between the new classes and the original ones. Thus, they used the *Jaccard distance*, which Anquetil and Lethbridge [AL99] said to produce good results in software remodularization.

To use the *Jaccard distance*, Fokaefs *et al.* [FTSC11] needed to employ the notion of entity sets. For that, they used the notion described by Tsantalis and Chatzigeorgiou [TC09a]. According to them, an entity set of an attribute contains:

- the attribute itself;
- the methods directly accessing the attribute that belongs to the same class as the attribute;
- the methods accessing the attribute through public accessors.

The entity set of a method contains:

- the method itself;
- the attributes accessed by the method;
- the methods accessed by the method.

Considering the way they defined the entity sets, the *Jaccard distance* between two entities α and β with entity sets A and B is measured as follows:

$$d_{\alpha,\beta} = 1 - \frac{|A \cap B|}{|A \cup B|}$$

They also defined several rules to eliminate illegal *Extract Class* candidates:

1. The class to be extracted should contain more than one entity;
2. The class to be extracted should contain at least one method;
3. Abstract methods should not be extracted;
4. Fields that have a visibility higher than private and are used by a class other than the source class should not be extracted;
5. Methods that override an abstract or a concrete method of the super class of the original class should not be extracted;
6. The class to be extracted should not contain a method that makes any super method invocations;
7. Methods that are synchronized or contain a synchronized block should not be extracted.

The following example summarizes the approach used by Fokaefs *et al.* [FTSC11] to identify *Extract Class* refactorings. Considering all the steps and rules, only clusters C_3 , C_4 , and C_5 can be defined as possible *Extract Class* opportunities.

```

1 public class Person {
2     private String name;
3     private String job;
4     private String areaCode = "555";
5     private String officeNumber;

```

```

6
7  public Person(String officeNumber) {
8      this.officeNumber = officeNumber;
9  }
10
11 public String getPhoneNumber() {
12     String phone = areaCode + "-" + officeNumber;
13     name += ", " + phone;
14     job += ", " + phone;
15     return phone;
16 }
17
18 public void changeJob(String newJob) {
19     if(!newJob.equals(job)) {
20         this.job = newJob;
21     }
22     name += ", " + newJob;
23 }
24
25 public void modifyName(String newName) {
26     if(!newName.equals(name)) {
27         this.name = newName;
28     }
29     job += newName + ", " + job;
30 }
31 }

```

Listing 4.2: Example of a class in Java.

For the class presented in Figure 4.2, (p. 113), we have the distance matrix represented in Table 4.3, (p. 114) and the corresponding graph (cf. Figure 4.3, p. 115) and dendrogram (cf. Figure 4.4, p. 115).

Table 4.3: Distance matrix of the class represented in Figure 4.2, (p. 113) (based on Table 1 of [FTSC11]).

	<i>name</i>	<i>job</i>	<i>areaCode</i>	<i>officeNumber</i>	<i>changeJob()</i>	<i>modifyName()</i>
<i>job</i>	0.4	-	-	-	-	-
<i>officeAreaCode</i>	0.8	0.8	-	-	-	-
<i>officeNumber</i>	0.8	0.8	0.67	-	-	-
<i>changeJob()</i>	0.6	0.6	1	1	-	-
<i>modifyName()</i>	0.6	0.6	1	1	0.5	-
<i>getPhoneNumber()</i>	0.71	0.71	0.6	0.6	0.67	0.67

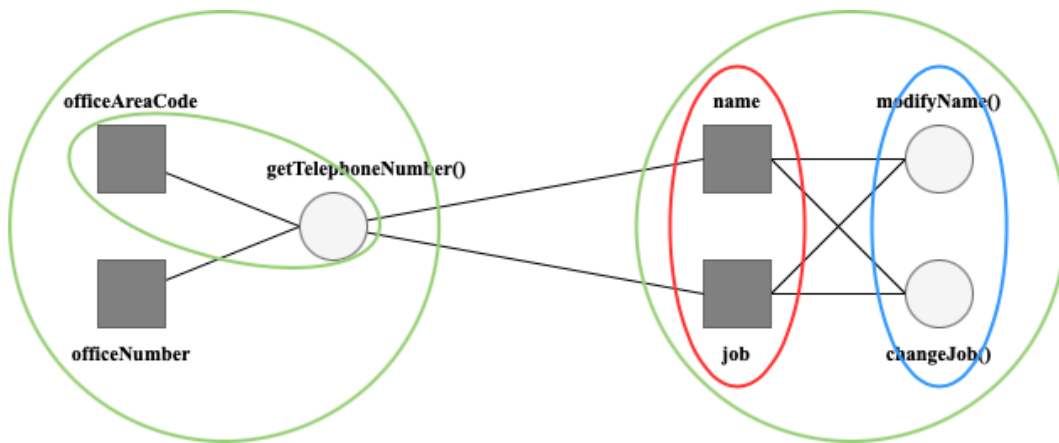


Figure 4.3: Graph of the class represented in Figure 4.2, (p. 113) (based on Figure 2 of [FTSC11]).

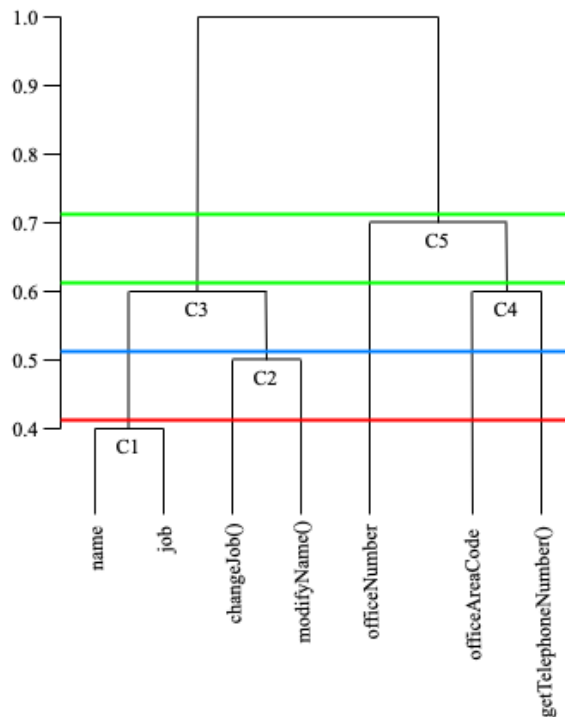


Figure 4.4: Dendrogram of the class represented in Figure 4.2, (p. 113) (based on Figure 3 of [FTSC11]).

Then, after implementing this approach in our *Live Refactoring Environment*, we tested it on a Java file with more than 345 lines of code. As happened with the *Extract Method* refactoring, our solution presented better results than the solution from Fokaefs *et al.* [FTSC11]. As can be seen in Table 4.4, (p. 116), our solution finds more *Extract Class* opportunities than the approach presented by Fokaefs *et al.* [FTSC11], creating more refactoring possibilities and, therefore, more possibilities to improve code quality. Besides, our solution takes less time to present more refactoring candidates. Thus, we decided to go with our approach.

Table 4.4: Number of *Extract Method* and *Extract Class* candidates and the time needed to present them by approach.

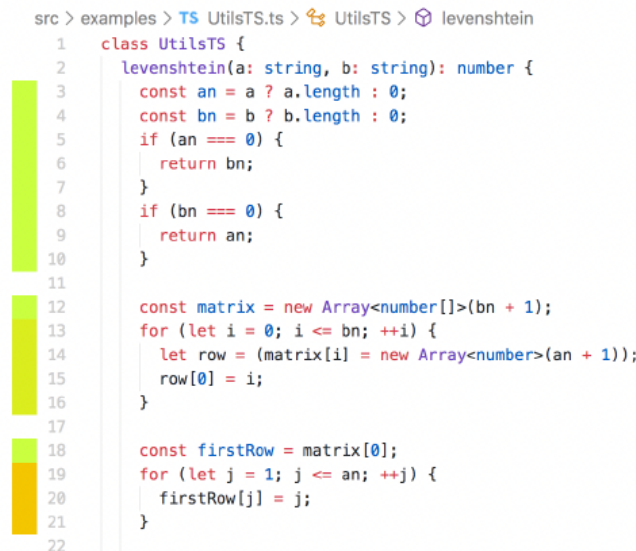
<i>Our Extract Method</i>		<i>Our Extract Class</i>	
<i>Number of Candidates</i>	<i>Time</i>	<i>Number of Candidates</i>	<i>Time</i>
9 candidates	2 seconds	29 candidates	3 seconds
<i>Barbosa [Bar20]</i>		<i>Fokaefs et al. [FTSC11]</i>	
<i>Number of Candidates</i>	<i>Time</i>	<i>Number of Candidates</i>	<i>Time</i>
3 candidates	4 seconds	12 candidates	45 seconds

4.3 Refactoring Visualizer

Software visualization techniques use various methods to represent and analyze software systems in a visual format. These techniques are widely used in software engineering, as they can help developers better understand the structure, behavior, and performance of software systems [SWM00, FLC⁺02, HCL05].

In our case, to develop a *Live Refactoring Environment*, we believe that it is essential to apply visual methodologies mainly to the suggestion of the identified refactorings to help developers quickly and easily understand each refactoring and its importance.

Our visualizer is based on the approach of Barbosa [Bar20] created for suggesting *Extract Method* refactorings in Visual Studio Code (cf. Figure 4.5).



```

src > examples > TS UtilsTS.ts > UtilsTS > levenshtein
1  class UtilsTS {
2      levenshtein(a: string, b: string): number {
3          const an = a ? a.length : 0;
4          const bn = b ? b.length : 0;
5          if (an === 0) {
6              return bn;
7          }
8          if (bn === 0) {
9              return an;
10         }
11
12         const matrix = new Array<number[]>(bn + 1);
13         for (let i = 0; i <= bn; ++i) {
14             let row = (matrix[i] = new Array<number>(an + 1));
15             row[0] = i;
16         }
17
18         const firstRow = matrix[0];
19         for (let j = 1; j <= an; ++j) {
20             firstRow[j] = j;
21         }
22     }

```

Figure 4.5: Example of the visual approach used by Barbosa (Figure 4.5 of [Bar20]).

After identifying and organizing the code areas that need to be refactored, we assessed the urgency of each one. We used a scale from 1 to 9, with 9 being the most pressing and requiring immediate attention. To determine where each candidate fit on the scale, we applied Equation 4.7,

(p. 117) to calculate their severity. In this calculus, *index* represents the index of the candidate, *a* represents the first value of the scale (1, in our case), *b* the last value of the scale (9, in our case), *min* the lowest index of the set of candidates, and *max* the highest index of the same set.

$$severity = (b - a) * \frac{index - min}{max - min} + a \quad (4.7)$$

Our scale for evaluating refactoring candidates ranges from 1 to 9, since we have a gradient of ten colors to represent the urgency of each candidate. This scale ranges from light green to dark red. The light green of the scale represents candidates that are less severe and do not need immediate attention. In contrast, the dark red represents candidates that should be refactored as soon as possible (cf. Figure 4.6).



Figure 4.6: Color gradient used in our visual solution.

To make the process more visually appealing and user-friendly, we have incorporated these colors into clickable gutters on the left side of the text editor. These gutters, shown in Figure 4.7, (p. 117), are small PNG images representing each candidate's severity with the appropriate color. We decided to shrink the size of the gutters even more than Barbosa [Bar20] to minimize distractions for the tool's users while still providing a clear and easy-to-use visual representation of the refactoring candidates.

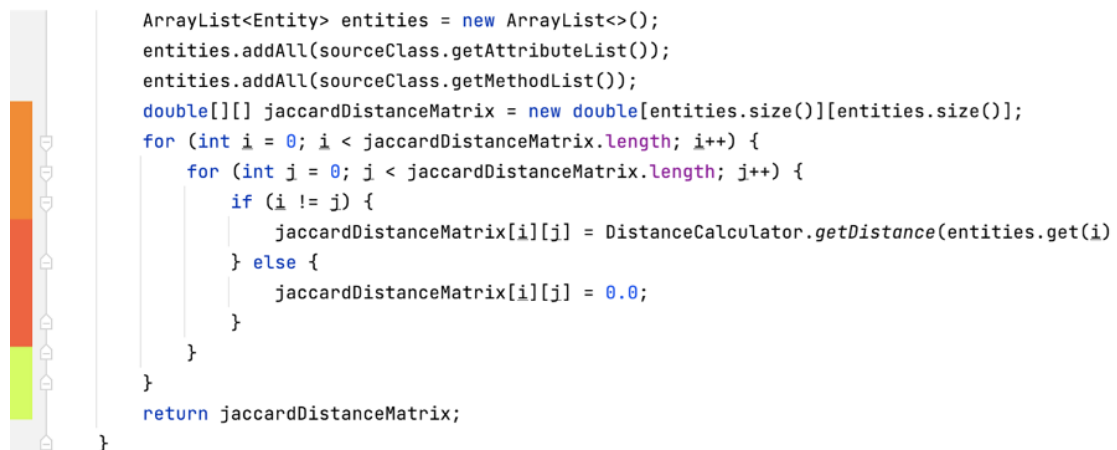


Figure 4.7: An example of our visual approach being used on IntelliJ.

Then, by clicking on a gutter, we can access a refactoring menu to choose the most appropriate refactoring for our code. In some cases, a single statement may be eligible for multiple refactorings, so we only show the color of the refactoring with the highest severity in that group. In the refactoring menu, we list all the overlapping refactorings relevant to that specific position in the code (cf. Figure 4.8, p. 118).

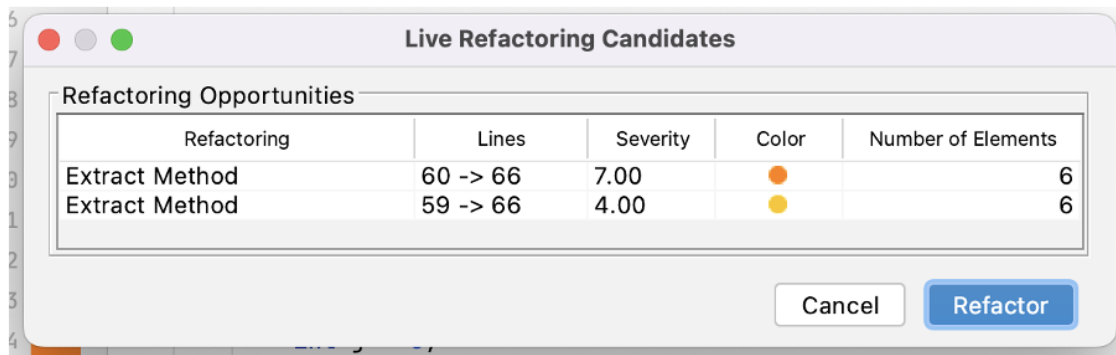


Figure 4.8: An example of our refactoring menu.

Considering the colorblind users, we created a second visual approach using (cf. Figure 4.9). Our tool uses this second visual option if users activate the “*Colorblind*” checkbox in the configuration menu (cf. Figure B.9, p. 236). Then, each refactoring will be represented through its respective symbol.

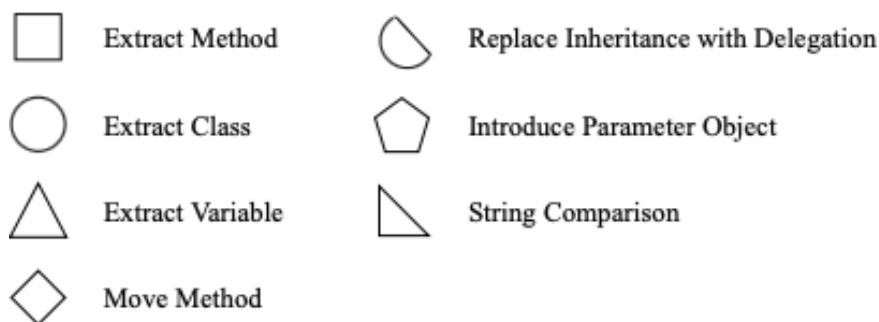


Figure 4.9: Visual approach for colorblind users.

To represent the severity of each refactoring, we converted the color gradient present in Figure 4.6, (p. 117) into a size scale of each symbol. Refactoring candidates with lower severity, which would be represented with greenish colors, are suggested to a colorblind user through smaller icons. More severe refactorings, suggested through darker red colors, are pointed out using larger icons. Figure 4.10, (p. 118) presents the conversion of our color gradient to the symbols’ scale for the *Extract Method* refactoring type.

Figure 4.10: Conversion of the color gradient to the symbols’ scale used to suggest *Extract Method* candidates.

For colorblind people, the refactoring menu is similar to the one from Figure 4.8, (p. 118). The only difference is that for colorblind users, the menu does not have the fourth column (column “color”).

4.4 Liveness Features

To implement a *Live Refactoring Environment*, we also have to consider that the refactoring-loop needs to be the fastest possible, shortening the time between coding or other events performed by the developers and the identification, suggestion, and application of each refactoring.

When the refactoring environment starts, it measures the code quality metrics of the code being displayed in the text editor and visually identifies and suggests the refactoring candidates. Then, after applying a refactoring or even after a coding action, our approach starts a new inspection process in the new code to find new refactoring opportunities.

This happens because we included specific live mechanisms. These mechanisms are triggers provided by IntelliJ that allow knowing when something has changed in the code under analysis. These triggers enable us to initiate our approach at the right time when a significant change occurs at one of the children of the PsiTree that represents the code. For that, we used two different triggers. The first initializes the analysis process after implementing a refactoring or after any coding action (cf. Algorithm 11, p. 120). We can distinguish normal coding actions from refactorings using specific adapters. For coding actions, we used the *PsiTreeChangeListener*¹³, which is called whenever the PsiTree changes. Since this adapter is always called when we change a character on the code, and for performance reasons, we only start a new refactoring analysis after 15 or more characters are changed. For refactorings, we internally print the phrase “*Refactoring applied to the code*”. For that, we used the method *setOut()* from the module *System* that helps catch the respective print.

The second trigger is based on the *FileEditorManagerAdapter*¹⁴. With it, we know whenever we switch the file focused on the text editor to start a new inspection to identify new refactoring candidates (cf. Algorithm 12, p. 120). To shorten the refactoring-loop, in the first trigger, we only measure the metrics related to the method extracted and not all metrics from that file’s methods.

With these triggers, we believe our approach is very reactive to the changes made in the source code and the text editor. Besides, they provide quick and continuous feedback, support, and guidance about the software under analysis, considering any change made to the code.

¹³PsiTreeChangeListener, <https://github.com/JetBrains/intellij-community/blob/master/platform/core-api/src/com/intellij/psi/PsiTreeChangeListener.java>

¹⁴FileEditorManagerAdapter, <https://github.com/JetBrains/intellij-community/blob/master/platform/platform-api/src/com/intellij/openapi/fileEditor/FileEditorManagerAdapter.java>

Algorithm 11 Sequence of code that starts a new refactoring inspection whenever a refactoring or code action happens.

```

1: metricsBefore  $\leftarrow$  getOldMetrics()  $\triangleright$  Gets the metrics measured before applying a refactoring
   to the code
2: metricsAfter  $\leftarrow$  before.changeMetrics(refactoring)  $\triangleright$  Only changes the metrics related to
   the methods involved in the refactoring previously performed
3: newSourceFile  $\leftarrow$  editor.currentFile  $\triangleright$  Gets the new source file with the refactoring already
   applied to the code
4: refactorings  $\leftarrow$  []
5: refactorings  $\leftarrow$  getCandidates(newSourceFile)  $\triangleright$  Finds the new refactoring set in the new
   source code

```

Algorithm 12 Sequence of code that starts a new refactoring inspection whenever we change the focus between text files.

```

1: sourceFile  $\leftarrow$  editor.currentFile
2: metricsBefore  $\leftarrow$  FileMetrics(sourceFile)
3: refactorings  $\leftarrow$  getCandidates(sourceFile)  $\triangleright$  Finds the new refactoring set in the new source
   code

```

4.5 Limitations

One of the limitations of our *Live Refactoring Environment* is the small number of refactoring included in it. Until now, our approach only supports the *Extract Method*, *Extract Class*, *Extract Variable*, *Move Method*, *Introduce Parameter Object*, *Replace Inheritance with Delegation*, and *String Comparison* refactorings. So, we believe that by having a small number of refactoring supported by our environment, we are decreasing the range of code smells that could be mitigated and, therefore, reducing the opportunities to improve the code quality and its readability, adaptability, and maintainability.

Another limitation is the time it takes to inspect the code and present the refactoring opportunities. Our approach takes a few seconds to identify and suggest most of the refactorings including on it. Refactorings like *Extract Class* or *Move Method* take more time, since they simultaneously analyze multiple factors, like method combination or the distance of a method to the classes of the software. Besides, as the size of the code increases, the time required to inspect it also increases. To consider our approach a fully live refactoring environment, it should only take up to one or two seconds to analyze code and suggest possible refactorings in all the cases. So, we considered it a limitation that should be addressed in future work.

4.6 Summary

The proposed solution to our research problem consisted of a *Live Refactoring Environment*. We developed it as a Java plugin for IntelliJ IDE. It allowed developers to receive visual feedback and guidance on how to refactor their code, while programming. This solution was mainly based

on assessing specific code quality metrics, such as cyclomatic complexity, Halstead Effort, or lack of cohesion.

In this approach, we included some of the most well-known refactorings from the catalog defined by Martin Fowler [Fow18]. Thus, our plugin identifies, suggests, and applies refactorings like the *Extract Method*, *Extract Class*, *Move Method*, or *Introduce Parameter Object*.

In Table 4.5, (p. 121), we summarize the refactorings supported by our *Live Refactoring Environment*, briefly describing the main problems they aim to reduce. We also synthesize the metrics and respective thresholds used to identify these refactorings.

Table 4.5: Summary of the refactorings and metrics included in this research project and the metrics and thresholds used to identify them.

<i>Refactoring</i>	<i>Problem</i>	<i>Metrics</i>
Extract Method	Block of code fragments that can be grouped in a new method	Lines of Code ≥ 30 , Method Length ≥ 60 , Cyclomatic Complexity ≥ 15 , Halstead Effort ≥ 300
Extract Class	Class that does the work of two or more classes	Number of Methods ≥ 4 , Lack of Cohesion ≥ 0.33 , Foreign Data ≥ 2
Extract Variable	Complex expression that is hard to understand	Expression Length ≥ 12 characters
Move Method	Method that is used more in another class than in its own class	Number of External References $\geq$ Number of Own References
Replace Inheritance with Delegation	Subclass that uses only a portion of the methods of its superclass	Protect Fields + Protected Methods ≥ 1 , Public Fields + Public Methods ≥ 1 , Ratio Override Data $\geq 33\%$, Ratio Inherited Data $\geq 33\%$
Introduce Parameter Object	Method containing a repeating group of parameters or a large number of parameters	Number of Parameters ≥ 4
String Comparison	Expression that uses operator “==” - to compare strings instead of method <i>equals</i>	

In the next chapter, Chapter 5, (p. 123), we present the empirical experiments that we performed as an attempt to evaluate our solution, validate our hypothesis (*cf.* Section 3.3, p. 82), and answer

our research questions (*cf.* Section 3.4, p. 82). There, we also summarize all the data collected from the experiments and discuss the results achieved from them.

Chapter 5

Empirical Experiments

5.1	Live Extract Method Experiment	124
5.2	Live Refactoring Experiment	143
5.3	Automated Experiment	164
5.4	Answering the Research Questions	184
5.5	Threats to Validity	187
5.6	Summary	190

As described in Chapter 4, (p. 91), we developed a *Live Refactoring Environment* to address the challenges outlined in our research problem (*cf.* Chapter 3, p. 79). This environment inspects code, visually recommends specific refactoring techniques, and applies them to enhance code quality more efficiently.

In engineering research, validating a proposed solution is a critical step, often conducted during or after its implementation [ZW98]. Our project adhered to this principle.

We conducted three experiments to validate our *Live Refactoring Environment*. The first experiment was carried out during the initial development stages and was focused on the *Extract Method* refactoring. It aimed to evaluate the tool’s usability and reliability (*cf.* Section 5.1, p. 124). The second experiment occurred after the tool’s development (*cf.* Section 5.2, p. 143). The third experiment, also conducted at the end of the project, involved an automated bot that crawled and refactored GitHub projects (*cf.* Section 5.3, p. 164).

To ensure the integrity of the data collected in these experiments—covering accuracy, completeness, legibility, and timeliness—we recorded and preserved all data locally on an external drive, Firebase databases, and Google Drive folders. By providing access to this data, we facilitate replication of our experiments by other researchers.

To address our research questions, we performed some hypothesis testing, defined by the following null and alternative hypotheses.

RQ1 “*Is it possible to inform developers, using a color-coded system, about possible refactoring opportunities in their code?*”

- **Null Hypothesis.** A color-coded system cannot be used to inform developers about possible refactoring candidates in their code.
- **Alternative Hypothesis.** A color-coded system can be used to inform developers about possible refactoring candidates in their code.

RQ2 “*What is the impact of real-time refactoring identification on the developers’ awareness of their code quality?*”

- **Null Hypothesis.** A real-time refactoring identification approach has no impact on the developers’ awareness of their code quality.
- **Alternative Hypothesis.** A real-time refactoring identification approach positively impacts the developers’ awareness of their code quality.

RQ3 “*What is the impact of a live refactoring environment on their code quality?*”

- **Null Hypothesis.** A real-time refactoring environment has no impact on code quality.
- **Alternative Hypothesis.** A real-time refactoring environment positively impacts code quality.

RQ4 “*What is the impact of a live refactoring environment on the time required to achieve high-quality programming solutions?*”

- **Null Hypothesis.** A live refactoring environment has no impact on the time needed to reach good programming solutions.
- **Alternative Hypothesis.** A live refactoring environment positively impacts the time needed to reach good programming solutions.

5.1 Live Extract Method Experiment

In an attempt to check if our *Live Refactoring Environment* helped developers converge to better code quality, faster, we performed an initial experiment. This experiment was carried out in the initial stage of our tool, only including the *Extract Method* refactoring.

To fulfill our experiment goals, we predefined a set of programming tasks (*cf.* Section 5.1.2, p. 125) that were performed by different predefined experimental groups (*cf.* Section 5.1.1, p. 125). Depending on the experimental group, the participants had access to a specific version of our tool, its user guide (*cf.* Section B, p. 233), a short video tutorial, and the experiment guide describing each programming task (*cf.* Section C.3.1, p. 261). These artifacts were saved in a Google Drive folder. After finishing their tasks, the participants had to fill out a Google Form with several questions on the experiment and the tool’s usability (*cf.* Section C.3.2, p. 282).

5.1.1 Experimental Groups

This experiment involved 56 undergraduate programming students. Their identities were kept anonymous in all records collected during the experiment. The only requirement to participate in this experiment was to have basic knowledge of programming using Java and the IntelliJ IDE.

We equally divided the experiment participants into four groups (*cf.* Figure 5.1). Group *A1* had access to our *Live Refactoring Environment* that displayed all the *Extract Method* refactoring candidates found for a specific code. The group *A2* also had access to the same approach, but it only displayed the most severe *Extract Method* candidate per iteration. Group *A3* had the same approach as group *A1*, but the *Extract Methods* were pointed out in an HTML file that had to be opened outside the IDE. Finally, group *B* had no access to any refactoring tool. The participants from this group only used a complementary tool that measured the evolution of specific code quality metrics over each programming task.

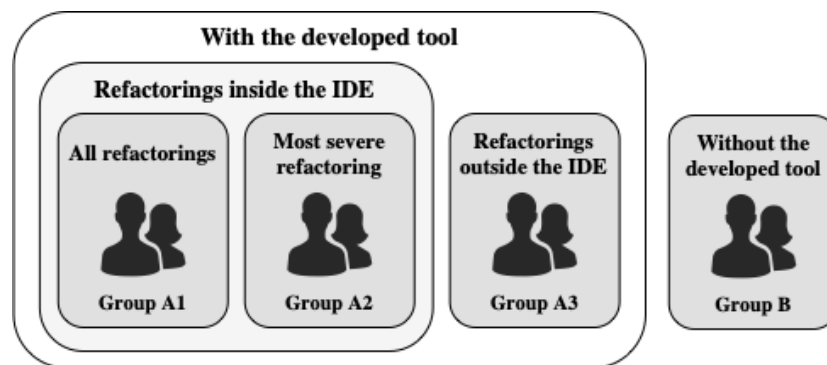


Figure 5.1: Synthesis of the groups that participated in our first empirical experiment.

With this group division, our goal was to check if a *Live Refactoring Environment* helped improve the code quality more efficiently. This was done by comparing the results of groups *A1*, *A2*, and *A3* with those of group *B*. Additionally, we aimed to determine the most effective way to present refactoring candidates and help to converge to better programming solutions. To achieve this, we compared the results of groups *A1*, *A2*, and *A3*.

5.1.2 Programming Tasks

The participants of this experiment needed to install our refactoring environment and carry out a series of tasks. These tasks were identical for all experimental groups.

They consisted of three Java projects with different levels of difficulty. We provided a brief description and a UML diagram of each project to help the participants understand them. During the tasks, they could apply all the suggested refactorings or only the ones they deemed relevant to their code.

Task 1: The first task was a warm-up exercise to ensure that each participant understood the tool they were using. The task involved a minimal and simple movie rental system and was based on one of the examples for *Extract Method* refactoring presented by Fowler on his book [FB99]¹.

Task 2: For this task, we provided a “*Space Invaders*” project² to the participants work on. They were required to implement a method with the game cycle (class *Board*) and implement *Extract Methods* while programming the respective method. The code was provided with comments explaining all the necessary implementation steps.

Task 3: For this task, we focused on the *JHotDraw* project³ which is a large and complex project with multiple classes containing more than 1,000 lines of code, long methods, and duplicated code. *JHotDraw* is used in other research projects as a code source for empirical experiments [AI 15]. The goal was to identify and apply *Extract Methods* in two different Java files (classes *DrawApplication* and *StandardDrawingView*). This task aimed to check if the code improves and becomes more readable, adaptable, and maintainable after refactoring it.

5.1.3 Outcomes

Our experiment resulted in four main types of outcomes. All these outcomes helped us answer our research questions (cf. Section 3.4, p. 82) using hypothesis tests.

Outcome A: This outcome contained the characterization of our experiment participants. For that, we asked them to fill out a Google Form with several questions that helped us understand their programming background.

Outcome B: This outcome included the evolution of several code quality metrics (e.g., Halstead metrics or the cyclomatic complexity) over each refactoring applied. The values of these metrics helped us compare the experimental groups to check which one converged to a better code. With it, we aimed to answer the research questions RQ2 and RQ3.

Outcome C: This outcome is related to the time needed to perform each refactoring to the code in each task. We aimed to evaluate the time efficiency of our approach and compare the times measured by each experimental group. With this outcome, we wanted to answer the research question RQ4.

Outcome D: This outcome included the answers to a questionnaire based on the System Usability Scale (SUS) [Bro13]. With this questionnaire, we aimed to evaluate not only our tool’s usability but also its reliability. With this outcome, we sought to answer the research question RQ1.

¹Refactoring example - <https://www.cs.unc.edu/~stotts/COMP204/refactor/chap1.html>

²Space Invaders - <https://github.com/tatilattanzi/java-space-invaders>

³JHotDraw - <https://github.com/wrandelshofer/jhotdraw>

5.1.4 Results and Discussion

After finishing this experiment, we analyzed and discussed the data collected from it. In this section, we detail those results and interpret and discuss them in order to validate the project's hypothesis (*cf.* Section 3.3, p. 82) and to answer its research questions (*cf.* Section 3.4, p. 82).

5.1.4.1 Participants Characterization

As previously said, our experiment participants were bachelor students from programming classes (*cf.* Section 5.1.1, p. 125). To know more about their programming background, we asked them about their programming experience and experience using Java programming language or IntelliJ IDE.

Figure 5.2, (p. 129) summarizes the answers for each question made. Notice that each question was answered in a range from 1 to 5, in which 1 represented *Very Poor Experience* and 5 represented *Very Good Experience*.

- **Programming experience:** For the first question, in which we tried to evaluate the participants' overall programming experience (*cf.* Figure 5.2a, p. 129), we verified that 54% said having good programming experience. The other 46% gave a neutral answer.
- **Experience programming with Java:** For the second question, in which we aimed to check if the participants already knew how to program using Java (*cf.* Figure 5.2b, p. 129), we verified that 39% said having a good programming knowledge on the program using Java. 59% said to have a neutral experience. And only 2% mentioned having a lack of knowledge on how to code using this programming language.
- **Experience using different data structures:** For the third question, we wanted to check if they knew how to use different Java data structures like *ArrayList*, *Set*, or *HashMap* (*cf.* Figure 5.2c, p. 129). With it, we verified that only 36% said to have good knowledge of using different data structures. 62% said to have neutral knowledge. And, only 2% said to have a poor knowledge of how to use them.
- **Experience with IntelliJ IDE:** For the fourth question, in which we aimed to verify if the participants already knew how to use IntelliJ IDE (*cf.* Figure 5.2d, p. 129), we checked that only 29% mentioned to have good knowledge on how to use IntelliJ IDE. 64% gave a neutral answer. And 7% of the participants said to have poor or no knowledge of how to use it.
- **Experience using refactoring tools:** For the fifth question, in which we wanted to know if the participants already used a refactoring tool or not (*cf.* Figure 5.2e, p. 129), we verified that they had a poor experience using this type of tools (36% of the answers), never used one (61% of the answers), or have a neutral experience (3% of the answers).

- **Experience with code metrics analysis tools:** For the sixth question, in which we wanted to know if the participants already used a code metrics analysis tool or not (*cf.* Figure 5.2f, p. 129), we verified that they most or never used this kind of tool (with 80% of the answers) or had a poor experience (20% of the answers).

5.1.4.2 Code Quality

Using Outcome B (*cf.* Section 5.1.3, p. 126), we computed the impact of our *Live Refactoring Environment* on several code quality metrics. For that, we compared the evolution of those metrics over the refactorings applied by each experimental group in each task. We did not access the evolution of the metrics for the first task since it was just a start-up exercise to get the participants used to the developed tool and IntelliJ IDE.

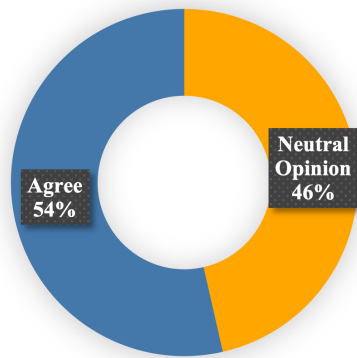
When we started checking the collected results, we verified that the participants sometimes decided to change the projects' code before starting the experiment itself. To reduce biased results, we computed the metrics to make each start on the same point for each experimental group. We also normalized the results to be between 0 and 1 to simplify their visualization on the following charts. Besides, notice that each chart has lines displayed with dots. The dotted parts of the charts represented the continuity of that metric because, in that experimental group, no more refactorings were applied.

Figure 5.3, (p. 130) presents the charts containing the evolution of metrics like the Halstead Metrics, cyclomatic complexity, or maintainability index for the second programming task. As can be seen on those charts, the experimental group A1 was the one that reached better results for each metric at the end of the experiment. In fact, the final code of this group was comparatively shorter, less complex, effortless to understand, and easier to maintain. Besides, when comparing each metric over time, they were also the ones that produced better code in earlier stages. This means that probably having a tool that points out several refactorings at a time helps to make better refactoring decisions that, consequently, help create code with more quality.

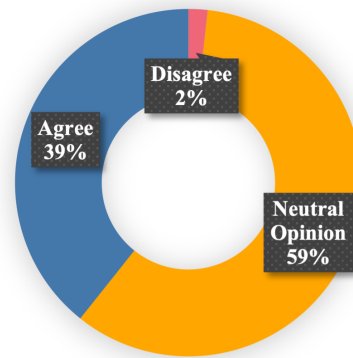
By further analyzing the charts of Figure 5.3, (p. 130), we also verified that in most metrics, group A2 was the one that got the second best results, followed by group A3. This means that breaking the “*programming mindset*” by placing the refactoring suggestions outside of the IDE creates worse code than by placing them inside, near the code, even if the tool only suggests one refactoring at a time. Also, as expected, and since its participants did not have any refactoring help, group B was the one that converged to worse code quality.

However, these facts do not hold when analyzing the evolution of the same metrics in the third task. Figure 5.4, (p. 131) and Figure 5.5, (p. 132) show the charts for the first and second files used on this task.

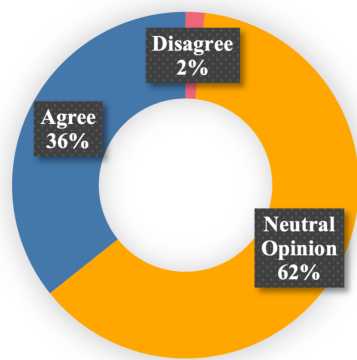
Interestingly, in some metrics like the cyclomatic complexity, Halstead difficulty, and maintainability for the first file, and Halstead difficulty for the second file, group B achieved better values. We do not have a clear explanation for this fact. Probably, the participants of this group made better refactoring decisions when doing it manually.



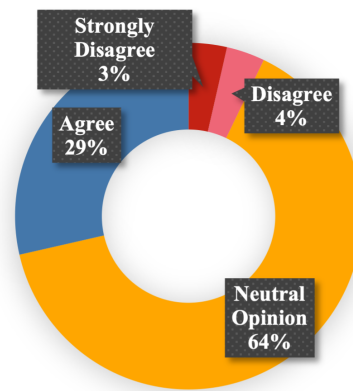
(a) Programming experience.



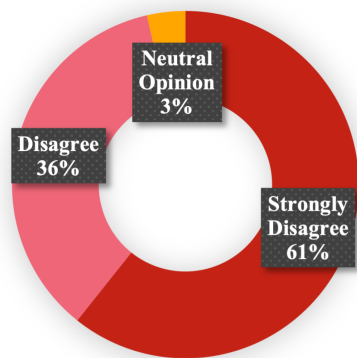
(b) Experience in programming with Java.



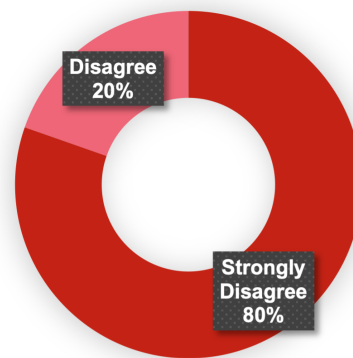
(c) Experience using different data structures.



(d) Experience using IntelliJ IDE.



(e) Experience using refactoring tools.



(f) Experience using code metrics analysis tools.

Figure 5.2: Characterization of the participants from the first experiment based on their experience.

On the remaining metrics, experimental groups *A1*, *A2*, or *A3* achieved similar quality results. This means that most of the time, our approach, independently of its version, was effective and helped converge to better programming solutions.

By putting all this data together, despite some unexpected results, we thought it was possible to

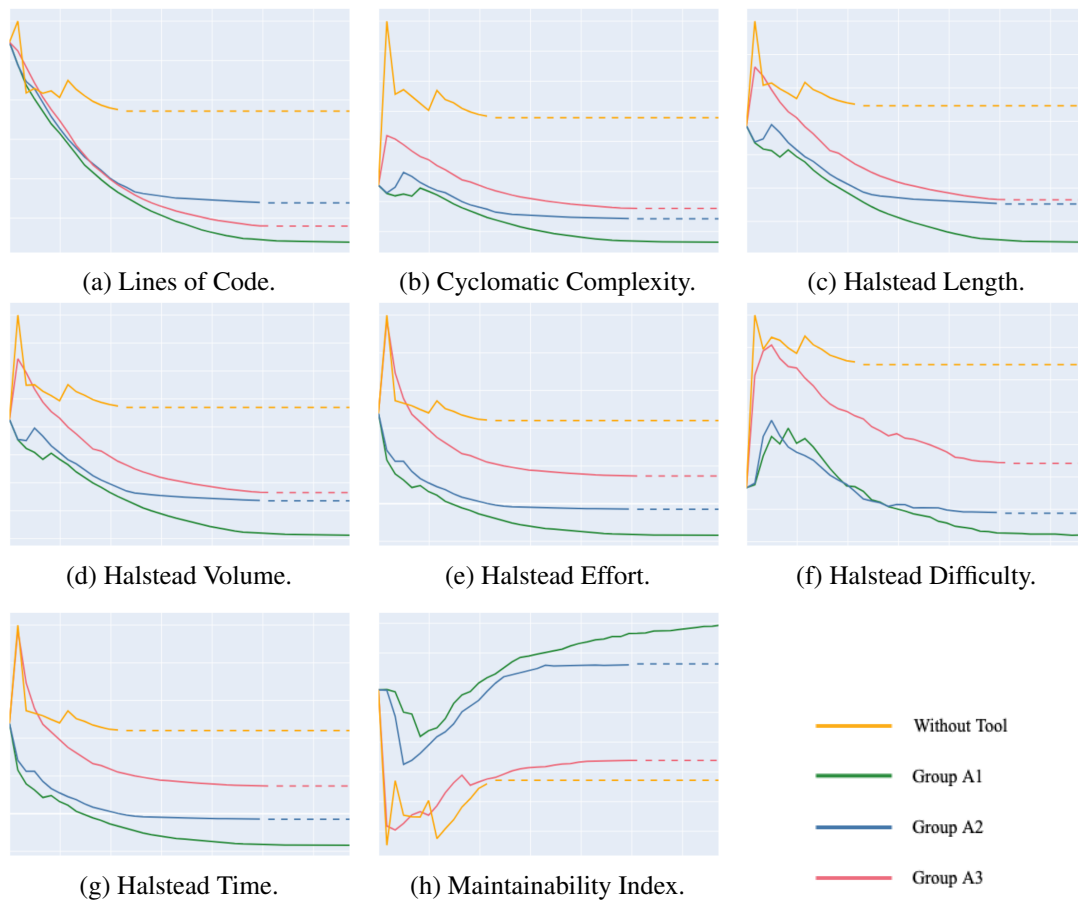


Figure 5.3: Metrics evolution by experimental group on Task 2.

validate our beliefs on our *Live Refactoring Environment*. Hence, we can conclude that a refactoring environment that supports and directs software developers on what and how to refactor their code aids them in reaching improved programming solutions with higher code quality.

5.1.4.3 Number of Refactorings

To determine the average number of refactorings performed by each experimental group, we examined the results obtained from Outcome B (*cf.* Section 5.1.3, p. 126). Additionally, we saved the number of refactorings carried out during each code iteration in a JSON file, which stores the progress of the code quality metrics. This enabled us to ascertain the total number of refactorings implemented by each participant in a particular group and project and subsequently compute their average value (*cf.* Table 5.1.4.4, p. 132).

The results presented in Table 5.1.4.4, (p. 132) reveal that, on average, the group *A1* executed a higher number of refactorings than groups *A2*, *A3*, and *B* (as can also be verified on the charts presented in Section 5.1.4.2, (p. 128)). Notably, the difference between the number of refactorings implemented by groups *A1*, *A2*, and *A3* is relatively small. This finding suggests that by implementing the most effective refactoring at a time, we can achieve similar improvements in code quality

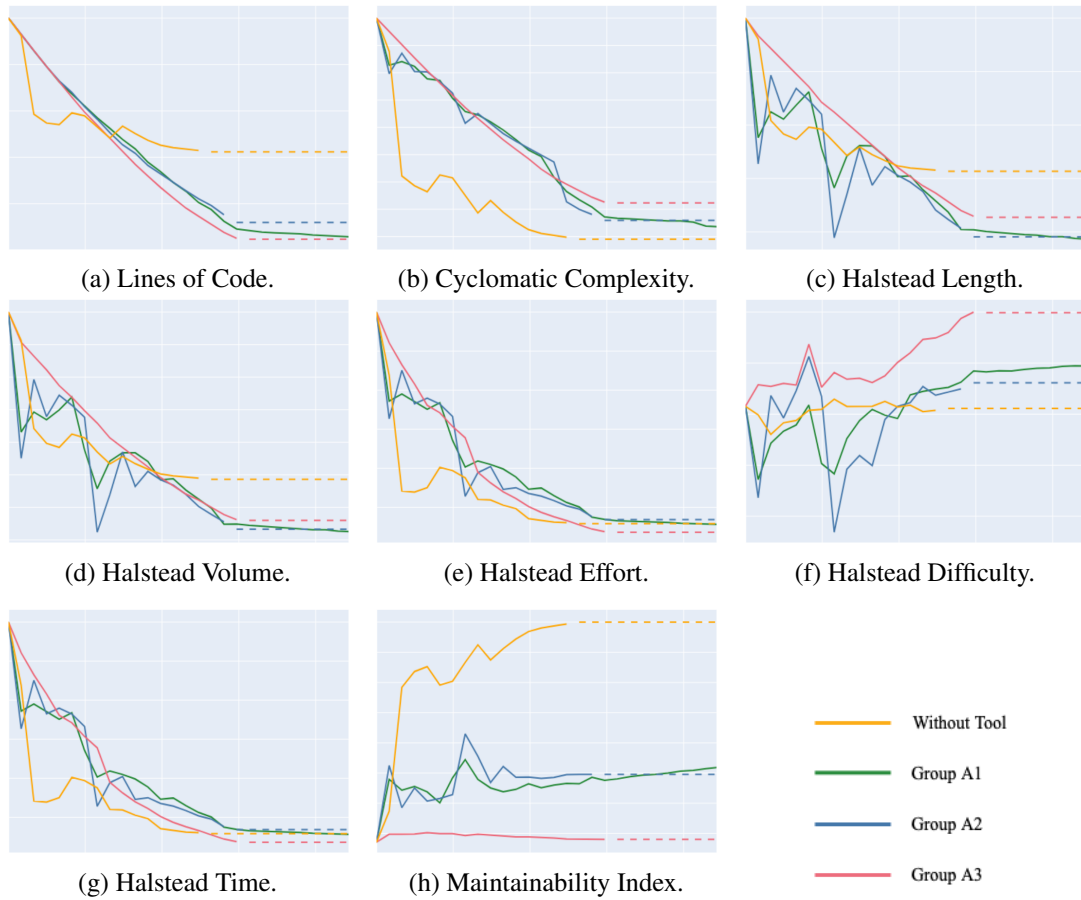


Figure 5.4: Metrics evolution by experimental group on Task 3 File 1.

metrics, regardless of whether the suggestions are integrated within the IDE or not.

Combining this analysis with the one presented by the charts of Figure 5.1.4.2, (p. 128), we concluded that participants in group *A1* achieved significant improvements across all quality metrics with fewer refactorings than the other groups.

In contrast, the group *B* only executed half the refactorings the other groups performed. This implies that certain blocks of their code may have remained unrefactored. The corresponding charts in Figure 5.1.4.2, (p. 128) validate this conclusion, as the lower number of refactorings implemented by group *B* resulted in less substantial improvements across all metrics than the other groups.

5.1.4.4 Time Efficiency

In Section 5.1.3, (p. 126), we present Outcome C, which evaluates the time taken to execute an *Extract Method* to the code across the different experimental groups. These measurements provide insights into the practical efficiency of our *Live Refactoring Environment* [Moo03]. Table 5.1.4.4, (p. 132) presents an overview of the average time required to implement an *Extract Method* in each experimental group.

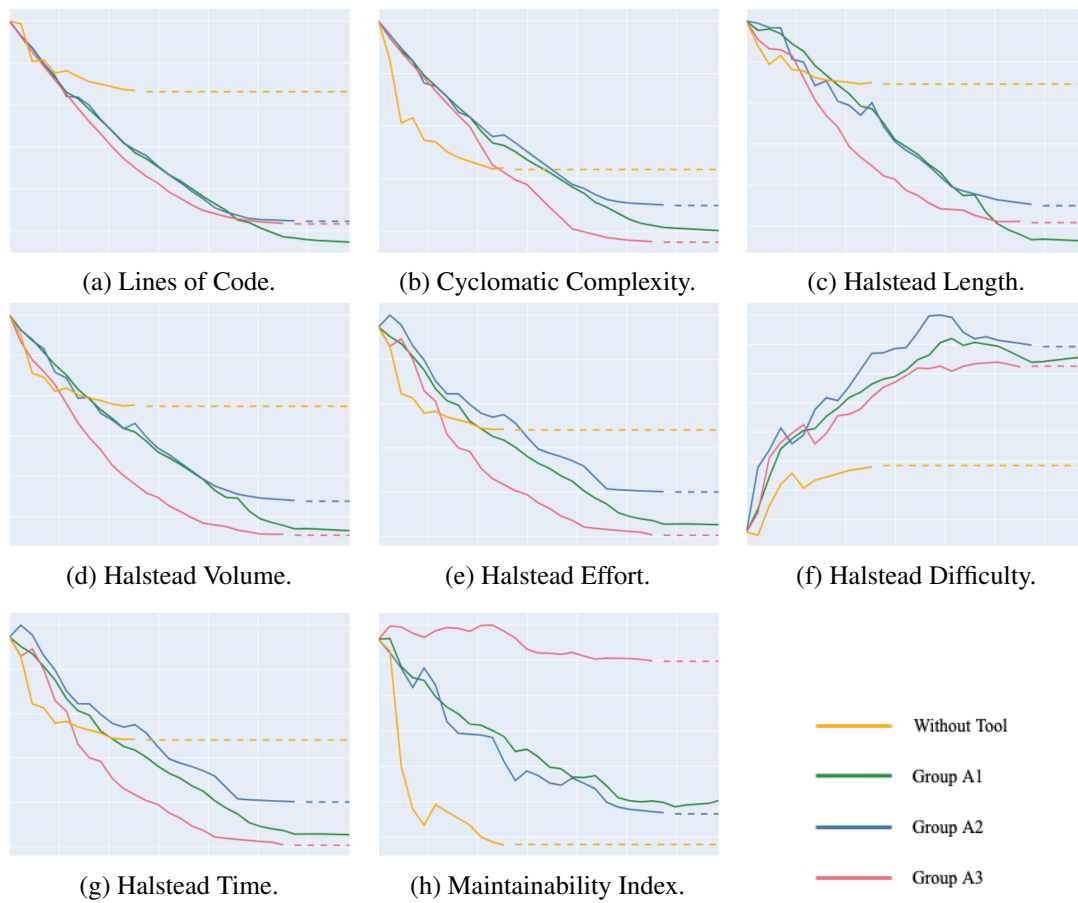


Figure 5.5: Metrics evolution by experimental group on Task 3 File 2.

The results in Table 5.1.4.4, (p. 132) show a difference of over 30 *seconds* in the average time required to implement an *Extract Method* between the experimental groups that utilized our approach (groups A1, A2, and A3) and group B. This supports the idea that participants who received refactoring feedback are more likely to accept and apply the proposed refactoring more quickly. This also validates our hypothesis that a real-time refactoring environment that identifies, suggests, and implements refactorings can accelerate the refactoring process, enabling developers to achieve cleaner, more readable, adaptable, and maintainable code faster than by manually refactoring code without external guidance or assistance.

Table 5.1: Average number of refactorings applied and time efficiency by experimental group.

	<i>Group A1</i>	<i>Group A2</i>	<i>Group A3</i>	<i>Group B</i>
Number Refactorings	16	13	14	6
Time Efficiency	29	36	48	60

5.1.4.5 Comparing Experimental Groups

To check if a *Live Refactoring Environment* was capable of helping developers converge to better code, faster, we conducted several hypothesis tests using specific code quality metrics. Notice that we performed each test for the data collected from each task and each experimental group. Besides, we only used the results collected from Tasks 2 and 3. Again, we did not use any data from Task 1 since it was only a simple warm-up task with no value for our primary goal.

We started this process by verifying whether the data collected from this experiment in each experimental group followed a normal distribution. For that, we applied the *Shapiro-Wilk test* [SW65], with the null hypothesis that the data was normally distributed (*cf.* Table 5.2, p. 133). With this test, we verified that most of the p-values were lower than 0.05 (cases painted in blue). Even though there were cases where the p-value was higher than 0.05, they were not enough to assume our data followed a normal distribution. Therefore, we were able to reject the null hypothesis for this test and concluded that our samples were not parametric.

Table 5.2: Null and alternative hypotheses of the tests used to compare the experimental groups from the first experiment.

<i>Shapiro-Wilk Test</i>	
Null Hypothesis	Data follow a normal distribution.
Alternative Hypothesis	Data does not follow a normal distribution.
<i>Wilcoxon Rank Sum Test</i>	
Null Hypothesis	The experimental groups are symmetric.
Alternative Hypothesis	The experimental groups are not symmetric.

Table 5.3: *Shapiro-Wilk test* on the data collected on the preliminary experiment.

		<i>Group A1</i>	<i>Group A2</i>	<i>Group A3</i>	<i>Group B</i>
Lines of Code	T2	0.00455	0.23132	0.11129	0.14503
	T3-F1	0.00264	0.00106	0.00491	0.70948
	T3-F2	0.04242	0.00248	0.718757	0.00316
Cyclomatic Complexity	T2	0.00369	0.36367	0.44176	0.03559
	T3-F1	0.00827	0.00286	0.00597	0.92575
	T3-F2	0.02116	0.25767	0.14265	0.02336
Cognitive Complexity	T2	0.07372	0.59710	0.04065	0.46798
	T3-F1	0.00067	0.00043	0.01467	0.00963

Continued on next page

Table 5.3: *Shapiro-Wilk test* on the data collected on the preliminary experiment. (Continued)

		<i>Group A1</i>	<i>Group A2</i>	<i>Group A3</i>	<i>Group B</i>
	T3-F2	0.06731	0.00081	0.12858	0.10215
Halstead Length	T2	0.00256	0.20296	0.24786	0.01110
	T3-F1	0.00051	0.00029	0.00853	0.38739
	T3-F2	0.18514	0.51172	0.24002	0.21210
Halstead Volume	T2	0.00143	0.18134	0.19338	0.01731
	T3-F1	0.00033	0.00029	0.00398	0.24276
	T3-F2	0.01562	0.14790	0.49618	0.08613
Halstead Effort	T2	0.00062	0.07709	0.33600	0.08568
	T3-F1	0.00064	0.00116	0.00465	0.29568
	T3-F2	0.98190	0.05255	0.76399	0.00182
Halstead Difficulty	T2	0.91450	0.11063	0.17499	0.44169
	T3-F1	0.01296	0.00457	0.13966	0.46809
	T3-F2	0.05531	0.03600	0.01713	0.06778
Halstead Time	T2	0.00062	0.07709	$7.90e^{-33}$	0.08568
	T3-F1	0.00064	0.00116	0.00811	0.29568
	T3-F2	0.98190	0.05255	0.00001	0.00182
Maintainability	T2	0.03179	0.39050	0.01286	0.43528
	T3-F1	0.00443	0.00059	0.00388	0.50748
	T3-F2	0.05350	0.00470	0.79907	0.04295

We also decided to verify if our experimental groups were symmetrical or not. For that, and since our data was not parametric, we decided to use the *Wilcoxon Rank-Sum test* (also known as the *Mann Whitney U test*) [Zim14]. This test was based on the null hypothesis that the data from the different experimental groups were symmetrical(cf. Table 5.2, p. 133).

Notice that hypothesis tests can have false positives, especially when comparing different experimental groups. To reduce those cases, we can use the *Bonferroni Correction method*, a common method for controlling the family-wise error rate when testing multiple hypotheses simultaneously or comparing experimental groups. When we perform multiple tests, it adjusts the p-values to account for the increased risk of *Type I errors*. *Type I errors*, also known as “false positives” or “alpha errors”, are a common concept in hypothesis testing and statistics. They occur when we incorrectly reject a null hypothesis when it is actually true. In other words, we conclude that there is a significant effect or difference when there is not one [PDvdL05, Hay13].

Thus, we applied the *Bonferroni Correction method* to the results obtained through the *Wilcoxon Rank-Sum test*. As can be seen in Table 5.18, (p. 158), even after applying the *Bonferroni method*, in most cases, the p-values were lower than 0.05 (cases painted in green), indicating that we could reject the null hypothesis for this test. Thus, we concluded that our experimental groups were mainly asymmetrical.

Our data does not conform to a normal distribution, and our groups are asymmetric, so we used the *Permutation test* to compare our experimental groups. This statistical method is advantageous to compare non-parametric data that do not comply with the symmetry assumption [PS10].

We executed the *Permutation test* with 1000 permutations. Besides, we based our test on different alternative hypotheses (cf. Table 5.5, p. 136). For metrics like cognitive complexity, any Halstead Metric, or the time elapsed between refactorings, we used a one-tailed alternative hypothesis to check if the metrics from the first group being compared were significantly lower than those from the second group. For the maintainability index, we used a one-tailed alternative hypothesis but, in this case, in the opposite direction. For the number of refactorings, we performed two-tailed tests.

Table 5.4: *Wilcoxon Rank Sum test* results after applying the *Bonferroni Correction method* to check if the experimental groups from the preliminary experiment were symmetric.

		<i>A1-A2</i>	<i>A1-A3</i>	<i>A2-A3</i>	<i>A1-B</i>	<i>A2-B</i>	<i>A3-B</i>
Lines of Code	T2	0.03263	0.32321	0.04564	0.00015	0.00472	0.00022
	T3-F1	0.19486	0.35305	0.66046	0.00782	0.03562	0.00300
	T3-F2	0.04543	0.07686	0.39511	0.00012	0.05727	0.00010
Cyclomatic Complexity	T2	0.03263	0.03908	0.00408	0.00022	0.00086	0.00165
	T3-F1	0.75014	0.00083	0.02750	0.00812	0.00182	0.06401
	T3-F2	0.44775	0.62944	0.66221	0.20595	0.32242	0.34583
Cognitive Complexity	T2	0.76520	0.00165	0.00140	0.00031	0.00037	0.00223
	T3-F1	0.47890	0.00525	0.03990	0.00938	0.00520	0.00140
	T3-F2	0.16069	0.13531	0.02289	0.80041	0.20608	0.37013
Halstead Length	T2	0.01077	0.00018	0.03263	0.01415	0.00018	0.00165
	T3-F1	0.34202	0.00741	0.01735	0.00741	0.00100	0.01735
	T3-F2	0.25983	0.13531	0.73028	0.00031	0.00037	0.00031
	T2	0.01228	0.00022	0.03656	0.01415	0.00018	0.00165

Continued on next page

Table 5.4: *Wilcoxon Rank Sum test* results after applying the *Bonferroni Correction method* to check if the experimental groups from the preliminary experiment were symmetric. (Continued)

			<i>A1-A2</i>	<i>A1-A3</i>	<i>A2-A3</i>	<i>A1-B</i>	<i>A2-B</i>	<i>A3-B</i>
Halstead Volume	Vol-	T3-F1	0.60995	0.00741	0.01735	0.00741	0.01735	0.01735
		T3-F2	0.17492	0.44831	0.42122	0.00073	0.00086	0.02628
		T2	0.01077	0.00018	0.01396	0.01752	0.00022	0.00304
Halstead Effort		T3-F1	0.54364	0.00741	0.00741	0.00741	0.01735	0.32315
		T3-F2	0.24095	0.83617	0.17515	0.01224	0.22312	0.01075
		T2	0.16109	0.69613	0.05085	0.00037	0.00120	0.03234
Halstead Difficulty		T3-F1	0.98056	0.01849	0.00717	0.00741	0.01735	0.01735
		T3-F2	0.50485	0.53495	0.26015	0.03254	0.01073	0.02029
		T2	0.01077	0.00744	0.06653	0.01752	0.00022	0.02156
Halstead Time		T3-F1	0.54363	0.00000	0.00000	0.00741	0.01735	0.00078
		T3-F2	0.24095	0.00000	0.00248	0.01224	0.22312	0.00743
		T2	0.02907	0.00353	0.59722	0.03908	0.00062	0.00062
Maintainability		T3-F1	0.23391	0.00124	0.03256	0.00011	0.00010	0.00010
		T3-F2	0.06942	0.10281	0.80044	0.00022	0.00012	0.04784
		T2	0.03428	0.69537	0.04754	0.00027	0.00171	0.01482
#Refactorings		T3-F1	0.38255	0.35313	0.90061	0.00156	0.00997	0.00997
		T3-F2	0.11652	0.07990	0.20527	0.00020	0.00017	0.02493
		T2	0.00000	0.00000	0.00000	0.00000	0.00000	0.67497
Time Elapsed		T3-F1	0.04312	0.00000	0.00000	0.00000	0.00000	0.00000
		T3-F2	0.00000	0.00000	0.00000	0.00000	0.00000	0.00022

Table 5.5: Null and alternative hypotheses used on the *Permutation tests* for the preliminary experiment.

One-Tailed - Lower Direction	<i>Lines of Code, Cyclomatic Complexity, Cognitive Complexity, Halstead Metrics, Time Elapsed</i>

Continued on next page

Table 5.5: Null and alternative hypotheses used on the *Permutation tests* for the preliminary experiment. (Continued)

Null Hypothesis	Metrics from the first group are higher than the ones from the second group being compared.
Alternative Hypothesis	Metrics from the first group are lower than the ones from the second group being compared.
One-Tailed - Higher Direction	<i>Maintainability</i>
Null Hypothesis	Metrics from the first group are lower than the ones from the second group being compared.
Alternative Hypothesis	Metrics from the first group are higher than the ones from the second group being compared.
Two-Tailed	<i>Number of Refactorings,</i>
Null Hypothesis	Metrics from the first group are not statistically different from the second group being compared.
Alternative Hypothesis	Metrics from the first group are statistically different from the second group being compared.

Just as it happened with the *Wilcoxon Rank Sum test* (cf. Table 5.4, p. 135), we also applied the *Bonferroni Correction method* to the results achieved through the *Permutation test*. Table 5.6, (p. 138) lists the p-values measured after applying the *Bonferroni method*.

As can be checked in Table 5.6, (p. 138), the majority of p-values for the evaluated metrics, when comparing the experimental groups for each programming task, were above 0.05. However, there were some outliers (painted in red). Most outliers happen when comparing groups A1, A2, and A3 between them. We expected that there could be some outliers when comparing these groups since they used a similar approach. The difference between the tools was in the number and manner in which the refactoring candidates were suggested to the experiment participants. There were also some cases in which we compared the groups that used a refactoring tool with the ones that did not (group B), in which the p-values were not lower than 0.05. These mainly happened on several metrics for task *Task 3 File 2*.

Despite unexpected p-values, the majority were below 0.05. This indicates that the experimental groups were statistically different. By further analyzing the code quality metrics, we also checked that groups A1, A2, and A3 differed significantly from group B. Therefore, we could reject the null hypotheses for the *Permutation test* and positively answer the research questions RQ1 and RQ2 (cf. Section 5.4, p. 184).

Besides, most p-values obtained from comparing the time required to apply a refactoring between all groups were lower than 0.05. The only exceptions were found when comparing groups A1, A2, and A3 with each other in task 2, and group A3 with group B in the first file of task 3. As most p-values were lower than 0.05, we concluded that groups A1, A2, A3, and B were statistically

distinct. Therefore, we could reject the null hypothesis for research question RQ4 (*cf.* Section 5.4, p. 184).

Similar findings were observed for the average number of refactorings applied to the code. In most cases, p-values obtained from comparing groups *A1*, *A2*, and *A3* were higher than 0.05. This may be because they were using similar versions of the same approach. However, the remaining p-values obtained from comparing groups *A1*, *A2*, and *A3* with group *B* were less than 0.05, indicating that they were statistically different. These results complement the data that helped us reject the null hypothesis for research question RQ2 (*cf.* Section 5.4, p. 184).

Table 5.6: *Permutation tests* performed to compare the experimental groups of the preliminary experiment after applying the *Bonferroni Correction method*.

		<i>A1-A2</i>	<i>A1-A3</i>	<i>A2-A3</i>	<i>A1-B</i>	<i>A2-B</i>	<i>A3-B</i>
Lines of Code	T2	0.02597	0.28172	0.97103	0.00200	0.00100	0.00100
	T3-F1	0.15884	0.35365	0.77822	0.00100	0.01898	0.00200
	T3-F2	0.16484	0.06094	0.28272	0.00100	0.00100	0.00100
Cyclomatic Complexity	T2	0.02398	0.00100	0.00100	0.00100	0.00100	0.00200
	T3-F1	0.16683	0.00100	0.00100	0.99800	0.99900	1.00000
	T3-F2	0.37562	0.58741	0.74426	0.30969	0.42058	0.28172
Cognitive Complexity	T2	0.28771	0.00100	0.00100	0.00100	0.00100	0.00100
	T3-F1	0.19880	0.00200	0.03098	0.49750	0.50050	0.51748
	T3-F2	0.28372	0.93407	0.99401	0.80520	0.93007	0.18581
Halstead Length	T2	0.00799	0.00100	0.00100	0.00100	0.00100	0.00100
	T3-F1	0.17682	0.00100	0.00100	0.00100	0.00100	0.00100
	T3-F2	0.22977	0.13487	0.44056	0.00100	0.00100	0.00100
Halstead Volume	T2	0.00599	0.00100	0.01499	0.00100	0.00100	0.00100
	T3-F1	0.17882	0.00100	0.00100	0.00100	0.00100	0.00100
	T3-F2	0.20080	0.37263	0.78522	0.00100	0.00100	0.00100
Halstead Effort	T2	0.01798	0.00100	0.00500	0.00100	0.00100	0.00100
	T3-F1	0.22378	0.00100	0.00100	0.00100	0.00100	0.23377

Continued on next page

Table 5.6: *Permutation tests* performed to compare the experimental groups of the preliminary experiment after applying the *Bonferroni Correction method*. (Continued)

		<i>A1-A2</i>	<i>A1-A3</i>	<i>A2-A3</i>	<i>A1-B</i>	<i>A2-B</i>	<i>A3-B</i>
	T3-F2	0.15485	0.35265	0.80020	0.00400	0.09291	0.01099
	T2	0.08791	0.45954	0.96104	0.00100	0.00100	0.00100
Halstead Diffi-	T3-F1	0.79820	0.00799	0.00300	0.00100	0.00100	0.00100
culty	T3-F2	0.50485	0.76723	0.90210	0.99700	1.00000	0.92508
	T2	0.02098	0.00100	0.01399	0.00100	0.00100	0.08591
Halstead Time	T3-F1	0.23077	0.00100	0.00100	0.00100	0.00100	0.99900
	T3-F2	0.13986	0.00100	0.00100	0.00599	0.08591	1.00000
	T2	0.01199	0.00500	0.49451	0.00100	0.00100	0.00100
Maintainability	T3-F1	0.16883	0.00100	0.00200	0.00100	0.00100	0.00100
	T3-F2	0.16283	0.10589	0.39960	0.00100	0.00100	0.00100
	T2	0.02398	0.55944	0.04196	0.00200	0.00400	0.00200
#Refactorings	T3-F1	0.35764	0.62138	0.47752	0.00200	0.00999	0.00799
	T3-F2	0.43956	0.14386	0.37163	0.00200	0.00200	0.00200
	T2	0.05095	0.39461	0.94705	0.00400	0.08292	0.00200
Time Elapsed	T3-F1	0.00100	0.00100	0.00100	0.00100	0.00100	0.00100
	T3-F2	0.00100	0.00100	0.00100	0.00100	0.00100	0.99900

5.1.4.6 Usability

At the end of this experience, the participants had to answer a set of *11 usability questions* to point out aspects that deserve attention regarding the usability of the tool developed in this project. Usually, a SUS questionnaire has *10 questions* [Bro13]. However, we decided to add a new one to evaluate if using the tool on each project used in the experiment was hard. The questions are the following:

1. If they would use the tool frequently in the future;
2. If they found the tool unnecessarily complex;
3. If they found the tool easy to use;
4. If they would need more technical support to understand and use the tool;
5. If they thought that all the functionalities of the tool were well integrated;

6. If they were inconsistencies in the tool;
7. If they learned how to use the tool quickly;
8. If they found it difficult to use the tool on the correspondent project;
9. If they found the tool very cumbersome to use;
10. If they felt confident using the tool;
11. If they needed to learn a lot of things before using the tool.

Notice that we only asked these questions to the participants of experimental groups *A1* and *A2* since they were the ones using our tool as implemented it (*cf.* Section 4.3, p. 116).

In Figure 5.6, (p. 140), we show the percentages of each type of answer given to each of the 11 usability questions. To make the results easier to read, we used colors somewhere between red and green. Thus, values close to 0% are displayed in *red*. Values close to 100% are represented with a color close to *green*.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Strongly Disagree	0%	82%	0%	64%	0%	75%	0%	96%	96%	0%	79%
Disagree	0%	18%	0%	36%	0%	25%	0%	4%	4%	0%	21%
Neutral	0%	0%	0%	0%	7%	0%	0%	0%	0%	11%	0%
Agree	61%	0%	29%	0%	68%	0%	39%	0%	0%	75%	0%
Strongly Agree	39%	0%	71%	0%	25%	0%	61%	0%	0%	14%	0%

Figure 5.6: Answers to our SUS Questionnaire on the first experiment.

As can be checked in Figure 5.6, (p. 140), most participants answered very positively to each question. However, a small percentage of participants gave neutral answers to questions related to whether the functionalities of our tool were well integrated and if they felt confident while using it. Therefore, we concluded that our tool has good usability. However, there is still space to improve it.

We also decided to ask other usability questions to the participants, mainly to evaluate more specific visual aspects of our tool and their goals. These questions are the following:

1. If they found the tool useful;
2. If displayed suggestions helped understand where they should improve their code;
3. If the refactoring suggested and applied helped create better code;
4. If the approach helped developers achieve better code, faster;
5. If the color scheme used to represent each refactoring was adequate;
6. If the placement of the interface was adequate;

7. If the placement of the refactoring suggestions was adequate;
8. If the number of suggestions was adequate;
9. If they believed that our approach helped reduce the effort needed to create better code;
10. If they believed that our approach would make it easier for developers to verify if their code has quality or not;
11. If they thought that using our approach would make it more difficult to maintain large and complex software systems.

The answers to these questions are synthesized in Figure 5.7, (p. 141). By analyzing their results, we verified that most participants agreed that our tool helped create better code, faster. Besides, it helped inform the participants of their code flaws and maintain large and complex code. Also, the participants seemed to like the color scheme used to point out the refactoring candidates and their placement on the IDE.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Strongly Disagree	0%	0%	0%	0%	0%	0%	0%	18%	0%	0%	75%
Disagree	0%	0%	0%	0%	0%	0%	0%	25%	0%	0%	25%
Neutral	0%	0%	4%	4%	0%	0%	0%	14%	0%	4%	0%
Agree	43%	57%	57%	64%	18%	29%	32%	18%	75%	71%	0%
Strongly Agree	57%	43%	39%	32%	82%	71%	68%	25%	25%	25%	0%

Figure 5.7: Answers to other usability questions on our tool on the first experiment.

5.1.4.7 Reliability

Based on some of the questions made on our SUS Questionnaire [Bro13], we were able to measure the average *Perceived Ease of Use*, *Perceived Usefulness*, and *Intention to Use* of the *Live Refactoring Environment* by groups A1 and A2.

The SUS questions used to measure these metrics were the ones listed in Table 5.7, (p. 141).

Table 5.7: Usability questions used to assess reliability.

Perceived Ease of Use

1. I found the tool unnecessarily complex
 2. I thought the tool was easy to use
 3. I think that I would need the support of a technical person to be able to use this tool
 4. I would imagine that most people would learn to use this tool very quickly
-

Continued on next page

Table 5.7: Usability questions used to assess reliability. (Continued)

5. I thought it difficult to use the tool on the correspondent project
6. I found the tool very cumbersome to use
7. I felt very confident using the tool
8. I needed to learn a lot of things before I could get going with this tool
<i>Perceived Usefulness</i>
1. I believe that the tool reduces the effort needed to create better software systems
2. I believe that the tool will make it easier for developers to verify if their code has quality or not
3. I found the tool useful
4. I believe that using the tool will make it more difficult to maintain large and complex software systems
<i>Intention of Use</i>
1. I think that I would like to use this tool frequently

To assess the reliability metrics, we measured their *Cronbach's alphas*. The literature suggests that alpha values of 0.7 or higher are considered as acceptable [Moo03]. Notice that we only assessed these metrics on experimental groups A1 and A2 because they were the ones that used a tool containing our colored visual methodology as we implemented it (cf. Section 4.3, p. 116).

Table 5.8, (p. 142) shows that groups A1 and A2 understood that our tool is handy and showed the intention of using it in the future. However, we cannot confirm that they perceived correctly how to use it since the Cronbach's alpha for the *Perceived Ease of Use* for both groups was lower than 0.7. Although these values were very close to the standard 0.7, they suggest that the participants did not fully understand how easy it is to use our tool. Nonetheless, we discovered that the participants of group A1 had a stronger belief than those of group A2.

Therefore, we can only claim that our experiment participants comprehended our approach's usefulness and aim to use it in the future.

Table 5.8: *Cronbach's alphas* measured between Group A1 and A2.

	<i>Group A1</i>	<i>Group A2</i>
Perceived Ease of Use	0.60	0.55
Perceived Usefulness	0.84	0.76
Intention of Use	0.99	0.99

5.1.4.8 Other Aspects

Besides all the usability questions (*cf.* Section 5.1.4.6, p. 139), we also asked another question to assess the participants' satisfaction with this experiment.

As shown in Figure 5.8, (p. 143), there were no negative opinions on the experiment itself. In fact, 43 participants stated that they were satisfied with the experiment, being that 12 of them were *strongly satisfied*. Only 13 participants showed a neutral opinion.

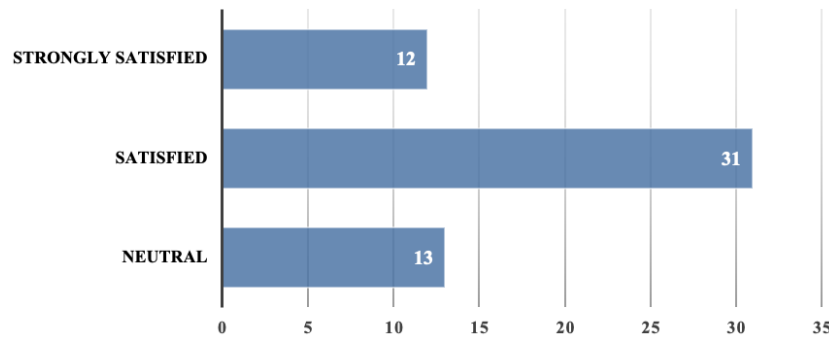


Figure 5.8: Participants satisfaction on the first experiment.

5.2 Live Refactoring Experiment

To validate our hypothesis and answer our research questions, we conducted a final empirical experiment after finishing the development of our solution. This experiment did not involve any predefined programming or refactoring tasks, nor did any previously defined experimental groups. Instead, we only identified our experimental groups when we started analyzing the data collected from the experiment itself (*cf.* Section 5.2.3.1, p. 145). Also, the only goal for the participants was to implement a specific project (*cf.* Section 5.2.1, p. 143).

This experiment only focused on using the developed *Live Refactoring Environment*. Each participant had access to the tool, its user guide (*cf.* Section B, p. 233), a small description of the experiment (*cf.* Section C.4.1, p. 298), and a short video tutorial via Google Drive. After developing their projects, the participants were asked to fill out a quick Google Form containing usability and reliability questions (*cf.* Section C.4.2, p. 309).

5.2.1 Project and Participants

The participants of our empirical experiment were students from programming Bachelor classes learning Java and its good programming practices. They worked in groups of two or three elements to develop a small Java integrator project that should be implemented in four weeks.

These students knew the experimental goals and that their participation was entirely voluntary. Besides, those who chose to participate were informed that their involvement in our study would not impact their final class grades.

The participants had to apply the SOLID principles to a specific project, use an MVC architectural pattern, and apply several design patterns to improve their code [Mar03, Mar15, Fow15]. Also, one of the main goals of this project was to have the students identify code smells and apply the correct refactorings to mitigate them, improve non-functional attributes, and increase code quality. They all used the IntelliJ Community Edition 2021 with Java 16 to develop their projects.

We did not combine the project groups into different experimental groups when we started the experiment. This is because we aimed to label them based on whether they used our *Live Refactoring Environment* or not, as well as on the week they started using the tool. With this type of division, we aimed to check if using a *Live Refactoring Environment* helps converge to better code, faster. The process to label the experimental groups is detailed in Section 5.2.3.1, (p. 145).

Notice that none of the data collected from this experiment included the participants' identities to reduce any possibility of biased results.

5.2.2 Outcomes

From this experiment, we gathered five different outcomes with specific aims. Two of the outcomes, namely *outcomes B* and *C*, were gathered through a JSON file saved in our Firebase database. With *outcomes B*, *C*, *D*, and *E*, we were able to answer the research questions that we listed in Section 3.4, (p. 82).

Outcome A: With this outcome, we aimed to collect information on the experiment's participants. We asked them to fill out a Google Form with questions to help us evaluate their programming proficiency, Java coding expertise, or familiarity with refactoring tools. These characteristics were evaluated on a scale from 1 to 5.

Outcome B: With this outcome, we focused on analyzing the evolution of specific code quality metrics over the four-week project development time. To compare the values between different experimental groups, we assessed metrics like the number of lines of code, cyclomatic complexity, Halstead metrics, or maintainability degree. Through this outcome, we sought to answer research questions RQ2, RQ3, and RQ4.

Outcome C: With this outcome, we wanted to check the number of refactorings and undone actions performed by each experimental group that used our tool. Basically, our goal was to verify if the participants trusted the refactorings recommended and executed by our tool and if they understood their relevance to help them achieve better code quality their code quality. Through this outcome, we aimed to answer research question RQ2.

Outcome D: With this outcome, we aimed to check the participants' projects' final grades and to verify if our tool helped them converge to better code quality and to code more readable, adaptable, and maintainable. This is because we assumed that better code would be better graded. By using the analysis of the data collected in this outcome, we sought to answer research question RQ3.

Outcome E: With this outcome, we aimed to assess the usability of our tool by applying a short questionnaire based on the System Usability Scale (SUS) [Bro13]. With some of the answers, we also aimed to assess specific reliability metrics like the perceived ease of use, usefulness, and intention to use our tool in the future. Through this outcome, we wanted to answer research question RQ1.

5.2.3 Results and Discussion

After completing this experiment, we had to analyze and discuss the data collected from it. Thus, in this section, we summarize all results achieved from the experiment, interpret and discuss them to verify if we were able to validate our hypothesis (*cf.* Section 3.3, p. 82), and answer our research questions (*cf.* Section 3.4, p. 82).

5.2.3.1 Experimental Groups

After finishing the experiment, we divided the participants into multiple experimental groups. Notice that, as mentioned in Section 5.2.1, (p. 143), the participants were already combined into different groups to implement their projects.

Initially, we divided the participants into two different groups: group (A) included all the project groups that used our *Live Refactoring Environment*; group (B) comprised all the groups who did not use it. Group (B) was labeled as group *No Tool*.

Then, we divided group (A) into more specific subgroups. As detailed in Section 5.2.1, (p. 143), the project should have been developed in four weeks. So, we used the development week, during which the project groups started using our tool. Thus, we labeled four subgroups: group *Week 1* included all the project groups that started using our *Live Refactoring Environment* since the beginning of their project development; group *Week 2* comprised the project groups that started using the approach since the second development week; group *Week 3* included the participants that started applying our tool to their project at the third week; group *Week 4* represented all the participants that only used the tool on the last week of project development.

Once we accurately labeled the experimental groups, we sought to assess whether participants' backgrounds impacted the final code quality results. This was done primarily to ensure that any observed variations between groups were solely due to the experiment and not influenced by differences in group characteristics, thereby minimizing potential biases.

Firstly, we started by verifying if the data collected from each group followed a normal distribution. For that, we used the *Shapiro-Wilk test* [SW65]. For this test, we based our analysis on the null and alternative hypothesis listed in Table 5.10, (p. 146).

Then, we analyzed the *p-values* returned by the *Shapiro-Wilk test* [SW65]. As can be seen in Table 5.9, (p. 146), there were cases in which the *p-values* were lower or equal to 0.05 (*p-values* painted in blue). This means that we could not refute the null hypothesis of this test. Therefore, we concluded that our results were not normally distributed.

Table 5.9: *Shapiro-Wilk test* on the data collected on every experimental group.

	<i>Week 1</i>	<i>Week 2</i>	<i>Week 3</i>	<i>Week 4</i>	<i>No Tool</i>
Lines of Code	0.87862	0.87220	0.07158	0.12230	0.00018
Cyclomatic Complexity	0.03420	0.32257	0.00239	0.00572	0.00011
Cognitive Complexity	0.01701	0.05970	0.01126	0.00228	0.00001
Lack of Cohesion	0.45740	0.00365	0.00047	0.00022	0.00009
Halstead Length	0.65514	0.46509	0.05497	0.00894	0.00020
Halstead Volume	0.94772	0.29908	0.03882	0.00506	0.00013
Halstead Effort	0.44086	0.71152	0.20900	0.00189	0.00007
Halstead Difficulty	0.94646	0.71152	0.20900	0.00189	0.04772
Halstead Time	0.16184	0.46391	0.00101	0.00006	0.00004
Final Grade	0.13349	0.13025	0.21633	0.45018	0.00741
AVG GPA	0.08544	0.39305	0.04867	0.45585	0.00426

After that, we decided to assess the relationship between the average GPA of each project group that comprised the different experimental groups (on a scale from 0 to 20) and the final code quality and project grade. We used the *Spearman Correlation test* [CD10] to check if these values were correlated. For this test, we used the hypothesis of Table 5.10, (p. 146).

Table 5.10: Null and alternative hypotheses used on several tests used to characterize our experimental groups and their data.

<i>Shapiro-Wilk Test</i>	
Null Hypothesis	Data follows a normal distribution.
Alternative Hypothesis	Data does not follow a normal distribution.
<i>Spearman Correlation Test (Code Quality)</i>	
Null Hypothesis	Final code quality is not correlated with the group's GPA.
Alternative Hypothesis	Final code quality is correlated with the group's GPA.
<i>Spearman Correlation Test (Final Project Grade)</i>	
Null Hypothesis	Final project grade is not correlated with the group's GPA.
Alternative Hypothesis	Final project grade is correlated with the group's GPA.

We started by verifying if there was any relationship between GPA and final project grade by checking the *p-values* obtained from applying the *Spearman test*. Table 5.11, (p. 147) presents the *p-values* when comparing all groups without any restrictions. When analyzing those values, we verified multiple cases had *p-values* lower than 0.05. This means there is a correlation between the

GPA and that respective metric.

Then, we decided to restrict the groups by GPA. Table 5.12, (p. 147) present the *p-values* when comparing only the groups whose students have an average GPA of 10, 11, 12, 13, and 14. As can be checked in these tables, none of the *Spearman tests* retrieved only *p-values* higher than 0.05, to show that the results for each experimental group were not related to their GPAs.

Table 5.11: *Spearman Coefficient test* to verify if the final code quality and project grade were related to the groups' GPA when comparing all groups.

	<i>Week 1</i>	<i>Week 2</i>	<i>Week 3</i>	<i>Week 4</i>	<i>No Tool</i>
Lines of Code	0.74706	0.26570	0.13091	0.69403	0.48896
Cyclomatic Complexity	0.11761	0.49237	0.87441	0.98331	0.55695
Cognitive Complexity	0.04052	0.77406	0.40632	0.99583	0.65179
Lack of Cohesion	0.32172	0.51223	0.17539	0.27149	0.18656
Halstead Length	0.18812	0.20800	0.00818	0.88121	0.33690
Halstead Volume	0.18812	0.46848	0.00858	0.82985	0.44010
Halstead Effort	0.28476	0.32872	0.11303	0.60597	0.44010
Halstead Difficulty	0.28476	0.07240	0.01250	0.42202	0.36045
Halstead Time	0.28476	0.32872	0.11303	0.60597	0.44010
Maintainability	0.74706	0.26570	0.11589	0.77195	0.90121
Final Grade	0.74706	0.07240	0.00580	0.00181	0.00000

Table 5.12: *Spearman Coefficient test* to verify if the final code quality and project grade were related to the groups' GPA when restricting the groups' GPA.

<i>GPA >= 10</i>					
	<i>Week 1</i>	<i>Week 2</i>	<i>Week 3</i>	<i>Week 4</i>	<i>No Tool</i>
Lines of Code	0.74706	0.26570	0.13091	0.69403	0.48896
Cyclomatic Complexity	0.11761	0.49237	0.87441	0.98331	0.55695
Cognitive Complexity	0.04052	0.77406	0.40632	0.99583	0.65179
Lack of Cohesion	0.32172	0.51223	0.17539	0.27149	0.18660
Halstead Length	0.18812	0.20800	0.00818	0.88121	0.33690
Halstead Volume	0.18812	0.46848	0.00858	0.82985	0.44010
Halstead Effort	0.28476	0.32872	0.11303	0.60597	0.44010

Continued on next page

Table 5.12: *Spearman Coefficient test* to verify if the final code quality and project grade were related to the groups' GPA when restricting the groups' GPA.
(Continued)

Halstead Difficulty	0.28476	0.07240	0.01250	0.42202	0.36045
Halstead Time	0.28476	0.32872	0.11303	0.60597	0.44010
Maintainability	0.74706	0.26570	0.11589	0.77195	0.90121
Final Grade	0.74706	0.07240	0.00580	0.00181	0.00000
<i>GPA >= 11</i>					
	<i>Week 1</i>	<i>Week 2</i>	<i>Week 3</i>	<i>Week 4</i>	<i>No Tool</i>
Lines of Code	0.74706	0.26570	0.13091	0.69403	0.48896
Cyclomatic Complexity	0.11761	0.49237	0.87441	0.98331	0.55695
Cognitive Complexity	0.04052	0.77406	0.40632	0.99583	0.65179
Lack of Cohesion	0.32172	0.51223	0.17539	0.27149	0.18660
Halstead Length	0.18812	0.20800	0.00818	0.88121	0.33690
Halstead Volume	0.18812	0.46848	0.00858	0.82985	0.44010
Halstead Effort	0.28476	0.32872	0.11303	0.60597	0.44010
Halstead Difficulty	0.28476	0.07240	0.01250	0.42202	0.36045
Halstead Time	0.28476	0.32872	0.11303	0.60597	0.44010
Maintainability	0.74706	0.26570	0.11589	0.77195	0.90121
Final Grade	0.74706	0.07240	0.00580	0.00181	0.00000
<i>GPA >= 12</i>					
	<i>Week 1</i>	<i>Week 2</i>	<i>Week 3</i>	<i>Week 4</i>	<i>No Tool</i>
Lines of Code	0.74706	0.26570	0.13091	0.31039	0.02305
Cyclomatic Complexity	0.11761	0.4924	0.87441	0.77902	0.66197
Cognitive Complexity	0.04052	0.77406	0.40632	0.75635	0.59872
Lack of Cohesion	0.32172	0.51223	0.17539	0.54104	0.13134
Halstead Length	0.18812	0.20800	0.00818	0.60967	0.05809
Halstead Volume	0.18812	0.46848	0.00858	0.66434	0.08027
Halstead Effort	0.28476	0.32872	0.11303	0.88820	0.08027
Halstead Difficulty	0.28476	0.07240	0.01250	0.85399	0.05143
Halstead Time	0.28476	0.32872	0.11303	0.88820	0.08027

Continued on next page

Table 5.12: *Spearman Coefficient test* to verify if the final code quality and project grade were related to the groups' GPA when restricting the groups' GPA.
(Continued)

Maintainability	0.74706	0.26570	0.11589	0.77834	0.58389
Final Grade	0.74706	0.07240	0.00580	0.00674	0.00090
GPA ≥ 13					
	Week 1	Week 2	Week 3	Week 4	No Tool
Lines of Code	0.74706	0.80000	0.57391	1.00000	0.64381
Cyclomatic Complexity	0.11761	0.68377	0.53587	0.59608	0.166020
Cognitive Complexity	0.04052	0.26214	0.49312	0.59608	0.09043
Lack of Cohesion	0.32172	0.22540	0.52906	0.41517	0.89376
Halstead Length	0.18812	0.00000	0.04670	0.64880	0.26218
Halstead Volume	0.18812	0.00000	0.02887	0.54194	0.42446
Halstead Effort	0.28476	0.20000	0.16015	0.37857	0.42446
Halstead Difficulty	0.28476	0.00000	0.20488	0.40298	0.59915
Halstead Time	0.28476	0.20000	0.16015	0.37857	0.42446
Maintainability	0.74706	0.80000	0.85254	0.69668	0.76725
Final Grade	0.74706	0.60000	0.07095	0.11211	0.00534
GPA ≥ 14					
	Week 1	Week 2	Week 3	Week 4	No Tool
Lines of Code	0.60000	0.80000	0.82872	0.80218	0.46951
Cyclomatic Complexity	0.22540	0.68377	0.29472	0.92541	0.15904
Cognitive Complexity	0.22540	0.26214	0.82178	0.92541	0.36011
Lack of Cohesion	0.26214	0.22540	0.75392	0.58196	0.20909
Halstead Length	0.60000	0.00000	0.16152	0.40470	0.85936
Halstead Volume	0.60000	0.00000	0.10759	0.40470	0.89874
Halstead Effort	0.80000	0.20000	0.32823	0.51089	0.89874
Halstead Difficulty	0.80000	0.00000	0.54355	0.46569	0.74538
Halstead Time	0.80000	0.20000	0.32823	0.51089	0.89874
Maintainability	0.60000	0.80000	0.48878	0.90736	0.29976
Final Grade	0.20000	0.60000	0.17394	0.21449	0.14697

However, when only comparing data from groups with GPA higher or equal to 14, we verified

that there were only three cases in which we were not able to reject the null hypothesis for the *Spearman test* related to code quality (cf. Table 5.10, p. 146). These cases happened for the *Halstead Length*, *Volume*, and *Difficulty* for group *Week 2*. Their *p-values* are equal to 0.0, meaning there is no monotonic relationship between the data being compared.

Thus, and since we could show that there were no correlations between the groups' GPAs and the final project grade when we used the results from groups with average GPAs higher or equal to 14, we decided only to contemplate the data collected from those groups (also as an attempt to eliminate possible biased results).

In Table 5.13, (p. 150), we summarize the different experimental groups and the number of groups included before and after removing the biased ones. As can be concluded from this table, most experiment participants did not use our approach. However, we also verified that the number of groups using our approach improved over time. This may indicate that some participants realized our tool could help them improve their complex and large code to achieve better programming solutions faster than manually refactoring the code.

Table 5.13: Number of participants and their GPA per experimental group.

<i>Number of Groups</i>					
	<i>Week 1</i>	<i>Week 2</i>	<i>Week 3</i>	<i>Week 4</i>	<i>No Tool</i>
All	5	6	16	18	86
GPA ≥ 14	4	4	10	10	22
<i>Average GPA</i>					
	<i>Week 1</i>	<i>Week 2</i>	<i>Week 3</i>	<i>Week 4</i>	<i>No Tool</i>
All	14.92	14.71	14.40	14.54	13.17
GPA ≥ 14	15.37	15.71	15.42	15.88	15.28

5.2.3.2 Participants Characterization

As mentioned in (cf. Section 5.2.3.1, p. 145), our experiment participants were students pursuing their bachelor's degrees. As happened in the first experiment (cf. Section 5.1, p. 124), and to gain further insight into their programming skills, we asked them a few questions on their level of experience in programming, their proficiency in using Java, or familiarity with IntelliJ IDE.

Figure 5.9, (p. 152) shows the charts summarizing all the answers for each question made. Note that each question was answered on a scale of 1 to 5, where 1 indicated a *Very Poor Experience* and 5 indicated a *Very Good Experience*.

- **Programming experience:** For the first question, in which we tried to evaluate the participants' overall programming experience (cf. Figure 5.9a, p. 152), we verified that 11.11% said having very good (3.70%) or good (7.41%) programming experience. In the other hand,

48.15% mentioned to have very poor (11.11%) or poor (37.04%) programming experience. The other 40.74% of the participants gave a neutral answers.

- **Experience programming with Java:** For the second question, in which we wanted to verify if the participants knew how to program in Java (cf. Figure 5.9b, p. 152), we checked that 3.70% of them said to have good knowledge using this programming language. However, 25.93% said to have very poor Java programming experience, and 55.56% said to have a poor experience. Only 14.81% gave a neutral answer.
- **Experience with IntelliJ IDE:** For the fourth question, in which we aimed to check if the participants already knew how to use IntelliJ IDE (cf. Figure 5.9c, p. 152), we verified that only 3.70% mentioned to have good knowledge on how to use IntelliJ IDE. On the other hand, 40.74% said to have a poor and 29.63% a very poor experience. 25.93% gave a neutral answer.
- **Experience using refactoring tools:** For the fifth question, in which we aimed to check if the participants already used a refactoring tool or not (cf. Figure 5.9d, p. 152), we verified that 77.78% of them never used one. 14.81% had poor experience and 3.70% gave a neutral answer. Only 3.70% said to have a good experience using refactoring tools.

5.2.3.3 Number of Refactorings and Undo Operations

Outcome B also allowed us to quantify the *number of refactorings* and *undone actions* executed by the experimental groups that used our *Live Refactoring Environment* (groups *Week 1*, *Week 2*, *Week 3* and *Week 4*). Table 5.14, (p. 152) summarizes the number of refactorings and undos performed, classified by refactoring type and experimental group.

As we expected, the participants of the group *Week 1* applied more refactorings than the other groups. Overall, over 57% of the applied refactorings were *Extract Variables*. This high percentage can be explained by the simplicity of this refactoring type and the tendency of developers to create complex variable expressions. However, this refactoring may increase the total number of variables and lines of code, which could be one of the reasons for some of the fluctuations observed in specific charts in from Figure 5.10, (p. 155).

Extract Method was the second most applied refactoring type (e.g., 24% of the refactorings were *Extract Methods*). This indicates that over 80% of the refactorings executed were either *Extract Variables* or *Extract Methods*. Moreover, as shown in Table 5.14, (p. 152), none of the experimental groups applied any *Extract Class* or *String Comparison* refactorings. The absence of *Extract Class* refactorings may have negatively impacted the cohesiveness of the final code of the experimental groups that used our tool.

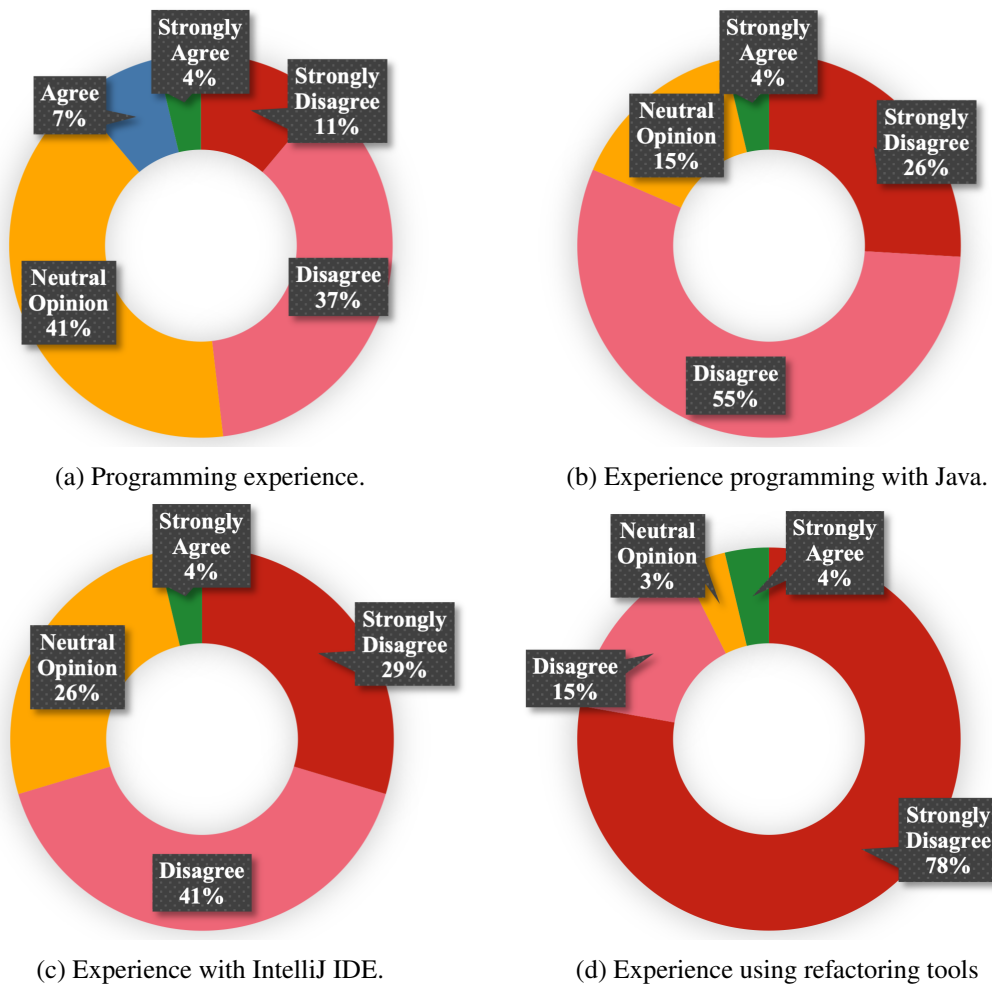


Figure 5.9: Characterization of the participants from the final experiment.

Table 5.14: Number of refactorings and undo operations applied by type and experimental group.

<i>Refactorings</i>								
	<i>EM</i>	<i>EC</i>	<i>EV</i>	<i>MM</i>	<i>IPO</i>	<i>ID</i>	<i>SC</i>	<i>Total</i>
Week 1	124	0	337	62	1	14	0	538
Week 2	20	0	67	7	0	3	0	97
Week 3	62	0	136	47	0	18	0	263
Week 4	109	0	226	71	2	20	0	428
Total	315	0	766	187	3	55	0	1326
<i>Undos</i>								
	<i>EM</i>	<i>EC</i>	<i>EV</i>	<i>MM</i>	<i>IPO</i>	<i>ID</i>	<i>SC</i>	<i>Total</i>

Continued on next page

Table 5.14: Number of refactorings and undo operations applied by type and experimental group. (Continued)

Week 1	47	-	91	6	0	3	-	147
Week 2	8	-	42	2	-	0	-	52
Week 3	17	-	77	13	-	5	-	112
Week 4	17	-	94	6	0	6	-	123
Total	89	-	304	27	0	14	-	434

Group *Week 1* also performed more undo operations. However, these actions only represented approximately 27% of the total refactorings applied to the code. They do not hold the record for the highest percentage of undos since they also executed more refactorings. In fact, group *Week 2* was the one with the highest percentage of undos, with more than 53% of their refactorings being undone, followed by group *Week 3* with over 42%, and finally group *Week 4* with almost 29% of undos. This high number of undone actions in this specific group could have caused the discrepancies found on the charts of Figure 5.10, (p. 155). Moreover, we confirmed that none of the groups undone of the *Introduce Parameter Object* refactorings they applied.

Based on analyzing the number of refactorings and undone actions (cf. Table 5.14, p. 152), and by complementing these data with one described in Section 5.2.3.4, (p. 153), we can conclude that the members of the group *Week 1* achieved better programming solutions by executing more refactorings and undoing a lower percentage of them, in comparison to the other groups.

5.2.3.4 Code Quality

Through Outcome B (cf. Section 5.2.2, p. 144), we assessed the impact on the code quality by using or not our *Live Refactoring Environment* during different development stages.

Figure 5.10, (p. 155) represent the evolution of several code quality metrics such as the number of lines of code, Halstead Metrics, or cyclomatic complexity. Notice that *Halstead Effort* and *Halstead Time*'s charts are the same since they are correlated and evolve in the same way.

As seen in Figure 5.10, (p. 155), most metrics for groups that explored our approach to develop their projects demonstrated considerable improvement over time. Besides, these groups produced better code by the end of the four-week development time.

Upon conducting a more extensive analysis, we have checked that participants from group *Week 1* converged to better programming solutions with more quality than the remaining groups. Their codes were comparatively shorter, less complex, easier to understand, and more maintainable. Besides, when comparing each metric over time, they were also the ones that produced better code in earlier development stages.

In contrast, the groups *Week 3* and *No Tool* were the ones that produced lower-quality code solutions. We anticipated that the participants who did not use our tool would converge to worse code. However, we did not expect those who began using our approach since the third week of

development would generate worse solutions than the ones from group *Week 4*. We believe this happened because of the high percentage of undos performed by *Week 3* compared to the one from *Week 4*. Probably, they undid some relevant refactorings or prioritized less relevant refactorings over others more suitable for their programming context. Moreover, as verified in Section 5.2.3.5, (p. 154), the group *Week 3* obtained the lowest final project grade. In contrast, the participants from group *Week 1* had the highest grades, as we expected.

Also, by analyzing the charts of Figure 5.10, (p. 155), it is evident that, at the same time, the groups using our tool since earlier development stages produced better code than the remaining groups. This underscores the notion that promptly refactoring code leads to improved code quality.

Therefore, we can conclude that a refactoring environment that assists and guides developers on what, how, and when to refactor their code helps them converge to better programming solutions with more quality and faster.

5.2.3.5 Project Grading

Our experiment participants were students from a Bachelor class who had to develop a Java project during a specific time frame (*cf.* Section 5.2.1, p. 143). Thus, when they ended up implementing their projects, they were evaluated by their respective professors.

This evaluation covered several parameters. The most relevant ones for our experiment are summarized in Table 5.15, (p. 156). Their professors evaluated the code itself, its complexity, and its architecture. For each one of these categories, they had a specific grade range. They used a scale from 0 to 2 to evaluate code and its architecture. They used a scale from 0 to 1 for code complexity. Then, they gave a final grade to each project on a scale from 0 to 20.

As shown in Table 5.15, (p. 156), on average, participants from group *Week 1* had better grades in each category. They also had the best final project grades. With this fact, we assume they benefited from using our *Live Refactoring Environment* since their project development began. Following them were the students from groups *Week 2* and *Week 4*.

Participants of group *Week 3* had the worst grades in each grading criteria, which reflected in also having the worst final grade. We believed that, and as mentioned in Section 5.2.3.4, (p. 153), they did not make good programming decisions based on the refactoring suggested by our tool. It is possible that they ignored some of the recommendations or prioritized less important refactorings over more relevant ones for their code.

Group *No Tool* had better grades than group *Week 3*. However, their grades were not as good as the ones of *Week 1*, *Week 2*, or *Week 4*.

Despite one of the groups that used our tool having lower grades than expected, we are confident that we were able to demonstrate that a *Live Refactoring Environment* is effective in helping developers converge to better programming solutions with better code quality. Also, with these results, we believe that we were able to show that using a refactoring suggester in earlier development stages leads to creating even better code.

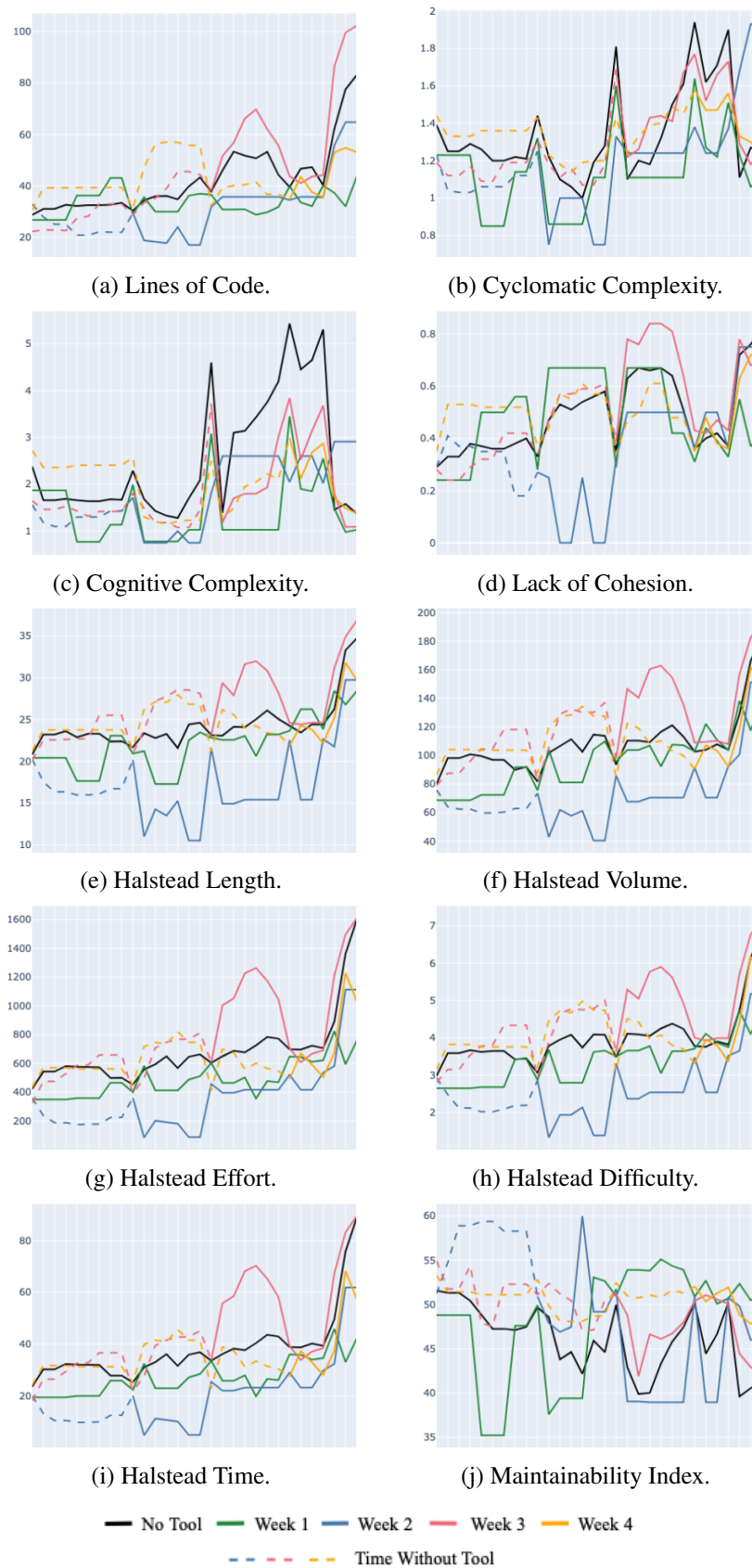


Figure 5.10: Metrics evolution over the four-week project development time.

Table 5.15: Final evaluation of the projects developed on our experiment per experimental group.

	<i>Code</i>	<i>Complexity</i>	<i>Architecture</i>	<i>Final Grade</i>
Week 1	1.84	0.94	1.91	18.56
Week 2	1.83	0.89	1.81	18.43
Week 3	1.63	0.71	1.59	16.50
Week 4	1.81	0.86	1.87	17.44
No Tool	1.65	0.88	1.70	16.80

5.2.3.6 Comparing Experimental Groups

To confirm our belief that live visual refactoring feedback, support, and guidance would benefit developers, we conducted some hypothesis tests on specific code quality metrics.

We started by checking if the data collected from the groups with GPAs higher or equal to 14 was normally distributed. We again applied the *Shapiro-Wilk test* [SW65], with the null hypothesis that the samples were normally distributed (*cf.* Table 5.17, p. 158). With this test, we verified that most of the p-values were higher than 0.05. This means that those samples were, in fact, normally distributed. On the other hand, there were some cases in which the p-value was lower than 0.05.

As seen in Section 5.2.3.1, (p. 145), when contemplating all groups with no GPA threshold, we verified that the results did not follow a normal distribution. However, we had to check if this fact was maintained after excluding the groups with GPAs lower than 14. To do so, we applied again the *Shapiro-Wilk test* [SW65] with the null hypothesis that the samples followed a normal distribution (*cf.* Table 5.17, p. 158). Then, we obtained the p-values shown in Table 5.16, (p. 156), where it can be checked that most of the p-values were greater than 0.05, indicating that the samples were normally distributed. However, there are some instances where the p-values were lower than 0.05 (painted in blue). This means that we could reject the null hypothesis for this test and show that our data did not follow a normal distribution.

Table 5.16: *Shapiro-Wilk test* results when the GPA is higher or equal to 14.

	<i>Week 1</i>	<i>Week 2</i>	<i>Week 3</i>	<i>Week 4</i>	<i>No Tool</i>
Lines of Code	0.80213	0.94504	0.52511	0.33135	0.016554
Cyclomatic Complexity	0.00124	0.60190	0.013281	0.00012	0.01231
Cognitive Complexity	0.00124	0.00657	0.00013	0.01045	0.00344
Lack of Cohesion	0.22194	0.00124	0.00005	0.00101	0.00251
Halstead Length	0.97051	0.76398	0.13075	0.03119	0.18095
Halstead Volume	0.93135	0.63375	0.07882	0.01654	0.24219

Continued on next page

Table 5.16: *Shapiro-Wilk test* results when the GPA is higher or equal to 14. (Continued)

	<i>Week 1</i>	<i>Week 2</i>	<i>Week 3</i>	<i>Week 4</i>	<i>No Tool</i>
Halstead Difficulty	0.81418	0.59007	0.52182	0.00966	0.19337
Halstead Effort	0.17810	0.88696	0.00481	0.00042	0.00042
Halstead Time	0.17813	0.88685	0.00481	0.00043	0.00042
Maintainability	0.47428	0.19706	0.32959	0.36633	0.00005
Project Grade	0.10112	0.97375	0.66967	0.94317	0.01215
AVG GPA	0.99933	0.63897	0.69250	0.69277	0.00676

Then, before applying any test to compare if our experimental groups were statistically different, we checked whether they were symmetrical. For that, since the size of our sample was not the same, we used the *Wilcoxon Rank-Sum test* (commonly referred to as the *Mann Whitney U test*) [Zim14]. We based this test on the null hypothesis that our samples were symmetrical (cf. Table 5.17, p. 158). After assessing the results from the *Mann Whitney U test*, we applied the *Bonferroni Correction method* to them to reduce the “false positive” cases [PDvdL05, Hay13]. Table 5.18, (p. 158) synthesizes those results.

As seen in Table 5.18, (p. 158), in most cases, the p-values were greater than 0.05, indicating that our groups are mainly symmetrical. However, there were cases where the p-value was smaller than 0.05 (painted in red).

Thus, since our data does not follow a normal distribution and our groups can be fully classified as symmetric, we used the *Permutation test* to compare our experimental groups. The *Permutation test* is a powerful and flexible statistical method that helps compare non-parametric experimental groups that do not comply with the assumption of symmetry [PS10]. Compared to other methods such as the *Mann-Whitney U test* [GC14], it is deemed to be more versatile and robust.

By applying the *Permutation test* and the *Bonferroni Correction method*, we calculated the p-values compiled in Table 5.20, (p. 160). Our test was carried out with 1000 *permutations* and multiple alternative hypotheses (cf. Table 5.19, p. 159). For metrics such as cyclomatic complexity or Halstead Effort, we implemented one-tailed alternative hypotheses to verify if the metrics from the first group were significantly lower than those from the second group. Conversely, we employed one-tailed alternative hypotheses for metrics such as maintainability or final project grade but in the opposite direction. Lastly, we conducted a two-tailed test for the GPAs since our objective was to evaluate any difference between the groups, regardless of the direction.

As can be checked in Table 5.20, (p. 160), most p-values for the evaluated metrics compared between the groups were greater than 0.05. For the groups that used our *Live Refactoring Environment*, we believe that happened because they used the same methodology no matter the development week they started using it. Nevertheless, there were a few situations where the p-value was less than 0.05 (e.g., most metrics when comparing *Week 1* and *Week 3*).

Similar results were observed when comparing the groups that used our approach with those that did not (group *No Tool*). Only group *Week 3* had no metrics statistically different from group *No Tool*. This fact could be because participants from the group *No Tool* may have resorted to the refactoring capabilities already offered by IntelliJ IDE. Consequently, although it is not feasible to demonstrate that all groups that used our tool differed significantly from the group that did not, we believe that the cases where the p-value was lower than 0.05 are enough to refute the null hypothesis of research questions RQ2, RQ3, and RQ4 (cf. Section 3.4, p. 82). This conclusion is supported by the results presented in Section 5.2.3.4, (p. 153) and Section 5.2.3.5, (p. 154), demonstrating that using our refactoring recommender helped quickly converge to better code quality and project grades.

Table 5.17: Null and alternative hypotheses used on the tests used to compare the experimental groups of the second experiment.

<i>Shapiro-Wilk Test</i>	
Null Hypothesis	Data follow a normal distribution.
Alternative Hypothesis	Data does not follow a normal distribution.
<i>Wilcoxon Rank Sum Test</i>	
Null Hypothesis	The experimental groups are symmetric.
Alternative Hypothesis	The experimental groups are not symmetric.
<i>Permutation Test</i>	
Null Hypothesis	The experimental groups are not statistically different.
Alternative Hypothesis	The experimental groups are statistically different.

Table 5.18: *Wilcoxon Rank Sum test* results after applying the *Bonferroni Correction method* to check if the experimental groups were equivalent.

	<i>W1-W2</i>	<i>W1-W3</i>	<i>W1-W4</i>	<i>W2-W3</i>	<i>W2-W4</i>
Lines of Code	0.14891	0.04771	0.47950	0.25790	0.47950
Cyclomatic Complexity	0.11235	0.43668	0.67137	0.07710	0.11979
Cognitive Complexity	0.38648	0.72367	0.47950	0.35797	0.52452
Lack of Cohesion	0.31232	0.28884	0.28884	0.88754	0.77730
Halstead Length	1.00000	0.06599	0.88754	0.20309	0.77730
Halstead Volume	1.00000	0.06599	0.77730	0.20309	0.67137
Halstead Effort	0.56370	0.03389	0.67137	0.32220	0.57161

Continued on next page

Table 5.18: *Wilcoxon Rank Sum test* results after applying the *Bonferroni Correction method* to check if the experimental groups were equivalent.
(Continued)

	<i>W1-W2</i>	<i>W1-W3</i>	<i>W1-W4</i>	<i>W2-W3</i>	<i>W2-W4</i>
Halstead Difficulty	0.56370	0.02365	0.57161	0.15730	0.88754
Halstead Time	0.56370	0.03390	0.67138	0.32220	0.57161
Maintainability	0.14891	0.03390	0.39614	0.20310	0.32220
Final Grade	0.77283	0.01621	0.20310	0.02366	0.15730
AVG GPA	0.24821	0.77730	0.57161	0.39614	0.83200
	<i>W3-W4</i>	<i>W1-NT</i>	<i>W2-NT</i>	<i>W3-NT</i>	<i>W4-NT</i>
Lines of Code	0.03764	0.10215	0.61886	0.33937	0.13251
Cyclomatic Complexity	0.62318	0.61886	0.08808	0.68433	0.91904
Cognitive Complexity	0.62318	0.69589	0.49959	0.71444	0.85484
Lack of Cohesion	0.87983	0.10215	0.83117	0.59713	0.41615
Halstead Length	0.04937	0.20083	0.47729	0.61132	0.14330
Halstead Volume	0.05878	0.20083	0.52243	0.65472	0.15475
Halstead Effort	0.06964	0.25551	0.56967	0.56923	0.27233
Halstead Difficulty	0.08210	0.08809	0.31978	0.51538	0.15476
Halstead Time	0.06964	0.25551	0.56968	0.56923	0.27233
Maintainability	0.01911	0.04660	0.47730	0.68433	0.02671
Final Grade	0.13058	0.03301	0.05501	0.41616	0.43984
AVG GPA	0.54536	0.54580	0.37437	0.46430	0.18640

Table 5.19: Null and alternative hypotheses used on the *Permutation tests*.

<i>One-Tailed - Lower Direction</i>	<i>Lines of Code, Cyclomatic Complexity, Cognitive Complexity, Halstead Metrics</i>
Null Hypothesis	Metrics from the first group are higher than the ones from the second group being compared.
Alternative Hypothesis	Metrics from the first group are lower than the ones from the second group being compared.
<i>One-Tailed - Higher Direction</i>	<i>Maintainability</i>

Continued on next page

Table 5.19: Null and alternative hypotheses used on the *Permutation tests*. (Continued)

Null Hypothesis	Metrics from the first group are lower than the ones from the second group being compared.
Alternative Hypothesis	Metrics from the first group are higher than the ones from the second group being compared.
Two-Tailed	<i>Average GPA</i>
Null Hypothesis	The GPAs of experimental groups are not statistically different.
Alternative Hypothesis	The GPAs of experimental groups are statistically different.

Table 5.20: Results collected from applying the *Permutation test* and the *Bonferroni Correction method* to our data (p-values lower than 0.05 are painted in green).

	W1-W2	W1-W3	W1-W4	W2-W3	W2-W4
Lines of Code	0.12858	0.02598	0.29171	0.13287	0.77023
Cyclomatic Complexity	0.07143	0.245756	0.38362	0.99001	0.97103
Cognitive Complexity	0.21429	0.34866	0.21479	0.95105	0.89311
Lack of Cohesion	0.24286	0.17783	0.16184	0.77023	0.69431
Halstead Length	0.42856	0.03097	0.42858	0.11389	0.49551
Halstead Volume	0.40000	0.03497	0.45755	0.14386	0.51649
Halstead Effort	0.27143	0.02698	0.36564	0.21779	0.60940
Halstead Difficulty	0.37143	0.01699	0.25975	0.06394	0.44456
Halstead Time	0.27143	0.02798	0.38762	0.20280	0.61239
Maintainability	0.10000	0.02998	0.25575	0.15985	0.76224
Project Grade	0.44286	0.00700	0.07093	0.01200	0.08892
AVG GPA	0.45715	0.88912	0.42358	0.47953	0.80320
	W3-W4	W1-NT	W2-NT	W3-NT	W4-NT
Lines of Code	0.99001	0.05895	0.29271	0.83917	0.04596
Cyclomatic Complexity	0.51149	0.28272	0.98902	0.49951	0.51249
Cognitive Complexity	0.21879	0.30170	0.91309	0.13287	0.51349
Lack of Cohesion	0.50450	0.04796	0.61739	0.40460	0.29571

Continued on next page

Table 5.20: Results collected from applying the *Permutation test* and the *Bonferroni Correction method* to our data (p-values lower than 0.05 are painted in green). (Continued)

	W1-W2	W1-W3	W1-W4	W2-W3	W2-W4
Halstead Length	0.96204	0.15185	0.22878	0.67533	0.13687
Halstead Volume	0.95905	0.15085	0.24976	0.64436	0.12088
Halstead Effort	0.87713	0.10290	0.30870	0.51649	0.14186
Halstead Difficulty	0.92008	0.09591	0.17083	0.65535	0.16384
Halstead Time	0.87013	0.08392	0.29871	0.51449	0.12088
Maintainability	0.98302	0.00800	0.16884	0.48552	0.00800
Project Grade	0.91409	0.01299	0.02598	0.67633	0.17983
AVG GPA	0.32968	0.84916	0.48552	0.65735	0.18982

5.2.3.7 Usability

At the end of this experience, the participants had to answer a set of 8 *usability questions* to give us feedback on our tool's usability. Usually, as already mentioned in Section 5.1.4.6, (p. 139), a SUS questionnaire has 10 *questions* [Bro13]. However, for this experiment, we decided to reduce the number of questions the participants had to answer. The questions are the following:

1. If they would like to use this tool frequently;
2. If they would like to use this tool in the future;
3. If they found the approach unnecessarily complex ;
4. If they found the approach easy to use;
5. If they thought that the various functions in the tool were well integrated;
6. If they were inconsistencies in the tool;
7. If they learned how to use the tool quickly;
8. If they felt confident using the approach.

In Figure 5.11, (p. 162), we present the percentages of each type of answer given to each one of the usability questions. As happened in Section 5.1.4.6, (p. 139), the results are painted on a color scale between *red* and *green*.

As seen in Figure 5.11, (p. 162), for most cases, the participants answered each question positively. However, there were some cases in which they gave a neutral or negative opinion, mainly on the future use of the tool, on finding inconsistencies in it and on feeling confident using it. Thus, we can conclude that our tool is usable. However, it has some aspects that should be improved.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Strongly Disagree	0%	0%	60%	0%	0%	78%	0%	0%
Disagree	4%	0%	33%	0%	0%	11%	0%	4%
Neutral	4%	7%	7%	4%	7%	7%	0%	26%
Agree	62%	19%	0%	11%	33%	4%	11%	44%
Strongly Agree	30%	74%	0%	85%	60%	0%	89%	26%

Figure 5.11: Answers to our SUS Questionnaire on the final experiment.

We also decided to ask the participants other usability questions, mainly to evaluate more specific visual aspects of our tool and their goals. These questions are the following:

1. If displayed suggestions helped understand the code that should be refactored;
2. If the refactoring suggested and applied helped create better code;
3. If the approach helped developers achieve better code, faster;
4. If the color scheme used to represent each refactoring was adequate;
5. If the placement of the interface was adequate;
6. If the placement of the refactoring suggestions was adequate;
7. If the number of suggestions was adequate;
8. If the tool was useful;
9. If they thought that using our approach would help maintain large and complex software systems.

The answers to these questions are summarized in Figure 5.12, (p. 162).

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Strongly Disagree	0%	0%	0%	0%	0%	0%	0%	0%	0%
Disagree	0%	0%	0%	0%	0%	0%	7%	0%	0%
Neutral	4%	11%	3%	0%	0%	4%	22%	0%	7%
Agree	33%	41%	56%	0%	7%	7%	19%	37%	30%
Strongly Agree	63%	48%	41%	100%	93%	89%	52%	63%	63%

Figure 5.12: Answers to other usability questions on our tool on the final experiment.

With these questions, we examined their outcomes. We observed that most participants agreed that our tool improved their programming solutions faster and helped them identify code issues and manage complex code. Additionally, the participants appreciated the color scheme we used to highlight the refactoring opportunities and their location within the IDE.

5.2.3.8 Reliability

To gather feedback on the usability of our tool, we asked our experiment participants from groups *Week 1*, *Week 2*, *Week 3*, and *Week 4* to answer a brief SUS Questionnaire [Bro13] (cf. Section 5.2.3.7, p. 161). With some of their answers, we were able to assess some reliability metrics, such as their average levels of *Perceived Ease of Use*, *Perceived Usefulness*, and *Intention to Use*. The questions used to assess these metrics are listed in Table 5.21, (p. 163).

Table 5.21: Usability questions used to assess reliability on the final experiment.

<i>Perceived Ease of Use</i>
1. I found the tool unnecessarily complex
2. I thought the tool was easy to use
3. I would imagine that most people would learn to use this tool very quickly
4. I felt very confident using the tool
<i>Perceived Usefulness</i>
1. The suggestions help understand the code that should be refactored
2. The refactorings suggested and applied helped create better code
3. The refactorings suggested and applied helped create better code, faster
<i>Intention of Use</i>
1. I think that I would like to use this tool frequently
1. I think that I would like to use this tool in the future

We calculated their *Cronbach's alpha* to evaluate these metrics. According to the literature, alphas higher or equal to 0.7 are considered as acceptable [Moo03].

As shown in Table 5.22, (p. 164), the *Cronbach's alphas* for the *Perceived Usefulness* and *Intention of Use* were higher than 0.7. This indicates the participants recognized the usefulness of our approach and intended to use it in the future. However, the *Cronbach's alpha* for the *Perceived Ease of Use* was slightly lower than 0.7, which indicates that they did not find our tool completely easy to use. Perhaps, if we provide a more comprehensive tutorial on using it correctly, they will understand how easy it is to use.

Based on these results, we can conclude that our *Live Refactoring Environment* is considered useful and a potential refactoring tool to be used in the future by programming students or even software developers.

Table 5.22: *Cronbach's alphas* measured to assess the reliability of our refactoring tool.

<i>Perceived Ease of Use</i>	<i>Perceived Usefulness</i>	<i>Intention of Use</i>
0.66	0.86	0.82

5.3 Automated Experiment

To verify if our *Refactoring Environment* helped create code with more quality, we executed an automated experiment. This experiment was carried out at the end of our tool's development and consisted of crawling and refactoring several GitHub projects.

To do so, we developed a bot that downloads and crawls (*cf.* Section 5.3.1, p. 164) and then inspects and refactors the different projects (*cf.* Section 5.3.2, p. 166).

Finally, after finishing the experiment, we analyzed the collected data. We asked several programming students how they felt about the changes created by our refactoring tool on the different projects (*cf.* Section 5.3.4, p. 171).

5.3.1 Crawler Implementation

In this experiment, we had to create a GitHub crawler. Usually, it consists of a tool designed to automatically search, analyze, and evaluate code repositories hosted on GitHub. It comprehensively examines the source code and various files within the repository to assess code quality, identify potential issues, and extract valuable insights about the codebase. The advantages of a GitHub crawler lie in allowing (i) perform automated code review, (ii) check if GitHub projects are using the best coding guidelines and standards, or (iii) quickly assess and analyze pull requests.

In our case, we designed it to open each Java file of a previously cloned project on IntelliJ IDE to be further analyzed and refactored with a specific version of our refactoring environment.

Our Java GitHub crawler was developed in Python using specific libraries offered by that programming language, such as *subprocess*⁴, *requests*⁵, or *os*⁶. As seen in Algorithm 13, (p. 165), it starts by getting a list of the ten projects evaluated with more stars on GitHub (Line 3, (p. 165) of Algorithm 13, (p. 165)), using the GitHub API⁷. Then, after receiving the projects, our crawler tries to clone each one of them (Line 15, (p. 165) of Algorithm 13, (p. 165)).

We repeated the cloning process daily for one week to ensure a consistently high-quality sample for this experiment. This approach allowed us always to have the best-classified projects and a larger sample to work with. Note that the cloning process does not allow the cloning a repository that already exists. In this way, projects that were analyzed in previous days were not inspected again.

⁴Subprocess Management, <https://docs.python.org/3/library/subprocess.html>

⁵Requests, <https://pypi.org/project/requests/>

⁶Miscellaneous operating system interfaces, <https://docs.python.org/3/library/os.html>

⁷GitHub REST API, <https://docs.github.com/en/rest?apiVersion=2022-11-28>

Algorithm 13 Fetching and Cloning GitHub Repositories.

```

1: procedure GETREPOSITORIES
2:    $params \leftarrow 'q': 'language:java', 'sort': 'stars', 'order': 'desc', 'per\_page': 10$ 
3:    $response \leftarrow$  Send GET request to GitHub API with API_URL, HEADERS, and params
4:   if  $response.status\_code = 200$  then
5:      $repositories \leftarrow$  Parse  $response.json()$  to get the list of repository items
6:     return  $repositories$ 
7:   else
8:     return []
9:   end if
10: end procedure
11:
12: procedure MAIN
13:    $repositories \leftarrow$  GETREPOSITORIES
14:   for  $repo$  in  $repositories$  do
15:     CLONEREPOSITORY( $repo[clone\_url]$ )    ▷ Cloning each repository using the “git
      clone” instruction.
16:   end for
17: end procedure

```

After finishing cloning the projects, our crawler starts its second phase by opening each project or sub-project on IntelliJ IDE (*cf.* Algorithm 14, p. 166). Note that by sub-projects, we are referring to smaller projects that were a part of a bigger one. To open them on the IDE, we had to use the command “*idea*” that allows us to open IntelliJ IDE through the command line (Line 4, (p. 166) of Algorithm 14, (p. 166)). Then, since it can take a while to finish the indexing process of the project opened on the IDE, we decided to stop code execution for 90 seconds. We chose this value after checking the average time it takes to fully open a project on this IDE.

After that, we crawl each Java file in the project’s directory. Suppose it’s a file placed inside the source code sub-directory (commonly known as “*src*”) and the file name is not equal to “*package-info.java*”, the file is opened on the IDE (Line 8, (p. 166) of Algorithm 14, (p. 166)). We decided to restrict the search and only analyze the source code and not any code used for testing purposes since we verified that the testing files were too simple and had lower complexity. Besides, we also decided to exclude “*package-info.java*” files since they only contained a few lines of code with the packages’ information.

When a file is opened on the IDE, our tool starts analyzing, and if it finds any refactoring opportunity, it tries to refactor the code automatically. This was possible because we installed our plugin before performing the crawling operations (*cf.* Section 5.3.2, p. 166). Then, and since we need to wait for our tool to end analyzing and refactoring the file opened on the IDE, we decided to create a while loop that checks a file where our tool writes all the prints resulting from analyzing a file and only stops when it finds the string “*No more refactoring.*” (Line 11, (p. 166) of of Algorithm 14, (p. 166)). After finding this string, the crawler progresses, opens a new file on the IDE, and starts a new inspection process.

Finally, only when the crawler finishes verifying each file of a project, it starts analyzing the next project in line.

Algorithm 14 Crawling the GitHub projects.

```

1: expected_print  $\leftarrow$  "Nomorerefactorings."
2: procedure MAIN
3:   for folder in directories do
4:     os.system("idea" + folder)
5:     time.sleep(90)
6:     java_files  $\leftarrow$  GlobalJavafilesrecursivelyinfolder
7:     for each file_path  $\in$  java_files do
8:       if file_path contains "/src/" and file_path equals "package - info.java" then
9:         os.system("idea" + folder + " " + file_path)
10:        found_print  $\leftarrow$  False
11:        while not foundPrint do
12:          prints  $\leftarrow$  Read prints file
13:          if prints equals expected_print then
14:            found_print  $\leftarrow$  True
15:          end if
16:        end while
17:      end if
18:    end for
19:  end for
20: end procedure

```

5.3.2 Automated Refactorings

As mentioned in Section 5.3.1, (p. 164), before performing the crawling process, we had to install our plugin on IntelliJ IDE to automatically inspect and refactor each file opened by the crawler. For that, we had to create a version of our plugin that performs the correct operations.

Most parts of this new version of our refactoring environment were reused from the original plugin. However, we had to change four specific details:

1. On the original version of our *Live Refactoring Environment*, we painted the refactoring suggestions right after getting all the candidates for a specific refactoring type. On our robotized tool, we only start the refactoring code after gathering all the refactoring opportunities independently of their type.
2. On the original version of our *Live Refactoring Environment*, users can choose the refactoring they want to apply to the code. However, with a robotized tool, we do not have a human capable of choosing the most suitable refactoring for a particular programming context. Thus, on the robotized version, we decided to randomly apply the refactorings without any specific order.

3. To control the flow of the robotized tool on the side of our Python crawler, we decided to save each Java print done by it. Then, when the tool does not find any refactoring opportunities in a file, it prints “*No more refactorings*”, and the Python crawler will know that it should switch files.
4. On the original version of our *Live Refactoring Environment*, we mainly used the refactoring capabilities offered by IntelliJ IDE. However, for the robotized tool, we did not want any dialog box opened when it tried to refactor the code. Thus, we had to implement code capable of performing each refactoring only by directly changing the code’s AST through PSI operations. These processes are described in the following subsections.

5.3.2.1 Extract Method

To perform an *Extract Method* refactoring, we should follow a set of steps Algorithm 15, (p. 167). First, we need to create a new method (Line 2, (p. 167) of Algorithm 15, (p. 167)). Then, we should move the code pointed out as an *Extract Method* opportunity to the new method (Line 3, (p. 167) of Algorithm 15, (p. 167)). After that, we should delete the block of code from its old method (Line 7, (p. 167) of Algorithm 15, (p. 167)). Finally, we need to place a call for the new method on the old one (cf. Algorithm 10, p. 167).

Algorithm 15 *Extract Method* applicer created for our robotized tool.

```

1: procedure EXTRACTMETHOD(candidate)
2:    $new\_Method \leftarrow createMethod(method\_Name, PsiType.VOID)$   $\triangleright createMethod$  is a
   method that can be accessed through the PsiTree.
3:   for each  $node \in candidate.nodes$  do
4:      $new\_Method.body.add(node)$ 
5:   end for
6:    $candidate.method.class.add(newMethod)$   $\triangleright$  By candidate, we are referring to the Extract
   Method candidate.
7:   for  $i \leftarrow 1 \rightarrow candidate.nodes.size() - 1$  do
8:      $candidate.nodes[i].delete()$ 
9:   end for
10:   $methodCallStatement \leftarrow createStatementFromText(method\_name + "()", null)$   $\triangleright$ 
    $createStatementFromText$  is a method that can be accessed through the PsiTree.
11:   $candidate.nodes[0].replace(methodCallStatement)$ 
12: end procedure

```

5.3.2.2 Extract Class

To execute an *Extract Class* refactoring with our robotized tool, we used the same *Extract Class* applicer used on the original version of our *Live Refactoring Environment*. As mentioned in

Section 4.2.3.2, (p. 100), to apply this refactoring type, we resorted to the *Extract Class* executor implemented on IntelliJDeodorant⁸.

5.3.2.3 Extract Variable

As seen in Algorithm 16, (p. 168), to apply an *Extract Variable* refactoring, we need to find the method call expression represented by the refactoring candidate we aim to execute (Line 3, (p. 168) of Algorithm 16, (p. 168)). Then, we should find the return type of that method call and create a new variable with that information (Line 5, (p. 168) of Algorithm 16, (p. 168)). Finally, we need to change the code to add the new variable (Line 7, (p. 168) of Algorithm 16, (p. 168)) and change the line in which the method call was placed (Line 8, (p. 168) of Algorithm 16, (p. 168)).

Algorithm 16 *Extract Variable* applier created for our robotized tool.

```

1: procedure EXTRACTVARIABLE(candidate)
2:   for each methodCallExpression  $\in$  candidate.method do
3:     if candidate.node == methodCallExpression then
4:       variableType  $\leftarrow$  methodCallExpression.method.type
5:       variableDeclaration  $\leftarrow$  variableType.text + "" + variableName + " = " +
        methodCallExpression.text + ";"  $\triangleright$  variableName is the random generated name of the new
        variable.
6:       text  $\leftarrow$  candidate.method.text
7:       text.add(variableDeclaration)
8:       text.replace(methodCallExpression.text, variableName + "()")
9:     end if
10:  end for
11: end procedure

```

5.3.2.4 Move Method

To apply an *Move Method* refactoring, we need to perform a set of operation on the PsiTree Algorithm 17, (p. 169). First, we must to create a new method with the same name, returning type, modifiers, parameters, and body (Line 5, (p. 169) of Algorithm 17, (p. 169)). Then, we should remove the original method from the PsiTree (Line 6, (p. 169) of Algorithm 17, (p. 169)) and add the new one to the target class (Line 7, (p. 169) of Algorithm 17, (p. 169)). After that, we need to change any reference that exists to the method when it was placed inside its original class (Line 8, (p. 169) of Algorithm 17, (p. 169)).

5.3.2.5 Replace Inheritance with Delegation

To perform a *Replace Inheritance with Delegation* refactoring (cf. Algorithm 18, p. 170), we first should create a field pointing out to the superclass and add it to the class being analyzed (Line 6,

⁸Extract Class Executor by IntelliJDeodorant, <https://github.com/JetBrains-Research/IntelliJDeodorant/blob/master/src/main/java/org/jetbrains/research/intellijdeodorant/ide/refactoring/extractClass/ExtractClassRefactoring.java>

Algorithm 17 *Move Method* applicer created for our robotized tool.

```

1: procedure MOVEMETHOD(candidate)
2:   method  $\leftarrow$  createMethod(candidate.method.name, candidate.method.type)
3:   method.modifierList.replace(candidate.method.modifierList)
4:   method.parameterList.replace(candidate.method.parameterList)
5:   method.body.replace(candidate.method.body)
6:   candidate.method.delete()
7:   candidate.targetClass.add(method)
8:   for each methodCallExpression  $\in$  classes do  $\triangleright$  classes represent all classes present on the
project.
9:     methodReference  $\leftarrow$  methodCallExpression.methodExpression
10:    resolvedMethod  $\leftarrow$  methodReference.method
11:    if resolvedMethod equals method then
12:      classReference  $\leftarrow$  methodReference.qualifierExpression
13:      classReference.delete()
14:      newReference  $\leftarrow$  createReferenceExpression(candidate.targetClass)  $\triangleright$ 
createReferenceExpression is a method that can be accessed through the PsiTree.
15:      newReference.getQualifierExpression().replace(classReference)
16:      methodReference.replace(newReference)  $\triangleright$  Replacing any reference to the old
method.
17:    end if
18:  end for
19: end procedure

```

(p. 170) of Algorithm 18, (p. 170)). Then, for each super method, we should create a delegated method with its name and return type (Line 8, (p. 170) of Algorithm 18, (p. 170)). After that, we need to create a statement that returns the call to the delegated method, through the field that we created on the first step (Line 10, (p. 170) of Algorithm 18, (p. 170)). Finally, we add that return statement to the body of the delegated method and add that method to the original class (Line 12, (p. 170) of Algorithm 18, (p. 170)).

5.3.2.6 Introduce Parameter Object

To execute an *Introduce Parameter Object* refactoring, we first need to create a class with a desired name (Line 4, (p. 170) of Algorithm 19, (p. 170)) and create an object type based on that new class (Line 5, (p. 170) of Algorithm 19, (p. 170)). After that, we need to create and add a set of fields to the new class based on the parameter we aim to replace by the new object (Line 7, (p. 170) of Algorithm 19, (p. 170)). Then, we have to create a new parameter with the new object (Line 10, (p. 170) of Algorithm 19, (p. 170)) and replace the old parameter with this new one (Line 11, (p. 170) of Algorithm 19, (p. 170)). Finally, as a last step, we should change all the calls to the method we were modifying and replace the parameters with the created object (Line 14, (p. 170) of Algorithm 19, (p. 170)). All these steps are synthesized in Algorithm 19, (p. 170).

Algorithm 18 *Replace Inheritance with Delegation* applier created for our robotized tool.

```

1: procedure REPLACEINHERITANCEDELEGATION(candidate)
2:   originalClass  $\leftarrow$  candidate.originalClass
3:   superClass  $\leftarrow$  candidate.targetClass
4:   superMethods  $\leftarrow$  superClass.methods
5:   field  $\leftarrow$  createField(superClass.name, superClass.type)  $\triangleright$  createField is a method that
      can be accessed through the PsiTree.
6:   originalClass.add(field)
7:   for each superMethod  $\in$  superMethods do
8:     delegateMethod  $\leftarrow$  createMethod(superMethod.name, superMethod.type)  $\triangleright$ 
      createMethod is a method that can be accessed through the PsiTree.
9:     methodName  $\leftarrow$  delegateMethod.body
10:    returnStatement  $\leftarrow$  createStatementFromText("return" + field.name + "." +
      superMethod.name + "();", delegateMethod)  $\triangleright$  createStatementFromText is a method that
      can be accessed through the PsiTree.
11:    methodName.add(returnStatement)
12:    originalClass.add(delegateMethod)
13:   end for
14: end procedure

```

Algorithm 19 *Introduce Parameter Object* applier created for our robotized tool.

```

1: procedure INTRODUCEPARAMETEROBJECT(candidate)
2:   parametersToReplace  $\leftarrow$  candidate.originalParameters
3:   parameterList  $\leftarrow$  candidate.method.parameterList
4:   parameterObjectClass  $\leftarrow$  createClass(parameterName)  $\triangleright$  createClass is a method that
      can be accessed through the PsiTree.
5:   parameterObjectType  $\leftarrow$  createType(parameterObjectClass)  $\triangleright$  createType is a method
      that can be accessed through the PsiTree.
6:   for each parameter  $\in$  parametersToReplace do
7:     field  $\leftarrow$  createField(parameter.name, parameter.type())  $\triangleright$  createField is a method
      that can be accessed through the PsiTree.
8:     parameterObjectClass.add(field)
9:   end for
10:  parameterObject  $\leftarrow$  createParameter(parameterName, parameterObjectType)
11:  methodCallExpression.method.parameterList.clear()
12:  parameterList.add(parameterObject)
13:  for each methodCallExpression  $\in$  classes do
14:    methodCallExpression.method.parameterList.clear()
15:    methodCallExpression.method.parameterList.add(parameterObjectClass)
16:  end for
17: end procedure

```

5.3.2.7 String Comparison

To perform a *String Comparison* refactoring with our robotized tool, we used the same methodology that we have already used on the original version of our *Live Refactoring Environment*. As mentioned in Section 4.2.3.7, (p. 109), we resorted to IntelliJ capabilities to replace each “==” operator when comparing strings with the “*equals()*” method (*cf.* Algorithm 10, p. 109).

5.3.3 Outcomes

Our experiment yielded two sets of results, both of which addressed the research questions outlined in (*cf.* Section 3.4, p. 82).

Outcome A: This set of results encompassed the analysis of various code quality metrics (*e.g.*, Halstead metrics or cyclomatic complexity) across each applied refactoring. These insights were aimed at addressing the research question RQ3.

Outcome B: This outcome included the answers to a questionnaire administered to several programming students. This questionnaire sought to gather their perspectives on the original and final versions of the different projects and whether our tool helped create better code. With it, we aimed to mainly answer research questions RQ2 and RQ3.

5.3.4 Results and Discussion

Upon concluding this experiment, we thoroughly analyzed and discussed the collected data. Therefore, in this section, we present a comprehensive summary of all the results obtained from the experiment. We interpret and discuss these findings to ascertain the validation of our hypothesis (*cf.* Section 3.3, p. 82) and to answer our research questions (*cf.* Section 3.4, p. 82).

Since this was an automated experiment, we did not have any participants, and therefore, we did not have any experimental groups either. Thus, we focused our results analysis on verifying if the robotized version of our *Live Refactoring Environment* could improve the code quality of each of the GitHub projects. We also asked several programming students with bachelor’s and master’s degrees if they preferred the code generated by our solution on the different projects or if they liked it most when it was not refactored.

5.3.4.1 GitHub Projects

As mentioned in Section 5.3.1, (p. 164), we simultaneously cloned the ten most well-ranked GitHub projects. However, knowing the volatility with which this ranking could change, we decided to repeat the cloning process every day for one week in order to always have the best-classified projects and a larger sample for this experience.

Therefore, we had 36 *projects*, with a total of 93 *project directories* (including main projects and sub-projects). Next, we list all the 36 main projects. As can be verified, most of them were projects focused on improving different aspects of the development of Android applications.

Android Percent Support Lib Sample. This project creates a package of the Android support library. It's available at:

<https://github.com/JulienGenoud/android-percent-support-lib-sample>.

AndroidUtilCode. This project is a robust and user-friendly Android library that efficiently encapsulates commonly used functions in Android development. It is available at:

<https://github.com/Blankj/AndroidUtilCode>.

Android-Volley. This project is an unofficial mirror, with minor bugfix for *Volley Android Framework*⁹. It is available at:

<https://github.com/mcxiaoke/android-volley>.

Apache Dubbo. This project is an open-source *RPC (Remote Procedure Call)* framework built on Java, known for its high-performance capabilities. It is available at:

<https://github.com/apache/dubbo>.

Apache RocketMQ. This is a distributed messaging and streaming platform known for its low latency, high performance, and exceptional reliability. It is available at:

<https://github.com/apache/rocketmq>.

Canal. This project is based on *MySQL database* incremental log parsing, providing incremental data subscription and consumption. It is available at:

<https://github.com/alibaba/canal>.

ChatGPT Spring Boot Starter. This project creates support for ChatGPT chat. It is available at:

<https://github.com/linux-china/chatgpt-spring-boot-starter>.

CircleDisplay. This project presents an Android view that enables the display and touch-based selection of values/percentages in a circular-shaped view, enhanced with captivating animations. It is available at:

<https://github.com/xuruoxuan/kkka.js>.

CircleRefreshLayout. This project features a custom pull-to-refresh layout that includes captivating animations. It is available at:

<https://github.com/tuesda/CircleRefreshLayout>.

Dart and Henson. This project is an Android library designed to organize the navigation layer of applications. It simplifies the creation and consumption of intents in a structured manner. It is available at:

<https://github.com/f2prateek/dart>.

⁹Volley Android Framework, <https://android.googlesource.com/platform/frameworks/volley>

FabulousFilter. This project creates a library that represents the implementation of the filter concept originally shared on *MaterialUp*¹⁰. It is available at:

<https://github.com/Krupen/FabulousFilter>.

Fresco. This project represents a system designed for displaying images in Android applications, simplifying the image loading and display process, eliminating manual handling. It is available at:

<https://github.com/facebook/fresco>.

GifLoadingView. This project helps create a loading view for GIFs for Android applications. It is available at:

<https://github.com/Rogero0o/GifLoadingView>.

Glide. This project is an Android open-source framework for media management and image loading that is both fast and efficient. It is available at:

<https://github.com/bumptech/glide>.

Guava. This project is a collection of essential Java libraries developed by Google, encompassing a variety of features such as new collection types (like multimap and multiset), immutable collections, a graph library, and utilities for concurrency, I/O, hashing, primitives, or strings. It is available at:

<https://github.com/google/guava>.

Apache Hive. This project presents a data warehouse software that simplifies the reading, writing, and managing large datasets stored in distributed storage through SQL queries. It is available at:

<https://github.com/apache/hive>.

JEECG BOOT. This project is a Java low-code platform for enterprise web applications. It is available at:

<https://github.com/jeecgboot/jeecg-boot>.

Logger. This project creates an amusing and powerful logger to be used on Android. It is available at:

<https://github.com/orhanobut/logger>.

MaterialProgressBar. This project creates a material design *ProgressBar* compatible with Android 4.0+. It is available at:

<https://github.com/zhanghai/MaterialProgressBar>.

Mockito. This project presents a mocking framework for Java unit tests. It is available at:

<https://github.com/mockito/mockito>.

¹⁰MaterialUp, <https://www.uplabs.com/posts/materialup>

MPAndroidChart. This is a chart library for Android, which makes it easy to create and use charts in Android applications. It is available at:

<https://github.com/PhilJay/MPAndroidChart>.

MyBatis. This project presents a MyBatis *SQL mapper* framework that simplifies the integration of relational databases with object-oriented applications. It is available at:

<https://github.com/mybatis/mybatis-3>.

Paginate. This project creates an Android library that facilitates the creation of straightforward pagination functionality (also known as infinite scrolling) for *RecyclerView* or *AbsListView* components. It is available at:

<https://github.com/MarkoMilos/Paginate>.

Phrase. This project is an Android string formatting. It is available at:

<https://github.com/square/phrase>.

Realm. This project presents a mobile database designed to operate directly within phones, tablets, or wearables. This project contains the source code for the Java version of *Realm*, which is currently compatible with Android platforms. It is available at:

<https://github.com/realm/realm-java>.

RecyclerView-FastScroll. This is a straightforward *FastScroller* designed for Android's *RecyclerView*. It is compatible with vertical *RecyclerViews*, whether using *LinearLayoutManager* or *GridLayoutManager* (with multiple spans). It is available at:

<https://github.com/timusus/RecyclerView-FastScroll>.

RecyclerViewItemAnimators. This project creates animators to be used on *RecyclerView* items. It is available at:

<https://github.com/gabrielemariotti/RecyclerViewItemAnimators>

Retrofit. This is a type-safe HTTP client crafted for both Android and Java environments. It is available at:

<https://github.com/square/retrofit>.

RxJava. This project presents a *Java Virtual Machine (VM)* implementation of *Reactive Extensions* that enables the composition of asynchronous and event-based programs through observable sequences. It is available at:

<https://github.com/ReactiveX/RxJava>.

Selenium. This comprehensive project encompasses various tools and libraries, allowing for web browser automation. It is available at:

<https://github.com/SeleniumHQ/selenium>.

SmartTubeNext. This project presents a feature-rich YouTube application designed specifically for Android TVs and TV boxes. It is available at:

<https://github.com/yuliskov/SmartTubeNext>.

Spring Boot. This project tries to simplify the process of creating Spring-powered, production-grade applications and services with utmost ease. It is available at:

<https://github.com/spring-projects/spring-boot>.

Termux. This Android application functions as a terminal emulator and provides a *Linux environment*. It is available at:

<https://github.com/termux/termux-app>.

Tinker. This is an Android hot-fix solution library that facilitates updates for libraries and resources without the need to reinstall the APK. It is available at:

<https://github.com/Tencent/tinker>.

Twitter Ambrose. This project functions as a platform designed for the visualization and real-time monitoring of *MapReduce* data workflows. It is available at:

<https://github.com/twitter-archive/ambrose>.

ZoomHeader. This project helps create zoom options for Android application headers. It is available at:

<https://github.com/githubwing/ZoomHeader>.

5.3.4.2 Number of Refactorings

Outcome A allowed us to quantify the *number of refactorings* executed by our robotized refactoring environment. Table 5.23, (p. 176) and Table 5.24, (p. 177) summarize the number of refactorings applied by refactoring type.

Table 5.23, (p. 176) provides the raw count of each refactoring type applied to each project. With it, we can verify that the total number of refactorings applied in each project varies significantly. The same happens with the number of refactoring applied by type. In specific projects, such as *Android-Volley* or *Dart and Henson*, the *Extract Method* was the most applied refactoring. In other cases like *Fresco* or *Glide* it was the *Extract Variable*. With this fact, we might conclude that the extent and type of changes introduced by the refactorings applied might differ across projects. Independently of this, the *Extract Variable* refactoring has the highest count across most projects, followed by *Extract Method* and *Move method*. Finally, projects like *Apache RocketMQ* and *Apache Dubbo* had higher total numbers of refactorings, indicating potentially complex codebases that required extensive restructuring.

In Table 5.23, (p. 176), we can all notice that none of the crawled projects applied an *String Comparison* refactorings. This means that the projects' developers already use the method "*equals()*" when comparing strings and not the equality operator "*==*".

Table 5.23: Number of refactorings applied in each project.

	<i>EM</i>	<i>EC</i>	<i>EV</i>	<i>MM</i>	<i>IPO</i>	<i>ID</i>	<i>SC</i>	<i>Total</i>
Android Percent Support Lib Sample	6	0	6	0	0	0	0	12
AndroidUtilCode	1	0	0	0	0	0	0	1
Android-Volley	46	0	26	0	7	0	0	79
Apache Dubbo	331	2	1713	153	19	22	0	2240
Apache Hive	24	0	39	2	0	0	0	65
Apache RocketMQ	618	7	6676	392	79	20	0	8792
Canal	566	8	3668	199	42	56	0	6539
ChatGPT Spring Boot Starter	18	0	23	44	0	0	0	85
CircleDisplay	2	0	8	0	0	0	0	10
CircleRefreshLayout	8	0	12	0	0	0	0	20
Dart and Henson	29	0	10	0	1	0	0	40
FabulousFilter	36	0	1	0	1	0	0	38
Fresco	406	5	1327	0	65	0	0	1803
GifLoadingView	1	0	0	0	0	0	0	1
Glide	242	1	917	6	35	0	0	1201
Guava	362	23	4798	1	55	0	0	5439
JEECG BOOT	517	19	1526	295	37	10	0	2404
Logger	6	0	1	0	0	0	0	7
MaterialProgressBar	9	0	44	0	0	0	0	53
Mockito	200	4	1858	1	18	0	0	2081
MPAndroidChart	302	4	121	0	38	0	0	465
MyBatis	247	2	987	258	61	41	0	1596
Paginate	7	0	14	0	2	0	0	23
Phrase	14	0	29	0	0	0	0	43
Realm	257	1	496	5	19	0	0	778
RecyclerView-FastScroll	16	0	22	0	1	0	0	39
RecyclerViewItemAnimators	2	0	4	0	0	0	0	6
Retrofit	36	0	141	0	6	0	0	183

Continued on next page

Table 5.23: Number of refactorings applied in each project. (Continued)

RxJava	152	41	1651	249	8	28	0	2129
Selenium	36	2	1167	103	9	17	0	1334
SmartTubeNext	207	0	335	1	10	0	0	553
Spring Boot	875	7	6867	8	112	0	0	7879
Termux	143	0	444	0	26	0	0	613
Tinker	419	2	372	7	69	0	0	869
Twitter Ambrose	0	0	20	1	0	0	0	21
ZoomHeader	9	0	0	0	0	0	0	9
Total	6523	138	29482	907	532	254	0	36136

Table 5.24, (p. 177) provides the average count of each refactoring type applied to each file of each project. When comparing the data from Table 5.23, (p. 176) and Table 5.24, (p. 177), we can identify projects where the average count deviates significantly from the total count. This happens with projects like *Apache Dubbo*, *Mockito*, or *Guava* where the total number of refactoring is close to the thousands and the average number of refactoring applied by file does not reach 30. This means that these projects had several files that were refactored. On the other hand, a project like *Phrase* had a lower discrepancy between counters. This could mean that the project has too few files or that too few files must be refactored. After checking the project repository, we verified it was the first hypothesis.

Table 5.24: Average number of refactorings applied in each file of each project.

	<i>EM</i>	<i>EC</i>	<i>EV</i>	<i>MM</i>	<i>IPO</i>	<i>ID</i>	<i>SC</i>	<i>Total</i>
Android Percent Support Lib Sample	3	0	3	0	0	0	0	6
AndroidUtilCode	1	0	0	0	0	0	0	1
Android-Volley	3	0	2	0	1	0	0	6
Apache Dubbo	2	0	8	1	1	0	0	13
Apache Hive	4	0	6	0	0	0	0	10
Apache RocketMQ	2	0	18	2	1	0	0	23
Canal	3	0	14	1	1	0	0	20
ChatGPT Spring Boot Starter	2	0	2	4	0	0	0	8
CircleDisplay	1	0	4	0	0	0	0	5
CircleRefreshLayout	4	0	6	0	0	0	0	10
Dart and Henson	4	0	2	0	1	0	0	7

Continued on next page

Table 5.24: Average number of refactorings applied in each file of each project. (Continued)

FabulousFilter	4	0	1	0	1	0	0	7
Fresco	2	0	6	0	1	0	0	9
GifLoadingView	1	0	0	0	0	0	0	1
Glide	2	0	8	0	1	0	0	11
Guava	2	0	20	0	1	0	0	24
JEECG BOOT	2	0	7	2	1	0	0	13
Logger	3	0	1	0	0	0	0	4
MaterialProgressBar	1	0	5	0	0	0	0	6
MPAndroidChart	3	0	2	0	0	0	0	5
MyBatis	2	0	6	2	1	0	0	12
Paginate	3	0	5	0	1	0	0	10
Phrase	7	0	15	0	0	0	0	22
Realm	5	0	10	1	1	0	0	18
RecyclerView-FastScroll	6	0	8	0	1	0	0	15
RecyclerViewItemAnimators	1	0	2	0	0	0	0	3
Retrofit	3	0	11	0	1	0	0	16
RxJava	1	1	11	2	1	0	0	16
Selenium	1	0	11	1	1	0	0	15
SmartTubeNext	3	0	5	0	0	0	0	8
Spring Boot	1	0	9	0	1	0	0	12
Termux	3	0	8	0	1	0	0	12
Twitter Ambrose	0	0	10	1	0	0	0	11
Total	136	1	327	23	19	0	0	506

Based on the analysis of both tables and by complementing these data with one described in the next section (*cf.* Section 5.3.4.3, p. 178), we can conclude that our robotized refactoring environment performed several refactorings in each project that ultimately helped them converge to better code quality.

5.3.4.3 Code Quality

We first aimed to verify whether our tool improved code quality when it inspected and refactored code automatically without human external actions. To do so, and through Outcome A (*cf.* Section 5.3.3, p. 171), we assessed how much the metrics improved on average with each refactoring

type.

As can be verified in Table 5.25, (p. 179), which summarizes the percentages in which each metric improved or decreased per refactoring type, most of the metrics evolved positively with each refactoring applied by our robotized tool. Most of the cases (*e.g.*, the number of lines of code, cyclomatic complexity, or the Halstead metrics) decreased their values as expected for most of the refactorings. On the other hand, the code maintainability improved as it should.

Nevertheless, there were cases in which the metrics did not suffer the expected impact (*e.g.*, the number of lines of code when applying an *Extract Variable* or *Move Method* refactoring or the lack of cohesion when applying a *Extract Method*, *Extract Class*, or *Replace Inheritance with Delegation* refactorings). We believe this happened because we use the metrics as thresholds to identify and sort each refactoring type. For instance, to detect an *Extract Method*, we focus on finding blocks of code inside methods with too many lines of code, and as you can see in Table 5.25, (p. 179), this metric decreased when applying this kind of refactoring. However, to detect it, we did not consider the lack of cohesion (*cf.* Section 4.2.3.1, p. 99); we only used it to sort the candidates out, and as listed in Table 5.25, (p. 179), this suffered a small but yet negative impact from applying an *Extract Method*. Besides, we have to remember that specific refactorings might improve the code quality of a particular method or class and, at the same time, decrease the code quality of other methods or classes (*e.g.*, when we move a method from one class to another or when we remove the inheritance and replace it with a delegation, we might be improving some metrics on the original class but at the same time decreasing others on the target one). Also, as seen in Table 5.23, (p. 176) and Table 5.24, (p. 177), the *Extract Variable* refactoring was the most applied one. This might justify why this refactoring negatively impacted several metrics. As known, when we extract a variable, we increase the number of lines of code and the number of variables inside a method, increasing most of the Halstead metrics and decreasing code maintainability. Finally, we cannot forget that these results were obtained by automatically applying a set of refactorings that were randomly chosen. Maybe these cases would be reduced or nonexistent if the tool was operated by a developer who could make the best refactoring choice for his code.

Table 5.25: Average improvement of each metric per refactoring type.

	<i>EM</i>	<i>EC</i>	<i>EV</i>	<i>MM</i>	<i>IPO</i>	<i>ID</i>	<i>SC</i>
Lines of Code	-8.52%	-2.90%	0.30%	2.51%	-0.20%	-71.10%	-
Cyclomatic Complexity	-5.91%	-5.30%	-0.10%	-7.04%	-0.05%	-23.60%	-
Cognitive Complexity	-12.90%	-6.30%	-0.10%	-10.75%	-0.10%	-22.70%	-
Lack of Cohesion	2.90%	-0.56%	-0.10%	-8.02%	0.00%	68.66%	-
Halstead Length	-3.90%	-3.10%	0.10%	-4.80%	-2.90%	-27.60%	-
Halstead Volume	-5.10%	-4.00%	0.20%	-6.40%	-3.80%	-35.50%	-
Halstead Effort	-11.00%	-8.90%	0.60%	-12.80%	-3.80%	-53.90%	-

Continued on next page

Table 5.25: Average improvement of each metric per refactoring type. (Continued)

Halstead Difficulty	-4.20%	-5.60%	0.30%	-5.60%	0.00%	-26.80%	-
Halstead Time	-11.00%	-8.90%	0.60%	-12.80%	-3.80%	-53.90%	-
Maintainability	0.05%	2.00%	-0.02%	3.50%	-0.02%	-0.72%	-

Even though we did not verify that our robotized tool positively impacted all metrics with any type of refactoring, we believe the results demonstrate that it helped improve code quality in most cases. We can also corroborate this belief with the results presented in Table 5.26, (p. 180), which shows the average improvement of each code quality metric when comparing each project's initial and final versions. As can be seen there, most metrics were positively impacted. The only cases in which the results were not expected were the ones obtained for cognitive complexity and lack of cohesion. Regardless, the negative impact caused on them was lower than 1.0%.

Table 5.26: Average improvement of each metric.

<i>Improvement (Original vs Refactored)</i>	
Lines of Code	-2.00%
Cyclomatic Complexity	-0.10%
Cognitive Complexity	0.06%
Lack of Cohesion	0.20%
Halstead Length	-0.07%
Halstead Volume	-0.10%
Halstead Effort	-0.21%
Halstead Difficulty	-0.10%
Halstead Time	-0.21%
Maintainability	1.96%

To check how our robotized tool impacted code quality, we also assessed the evolution of several code quality metrics (*e.g.*, number of lines of codes, Halstead metrics, or maintainability index) over each crawled project. We compiled all those results into Figure 5.13, (p. 182) and Figure 5.14, (p. 183). Besides, to simplify, we aggregated the results per project and not by sub-projects.

As seen in both Figure 5.13, (p. 182) and Figure 5.14, (p. 183), most project metrics improved considerably over each refactoring applied by our robotized approach. This means that our tool helped the projects converge to shorter, less complex, cleaner, or maintainable code.

There were some cases in which that did not happen, as is the case of projects such as *Apache Dubbo*, *Fresco*, *GifLoadingView*, *MaterialProgressBar*, *Mockito*, or *Twitter Ambrose*. This fact can be justified with the data summarized in Table 5.23, (p. 176) and Table 5.24, (p. 177). In these tables, we can check that the projects in which most code quality metrics did not merge to

better results were mostly refactored by extracting variables. Besides, as we verified in Table 5.25, (p. 179), some refactorings, such as the *Extract Variable*, decreased specific code quality metrics. Therefore, we believe that applying so many *Extract Variable* refactorings might have contributed to the outlier cases in which our tool did not help converge to better code.

5.3.4.4 Personal Opinions

As mentioned in Section 5.3.3, (p. 171), after our robotic experiment, we provided a Google form regarding each type of refactoring applied by our bot to programming students. In these forms, we gave an example of each kind of refactoring applied and asked the students if they thought the code improved and became less complex and easier to understand after each refactoring. The answers were measured using a *Likert Scale*, where 1 represented “*Strongly Disagree*” and 5 represented “*Strongly Agree*”. Each form was filled out by 32 students.

Figure 5.15, (p. 184) summarizes the answers collected from the 32 students. Note that, as seen in Section 5.3.4.2, (p. 175), our robotized refactoring tool did not apply any *String Comparison* refactoring. Then, we did not include this refactoring in this specific analysis.

In the case of *Extract Method* and *Extract Class* refactoring types, the majority of students agreed with the statement, with ratings of 4 or 5 (agree and strongly agree), indicating that they believed these refactoring techniques were well applied and resulted in improved code. Only a few students remained neutral with a rating of 3, and a minimal number disagreed with a rating of 2.

The answers were more mixed for the *Extract Variable* refactoring. While many students agreed or remained neutral, a notable proportion disagreed with a rating of 2. This suggests that *Extract Variable* refactoring might have presented some challenges or was perceived as less effective in the provided code example than other refactoring techniques.

In the case of *Move Method*, most students agreed (ratings of 4 or 5) that it was well applied and led to better code, with very few expressing disagreement.

Similarly, the *Introduce Parameter Object* refactoring received mixed responses, with more students in agreement or neutrality and some expressing disagreement. However, the level of disagreement was not as pronounced as with the *Extract Variable*.

Finally, the *Replace Inheritance with Delegation* refactoring type had a strong positive perception among the students, with most agreeing that it was well applied and resulted in improved code. In this case, disagreement was minimal.

Thus, with the results of the different forms, we verified that most programming students could recognize that the refactoring types applied by our robotized approach improved the code, making the solution less complex and easier to understand. The only refactoring that raised more doubts was the *Extract Variable*, which indicates that students did not see any relevance in applying to code.

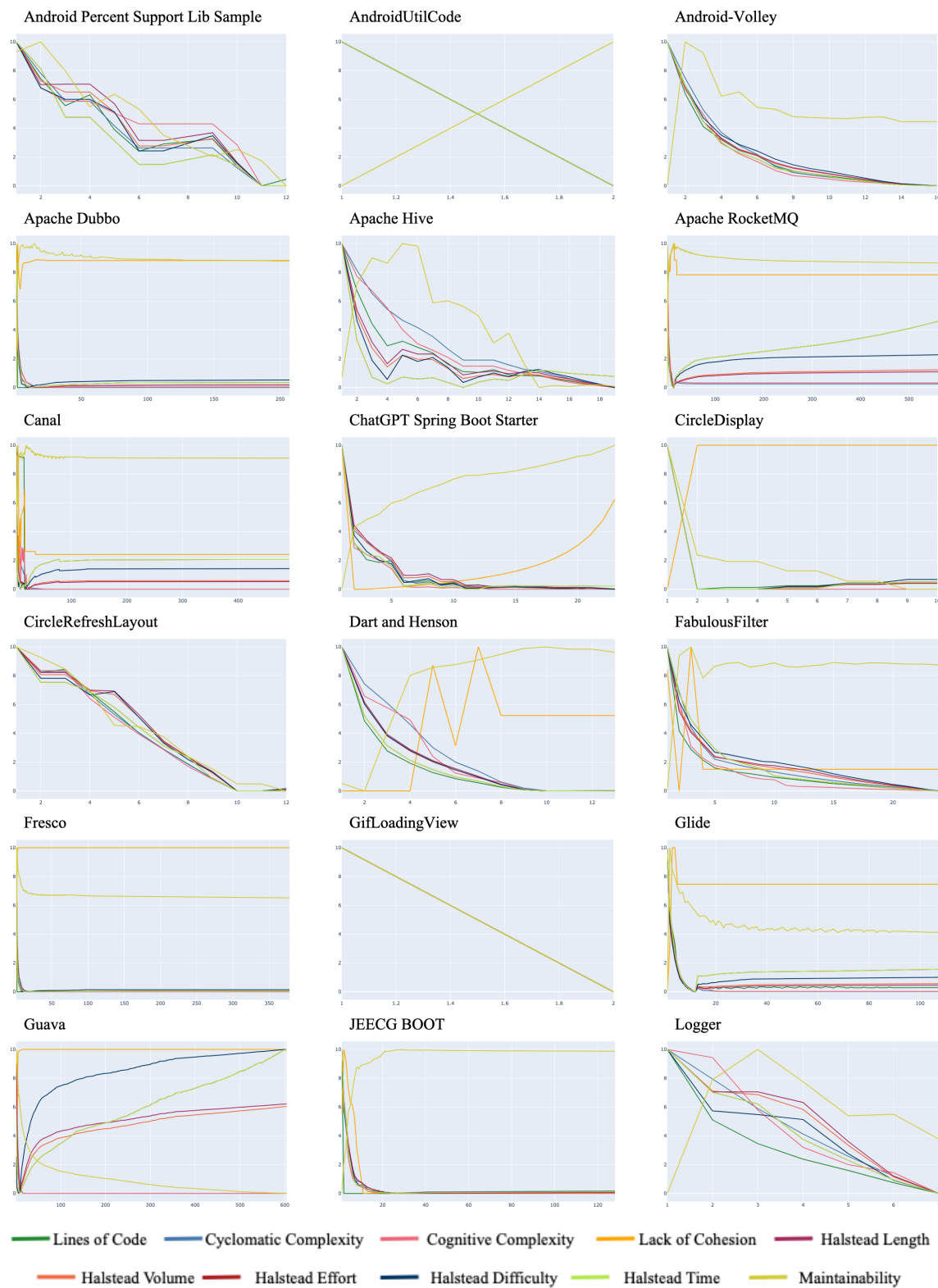


Figure 5.13: Metrics evolution in each project.

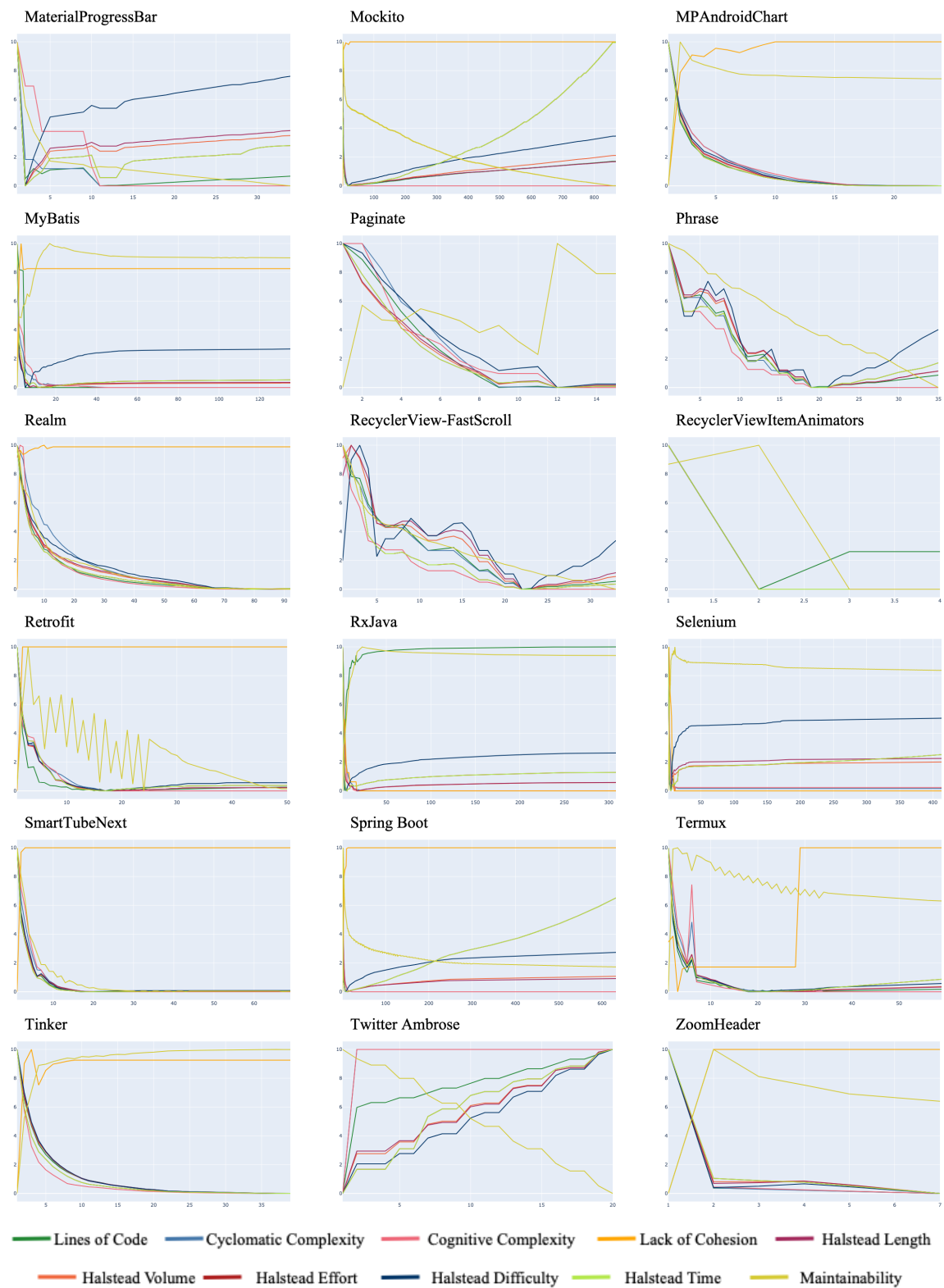


Figure 5.14: Metrics evolution in each project (continuation).

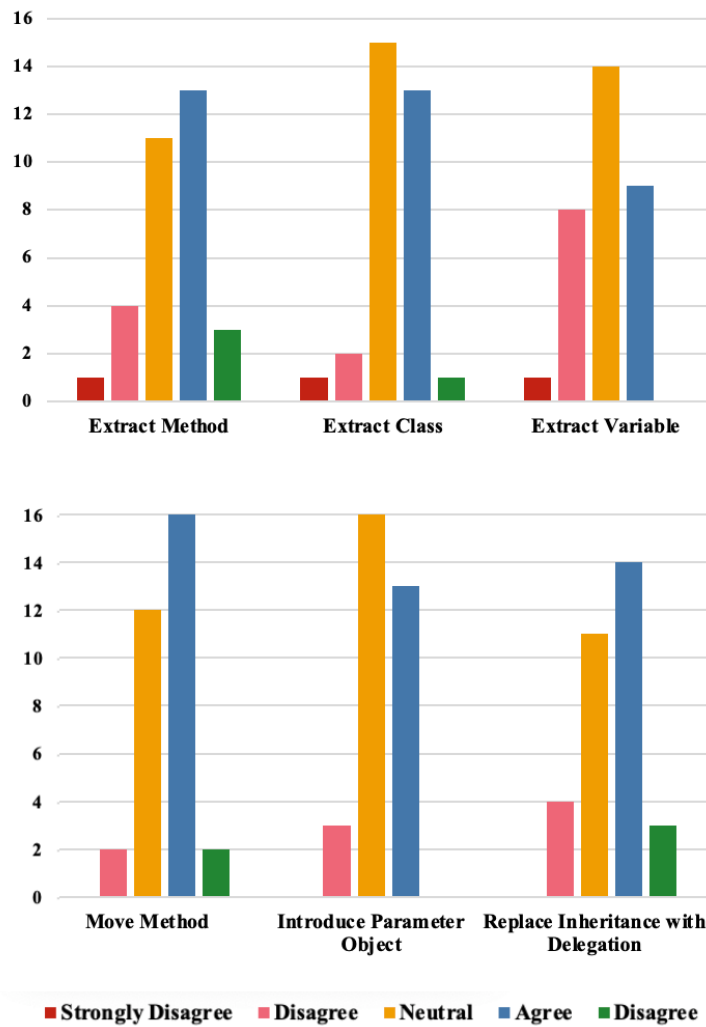


Figure 5.15: Personal opinions on the refactorings applied.

5.4 Answering the Research Questions

After gathering all data from our empirical experiments and analyzing and discussing the results, we can finally answer our research questions and check if we can reject the null hypotheses for each research question.

RQ1 *“Is it possible to inform developers, using a color-coded system, about possible refactoring opportunities in their code?”*

For both experiments, we assessed our tool’s usability and reliability. As can be checked in Section 5.1.4.6, (p. 139), the experiment participants thought our tool had good usability, with some points for possible improvement. For the same experiment, in Section 5.1.4.7, (p. 141), we verified that the participants perceived our tool’s usefulness and intended to use it in the future. However, they did not completely understand how easy it is to use it.

The same happened with the results detailed in Section 5.2.3.7, (p. 161). The participants of this experiment have shown to like our approach and classified it as having good usability. In terms of reliability (*cf.* Section 5.2.3.8, p. 163), we also verified that the experiment participants understood our tool's usefulness and intend to use it in the future. However, even though they had a better value, their perceived ease of use of our tool did not achieve the expected value.

In both experiments, we also assessed the participants' opinions on the visual methodology used to suggest the refactorings included in our tool. We verified that they liked how the refactorings were suggested and appreciated our visual approach, color scheme, and placement on the IDE.

Therefore, and even though our tool still has room for improvement, we think we could reject the null hypothesis of this research question since it is possible to identify, suggest, and apply refactorings using a color-coded approach to help developers create better code with more quality. And that is our *Live Refactoring Environment*.

RQ2 “What is the impact of real-time refactoring identification on the developers' awareness of their code quality?”

In the initial experiment, as outlined in Section 5.1.4.2, (p. 128), we confirmed that our approach improved all of the evaluated code quality metrics, regardless of the version of the tool used by the participants. Additionally, upon examining the first row of Table 5.1, (p. 132), we determined that using a *Live Refactoring Environment* resulted in participants performing more refactorings, likely due to increased visual awareness of potential flaws in their code.

In the second experiment, as outlined in Section 5.2.3.3, (p. 151), we observed that all experimental groups using our tool implemented numerous refactorings while developing their code. This suggests a clear understanding of the significance and benefits of refactoring in improving code quality. The percentage of undo operations performed after a specific refactoring, as shown in Table 5.14, (p. 152), further supports this conclusion, with over 65% of the 1326 refactorings applied being maintained on the code. Additionally, the results reported in Section 5.2.3.4, (p. 153) consistently indicate significant improvements in most evaluated code quality metrics, regardless of when participants began using the tool. Finally, as noted in Section 5.2.3.5, (p. 154), the use of a *Live Refactoring Environment* was found to have a positive impact on the creation of high-quality software, as demonstrated by improved project grades.

In the third experiment, as outlined in Section 5.3.4.4, (p. 181), we verified that most of the inquired programming students believed that the refactoring types applied by our robotized refactoring approach were capable of creating better code with more quality, less complex, and easier to be read, adapted, and maintained. The only refactoring that raised more doubts was the *Extract Variable*. This probably happens because students do not recognize its usefulness, as it creates more lines of code and introduces another variable into the code. Thus, if we consider that an automated approach that randomly applies refactorings can enhance code quality, it stands to reason that a real-time approach that detects code smells and tries to mitigate them could achieve similar improvements more swiftly and with better outcomes.

To conclude, based on the results obtained from these experiments, we are confident in rejecting the null hypothesis for this research question. That's because our findings indicate that the real-time identification of refactoring candidates positively impacts developers' awareness of their code quality.

RQ3 “What is the impact of a live refactoring environment on their code quality?”

When comparing the experimental groups *A1*, *A2*, and *A3*, in the first experiment, in Section 5.1.4.5, (p. 133), we verified that those groups were not statistically different. That is because the logic behind the approaches they used was the same. This fact does not happen when comparing these groups with group *B*, which did not use any refactoring tool, in which we verified that this group was statistically different from the remaining ones. Besides, as seen in Section 5.1.4.2, (p. 128), the complete version of our approach used by group *A1* had better code quality results in most cases.

In the second experiment, as reported in Section 5.2.3.6, (p. 156), several quality metrics measured by the experimental group showed statistically significant differences. Most of these differences occurred when comparing groups that utilized our tool (groups *Week 1*, *Week 2*, *Week 3*, and *Week 4*) with those that did not use it (group *No Tool*). Additionally, as shown in Figure 5.10, (p. 155), participants who used our solution since earlier development stages achieved better results in several code quality metrics and produced better programming solutions than those who did not use any refactoring approach.

In the third experiment, as reported in Section 5.3.4.3, (p. 178), we verified that our robotized refactoring approach, even though not working as a live tool, was capable of creating better code in most of the refactoring types applied since most of the assessed code quality metrics improved their values. Additionally, as demonstrated in Section 5.3.4.4, (p. 181), most of the surveyed programming students believed that most refactoring techniques employed through this approach could enhance code quality, reduce complexity, and make the code more readable, adaptable, and maintainable. Considering that a robotized approach that randomly applies refactorings can improve code quality, we can infer that a real-time approach, combined with the power of user decision, would also be able to do so more quickly and with better results.

With all the data collected from these experiments, we could reject the null hypothesis for this research question. Thus, we can conclude that a *Live Refactoring Environment* positively impacts code quality.

RQ4 “What is the impact of a live refactoring environment on the time required to achieve high-quality programming solutions?”

As confirmed in Section 5.1.4.5, (p. 133), one the first experiment, the time values obtained from comparing groups *A1*, *A2*, and *A3* were statistically different from those of group *B*. Moreover, by analyzing the second row of Table 5.1, (p. 132), we concluded that the time required to perform a refactoring using any version of our approach was faster than when participants did not use any kind of refactoring tool. Additionally, group *A1* demonstrated the quickest application of

each refactoring. As previously mentioned, using our approach, participants were able to produce better-quality code.

In the final experiment, we addressed the evolution of the metrics described in Figure 5.10, (p. 155). The analysis showed that the groups using our tool (regardless of the week in which they started using it) could achieve better programming solutions with higher code quality and at a faster rate than the participants who did not use any refactoring tool. Additionally, by monitoring each metric daily, it was observed that most of the time, the groups using our approach were developing code with better quality than those who did not use it.

Therefore, based on the results of our experiments, we can reject the null hypothesis for this research question and conclude that using a *Live Refactoring Environment* positively impacts the time required to achieve better programming solutions.

Based on the evidence summarized above, we are confident that not only were we able to answer positively to each research question, but we were also able to validate our hypothesis (*cf.* Section 3.3, p. 82).

5.5 Threats to Validity

Our empirical experiments can be characterized by the same type of threats to their validity. These threats can be divided into different categories: *internal*, *external*, *construct*, and *conclusion* threats [See12, SCC01, CC94, GP12]. Next, we described each one of them in more detail.

5.5.1 Internal Validity

Internal validity refers to the extent to which an experiment design is free from systematic error or bias that could affect the results. Internal threats to validity include selection bias, experimenter bias, and the influence of extraneous variables that could confound the relationship between the independent and dependent variables [See12, SCC01]. On this type of validity, we identified three main threats.

Order of the refactorings applied: We assess the severity of each refactoring opportunity by measuring specific code quality metrics. Thus, our candidates may change once a refactoring is applied. Consequently, how refactorings are carried out can affect the resulting outcomes. To mitigate this potential threat, we propose checking the effects of applying several refactorings on code quality and using this data to predict future refactoring candidates and the best order to apply specific refactoring types to achieve the best code possible,

Number of refactorings applied: We believe that identifying, suggesting, and applying refactorings to software, in real-time, can help developers create better code in less time. However, due to the novelty of our approach, it is challenging to determine the exact number of refactorings that should be suggested and applied to improve code quality.

Code quality metrics: In our experiment, we employed a set of metrics to evaluate the impact of our approach on code quality. However, this analysis only approximates the attributes captured by the metrics. Despite that, proven results confirm the reliability of using these metrics to evaluate code quality [BBM96, RMT09].

5.5.2 External Validity

External validity refers to the extent to which the findings of a study can be generalized to other populations, settings, and times. External threats to validity include factors such as the use of non-representative samples, the use of artificial experimental settings, and the influence of historical or cultural events that may limit the generalizability of the results to other contexts [See12, SCC01]. On this type of validity, we identified four main threats mainly focused on the participants and projects used in our experiment.

Participants' knowledge: The inclusion of bachelor students in our study may have introduced potential bias in our results. Some participants exhibited limited programming proficiency and lack of refactoring knowledge. Besides, some were novices in Java programming or using the IntelliJ IDE. As a result, evaluating the effectiveness of a live refactoring tool with this specific demographic might yield outcomes markedly different from those obtained with a more seasoned and experienced audience of programmers.

Pressure: Each project group was tasked with developing a project with specific goals, using the best programming practices possible. However, they may have felt pressured to use our *Live Refactoring Environment* to achieve better programming solutions and a higher project grade. Although this is a complex threat to mitigate.

IDE capabilities: The experiment participants were instructed not to use the refactoring tools included in the IntelliJ IDE. However, there may have been situations where the groups did not follow this directive, which could have impacted their final results. Unfortunately, there is no clear way to solve this threat, as it is up to the participants to decide whether or not to use the refactoring options offered by the IDE.

Selected projects: We examined several Java projects and tested our approach on them before providing it to the experiment participants to use in their projects. Nevertheless, due to time constraints, we cannot guarantee that our approach works seamlessly with all Java software systems and every Java programming aspect. We believe we would mitigate this threat if we had more time to analyze all Java programs that would make sense to be used with our refactoring environment.

Students' opinions: The conclusions drawn from the questionnaires applied on the robotized experiment (*cf.* Section 5.3.4.4, p. 181) are based on the subjective perceptions of the students and may not necessarily reflect the objective effectiveness of refactoring techniques included in our refactoring tool. Students may have varying experiences and understanding of refactoring, which

can influence their answers. Besides, the quality of the code examples provided could impact the students' perceptions.

Lack of geographical diversity: Most participants come from the same region or country. Thus, our results may not generalize to programming students or programmers in other areas. To reduce this threat, we should extend our experiment to other regions or countries to include as many participants as possible with a more diverse geographical diversity.

5.5.3 Construct Validity

Construct validity pertains to the extent to which the methods, tools, or instruments used in a study accurately capture and represent the underlying theoretical constructs or concepts of interest. Essentially, it is about ensuring that our measurements align with the abstract ideas we aim to study [CC94]. Imagine that a researcher seeks to investigate a specific topic, and for that, he needs to apply a survey. If the survey is poorly prepared, its final results may be wrong.

In this specific validity, we found one potential in our experiments:

Questionnaires construction: For each experiment, we had a different survey that the participants should fill out. In the first two experiments, they had to answer a SUS questionnaire to assess the usability of our approach. For our automated experiment, we asked for feedback from a group of students to check whether a specific set of refactorings positively impacted the code. However, it is possible that the format or description of the questions in each survey may have some flaws or may not be precise enough. These factors could have resulted in inaccurate answers, potentially affecting the integrity of our data and, consequently, our research outcomes. To address this potential threat, we can consider conducting pilot tests with a small sample of participants in our future experiments. This will allow us to identify, rectify, and optimize several critical aspects of our questionnaires, ensuring our data collection is more precise and reliable.

5.5.4 Conclusion Validity

Conclusion validity stands as a cornerstone of research integrity, holding the key to ensuring that the conclusions we draw from our studies accurately reflect the underlying reality we aim to explore. It is the steadfast guardian of the inferences we make and the trust we place in our findings. In the intricate tapestry of research, conclusion validity safeguards against errors that might mislead us. Imagine a scientist delving into the effects of a groundbreaking new drug. If conclusion validity falters, it could mean the difference between believing the drug to be a medical breakthrough or a mere placebo [GP12].

In our case, this kind of validity primarily pertains to the conclusions we derive from the data we have gathered and the statistical analysis we have conducted on our second experiment.

Experiment participants: The division of our second experiment participants into different groups was not random, and it might have been influenced by some characteristic related to

the outcome (*e.g.*, final project grade). For instance, if the participants self-select into an experimental group because they are more motivated, this may lead to overestimating the experiment's effectiveness (*e.g.*, by creating overestimating code quality and, probably, introducing biased results on our statistical analysis).

5.6 Summary

We performed three empirical experiments in different stages of our solution to validate our approach and beliefs and answer our research questions.

The first one was carried out in an earlier development stage, in which our *Live Refactoring Environment* only included the *Extract Method* refactoring. We performed this experiment as a try to check if our approach helped improve code quality, faster. We also used its data to evaluate the tool's usability and reliability.

The second one consisted of a final experiment, in which the participants could use our *Live Refactoring Environment* as presented in Chapter 4, (p. 91). With this experiment, we aimed to verify each detail of our approach and collect enough data to help us answer each research question and validate our hypothesis.

The third one was an automated experiment with a GitHub crawler, which inspected and refactored several directories of the most well-evaluated Java GitHub projects. With this experiment, we were to verify that, in most cases, we could improve several code quality metrics and validate that a refactoring environment like ours positively impacts code quality.

With the large amount of data we collected, we could check in both experiments that our approach helped converge to better code, faster. We also verified that it helped create awareness on the blocks of code that should be refactored to improve code quality. We also verified that our live refactoring recommender has good usability and reliability. However, there is still space for improvement.

Even though we had good results that comprised our guidelines and answered each research question positively, we also had to point out several threats to validity that could have harmed our results and conclusions.

In the next chapter, Chapter 6, (p. 191), we summarize all the conclusions and outcomes compiled from this research project.

Chapter 6

Conclusions

6.1	Thesis Statement	191
6.2	Research Questions	192
6.3	Outcomes	194
6.4	Future Work	196

Developers must account for various stages of the software development cycle when implementing their code. The duration of this cycle depends on the time required for each stage, with the implementation phase often being the most time-consuming. This is particularly true for large and complex systems, where the phase’s duration directly correlates with the system’s complexity and size [Cli03]. Low-quality programs exacerbate these challenges, making them harder to understand, adapt, and maintain [LF95, ND86, MS06].

To mitigate these issues, developers frequently review their code to identify refactoring opportunities, addressing poor programming practices or code smells [MG18, Fow18, MCSW07, SLZ08, ADA18]. However, detecting these opportunities can be particularly difficult when the readability of the code is low.

To streamline this process, we proposed an environment designed to identify, suggest, and apply refactorings in real-time. By reducing the effort required to improve code quality, this environment aims to enhance software systems’ readability, adaptability, and maintainability, ultimately enabling developers to create better-quality code more efficiently.

6.1 Thesis Statement

The main guideline of this project was that a *Live Refactoring Environment*, which automatically and continuously reviews and refactors software to enhance them, would positively impact the

feedback and guidance loop between developers and their code. This would result in more readable, adaptable, and maintainable code with more quality, faster.

Thus, this research project was supported by the following statement:

“A development environment that continuously inspects source code, detects and visualizes code smells, and recommends refactoring techniques, in real-time, helps software developers to achieve code with more quality and faster.”

We hypothesized that continuously analyzing software systems in real-time would favor the software development cycle. This approach would enable developers to receive feedback on the outcomes of their programming actions as they implement them, leading to improved support and guidance.

Besides, by continuously identifying code smells and refactoring methods while coding, developers would gain a heightened awareness of the state and quality of their systems. This awareness would allow them to make more informed programming decisions during earlier development stages.

Through efficient and expedited development, developers would be better equipped to oversee their solutions’ final characteristics, architecture, code, and outcomes. Their increased knowledge and awareness of their systems while programming would enable them to create highly readable, adaptable, and maintainable code.

6.2 Research Questions

The main goal of this research project was to enhance and speed-up the feedback, assistance, and guidance offered to developers concerning their code quality. This was accomplished by providing visual recommendations for refactorings that can be automatically applied to the code. To achieve this goal, we needed to incorporate concepts such as liveness, analysis of code quality metrics, and visual identification of refactoring opportunities. To align with the hypothesis stated in Section 3.3, (p. 82) and Section 6.1, (p. 191), we formulated four research questions:

RQ1 *“Is it possible to inform developers, using a color-coded system, about possible refactoring opportunities in their code?”*

This research question aimed to explore if we could use a color-coded approach to present refactoring candidates to developers in a non-intrusive and user-friendly manner while they were programming.

As concluded in Section 5.4, (p. 184), we were able to reject the null hypothesis of this research question. In both experiments, we evaluated the usability and reliability of our tool, and we verified that the participants believed that our tool had good usability. They also concluded that our tool was helpful and intend to use it in the future.

We also evaluated the participants' feedback on our visual methodology to suggest refactorings. The results indicated that they had a favorable opinion of our approach and appreciated the color scheme and placement of the suggestions in the IDE.

Thus, we have concluded that using a color-coded technique like our *Live Refactoring Environment* can aid developers in producing higher-quality code by identifying, recommending, and implementing different refactorings in real-time.

RQ2 “What is the impact of real-time refactoring identification on the developers' awareness of their code quality?”

With this question, we aimed to assess the effect of identifying refactorings, in real-time, on developers' awareness of code quality. Specifically, we seek to check (i) whether the presence of code smells would indeed degrade code quality, and (ii) whether identifying refactorings to mitigate them could assist developers in recognizing when their code had poor quality, making it harder to read, understand, modify, and maintain.

As verified in our empirical experiments, we could reject the null hypothesis for this question. As detailed in Section 5.1.4.2, (p. 128) and as shown in Table 5.14, (p. 152), we verified that our approach improved all evaluated code quality metrics, regardless of the experimental group. Moreover, with Table 5.1, (p. 132) and Section 5.2.3.3, (p. 151), we were able to check that using our *Live Refactoring Environment* helped the participants perform more refactorings, likely due to an increased visual awareness of potential blocks of code with lower quality. Besides, as shown in Table 5.14, (p. 152), in the second experiment, the percentage of undos performed after refactoring the code was very small. As shown in Section 5.2.3.5, (p. 154), using our *Live Refactoring Environment*, the participants from the second experiment created better programming solutions, resulting in better final project grades. Finally, in the third experiment, as seen in Section 5.3.4.4, (p. 181), through the questionnaires applied to programming students regarding their beliefs on the code created by using specific refactoring types, we verified that most of them understood the relevance of each refactoring in improving code quality.

Therefore, we showed that pointing out refactoring candidates, in real-time, positively impacts developers' awareness of their code quality.

RQ3 “What is the impact of a live refactoring environment on their code quality?”

This research question aimed to investigate how constant exposure to refactoring recommendations affects the final quality of the programming solutions.

As mentioned in Section 5.1.4.2, (p. 128) and Section 5.2.3.4, (p. 153), we were able to conclude that by using our *Live Refactoring Environment*, our experiments' participants were able to create code with more quality. Also, by comparing the experimental groups in Section 5.1.4.5, (p. 133) and Section 5.2.3.6, (p. 156), we were able to check that, in fact, for the majority of the code quality metrics assessed in both experiments, the groups that used our approach were statistically different from the ones that did not use it. Finally, in section Section 5.3.4.3, (p. 178), we were able to check that, even when using a robotized refactoring tool that randomly applies refactorings to code,

code quality improves, and the final programming solutions are less complex, more adaptable, and maintainable.

Thus, by observing these results, we confirmed a favorable effect on the code quality. Consequently, we rejected the null hypothesis for this research question and concluded that a *Live Refactoring Environment* positively impacts code quality.

RQ4 “What is the impact of a live refactoring environment on the time required to achieve high-quality programming solutions?”

With this research question, we aimed to study if a *Live Refactoring Environment* impacted the time needed to converge to better programming solutions with more quality.

For that, in the first experiment, we measured the time needed to perform refactoring (second row of Table 5.1, (p. 132)), and we verified that when using our approach, the participants took less time to refactor their code and, therefore, to achieve better-quality solutions. Besides, in Section 5.1.4.5, (p. 133), we also confirmed that the participants who used our tool were statistically different from those who did not. In the second experiment, we compared the evolution of specific code quality metrics over time (cf. Figure 5.10, p. 155), and we verified that the participants that started using our tool in earlier development stages converged to better code, faster.

Then, considering the conclusions drawn in each experiment, we could reject the null hypothesis for this research question. We also concluded that using a *Live Refactoring Environment* positively impacts the time needed to converge to better code.

6.3 Outcomes

This research project contributed to the area of *Software Engineering*, mainly to the topic of *Live Refactoring*.

From this work, we identified four main outcomes: (i) an extensive literature review on the main concepts of our project, (ii) the development of a *Live Refactoring Environment* that helps developers converge to better code, faster, (iii) an empirical experiment that the scientific community can replicate, and (iv) a set of scientific publications summarizing all the work done.

6.3.1 Literature Review

To understand what kind of projects already exist on the main topics of our research, we performed an extensive and comprehensive evaluation of the existing state-of-the-art (cf. Section 2, p. 9). With it, we aimed to identify, analyze, and summarize relevant projects, approaches, or tools on concepts like *software inspection*, *code smells* and *refactorings*, *code quality metrics*, *software visualization*, and *liveness*.

Through it, we could evaluate the strengths, weaknesses, and gaps of several approaches, identifying the key challenges of our main topics and areas where we could go further in our study. We also synthesized the key methodologies used mainly to detect *refactoring candidates* on the

source code, which was a significant part of our project. Finally, by analyzing the state-of-the-art, we were also able to provide a foundation for future research opportunities for ourselves and other researchers.

This literature review allowed us to create a new *refactoring environment* that works in real-time, while programming. With it, we tried to mitigate several open issues, mainly on the visual suggestion and application of specific refactoring candidates (*cf.* Section 4, p. 91).

6.3.2 Live Refactoring Environment

As mentioned in Section 4, (p. 91), dealing with large and complex software can be challenging, mainly when its code has several code smells to mitigate [MHB10, CAGA11].

Development environments like IntelliJ IDE, Eclipse, or Visual Studio Code already offer feedback and direction on enhancing code by refactoring it [Tyn17]. Besides, several tools have already been implemented for those IDEs to also do exactly that [TC09a, FGLD13, Cra, Man]. However, most of them are characterized by their lack of liveness. This implies that developers may not be able to take advantage of these tools until they have already developed code, which could result in suboptimal code quality.

We assume this gap could be filled by providing quicker refactoring guidelines to developers to help them modify, adapt, and maintain their code more efficiently and swiftly, resulting in faster and better code quality.

Thus, another outcome from this research project was the development of a *Live Refactoring Environment*. This environment consisted of a Java plugin for IntelliJ IDE.

Our solution allows inspecting code, in real-time, while programming, to identify some of the most well-known refactoring opportunities, such as *Extract Method*, *Extract Class*, or *Move Method* [Fow18]. To detect these kind of candidates, our environment resorts to several specific algorithms complemented by assessing specific code quality metrics (*cf.* Section 4.2, p. 93). After detecting these candidates, our tool visually suggests them to its users. For that, we developed a visual approach inside the IDE to make more appealing and user-friendly recommendations (*cf.* Section 4.3, p. 116). Then, the tool automatically applies each refactoring when its users accept the suggestions.

6.3.3 Empirical Experiments

Empirical experiments are a direct result of a research project. They are usually conducted to validate the proposed solution or approach to a particular problem. The results of these experiments provide evidence to support the claims made in the thesis and demonstrate the feasibility and effectiveness of the proposed solution [AIUN18, PPV00].

Besides, the experiments conducted during a project may also contribute to advancing knowledge in the field of study. By gathering and analyzing data, researchers can gain new insights, leading to new discoveries or recommendations for future investigation [AIUN18, PPV00].

Therefore, another outcome from this project was the empirical experiments carried out in different research stages (*cf.* Section 5, p. 123). We started by performing a preliminary experiment when our tool only included the *Extract Method* refactoring. This was a small experiment with pre-defined programming and refactoring tasks and with different experimental groups (*cf.* Section 5.1, p. 124). After finishing the development of our refactoring environment and considering the results achieved from the first experiment, we carried out a second experiment. This one was carried out by more participants, without any task or experimental groups defined when it started (*cf.* Section 5.2, p. 143). Finally, we also performed a robotized experiment with an automated version of our refactoring environment that crawled, inspected, and refactored Java GitHub projects (*cf.* Section 5.3, p. 164).

From these experiments, we collected data that helped us validate our assumptions and hypothesis and answer our research questions (*cf.* Section 5.4, p. 184).

Even though their results were satisfactory, we have to point out some internal and external threats to their validity (*cf.* Section 5.5, p. 187), such as the number of refactorings applied, the participants' characterization, or the capabilities already offered by IntelliJ IDE.

6.3.4 Scientific Publications

As in most research projects [ZM22], we tried to submit our work to multiple scientific conferences and journals.

In fact, since our research project yielded significant outcomes, we could publish them in multiple conference proceedings. These publications reflected the depth and breadth of our research, showcasing our commitment to advancing knowledge in diverse domains. Each one of these publications is detailed in (*cf.* Appendix A, p. 225).

6.4 Future Work

Even though we feel pretty confident with the work carried out, we know that there are still some points related to our main contributions that could have been further explored. Next, we list some of those issues and how they can be addressed in future research.

Regarding the study of the state-of-the-art, we recognize that we may have missed the analysis of some refactoring approaches or tools. As known, much research is already focused on refactorings. Since the concepts of *code smell* and *refactoring* date back to the year of 1999 [FB99], we expect that many researchers have already studied these topics and their main limitations. Indeed, even having carried out an extensive literature review on them, there is probably still work that remains to be analyzed, which could have been relevant to this project. The same happens with the remaining topics and sub-topics that are included in this project, such as *code quality metrics*, *software visualization*, or *liveness*. However, we believe that in the future, we can carry out a more in-depth systematic literature review for each concept to obtain the most significant number of examples of related work possible to help us improve our approach and mitigate its limitations.

Considering our solution presented as a *Live Refactoring Environment*, we believe there is still room for improvement. Even having already validated our main assumptions and answered our research questions, we know there are limitations we should also explore in the future (cf. Section 4.5, p. 120). The first limitation is the number of refactorings included in our solution. Our approach identifies refactoring candidates like *Extract Method*, *Extract Class*, *Move Method*, or *Replace Inheritance with Delegation*. However, by including limited refactorings, we are probably detecting a narrower spectrum of code smells. Then, we might reduce the number of opportunities to enhance the code quality and its readability, adaptability, and maintainability.

Secondly, further research should also address the time needed to inspect code and identify, suggest, and apply each refactoring. Right now, our approach takes a few seconds to perform our refactoring cycle (e.g., no more than two seconds to find *Extract Variable* or *String Comparison* refactorings). However, refactorings like *Extract Class* may take more than that. We found that if we analyze multiple factors simultaneously, we might increase the time needed to point out a set of possible refactoring candidates. Besides, as the size and complexity of the code increase, the refactoring cycle also increases. We already use threads to allow our approach to detect and visually suggest each refactoring type in concurrent mode. However, we believe a solution like ours needs more than that to reduce the time required to identify each refactoring in just a few seconds or milliseconds.

We also believe several other aspects could be included in our approach to make it smarter and more accurate. The first is to predict how the quality metrics would look after each refactoring. This would help predict whether or not a refactoring would be beneficial to the code and help include it or not in the final set of refactorings to be suggested to developers. And it would also help to order them among themselves regardless of their type. Thus, refactorings with the greatest impact on the code would be suggested in red according to the range of colors used in our visual methodology (cf. Section 4.3, p. 116).

Another point that can be addressed in the future is the use of smart methodologies to not only identify each refactoring but also to suggest optimal refactoring sequences. We believe that using *Artificial Intelligence* algorithms such as those already used by Kurbatova *et al.* [KVGB20] or Tarwani and Chug [TC16, CT17] can help our approach detect better refactoring opportunities and exclude false-positive suggestions. Besides that, we also believe that using *Artificial Intelligence* or *Machine Learning* can help fine-tune the thresholds used to identify possible refactoring candidates, making our approach more adaptive, data-driven, and context-aware.

In the future, we can also try to identify refactoring opportunities based on other refactorings already applied to the code. Imagine that a developer applies more *Extract Method* refactorings than *Extract Classes*. We would give greater relevance to *Extract Methods* and suggest them first than *Extract Classes*. Or, we could consider the refactorings applied and suggest others that would complement them to improve even more the impact they have on the code (e.g., after applying a *Extract Variable*, we may automatically suggest an *Extract Method* on the method where we extracted a variable).

Still related to our solution, we think it would be a good idea to include an additional interface

in which developers could see the evolution of each metric over time and each refactoring applied like the one schematized in Figure 6.1, (p. 198). That way, they would have an idea of the quality of their code and a better perception of the impact of each refactoring and whether or not it would have been beneficial to their code.

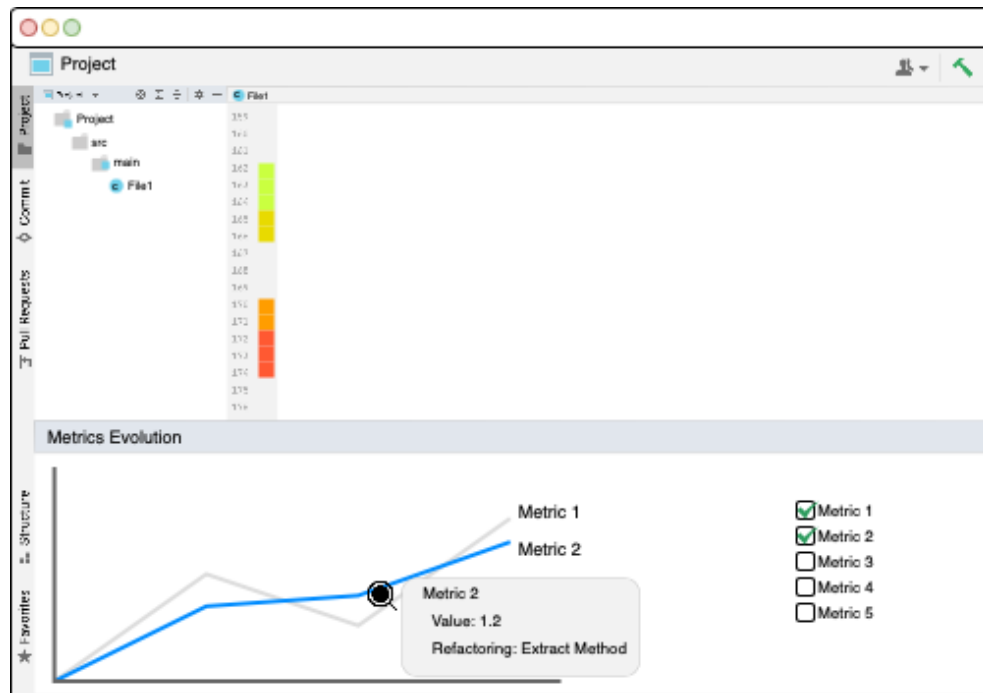


Figure 6.1: Mockup of a possible interface to display the evolution of specific code quality metrics.

In addition to future work related to our refactoring environment, we can also point out things to be addressed regarding our empirical experiment. Since the visual approach we implemented is a vital part of our approach (*cf.* Section 4.3, p. 116), we believe it will be essential to have an experiment solely focused on our tool's usability. We would also like to test our solution with colorblind developers to see if the colorblind approach we selected is inclusive enough to them.

Finally, we can also explore integrating large language models (LLMs) [BMR⁺20] to assist with aspects such as suggesting refactorings or explaining code transformations. However, the core of this project, which is the live refactoring driven by code quality metrics, ensuring that refactorings happen dynamically and in response to real-time feedback, would remain the same. While LLMs could enhance usability and automation, they would not change the fundamental focus on liveness and metric-based decision-making.

References

- [AAA⁺11] I. Alexandrov, Vasco Amaral, Antonio Amorim, E. Badescu, D. Burckhart, Mihai Caprini, L. Cohen, P. Duval, R. Hart, R Jones, A. Kazarov, Serguei Kolos, V. Kotov, D. Laugier, L. Mapelli, L Moneta, Zhanq Qian, C. Ribeiro, V. Rumyantsev, and Ilia Solovev. Impact of software review and inspection. *Computing In High Energy Physics And Nuclear Physics*, 05 2011.
- [AAG19] Mohammed Alqmase, Mohammad Alshayeb, and Lahouari Ghouti. Threshold extraction framework for software metrics. *Journal of Computer Science and Technology*, 34:1063–1078, 09 2019.
- [AAJ⁺18] Nour AlAbwaini, Amal Aldaaje, Tamara Jaber, Mohammad Abdallah, and Abdelfatah Tamimi. Using Program Slicing to Detect the Dead Code. In *2018 8th International Conference on Computer Science and Information Technology (CSIT)*, pages 230–233. IEEE, jul 2018.
- [ABKS99] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. OPTICS. *ACM SIGMOD Record*, 28(2):49–60, jun 1999.
- [Aby] Hatica Aravind Aby. ChatGPT-4 For Developers: How To Debug, Refactor, and Fix Code.
- [AD07] S. Alam and P. Dugerdil. Evospaces visualization tool: Exploring software architecture in 3d. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 269–270, Oct 2007.
- [ADA18] Jehad Al Dallah and Anas Abdin. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering*, 44(1):44–69, 2018.
- [AFE84] A. Frank Ackerman, Priscilla J. Fowler, and Robert G. Ebenau. Software inspections and the industrial production of software. In *Proc. of a Symposium on Software Validation: Inspection-Testing-Verification-Alternatives*, page 13–40, USA, 1984. Elsevier North-Holland, Inc.
- [AIK⁺23] Eman Abdullah Alomar, Anton Ivanov, Zarina Kurbatova, Yaroslav Golubev, Mohamed Wiem Mkaouer, Ali Ouni, Timofey Bryksin, Le Nguyen, Amit Kini, and Aditya Thakur. Anticopypaster: Extracting code duplicates as soon as they are introduced in the ide. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, ASE '22*, New York, NY, USA, 2023. Association for Computing Machinery.

- [AIUN18] Talat Ambreen, Naveed Ikram, Muhammad Usman, and Mahmood Niazi. Empirical research in requirements engineering: trends and opportunities:. *Requirements Engineering*, 23, 03 2018.
- [Al 12] Jihad Al Dallal. Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics. *Information and Software Technology*, 54(10):1125–1141, 2012.
- [Al 15] Jihad Al Dallal. Identifying refactoring opportunities in object-oriented code: A systematic literature review. *Information and Software Technology*, 58:231–249, 2015.
- [AL99] N. Anquetil and T.C. Lethbridge. Experiments with clustering as a software re-modularization method. In *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*, pages 235–255, 1999.
- [AL20] Nabil Almashfi and Lunjin Lu. Code smell detection tool for java script programs. In *2020 5th International Conference on Computer and Communication Systems (ICCCS)*, pages 172–176, 2020.
- [ALT⁺21] Hassan Atwi, Bin Lin, Nikolaos Tsantalis, Yutaro Kashiwa, Yasutaka Kamei, Naoyasu Ubayashi, Gabriele Bavota, and Michele Lanza. Pyref: Refactoring detection in python projects. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 136–141, 2021.
- [Alz20] Musaad Alzahrani. Measuring class cohesion based on client similarities between method pairs: An improved approach that supports refactoring. *IEEE Access*, 8:227901–227914, 2020.
- [Ama18] Diogo Amaral. Towards a live software development environment. Master’s thesis, Faculty of Engineering of the University of Porto, Porto, Portugal, 2018.
- [AOKC19] Vahid Alizadeh, Mohamed Amine Ouali, Marouane Kessentini, and Meriem Chater. Refbot: Intelligent software refactoring bot. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 823–834, 2019.
- [AP12] Saeed Aghaee and Cesare Pautasso. Englishmash: Usability design for a natural mashup composition environment. In Michael Grossniklaus and Manuel Wimmer, editors, *Current Trends in Web Engineering*, pages 109–120, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [AP13] Saeed Aghaee and Cesare Pautasso. Live mashup tools: Challenges and opportunities. *2013 1st International Workshop on Live Programming, LIVE 2013 - Proceedings*, pages 1–4, 2013.
- [ARC⁺19] Ademar Aguiar, André Restivo, Filipe Figueiredo Correia, Hugo Sereno Ferreira, and João Pedro Dias. Live software development — tightening the feedback loops. In *Proceedings of the 5th Programming Experience (PX) Workshop*, apr 2019.
- [ASK14] Everton L. G. Alves, Myoungkyu Song, and Miryung Kim. Refdistiller: A refactoring aware code review tool for inspecting manual refactoring edits. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014*, page 751–754, New York, NY, USA, 2014. Association for Computing Machinery.

- [ASM⁺18] Everton L. G. Alves, Myoungkyu Song, Tiago Massoni, Patrícia D. L. Machado, and Miryung Kim. Refactoring inspection support for manual refactoring edits. *IEEE Transactions on Software Engineering*, 44(4):365–383, 2018.
- [Ass13] World Medical Association. World Medical Association Declaration of Helsinki: Ethical Principles for Medical Research Involving Human Subjects. *JAMA*, 310(20):2191–2194, 11 2013.
- [AT16] Shumail Arshad and Christos Tjortjis. Clustering software metric values extracted from c# code for maintainability assessment. In *Proceedings of the 9th Hellenic Conference on Artificial Intelligence*, SETN '16, New York, NY, USA, 2016. Association for Computing Machinery.
- [Bae98] Ronald Baecker. Appears in *Software Visualization: Programming as a Multimedia Experience*. Technical report, MIT - Massachusetts Institute of Technology, 1998.
- [Bar20] João Paulo Moreira Barbosa. Towards a smart recommender for code refactoring. Master's thesis, Faculty of Engineering of the University of Porto, Porto, Portugal, 2020.
- [BBM96] V.R. Basili, L.C. Briand, and W.L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.
- [BD02a] J. Bansiya and C.G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, jan 2002.
- [BD02b] J. Bansiya and C.G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.
- [BDLM⁺12] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, Rocco Oliveto, and Fabio Palomba. Supporting extract class refactoring in eclipse: The aries project. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1419–1422, 2012.
- [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, 2011.
- [BFdH⁺13] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. It's alive! continuous feedback in ui programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 95–104, New York, NY, USA, 2013. Association for Computing Machinery.
- [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.
- [Bie89] Irving Biederman. "Recognition-by-components: A theory of human image understanding": Clarification. *Psychological Review*, 96(1), jan 1989.

- [BJB⁺20] Christopher Brown, Vladimir Janjic, Adam Barwell, J. Garcia, and Kenneth MacKenzie. Refactoring grppi: Generic refactoring for generic parallelism in c++. *International Journal of Parallel Programming*, 48, 08 2020.
- [BKR⁺21] Tom Beckmann, Eva Krebs, Patrick Rein, Stefan Ramson, and Robert Hirschfeld. Shortening feedback loops in a live game development environment. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–5, 2021.
- [BLD15] Benjamin Biegel, Benedikt Lesch, and Stephan Diehl. Live object exploration: Observing and manipulating behavior and state of Java objects. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 581–585. IEEE, sep 2015.
- [BMR⁺20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.
- [BOD⁺10] Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, Giuliano Antoniol, and Yann Gaël Gúehéneuc. Playing with refactoring: Identifying extract class opportunities through game theory. *IEEE International Conference on Software Maintenance, ICSM*, 2010.
- [Bri88] R. N. Britcher. Using inspections to investigate program correctness. *Computer*, 21(11):38–44, 1988.
- [Bro13] John Brooke. Sus: A retrospective. *J. Usability Studies*, 8(2):29–40, feb 2013.
- [Bry99] Bill Brykczynski. A survey of software inspection checklists. *SIGSOFT Softw. Eng. Notes*, 24(1):82, January 1999.
- [BT14] L. Bose and S. Thakur. Graft: Generic amp; reusable automation framework for agile testing. In *2014 5th International Conference - Confluence The Next Generation Information Technology Summit (Confluence)*, pages 761–766, Sep. 2014.
- [BV21] Rodrigo Brito and Marco Túlio Valente. RAID: tool support for refactoring-aware code reviews. *CoRR*, abs/2103.11453, 2021.
- [BVZ⁺19] Anke Brocker, Simon Voelker, Tony Zelun Zhang, Mathis Müller, and Jan Borchers. Flowboard: A visual flow-based programming environment for embedded coding. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI EA ’19, page 1–4, New York, NY, USA, 2019. Association for Computing Machinery.
- [BW93] Bill Brykczynski and David A. Wheeler. An annotated bibliography on software inspections. *SIGSOFT Softw. Eng. Notes*, 18(1):81–88, January 1993.
- [CAC⁺17] Sofia Charalampidou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Antonios Gkortzis, and Paris Avgeriou. Identifying extract method refactoring opportunities based on functional relevance. *IEEE Transactions on Software Engineering*, 43(10):954–974, 2017.

- [CAGA11] Keith Cassell, Craig Anslow, Lindsay Groves, and Peter Andreae. Visualizing the refactoring of classes via clustering. *Conferences in Research and Practice in Information Technology Series*, 113(Acsc):63–72, 2011.
- [CALO94] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, August 1994.
- [Cam18] G. Ann Campbell. Cognitive complexity: An overview and evaluation. In *Proceedings of the 2018 International Conference on Technical Debt*, TechDebt ’18, page 57–58, New York, NY, USA, 2018. Association for Computing Machinery.
- [CC94] Kendon J. Conrad and Karen M. Conrad. Reassessing validity threats in experiments: Focus on construct validity. *New Directions for Program Evaluation*, 1994(63):5–25, 1994.
- [CD10] Christophe Croux and Catherine Dehon. Influence functions of the spearman and kendall correlation measures. *Statistical Methods and Applications*, 19:497–515, 11 2010.
- [CFYH18] Eunjong Choi, Kenji Fujiwara, Norihiro Yoshida, and Shinpei Hayashi. A survey of refactoring detection techniques based on change history analysis. *JSSST journal Computer Software*, 08 2018.
- [Chr17] Google Chrome. Chrome DevTools, 2017. <https://developers.google.com/web/tools/chrome-devtools/>, 2017.
- [CK91] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. *ACM SIGPLAN Notices*, 26(11):197–211, nov 1991.
- [CK94] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, jun 1994.
- [Cle03] Tony Clear. The waterfall is dead.: Long live the waterfall!! *SIGCSE Bull.*, 35(4):13–14, dec 2003.
- [Cli03] Victor A. Clincy. Software development productivity and cycle time reduction. *J. Comput. Sci. Coll.*, 19(2):278–287, dec 2003.
- [CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- [Cod] CodeMR. Codemr. <https://plugins.jetbrains.com/plugin/10811-codemr>.
- [CP13] G. Ann Campbell and Patroklos P. Papapetrou. *SonarQube in Action*. Manning Publications Co., USA, 1st edition, 2013.
- [Cra] Thomas Craig. Super sharp. <https://marketplace.visualstudio.com/items?itemName=craigthomas.supersharpp>.
- [CRT18] Christian Marlon Souza Couto, Henrique Rocha, and Ricardo Terra. A quality-oriented approach to recommend move method refactorings. In *Proceedings of the 17th Brazilian Symposium on Software Quality*, SBQS, page 11–20, New York, NY, USA, 2018. Association for Computing Machinery.

- [CT17] Anuradha Chug and Sandhya Tarwani. Determination of optimum refactoring sequence using A* algorithm after prioritization of classes. *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 2017-Janua:1624–1630, sep 2017.
- [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.
- [DAN18] Atish Kumar Dipongkor, Iftekhar Ahmed, and Nadia Nahar. Move method recommendation using call frequency of methods and attributes. In *2018 Joint 7th International Conference on Informatics, Electronics & Vision (ICIEV) and 2018 2nd International Conference on Imaging, Vision & Pattern Recognition (icIVPR)*, pages 76–81, 2018.
- [DeL21] Robert A DeLine. Glinda: Supporting data science with live programming, guis and a domain-specific language. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI ’21, New York, NY, USA, 2021. Association for Computing Machinery.
- [dF24] Diogo Luís Araujo de Faria. Mining software repositories to improve refactoring assistants. Master’s thesis, Faculty of Engineering of the University of Porto, Porto, Portugal, 2024.
- [DL05] S. Ducasse and M. Lanza. The class blueprint: visually supporting the understanding of glasses. *IEEE Transactions on Software Engineering*, 31(1):75–90, 2005.
- [dO24] Margarida Rocha Raposo de Oliveira. Spatialized live refactoring. Master’s thesis, Faculty of Engineering of the University of Porto, Porto, Portugal, 2024.
- [DPT⁺18] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia. Detecting code smells using machine learning techniques: Are we there yet? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 612–621, 2018.
- [DSZ⁺23] Sunhao Dai, Ninglu Shao, Haiyuan Zhao, Weijie Yu, Zihua Si, Chen Xu, Zhongxiang Sun, Xiao Zhang, and Jun Xu. Uncovering chatgpt’s capabilities in recommender systems. In *Proceedings of the 17th ACM Conference on Recommender Systems*, RecSys ’23, page 1126–1132, New York, NY, USA, 2023. Association for Computing Machinery.
- [Dua22] Carlos Eduardo da Nova Duarte. Towards live refactoring to patterns. Master’s thesis, Faculty of Engineering of the University of Porto, Porto, Portugal, 2022.
- [DYD19] Ananta Kumar Das, Shikhar Yadav, and Subhasish Dhal. Detecting code smells using deep learning. In *TENCON 2019 - 2019 IEEE Region 10 Conference (TENCON)*, pages 2081–2086, 2019.
- [Eas07] Steve Easterbrook. Empirical research methods for software engineering. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ASE ’07, page 574, New York, NY, USA, 2007. Association for Computing Machinery.

- [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.
- [Fag76] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.
- [FAR22] Sara Fernandes, Ademar Aguiar, and André Restivo. A live environment to improve the refactoring experience. In *Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming*, Programming '22, page 30–37, New York, NY, USA, 2022. Association for Computing Machinery.
- [FAR23a] Sara Fernandes, Ademar Aguiar, and André Restivo. Liveref: A tool for live refactoring java code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.
- [FAR23b] Sara Fernandes, Ademar Aguiar, and André Restivo. Empirical evaluation of a live environment for extract method refactoring, 2023.
- [FB99] Martin Fowler and Kent Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [Fer19] Sara Filipa Couto Fernandes. Supporting software development through live metrics visualization. Master's thesis, Faculty of Engineering of the University of Porto, Porto, Portugal, 2019.
- [Fer21] Sara Fernandes. A live environment for inspection and refactoring of software systems. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1655–1659, New York, NY, USA, 2021. Association for Computing Machinery.
- [Fer23] Sara Fernandes. Towards a live environment for code refactoring. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.
- [FGLD13] Lyle Franklin, Alex Gyori, Jan Lahoda, and Danny Dig. LambdaFicator: From imperative to functional programming through automated refactoring. *Proceedings - International Conference on Software Engineering*, pages 1287–1290, 2013.
- [Fis13] Andrew Fischer. Introducing circa: A dataflow-based language for live coding. *2013 1st International Workshop on Live Programming, LIVE 2013 - Proceedings*, pages 5–8, 2013.
- [FLC⁺02] Carla M. D. S. Freitas, Paulo R. G. Luzzardi, Ricardo A. Cava, Marco Winckler, Marcelo S. Pimenta, and Luciana P. Nedel. On evaluating information visualization techniques. In *Proceedings of the Working Conference on Advanced Visual Interfaces - AVI '02*, page 373, New York, New York, USA, 2002. ACM Press.

- [FMS⁺17] Wolfram Fenske, Jens Meinicke, Sandro Schulze, Steffen Schulze, and Gunter Saake. Variant-preserving refactorings for migrating cloned products to a product line. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 316–326. IEEE, feb 2017.
- [FOP⁺20a] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. Small-step live programming by example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology, UIST '20*, page 614–626, New York, NY, USA, 2020. Association for Computing Machinery.
- [FOP⁺20b] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. Small-step live programming by example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology, UIST '20*, page 614–626, New York, NY, USA, 2020. Association for Computing Machinery.
- [Fow15] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley, New York, NY, 2015.
- [Fow18] Martin Fowler. *Refactoring: Improving the Design of Existing Code (2nd Edition)*. Addison-Wesley, 2018.
- [FRA20] Sara Fernandes, André Restivo, , and Ademar Aguiar. Live software inspection and refactoring. In *8th SEDES - Software Engineering Doctoral Symposium, '8*. CEUR Workshop Proceedings, 2020.
- [FRFA20] Sara Fernandes, André Restivo, Hugo Sereno Ferreira, and Ademar Aguiar. Helping software developers through live software metrics visualization. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming, '20*, page 209–210, New York, NY, USA, 2020. Association for Computing Machinery.
- [FTC07] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. JDeodorant: Identification and Removal of Feature Envy Bad Smells. *2007 IEEE International Conference on Software Maintenance*, pages 519–520, oct 2007.
- [FTSC11] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. JDeodorant: Identification and application of extract class refactorings. *Proceedings - International Conference on Software Engineering*, pages 1037–1039, 2011.
- [GAM19] Thirupathi Guggulothu and Salman Abdul Moiz. Detection of shotgun surgery and message chain code smells using machine learning techniques. *International Journal of Rough Sets and Data Analysis*, 6:34–50, 04 2019.
- [GC14] Jean Gibbons and Subha Chakraborti. Comparisons of the mann-whitney, student's t , and alternate t tests for means of normal distributions. *The Journal of Experimental Education*, 59:258–267, 04 2014.
- [Gel08] David Gelperin. Exploring agile. In *Proceedings of the 2008 International Workshop on Scrutinizing Agile Practices or Shoot-out at the Agile Corral, APOS '08*, page 1–3, New York, NY, USA, 2008. Association for Computing Machinery.
- [GGFN18] Pascal Gadiant, Mohammad Ghafari, Patrick Frischknecht, and Oscar Nierstrasz. Security code smells in android ICC. *CoRR*, abs/1811.12713, 2018.

- [GGGR18] Julián Grigera, Juan Cruz Gardey, Alejandra Garrido, and Gustavo Rossi. Live versioning of web applications through refactoring. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*, pages 872–875, New York, New York, USA, 2018. ACM Press.
- [GLB⁺21] Yaroslav Golubev, Jiawei Li, Timofey Bryksin, Viacheslav Bushev, and Iftekhar Ahmed. Changes from the trenches: Should we automate them? *CoRR*, abs/2105.10157, 2021.
- [GME05] Denis Gračanin, Krešimir Matković, and Mohamed Eltoweissy. Software visualization. *Innovations in Systems and Software Engineering*, 1(2):221–230, sep 2005.
- [GMH11] Xi Ge and Emerson Murphy-Hill. BeneFactor. *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH '11*, page 19, 2011.
- [GMNP15] Sumit Gulwani, Mikaël Mayer, Filip Niksic, and Ruzica Piskac. Strisynth: Synthesis for live programming. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 701–704, 2015.
- [Gol04] K.J. Goldman. Live software development with dynamic classes. *Washington University, Department of Computer Science and Engineering, Technical Report TR-2004-81*, 81:1–19, 2004.
- [GP12] Miguel García-Pérez. Statistical conclusion validity: Some common threats and simple remedies. *Frontiers in Psychology*, 3, 2012.
- [GS06] G. Gui and P. D. Scott. Coupling and cohesion measures for evaluation of component reusability. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, page 18–21, New York, NY, USA, 2006. Association for Computing Machinery.
- [Gug19] Thirupathi Guggulothu. Code smell detection using multilabel classification approach. *CoRR*, abs/1902.03222, 2019.
- [GWL16] F. Gao, L. Wang, and X. Li. Bovinspector: Automatic inspection and repair of buffer overflow vulnerabilities. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 786–791, 2016.
- [Hai59] Lois M. Haibt. A program to draw multilevel flow charts. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference, IRE-AIEE-ACM '59 (Western)*, pages 131–137, New York, NY, USA, 1959. ACM.
- [Hal77a] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., USA, 1977.
- [Hal77b] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.
- [Han03] Christopher Michael Hancock. *Real-time Programming and the Big Ideas of Computational Literacy*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2003. AAI0805688.

- [Hay13] Winston Haynes. *Bonferroni Correction*, pages 154–154. Springer New York, New York, NY, 2013.
- [HCL05] Jeffrey Heer, Stuart K. Card, and James A. Landay. prefuse - A Toolkit for Interactive Information Visualization. *Proceedings of the Conference on Human Factors in Computing Systems (CHI'05)*, pages 421–430, 2005.
- [HFJ⁺19] Rui Hao, Yang Feng, James A. Jones, Yuying Li, and Zhenyu Chen. Ctras: Crowd-sourced test report aggregation and summarization. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 900–911, 2019.
- [HJR⁺16] Felienne Hermans, Bas Jansen, Sohoni Roy, Efthimia Aivaloglou, Alaaeddin Swidan, and David Hoepelman. Spreadsheets are Code: An Overview of Software Engineering Approaches Applied to Spreadsheets. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 5:56–65, 2016.
- [HK81] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, SE-7(5):510–518, 1981.
- [HKA10] Nicolas Haderer, Foutse Khomh, and Giuliano Antoniol. SQUANER: A framework for monitoring the quality of software systems. *IEEE International Conference on Software Maintenance, ICSM*, pages 0–3, 2010.
- [HMC⁺18] Austin Z. Henley, Kİvanç Muçlu, Maria Christakis, Scott D. Fleming, and Christian Bird. Cfar: A tool to increase communication, productivity, and review quality in collaborative code reviews. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI '18*, page 1–13, New York, NY, USA, 2018. Association for Computing Machinery.
- [Hoa81] C.A.R Hoare. The Emperor's Old Clothes. <http://zoo.cs.yale.edu/classes/cs422/2011/bib/hoare81emperor.pdf>, 1981.
- [HS02] Brian Henderson-Sellers. *Object oriented metrics: Measures of complexity*. Prentice Hall PTR, 2002.
- [HSHV12] Cameron Hine, Jean-Guy Schneider, Jun Han, and Steve Versteeg. Quokka: visualising interactions of enterprise software environment emulators. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, page 370, New York, New York, USA, 2012. ACM Press.
- [HSS17] Thorsten Haendler, Stefan Sobernig, and Mark Strembeck. Towards triaging code-smell candidates via runtime scenarios and method-call dependencies. In *Proceedings of the XP2017 Scientific Workshops, XP '17*, New York, NY, USA, 2017. Association for Computing Machinery.
- [HvdS16] Felienne Hermans and Tijs van der Storm. Truegrid: Code the table, tabulate the data. In *Software Technologies: Applications and Foundations*, volume 9946, pages 388–393, 07 2016.
- [IHHK17] Ayaka Imazato, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Finding extract method refactoring opportunities by analyzing development history. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 190–195, 2017.

- [IMA15] Tomoki Imai, Hidehiko Masuhara, and Tomoyuki Aotani. Making live programming practical by bridging the gap between trial-and-error development and unit testing. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH Companion 2015, page 11–12, New York, NY, USA, 2015. Association for Computing Machinery.
- [IW00] Pourang Irani and Colin Ware. Diagrams based on structural object perception. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '00, page 61–67, New York, NY, USA, 2000. Association for Computing Machinery.
- [JC18] J. Jenkins and H. Cai. Icc-inspect: Supporting runtime inspection of android inter-component communications. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 80–83, 2018.
- [JRe] JRebel. The best java ides to use in 2023.
- [Kay93] Alan C. Kay. The early history of smalltalk. *SIGPLAN Not.*, 28(3):69–95, March 1993.
- [KBG16] Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey. Code review quality: How developers see it. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 1028–1038, New York, NY, USA, 2016. Association for Computing Machinery.
- [KECK12] Ekin Koc, Nur Ersoy, Zelal Seda Camlidere, and Hurevren Kilic. A web-service for automated software refactoring using artificial bee colony optimization. In Ying Tan, Yuhui Shi, and Zhen Ji, editors, *Advances in Swarm Intelligence*, pages 318–325, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [KG18] Jun Kato and Masataka Goto. Deployground: A framework for streamlined programming from api playgrounds to application deployment. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 259–263, 2018.
- [KHFG16] István Kádár, Péter Hegedundefineds, Rudolf Ferenc, and Tibor Gyimóthy. A manually validated code refactoring dataset and its assessment regarding software maintainability. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2016, New York, NY, USA, 2016. Association for Computing Machinery.
- [KHIK12] Shuhei Kimura, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Move code refactoring with dynamic analysis. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 575–578, 2012.
- [KK15] Tae Kyun Kim. T test as a parametric statistic. *Korean Journal of Anesthesiology*, 68:540, 11 2015.
- [KKS⁺21] Z. Kurbatova, V. Kovalenko, I. Savu, B. Brockbernd, D. Andreescu, M. Anton, R. Venediktov, E. Tikhomirova, and T. Bryksin. Refactorinsight: Enhancing ide representation of changes in git with refactorings information. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1276–1280, Los Alamitos, CA, USA, nov 2021. IEEE Computer Society.

- [KM00] 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, July 2000.
- [Knu63] Donald E. Knuth. Computer-drawn flowcharts. *Commun. ACM*, 6(9):555–563, September 1963.
- [Kon] Jasper Koning. Coremetrics. <https://marketplace.visualstudio.com/items?itemName=jasper.coremetrics>.
- [Kos] Vincent Kos. Elegant php. <https://marketplace.visualstudio.com/items?itemName=vincentkos.elegant-php>.
- [KRB18] J. Kubelka, R. Robbes, and A. Bergel. The road to live programming: Insights from the practice. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1090–1101, 2018.
- [Kuh13] Adrian Kuhn. On extracting unit tests from interactive live programming sessions. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1241–1244, 2013.
- [Kum14] R. Kumar. *Research Methodology: A Step-by-Step Guide for Beginners*. Sage Publications, 4th edition, 2014.
- [KVGB20] Zarina Kurbatova, Ivan Veselov, Yaroslav Golubev, and Timofey Bryksin. Recommendation of move method refactoring using path-based representation of code, 2020.
- [LCJ13] Yun Young Lee, Nicholas Chen, and Ralph E. Johnson. Drag-and-drop refactoring: Intuitive and efficient program transformation. *Proceedings - International Conference on Software Engineering*, pages 23–32, may 2013.
- [LD01] Michele Lanza and Stéphane Ducasse. A categorization of classes based on the visualization of their internal structure: The class blueprint. *SIGPLAN Not.*, 36(11):300–311, October 2001.
- [Ler20a] Sorin Lerner. Focused live programming with loop seeds. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology, UIST '20*, page 607–613, New York, NY, USA, 2020. Association for Computing Machinery.
- [Ler20b] Sorin Lerner. Projection boxes: On-the-fly reconfigurable visualization for live programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems, CHI '20*, page 1–7, New York, NY, USA, 2020. Association for Computing Machinery.
- [LF95] Henry Lieberman and Christopher Fry. Bridging the gulf between code and behavior in programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '95*, pages 480–486, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.
- [Li95] Xiaolin Li. A comparison-based approach for software inspection. In *Proceedings of the 1995 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON '95*, page 41. IBM Press, 1995.

- [LJX⁺21] Hui Liu, Jiahao Jin, Zhifeng Xu, Yanzhen Zou, Yifan Bu, and Lu Zhang. Deep learning based code smell detection. *IEEE Transactions on Software Engineering*, 47(9):1811–1837, 2021.
- [LL07] Mircea Lungu and Michele Lanza. Exploring inter-module relationships in evolving software systems. In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 91–102, 2007.
- [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.
- [LLG⁺17] Yibin Liu, Yanhui Li, Jianbo Guo, Yuming Zhou, and Baowen Xu. Connecting software metrics across versions to predict defects. *CoRR*, abs/1712.09835, 2017.
- [LM16] P. Lakshmi and T Maheswari. An effective rank approach to software defect prediction using software metrics. *2016 10th International Conference on Intelligent Systems and Control (ISCO)*, pages 1–5, 2016.
- [LMD06] Michele Lanza, Radu Marinescu, and Stéphane Ducasse. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
- [Lou18] Pedro Lourenço. Towards a live management of cloud infrastructures. Master’s thesis, Faculty of Engineering of the University of Porto, Porto, Portugal, 2018.
- [LT11] Huiqing Li and Simon Thompson. Incremental clone detection and elimination for erlang programs. In Dimitra Giannakopoulou and Fernando Orejas, editors, *Fundamental Approaches to Software Engineering*, pages 356–370, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [LTOT08] Huiqing Li, Simon Thompson, George Orösz, and Melinda Tóth. Refactoring with wrangler, updated data and process refactorings, and integration with eclipse. In *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*, pages 61–72, 01 2008.
- [LXW18] Jeremy Ludwig, Steven Xu, and Frederick Webber. Static software metrics for reliability and maintainability. In *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 53–54, 2018.
- [MAB⁺12a] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, and Esma Aïmeur. Smurf: A svm-based incremental anti-pattern detection approach. In *2012 19th Working Conference on Reverse Engineering*, pages 466–475, 2012.
- [MAB⁺12b] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Esma Aïmeur. Support vector machines for anti-pattern detection. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 278–281, 2012.
- [Mac] Virtual Machinery. Maintainability index. <http://www.virtualmachinery.com/sidebar4.htm>.

- [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.
- [Man] Dan Mannock. Vscod fsharp refactor. <https://marketplace.visualstudio.com/items?itemName=danmannock.vscod-fsharp-refactor>.
- [Mar03] Robert C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, Hoboken, New Jersey, USA, 2003.
- [Mar04] R. Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 350–359, 2004.
- [Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, USA, 1 edition, 2008.
- [Mar15] Ethan Marcotte. *Responsive design: Patterns and Principles*. A Book Apart, New York, NY, 2015.
- [MBM⁺19] Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo Pinto. Are static analysis violations really fixed? a closer look at realistic usage of sonarqube. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 209–219, 2019.
- [MC12] Iman Hemati Moghadam and Mel Ó Cinnéide. Automated refactoring using design differencing. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 43–52, 2012.
- [MC19] Andrew R Martin and Simon Colton. Towards liveness in game development. In *2019 IEEE Conference on Games (CoG)*, pages 1–4, 2019.
- [McC76] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.
- [McD07] Sean McDirmid. Living it up with a live programming language. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, OOPSLA ’07*, page 623–638, New York, NY, USA, 2007. Association for Computing Machinery.
- [McD13a] Sean McDirmid. Usable live programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! 2013*, page 53–62, New York, NY, USA, 2013. Association for Computing Machinery.
- [McD13b] Sean McDirmid. Usable live programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2013*, page 53–62, New York, NY, USA, 2013. Association for Computing Machinery.

- [MCSW07] Erica Mealy, David Carrington, Paul Strooper, and Peta Wyeth. Improving Usability of Software Refactoring Tools. In *2007 Australian Software Engineering Conference (ASWEC'07)*, pages 307–318. IEEE, apr 2007.
- [Mea12] Panita Meananeatra. Identifying Refactoring Sequences for Improving Software Maintainability Categories and Subject Descriptors The Scope of Our Approach. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 406–409, 2012.
- [MF16] Sebastian C. Müller and Thomas Fritz. Using (bio)metrics to predict code quality online. In *Proceedings of the 38th International Conference on Software Engineering, ICSE '16*, page 452–463, New York, NY, USA, 2016. Association for Computing Machinery.
- [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–ff, New York, NY, USA, 2003. Association for Computing Machinery.
- [MG18] Michael Mohan and Des Greer. A survey of search-based refactoring for software maintenance. *Journal of Software Engineering Research and Development*, 6(1):3, Feb 2018.
- [MGM19] Michael Mohan, Des Greer, and Paul McMullan. Maximizing refactoring coverage in an automated maintenance approach using multi-objective optimization. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWorR)*, pages 31–38, 2019.
- [MGV10] Radu Marinescu, George Ganea, and Ioana Verebi. Incode: Continuous quality assessment and improvement. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 274–275, 2010.
- [MHB08] Emerson Murphy-Hill and Andrew P Black. Refactoring tools: Fitness for purpose. *IEEE software*, 25(5):38–44, 2008.
- [MHB10] Emerson Murphy-Hill and Andrew P. Black. An interactive ambient visualization for code smells. *Proceedings of the 5th international symposium on Software visualization - SOFTVIS '10*, page 5, 2010.
- [MHM⁺11] Leandra Mara, Gustavo Honorato, Francisco Dantas Medeiros, Alessandro Garcia, and Carlos Lucena. Hist-inspect: A tool for history-sensitive detection of code smells. In *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development Companion, AOSD '11*, page 65–66, New York, NY, USA, 2011. Association for Computing Machinery.
- [ML13] Roberto Minelli and Michele Lanza. Software analytics for mobile applications—insights & lessons learned. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 144–153, 2013.
- [MMM⁺05] Cristina Marinescu, Radu Marinescu, Petru Mihancea, Daniel Ratiu, and Richard Wettel. iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design. In *Proceedings of the 21st IEEE International Conference on Software Maintenance ICSM 2005*, pages 77–80, 2005.

- [Moo03] Daniel L. Moody. The method evaluation model: a theoretical model for validating information systems design methods. In *Proceedings of the 11th European Conference on Information Systems, ECIS 2003, Naples, Italy 16-21 June 2003*, pages 1327–1336, USA, 2003. Association for Information Systems AIS Electronic Library (AISeL).
- [MPF08] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.
- [MRA11] Panita Meananeatra, Songsakdi Rongviriyapanish, and Taweessup Apiwattanapong. Using software metrics to select refactoring for long method bad smell. In *The 8th Electrical Engineering/ Electronics, Computer, Telecommunications and Information Technology (ECTI) Association of Thailand - Conference 2011*, pages 492–495. IEEE, may 2011.
- [MRH17] Toni Mattis, Patrick Rein, and Robert Hirschfeld. Edit transactions: Dynamically scoped change sets for controlled updates in live programming. *The Art, Science, and Engineering of Programming*, 1(2), apr 2017.
- [MS06] E. Mealy and P. Strooper. Evaluating software refactoring tool support. In *Australian Software Engineering Conference (ASWEC’06)*, pages 10 pp.–340. IEEE, 2006.
- [MT04] Tom Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, feb 2004.
- [MTZ20] Ricardo Moran, Matías Teragni, and Gonzalo Zabala. Physical bits: A live programming environment for educational robotics. In *Robotics in Education*, 2020.
- [Mye97] Roger B. Myerson. *Game theory - Analysis of Conflict*. Harvard University Press, 1997.
- [NBGL18] Alexandra Nilles, Mattox Beckman, Chase Gladish, and Amy LaViers. Improv: Live coding for robot motion design. In *MOCO ’18: Proceedings of the 5th International Conference on Movement and Computing*, pages 1–6, 06 2018.
- [ND86] Donald A. Norman and Stephen W. Draper. *User Centered System Design; New Perspectives on Human-Computer Interaction*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1986.
- [Nik] Nikolaos Tsantalis. JDeodorant. <https://github.com/tsantalis/JDeodorant>.
- [NN22] Nhan Nguyen and Sarah Nadi. An empirical evaluation of github copilot’s code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories, MSR ’22*, page 1–5, New York, NY, USA, 2022. Association for Computing Machinery.
- [NND⁺19] Hoan Anh Nguyen, Tien N. Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 819–830, 2019.

- [NNN⁺12] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. Detection of embedded code smells in dynamic web applications. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, page 282, 2012.
- [NPT13] Christian Napoli, Giuseppe Pappalardo, and Emiliano Tramontana. Using modularity metrics to assist move method refactoring of large systems. In *Proceedings - 2013 7th International Conference on Complex, Intelligent, and Software Intensive Systems, CISIS 2013*, pages 529–534, 07 2013.
- [NRE⁺20] Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. Example-based live programming for everyone: Building language-agnostic tools for live programming with lsp and graalvm. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2020*, page 1–17, New York, NY, USA, 2020. Association for Computing Machinery.
- [Nub] Robert Nubel. Refactoring with GitHub Copilot.
- [Oet] Christian Oetterli. refactorix. <https://marketplace.visualstudio.com/items?itemName=christianoetterli.refactorix>.
- [OMB13] Stephen Oney, Brad A. Myers, and Joel Brandt. Euclase: A live development environment with constraints and FSMs. *2013 1st International Workshop on Live Programming, LIVE 2013 - Proceedings*, pages 15–18, 2013.
- [OMIA17] Akio Oka, Hidehiko Masuhara, Tomoki Imai, and Tomoyuki Aotani. Live data structure programming. In *Companion to the First International Conference on the Art, Science and Engineering of Programming, Programming '17*, New York, NY, USA, 2017. Association for Computing Machinery.
- [Opd92] William F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
- [Ora14] Oracle. Java Platform Debugger Architecture, 2014. <https://docs.oracle.com/javase/7/docs/technotes/guides/jpda/>, 2014.
- [ÓTH⁺12] Mel Ó Cinnéide, Laurence Tratt, Mark Harman, Steve Counsell, and Iman Hemati Moghadam. Experimental assessment of software metrics using automated refactoring. *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '12*, page 49, 2012.
- [OVH⁺17] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. Toward semantic foundations for program editors. *CoRR*, abs/1703.08694, 2017.
- [Pal15] F. Palomba. Textual analysis for code smell detection. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 769–771, 2015.
- [Pan19] Jevgenija Pantiuchina. Towards Just-In-Time Rational Refactoring. *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 180–181, may 2019.

- [PAN⁺20] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. Tsdetect: An open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020*, page 1650–1654, New York, NY, USA, 2020. Association for Computing Machinery.
- [Pax02] John Paxton. Live programming as a lecture technique. *J. Comput. Sci. Coll.*, 18(2):51–56, dec 2002.
- [PBTP18] Jevgenija Pantiuchina, Gabriele Bavota, Michele Tufano, and Denys Poshyvanyk. Towards just-in-time refactoring recommenders. *Proceedings of the 26th Conference on Program Comprehension - ICPC '18*, pages 312–315, 2018.
- [PDNP⁺17] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. Lightweight detection of android-specific code smells: The adocor project. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 487–491, 2017.
- [PDvdL05] K. S. Pollard, S. Dudoit, and M. J. van der Laan. *Multiple Testing Procedures: the multtest Package and Applications to Genomics*, pages 249–271. Springer New York, New York, NY, 2005.
- [PGN08] Chris Parnin, Carsten Görg, and Ogechi Nnadi. A catalogue of lightweight visualizations to support code smell inspection. In *Proceedings of the 4th ACM symposium on Software visualization - SoftVis '08*, page 77, New York, New York, USA, 2008. ACM Press.
- [PPV00] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. Empirical studies of software engineering: a roadmap. In *International Conference on Software Engineering*, 2000.
- [PQV07] Thomas Panas, Dan Quinlan, and Richard Vuduc. Tool support for inspecting the code quality of hpc applications. In *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing Applications, SE-HPC '07*, page 2, USA, 2007. IEEE Computer Society.
- [PS10] Fortunato Pesarin and Luigi Salmaso. *Permutation Tests for Complex Data: Theory, Applications and Software*. Kohn Wiley & Sons, Hoboken, New Jersey, USA, 2010.
- [PSM09] Harkirat Padda, Ahmed Seffah, and Sudhir Mudur. Investigating the comprehension support for effective visualization tools – a case study. In *2009 Second International Conferences on Advances in Computer-Human Interactions*, pages 283–288, 2009.
- [PT13] Giuseppe Pappalardo and Emiliano Tramontana. Suggesting extract class refactoring opportunities by measuring strength of method interactions. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, volume 2, pages 105–110, 2013.
- [PWB09] Kai Petersen, Claes Wohlin, and Dejan Baca. The waterfall model in large-scale development. In Frank Bomarius, Markku Oivo, Päivi Jaring, and Pekka Abrahamsson, editors, *Product-Focused Software Process Improvement*, pages 386–400, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

- [RC15] Sudha Rajesh and A. Chandrasekar. Metrics measurement model: To measure the object oriented design metrics. In *2015 Seventh International Conference on Advanced Computing (ICoAC)*, pages 1–6, 2015.
- [Rea17] React. React – A JavaScript library for building user interfaces, 2017. <https://reactjs.org/>, 2017.
- [Rei95] Steven P. Reiss. An engine for the 3d visualization of program information. *Journal of Visual Languages & Computing*, 6(3):299 – 323, 1995.
- [Rei01] Steven P. Reiss. An overview of bloom. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '01*, page 2–5, New York, NY, USA, 2001. Association for Computing Machinery.
- [Resa] JetBrains Research. IntelliJDeodorant.
- [Resb] JetBrains Research. kotlinRMiner.
- [Resc] JetBrains Research. LLMs guide IDE refactorings - Extract Function.
- [RG09] Lukas Renggli and Tudor Gîrba. Why smalltalk wins the host languages shootout. In *Proceedings of the International Workshop on Smalltalk Technologies, IWST '09*, page 107–113, New York, NY, USA, 2009. Association for Computing Machinery.
- [RKM15] Songsakdi Rongviriyapanish, Noppawit Karunlanchakorn, and Panita Meananeatra. Automatic code locations identification for replacing temporary variable with query method. In *2015 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, pages 1–6, 2015.
- [RLR⁺19] Patrick Rein, Jens Lincke, Stefan Ramson, Toni Mattis, Fabio Niephaus, and Robert Hirschfeld. Implementing babylonian/s by putting examples into contexts: Tracing instrumentation for example-based live programming as a use case for context-oriented programming. In *Proceedings of the 11th ACM International Workshop on Context-Oriented Programming, COP '19*, page 17–23, New York, NY, USA, 2019. Association for Computing Machinery.
- [RMT09] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. A systematic review of software maintainability prediction and metrics. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 367–377, 2009.
- [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.
- [RRL⁺18] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. Exploratory and live, programming and coding: A literature study comparing perspectives on liveness. *The Art, Science, and Engineering of Programming*, 3(1), July 2018.
- [RRRC16] Mohammad Masudur Rahman, Chanchal K. Roy, Jesse Redl, and Jason A. Collins. Correct: Code reviewer recommendation at github for vendasta technologies. In

- Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, page 792–797, New York, NY, USA, 2016. Association for Computing Machinery.
- [RS19] Riemer Rozen and Tijs Storm. Toward live domain-specific languages. *Softw. Syst. Model.*, 18(1):195–212, feb 2019.
- [RTSB17] Guillermo Rodriguez, Alfredo Teyseyre, Alvaro Soria, and Luis Berdun. A visualization tool to detect refactoring opportunities in SOA applications. In *2017 XLIII Latin American Computer Conference (CLEI)*, pages 1–10. IEEE, sep 2017.
- [RXH18] Steven P. Reiss, Qi Xin, and Jeff Huang. Seede: Simultaneous execution and editing in a development environment. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE '18*, page 270–281, New York, NY, USA, 2018. Association for Computing Machinery.
- [RZ79] H. Remus and S. Zilles. Prediction and management of program quality. In *Proceedings of the 4th International Conference on Software Engineering, ICSE '79*, page 341–350. IEEE Press, 1979.
- [SA19] Alejandra Siles Antezana. Toad: A tool for recommending auto-refactoring alternatives. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 174–176, 2019.
- [Sal20] Sérgio António Dias Salgado. Towards a live refactoring recommender based on code smells and quality metrics. Master’s thesis, Faculty of Engineering of the University of Porto, Porto, Portugal, 2020.
- [SBF14] IEEE Computer Society, Pierre Bourque, and Richard E. Fairley. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. IEEE Computer Society Press, Los Alamitos, CA, USA, 3rd edition, 2014.
- [SCC01] William R. Shadish, Thomas D. Cook, and Donald T. Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Cengage Learning, 2001.
- [SE11] K. T. Stolee and S. Elbaum. Refactoring pipe-like mashups for end-user programmers. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 81–90, May 2011.
- [See12] Norbert M. Seel. *Experimental and Quasi-Experimental Designs for Research on Learning*, pages 1223–1229. Springer US, Boston, MA, 2012.
- [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 Jordi Cabot and Eelco Visser, editors, *Theory and Practice of Model Transformations*, pages 213–227, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [Sha01] M. Shaw. The coming-of-age of software architecture research. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 657–664a, 2001.

- [SHP19] Mark Santolucito, William T. Hallahan, and Ruzica Piskac. Live programming by example. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI EA '19, page 1–4, New York, NY, USA, 2019. Association for Computing Machinery.
- [SLZ08] Yang Yi Sui, Jun Lin, and Xiao Tuo Zhang. An automated refactoring tool for dataflow visual programming language. *ACM SIGPLAN Notices*, 43(4):21–28, apr 2008.
- [SMR⁺20] Railana Santana, Luana Martins, Larissa Rocha, Tássio Virgínio, Adriana Cruz, Heitor Costa, and Ivan Machado. Raide: A tool for assertion roulette and duplicate assert identification and refactoring. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, SBES '20, page 374–379, New York, NY, USA, 2020. Association for Computing Machinery.
- [So21] Richard So. What makes vscode so popular?, Mar 2021.
- [Som10] Ian Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, USA, 9th edition, 2010.
- [SS18] Tushar Sharma and Diomidis Spinellis. A survey on software smells. *Journal of Systems and Software*, 138:158–173, apr 2018.
- [STA⁺14] Jeffrey Schiller, Franklyn Turbak, Hal Abelson, José Dominguez, Andrew McKinney, Johanna Okerlund, and Mark Friedman. Live programming of mobile apps in app inventor. In *Proceedings of the 2nd Workshop on Programming for Mobile and Touch*, PROMOTO '14, page 1–8, New York, NY, USA, 2014. Association for Computing Machinery.
- [Stea] Chris Stead. cmstead/js-refactor. <https://github.com/cmstead/js-refactor>.
- [Steb] Chris Stead. Js refactor. <https://marketplace.visualstudio.com/items?itemName=cmstead.jsrefactor>.
- [Stec] Stepsize. Tech debt tracker. <https://marketplace.visualstudio.com/items?itemName=Stepsize.tech-debt-tracker>.
- [STPWR20] Haven Skinner, Rafael Trapani Possignolo, Sheng-Hong Wang, and Jose Renau. Livesim: A fast hot reload simulator for hdl. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 126–135, 2020.
- [SVH10] Jean-Guy Schneider, Rajesh Vasa, and Leonard Hoon. Do metrics help to identify refactoring? In *IWPSE-EVOL '10*, 2010.
- [SW65] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.
- [SW19] Markus Schütz and Michael Wimmer. Live coding of a vr render engine in vr. In *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pages 1150–1151, 2019.

- [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.
- [SZZ⁺19] Bo Shen, Wei Zhang, Haiyan Zhao, Guangtai Liang, Zhi Jin, and Qianxiang Wang. Intellimerge: A refactoring-aware software merging technique. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.
- [Tam] Kiss Tamás. Codemetrics. <https://marketplace.visualstudio.com/items?itemName=kisstkondoros.vscode-codemetrics>.
- [Tan90] Steven L. Tanimoto. Viva: A visual language for image processing. *Journal of Visual Languages and Computing*, 1(2):127 – 139, 1990.
- [Tan13] Steven L. Tanimoto. A perspective on the evolution of live programming. In *Proceedings of the 1st International Workshop on Live Programming*, LIVE ’13, pages 31–34, Piscataway, NJ, USA, 2013. IEEE Press.
- [TC09a] N. Tsantalis and A. Chatzigeorgiou. Identification of Move Method Refactoring Opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, may 2009.
- [TC09b] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 119–128, 2009.
- [TC16] Sandhya Tarwani and Anuradha Chug. Sequencing of refactoring techniques by Greedy algorithm for maximizing maintainability. *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 1397–1403, sep 2016.
- [TCC08] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. JDeodorant: Identification and Removal of Type-Checking Bad Smells. *2008 12th European Conference on Software Maintenance and Reengineering*, pages 329–331, apr 2008.
- [TCC18] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. Ten years of JDeodorant: Lessons learned from the hunt for smells. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 4–14. IEEE, mar 2018.
- [TG02] Craig K. Tyran and Joey F. George. Improving software inspections with group process support. *Commun. ACM*, 45(9):87–92, September 2002.
- [Thi14] David V. Thiel. *Research Methods for Engineers*. Cambridge University Press, 2014.
- [TKD22] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, 48(3):930–950, 2022.
- [TME⁺18] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinianian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE ’18, pages 483–494, New York, NY, USA, 2018. ACM.

- [TP07] Walter F. Tichy and Frank Padberg. Empirical methods in software engineering research. In *Companion to the Proceedings of the 29th International Conference on Software Engineering*, ICSE COMPANION '07, page 163–164, USA, 2007. IEEE Computer Society.
- [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.
- [Tyn17] Yuriy Tynchuk. *Quality Aware Tooling*. PhD thesis, University of Bern, 11 2017.
- [UOII17] Naoya Ujihara, Ali Ouni, Takashi Ishio, and Katsuro Inoue. C-JRefRec: Change-based identification of Move Method refactoring opportunities. *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 482–486, 2017.
- [VNB⁺12] S. Vimala, H. Khanna Nehemiah, R.S. Bhuvaneswa, G. Saranya, and A. Kannan. Applying Game Theory to Restructure PL/SQL Code. *International Journal of Soft Computing*, 7(6):264–270, jun 2012.
- [VNP17] Juraj Vincur, Pavol Navrat, and Ivan Polasek. Vr city: Software analysis in virtual reality environment. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 509–516, 2017.
- [Vue14] Vue. Vue.js, 2014. <https://vuejs.org/>, 2014.
- [WB19] Marvin Wyrich and Justus Bogner. Towards an autonomous bot for automatic source code refactoring. In *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*, pages 24–28, 2019.
- [Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 439–449. IEEE Press, 1981.
- [WGS⁺21] Tongjie Wang, Yaroslav Golubev, Oleg Smirnov, Jiawei Li, Timofey Bryksin, and Iftekhar Ahmed. Pynose: A test smell detector for python. *CoRR*, abs/2108.04639, 2021.
- [WHTK15] Chunyan Wang, Shoichi Hirasawa, Hiroyuki Takizawa, and Hiroaki Kobayashi. Combining code refactoring and auto-tuning to improve performance portability of high-performance computing applications. In *The Sixth International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking*, 2015.
- [Wixa] Wix. glean. <https://marketplace.visualstudio.com/items?itemName=wix.glean>.
- [Wixb] Wix. wix/vscode-glean. <https://github.com/wix/vscode-glean>.
- [WLR11] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities. *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 551, 2011.

- [WQB⁺11] Bin Wu, Kai Qian, Prabir Bhattacharya, Minzhe Guo, and Wanjun Hu. Live programming learning objects on cloud. In *2011 IEEE 11th International Conference on Advanced Learning Technologies*, pages 362–363, 2011.
- [Wuy02] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, Faculteit Wetenschappen - Departement Informatica, 02 2002.
- [XGLX17] Sihan Xu, Chenkai Guo, Lei Liu, and Jing Xu. A log-linear probabilistic model for prioritizing extract method refactorings. In *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, pages 2503–2507, 2017.
- [YC79] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., USA, 1st edition, 1979.
- [YGM⁺18] Ruru Yue, Zhe Gao, Na Meng, Yingfei Xiong, Xiaoyin Wang, and J. David Morgen-thaler. Automatic clone recommendation for refactoring based on the present and the past, 2018.
- [YM98a] P. Young and M. Munro. Visualising software in virtual reality. In *Program Comprehension, 1998*, pages 19–26. IWPC '98. Proceedings., 6th International Workshop, Jun 1998.
- [YM98b] P. Young and M. Munro. Visualising software in virtual reality. In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*, pages 19–26, June 1998.
- [YM13] A. Yamashita and L. Moonen. Do developers care about code smells? an exploratory survey. *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 242–251, 2013.
- [YNCI19] Norihiro Yoshida, Seiya Numata, Eunjong Choiz, and Katsuro Inoue. Proactive clone recommendation system for extract method refactoring. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWor)*, pages 67–70, 2019.
- [YSC⁺16] N. Yoshida, T. Saika, E. Choi, A. Ouni, and K. Inoue. Revisiting the relationship between code smells and refactoring. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–4, 2016.
- [ZH22] Xing Zhang and Zhenjiang Hu. Towards bidirectional live programming for incomplete programs. In *Proceedings of the 44th International Conference on Software Engineering, ICSE '22*, page 2154–2164, New York, NY, USA, 2022. Association for Computing Machinery.
- [Zim14] Donald Zimmerman. Comparative power of student t test and mann-whitney u test for unequal sample sizes and variances. *The Journal of Experimental Education*, 55:171–174, 04 2014.
- [ZM22] Lixin Zhao and Yinghong Ma. The matching degrees of research projects with their papers. *IFAC-PapersOnLine*, 55(3):31–36, 2022. 16th IFAC Symposium on Large Scale Complex Systems: Theory and Applications LSS 2022.

- [ZMRA13] Marie Lisandra Zepeda-Mendoza and Osbaldo Resendis-Antonio. *Hierarchical Agglomerative Clustering*, pages 886–887. Springer New York, New York, NY, 2013.
- [ZSF⁺22] Chunqi Zhao, I-Chao Shen, Tsukasa Fukusato, Jun Kato, and Takeo Igarashi. Oden: Live programming for neural network architecture editing. In *27th International Conference on Intelligent User Interfaces, IUI '22*, page 392–404, New York, NY, USA, 2022. Association for Computing Machinery.
- [ZSK14] Tianyi Zhang, Myoungkyu Song, and Miryung Kim. Critics: An interactive code review tool for searching and inspecting systematic changes. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014*, page 755–758, New York, NY, USA, 2014. Association for Computing Machinery.
- [ZW98] M. V. Zelkowitz and D. R. Wallace. Experimental models for validating technology. *Computer*, 31(5):23–31, 1998.
- [ZXX⁺22] Zejun Zhang, Zhenchang Xing, Xin Xia, Xiwei Xu, and Liming Zhu. Making python code idiomatic by automatic refactoring non-idiomatic python code with pythonic idioms. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022*, page 696–708, New York, NY, USA, 2022. Association for Computing Machinery.
- [ZZZS20] Weijie Zhou, Yue Zhao, Guoqiang Zhang, and Xipeng Shen. Harp: Holistic analysis for refactoring python-based analytics programs. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 506–517, 2020.

Appendix A

Scientific Publications

This appendix describes all the scientific publications related to this research project. We present not only papers that directly resulted from this project but also previous papers from previous work, unpublished papers, or master theses related to the main topics of this project.

A.1 Publications from this Research Project

Table A.1, (p. 225) lists all the scientific publications that resulted from this research project. These papers are related to the topic of *Live Refactoring*, mainly regarding our *Live Refactoring Environment*.

Table A.1: Summary of the publications resulting from this research project.

	<i>Title</i>	<i>Year</i>	<i>Conference</i>	<i>Core</i>
[FRA20]	Live Software Inspection and Refactoring	2020	SEDES	-
[Fer21]	A Live Environment for Inspection and Refactoring of Software Systems	2021	ESEC	-
[FAR22]	A Live Environment to Improve the Refactoring Experience	2022	PX	-
[FAR23a]	LiveRef: a Tool for Live Refactoring Java Code	2023	ASE	A*
[Fer23]	Towards a Live Environment for Code Refactoring	2023	ASE	A*

A.1.1 Live Software Inspection and Refactoring

Sara Fernandes, Ademar Aguiar, and André Restivo. 2020. *Live Software Inspection and Refactoring*. In *Proceedings of the 8th Software Engineering Doctoral Symposium (SEDES 2020)*. [FRA20]

Abstract. With the increasing complexity of software systems, software developers would benefit from instant and continuous guidance about the system they are maintaining and evolving. Despite existing several solutions providing feedback and suggesting improvements, many tools require explicit invocation, leading to developers missing improvement opportunities, such as important refactorings, due to lost of train of thought. Therefore, to address these limitations, we propose an approach where developers receive instant and continuous feedback about their software systems. This guidance would include the detection of code smells and the suggestion of refactorings to improve the system, justified by relevant software quality metrics related to the recommendations. This research aims to improve the experience of developing and maintaining software systems by providing a live environment for continuous inspection and refactoring of software systems, that is informative, responsive, and tactically predictive, and thus helping developers to identify and solve quality problems in a quicker and better way.

A.1.2 A Live Environment for Inspection and Refactoring of Software Systems

Sara Fernandes. 2021. A live environment for inspection and refactoring of software systems. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021). [Fer21]

Abstract. Refactoring helps to improve the design of software systems, making them more readable, maintainable, cleaner, and easy to expand. Most of the tools that already exist on this concept allow developers to select and execute the best refactoring techniques for a particular programming context. However, they aren't interactive and prompt enough, providing a poor programming experience. In this gap, we can introduce and combine the topic of liveness with refactoring methods. Live Refactoring allows to know continuously, while programming, the blocks of code that we should refactor and why they were classified as problematic. Therefore, it shortens the time needed to create high-quality systems, due to early and continuous refactoring feedback, support, and guidance. This paper presents our research project based on a live refactoring environment. This environment is focused on a refactoring tool that aims to explore the concept of Live Refactoring and its main components — recommendation, visualization, and application.

A.1.3 A Live Environment to Improve the Refactoring Experience

Sara Fernandes, Ademar Aguiar, and André Restivo. 2022. A Live Environment to Improve the Refactoring Experience. In Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming (Programming'22). [FAR22]

Abstract. Refactoring helps improve the design of software systems, making them more understandable, readable, maintainable, cleaner, and self-explanatory. Many refactoring tools allow developers to select and execute the best refactorings for their code. However, most of them lack

quick and continuous feedback, support, and guidance, leading to a poor refactoring experience. To fill this gap, we are researching ways to increase liveness in refactoring. Live Refactoring consists of continuously knowing, in real-time, what and why to refactor. To explore the concept of Live Refactoring and its main components — recommendation, visualization, and application, we prototyped a Live Refactoring Environment focused on the Extract Method refactoring. With it, developers can receive recommendations about the best refactoring options and have support to apply them automatically. This work helped us reinforce the hypothesis that early and continuous refactoring feedback helps to shorten the time needed to create high-quality systems.

A.1.4 LiveRef: a Tool for Live Refactoring Java Code

Sara Fernandes, Ademar Aguiar, and André Restivo. 2023. LiveRef: a Tool for Live Refactoring Java Code. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE'22). [FAR23a]

Abstract. Refactoring software can be hard and time-consuming. Several refactoring tools assist developers in reaching more readable and maintainable code. However, most of them are characterized by long feedback loops that impoverish their refactoring experience. We believe that we can reduce this problem by focusing on the concept of Live Refactoring and its main principles: the live recommendation and continuous visualization of refactoring candidates, and the immediate visualization of results from applying a refactoring to the code. Therefore, we implemented a Live Refactoring Environment that identifies, suggests, and applies Extract Method refactorings. To evaluate our approach, we carried out an empirical experiment. Early results showed us that our refactoring environment improves several code quality aspects, being well received, understood, and used by the experiment participants.

A.1.5 Towards a Live Environment for Code Refactoring

Sara Fernandes. 2023. Towards a Live Environment for Code Refactoring. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE'22). [Fer23]

Abstract. Refactoring code manually can be complex. Several refactoring tools were developed to mitigate the effort needed to create more readable, adaptable, and maintainable code. However, most of them continue to provide late feedback, assistance, and support on how developers should improve their software. That's where the concept of Live Refactoring comes in. We believe the immediate and continuous suggestion of refactoring candidates to the code will help reduce this problem. Therefore, we prototyped a Live Refactoring Environment that identifies, recommends, and applies Extract Method refactorings. We carried out an empirical experiment that showed us that our approach helped developers reach better code, with more quality, improving their refactoring experience.

A.2 Publications from Previous Work

In Table A.2, (p. 228), we list all the publications that resulted from previous work that was the basis for this research project. These papers are mainly related to measuring code quality metrics, to assess code quality, in real-time.

Table A.2: Summary of the publications resulting from previous work.

	<i>Title</i>	<i>Year</i>	<i>Type</i>	<i>Conference</i>	<i>Core</i>
[Fer19]	Supporting Software Development through Live Metrics Visualization	2019	Master Thesis	-	-
[FRFA20]	Helping Software Developers through Live Software Metrics Visualization	2020	Research Paper	PX	-

A.2.1 Supporting Software Development through Live Metrics Visualization

Sara Filipa Couto Fernandes. Supporting software development through live metrics visualization. Master's thesis, Faculty of Engineering of the University of Porto, Porto, Portugal, 2019. [Fer19]

This master thesis aimed to solve a recurrent problem of several plugins for different IDEs, in which most tools need to be compiled and executed to provide feedback, assistance, and guidance to help developers improve their code. Considering this problem, Sara Fernandes [Fer19] developed a plugin for Visual Studio Code that provides live visual feedback to developers on their code quality through the evolution of specific quality metrics. Besides, it also suggests and applies some simple refactorings, such as the *Split Variable Declaration*. Lastly, it also provided some information on the quality of each Git version of the project under analysis.

To validate the implemented solution, Fernandes [Fer19] carried out a controlled experiment with some software developers, in which they had to perform several programming tasks on a TypeScript project with or without the developed tool. With this experiment, it was possible to verify the tool's usability and reliability and also if the developers who used the tool converged to better programming solutions faster than the others.

A.2.2 Helping Software Developers through Live Software Metrics Visualization

Sara Fernandes, André Restivo, Hugo Sereno Ferreira, and Ademar Aguiar. 2020. Helping software developers through live software metrics visualization. In Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming (Programming'20). [FRFA20]

Abstract. With the increasing complexity of software systems, software developers would benefit from instant and continuous feedback about the system they are maintaining and evolving.

Despite existing several solutions providing feedback and suggesting improvements, many tools require explicit invocations, leading developers to miss some improvement opportunities, such as important refactorings, due to the loss of their train of thought. Therefore, to address these limitations, we developed a Visual Studio Code plugin providing real-time feedback — and also information about each commit made to the version control system. This tool is also capable of recommending two types of refactorings. To validate this approach, we did a preliminary controlled experiment using hypothesis-tests to check specific results. However, in this initial stage, we didn't have enough data to confirm our research questions, and we weren't able yet to confirm the main hypothesis.

A.3 Unpublished Work

From this research project also resulted in an unpublished scientific paper (*cf.* Table A.3, p. 229). This publication was already presented on 6th *International Workshop on Refactoring*, which was a co-located event with the 37th *IEEE/ACM International Conference on Automated Software Engineering (ASE'22)*. However, it was not formally published in the proceedings of the respective workshop.

Table A.3: Summary of the unpublished papers resulting from this research project.

	<i>Title</i>	<i>Year</i>	<i>Conference</i>	<i>Core</i>
[FAR23b]	Empirical Evaluation of a Live Environment for Extract Method Refactoring	2022	IWoR	-

A.3.1 Empirical Evaluation of a Live Environment for Extract Method Refactoring

Sara Fernandes, Ademar Aguiar, and André Restivo. 2022. Empirical Evaluation of a Live Environment for Extract Method Refactoring. In International Workshop on Refactoring (IWoR'22). [FAR23b]

Abstract. Complex software can be hard to read, adapt, and maintain. Refactoring it can create cleaner and self-explanatory code. Refactoring tools try to guide developers towards better code, with more quality. However, most of them take too long to provide feedback, support, and guidance on how developers should improve their software. To reduce this problem, we explored the concept of Live Refactoring, focusing on visually suggesting and applying refactorings, in real-time. With this in mind, we developed a Live Refactoring Environment that visually identifies, recommends, and applies *Extract Method* refactorings. To validate it, we conducted an empirical experiment. Early results showed that our approach improved several code quality metrics. Besides, we also concluded that our results were significantly different and better than the ones from refactoring the code manually without further help.

A.4 Co-Supervised Master Theses

There were also four master theses related to *Live Refactoring* that were supervised by part of our research team (cf. Table A.4, p. 230). Besides, these theses helped us with our project.

Table A.4: Co-supervised Master’s theses.

	<i>Title</i>	<i>Year</i>	<i>Student</i>	<i>Supervisors</i>
[Bar20]	Towards a Smart Recommender for Code Refactoring	2020	João Barbosa	Ademar Aguiar and Sara Fernandes
[Sal20]	Towards a Live Refactoring Recommender Based on Code Smells and Quality Metrics	2020	Sérgio Salgado	Ademar Aguiar and Sara Fernandes
[dF24]	Mining Software Repositories to Improve Refactoring Assistants	2024	Diogo Faria	Ademar Aguiar and Sara Fernandes
[dO24]	Spatialized Live Refactoring	2024	Margarida Oliveira	Ademar Aguiar

A.4.1 Towards a Smart Recommender for Code Refactoring

João Paulo Moreira Barbosa. Towards a Smart Recommender for Code Refactoring. Master’s Thesis, Faculty of Engineering of the University of Porto, Porto, Portugal, 2020. [Sal20]

In this master thesis, the student João Barbosa explored the concept of *Live Refactoring*. To do so, he created a refactoring environment capable of inspecting code to identify *Long Method* smell, visually suggest, and apply the corresponding *Extract Method* refactoring [Fow18]. He created a JavaScript and TypeScript plugin for Visual Studio Code that identifies several *Extract Method* candidates using the OPTICS clustering algorithm considering how tightly coupled the statements of a method are [Bar20].

To validate his solution, he performed a controlled experiment and a usability survey. In the controlled experiment, he automatically used his tool on several public repositories to verify if his methodology to detect, recommend, and execute *Extract Method* refactorings helps converge to better code. With the survey, he aimed to check if live feedback on refactoring suggestions is a suitable and user-friendly approach to help developers understand their code and how they should improve it [Bar20].

A.4.2 Towards a Live Refactoring Recommender Based on Code Smells and Quality Metrics

Sérgio António Dias Salgado. Towards a Live Refactoring Recommender Based on Code Smells and Quality Metrics. Master’s Thesis, Faculty of Engineering of the

University of Porto, Porto, Portugal, 2020. [Bar20]

In this master's thesis, the student Sérgio Salgado proposed a refactoring recommender for TypeScript and JavaScript software as a Visual Studio Code plugin. It provided live visualization of several code quality metrics as well as the chance to apply optimal extract-related refactorings like *Extract Method*, *Extract Class*, and *Extract Variable*. For all the refactorings, the metrics and heuristics used to detect them were customizable by the tool's user. Also, by allowing the user to live observe the impact of each refactoring on the quality metrics, the student aimed to help them make better refactoring decisions and be more aware of how they should improve their code [Sal20].

With a usability survey, the student understood how its tool impacted the users' workflow through the information provided by the tool's interface. Salgado also executed his tool on multiple code files across multiple public repositories to check if his approach helped converge to more maintainable code [Sal20].

A.4.3 Mining Software Repositories to Improve Refactoring Assistants

Diogo Luís Araujo de Faria. Mining Software Repositories to Improve Refactoring Assistants. Master's Thesis, Faculty of Engineering of the University of Porto, Porto, Portugal, 2024. [dF24]

In this master's thesis, the student Diogo Faria worked on improving the *LiveRef* refactoring recommendation plugin by incorporating machine learning to enhance the identification and suggestion of refactorings, specifically *Extract Method* and *Extract Class*. Instead of using the traditional threshold-based approach, the student focused on implementing a dynamic classification model, allowing the system to continuously learn from usage and adapt to different software contexts. To achieve this, he leveraged data mining techniques on existing software repositories, creating a diverse dataset to train the model and ensure its applicability to real-world scenarios [dF24].

To validate the improvements, the student conducted an automated analysis comparing the plugin's suggestions before and after the modifications, using real-life developer refactoring data. The results showed that incorporating real-world data into the recommendation process is crucial, as textbook refactorings do not always reflect developers' day-to-day practices. This approach makes the tool more useful and adaptable, encouraging continuous use and contributing to better-maintained software systems [dF24].

A.4.4 Spatialized Live Refactoring

Margarida Rocha Raposo de Oliveira. Spatialized Live Refactoring. Master's Thesis, Faculty of Engineering of the University of Porto, Porto, Portugal, 2024. [dO24]

In this master's thesis, the student Margarida Oliveira worked on developing a toolset that enhances software visualization using the city metaphor, a 3D representation designed to improve

code understanding and refactoring. Building on previous research, the project aimed to increase the liveness of code cities by allowing developers to navigate and inspect software structures using extended reality. This approach reduces cognitive effort and improves usability by enabling developers to identify defects and areas for improvement more efficiently. The toolset integrates seamlessly with IDEs and employs spatial techniques to visualize software systems, facilitating live code refactoring and providing instant feedback on changes [dO24].

To validate the toolset, the student conducted controlled experiments to assess its effectiveness. The tool not only helps developers familiarize themselves with complex codebases but also computes and displays refactoring candidates in real-time during coding. By incorporating visualization techniques from existing research and leveraging Apple Vision Pro for extended reality integration, the toolset enhances the software inspection process, improving both code comprehension and the overall developer experience [dO24].

Appendix B

User Guide

In this appendix, we present the user guide of our tool. We explain how to install, use, and configure it correctly. This guide was provided to the participants of our *Live Refactoring Experiment* (cf. Section 5.2, p. 143). The participants of our *Live Extract Method Experiment* (cf. Section 5.1, p. 124) had access to a simplified version of this guide, which only included explanations regarding the *Extract Method* refactoring type.

For experimental reasons, we had to name our tool to allow others to use and test it. Therefore, considering that we were studying the topic of *Live Refactorings*, we decided to name it “*LiveRef*”.

B.1 Installation

To install **LiveRef**, we need to launch the IntelliJ IDE and install the *.zip* folder that contains the tool. Inside IntelliJ, we must select the IDE preferences/settings depending on the OS. Then, there we find the “*Plugins*” menu, where we must choose the option “*Install Plugin from Disk...*” (cf. Figure B.1). After that, we have to search for the folder where the *.zip* is saved and select it. Then, IntelliJ will install the plugin.

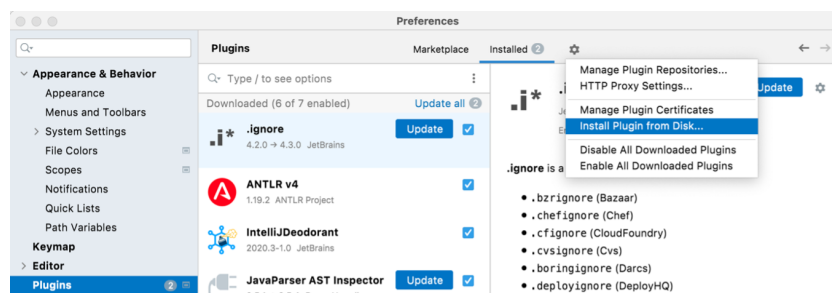


Figure B.1: How to install LiveRef.

B.2 Activation

After installing the plugin, we can activate the tool. The next steps depend on the IDE version we use:

- For IntelliJ 2021, we should click on the option “*Live Refactoring*” on the main menu (cf. Figure B.2);
- For IntelliJ 2022, we have a popup dialog displayed when you open a project and also a menu named “*Live Refactoring*” inside the “*Tools*” menu (cf. Figure B.3). If we activate the tool through the popup dialog, it starts automatically. If we choose not to activate it using the dialog, we can always start it using the second option (the “*Live Refactoring*” menu).

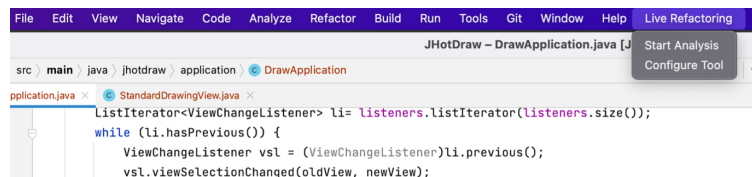


Figure B.2: *Live Refactoring* menu when using IntelliJ 2021.

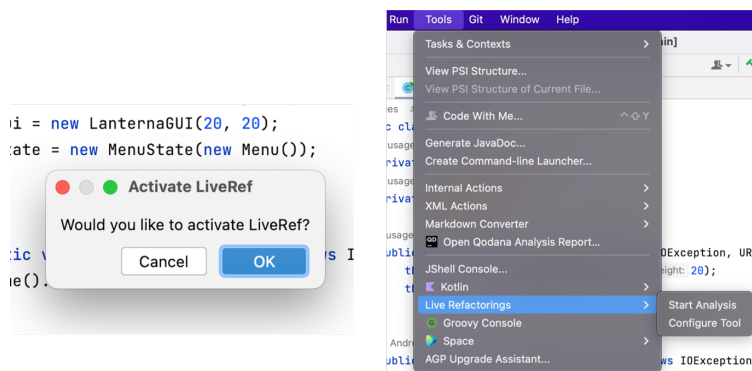


Figure B.3: Popup dialog and *Live Refactoring* menu when using IntelliJ 2022.

Then, on the “*Live Refactoring*” menu, we have access to two new options: (i) “*Start Analysis*”, which activates the tool and starts the analysis to find refactoring candidates (cf. Section B.2.1, p. 234), and (ii) “*Configure Tool*”, which allows configuring the tool and the thresholds for each refactoring (cf. Section B.2.2, p. 236).

B.2.1 Tool Workflow

After clicking on “*Start Analysis*” (or after activating the tool through the popup dialog), the tool inspects the code and presents the different refactoring candidates. As described in Section 4.3, (p. 116), to simplify the suggestion of each refactoring, we display each candidate through different colors ranging from light green to dark red (cf. Figure B.4, p. 235). The assigned colors were based

on the number of candidates per refactoring type. Light green represents less impactful candidates, and dark red represents the most impactful refactorings.

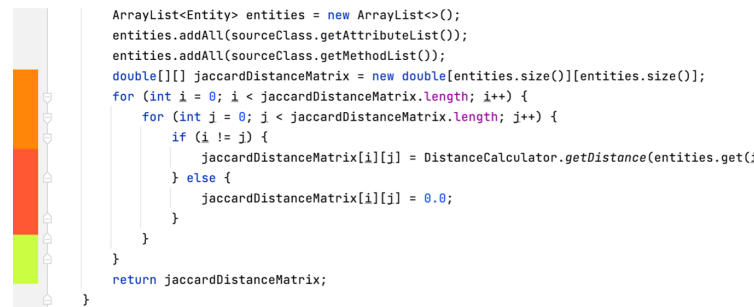


Figure B.4: Displaying refactorings using multiple colors.

Then, if we click on a color, we can access the refactoring menu (cf. Figure B.5). This menu lists the refactoring candidates related to the line of code represented by that color. Imagine that there are overlapping refactorings related to that line of code. To simplify, we display the color of just one of those refactorings. But on the menu, users can access all refactorings that include that line (cf. Section 4.3, p. 116).

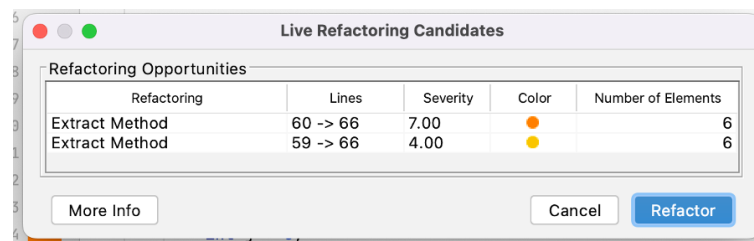


Figure B.5: Refactoring menu listing multiple refactoring candidates.

If we desire to perform one of the refactorings listed on the refactoring menu, you need to select it and click the “*Refactor*” button. Then, the refactoring activity will start. Note that the tool may not suggest all refactorings at the same time. As it identifies possible refactoring candidates by their type, it recommends them (e.g., there may be a time-lapse of seconds or less between listing Extract Variable and Extract classes candidates). If the tool does not find any refactoring candidates, a message like the one from Figure B.6, (p. 235) pops up in the bottom left corner of IntelliJ.

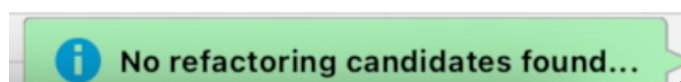


Figure B.6: Message displayed when no refactorings are found.

At any time, we stop the tool from inspecting your code. To do so, you need to click on the “*Stop Analysis*” option, which is only available after we start the tool and is displayed depending on whether we are using IntelliJ 2021 (cf. Figure B.7, p. 236) or 2022 (cf. Figure B.8, p. 236).

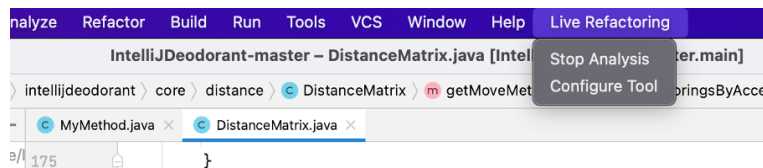


Figure B.7: Option to stop the tool when using IntelliJ 2021.

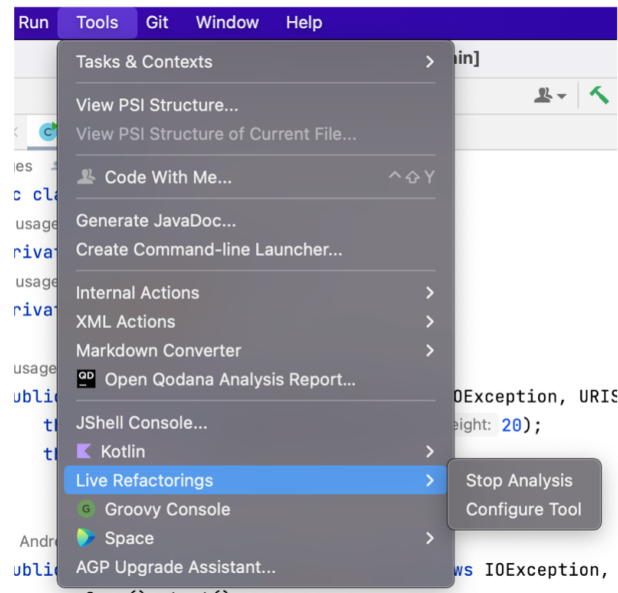


Figure B.8: Option to stop the tool when using IntelliJ 2022.

B.2.2 Tool Configuration

If we click on the “*Configure Tool*” option, we access the configuration menu (cf. Figure B.9). There, we can select the refactoring to include in the inspection process, change some thresholds used to detect or sort the refactoring candidates, limit the number of refactorings to be displayed (by refactoring type), or identify if we are colorblind. For colorblind people, the tool uses symbols instead of colors to list the refactoring candidates.

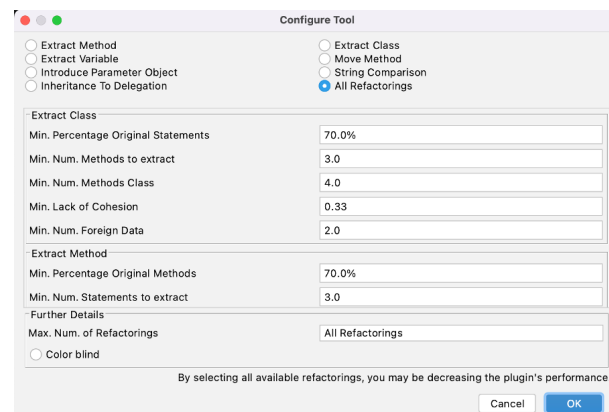


Figure B.9: Configuration Menu of LiveRef.

All thresholds to identify each refactoring candidate are described in the respective subsection in Section 4.2.3, (p. 97), where we detail the steps to detect each refactoring.

Appendix C

Experiments Artifacts

In this appendix, we present all the artifacts used in all experiments carried out to validate our research project.

C.1 Experiments Protocol

Before starting the experiments, we wrote a protocol document as an auxiliary basis for each. This protocol describes not only the procedures followed in each experiment but also their expected outcomes, how we expected to handle the collected data, and what kind of scientific publications we aimed to produce with the results achieved from them.

EXPERIMENT PROTOCOL

A Live Environment for Continuous Software Inspection and Refactoring



STUDY RESEARCHER(S)

Sara Fernandes

Ph.D. Student

Faculty of Engineering, University of Porto

Department of Informatics Engineering

Ademar Aguiar

Supervisor

Faculty of Engineering, University of Porto

Department of Informatics Engineering

André Restivo

Co-Supervisor

Faculty of Engineering, University of Porto

Department of Informatics Engineering

1. INTRODUCTION

This empirical experiment is focused on evaluating and validating a Live Refactoring Environment developed in the context of a Ph.D. research project.

With a Live Refactoring Environment, software developers will have access to several refactoring suggestions that aim to improve the quality of their software systems. This refactoring environment was developed for Java code on the IntelliJ IDE, and it focuses on analyzing specific quality attributes to detect particular code smells. By identifying different smells, this environment selects and applies possible refactoring methods to mitigate the respective smells.

2. MAIN PROBLEM

It is hard to achieve high-quality software systems, mostly because of the existence of some programming problems that reduce the code capability of being read, adapted, and maintained [1,2].

To reduce this problem, we need to inspect software to identify possible issues and provide guidance to developers towards possible solutions. However, how can we inspect software, provide feedback about its quality, and also guide and support developers towards better programming solutions?

Inspecting code helps find possible programming defects. However, it can be tricky and time-consuming when the system is complex, with many lines of code. We also know that most of these issues are caused by code smells and that they should be mitigated by refactoring the code, which can be hard [3].

Besides, most developers try to refactor their code manually, drastically increasing the total time needed to create good software, with quality [4].

3. AIM OF STUDY / RESEARCH QUESTIONS

This experiment aims to determine if a development environment that continuously inspects source code, detects, and visualizes code smells, and recommends refactoring techniques, in real-time, helps software developers achieve code with more quality and faster.

Taking this in mind, our study is focused on four distinct research questions:

RQ1: Which non-intrusive ways can be used to inform developers, in real-time, of possible code smells in their source code?

With it, we aim to verify how we can detect and visualize, in real-time, in a user-friendly way, the code smells present in the source code.

RQ2: What is the impact of real-time code smell detection on the developer's awareness of code quality?

With it, we aim to measure the impact of a live code smell detection approach on the developers' awareness of the quality of their systems.

RQ3: What is the impact of a real-time refactoring environment on the quality of source code?

With it, we aim to measure the impact of continuously exposing developers to refactoring suggestions on the quality of their code.

RQ4: What is the impact of a live refactoring environment in the time needed to reach good programming solutions?

With it, we aim to measure the impact of liveness in refactoring environment and if developers reach good programming solutions, faster than with conventional refactoring tools or when they refactor their code manually.

4. HYPOTHESIS

Our research project is focused on the following hypothesis:

A development environment that continuously inspects source code, detects and visualizes code smells, and recommends refactoring techniques, in real-time, helps software developers to achieve code with more quality and faster.

Then, considering this research statement and some of the previously described research questions, we enumerated possible hypothesis tests that will help us validate and evaluate our approach.

RQ1: *Which non-intrusive ways can be used to inform developers, in real-time, of possible code smells in their source code?*

H₀: *There are no user-friendly approaches that can be used to inform developers about possible code smells on the source code.*

H_A: *There are user-friendly approaches that can be used to inform developers about possible code smells on the source code.*

RQ2: *What is the impact of real-time code smell detection on the developer's awareness of code quality?*

H₀: *A real-time code smell detection approach has no impact on the developers' awareness of their code quality.*

H_A: *A real-time code smell detection approach impacts positively the developers' awareness of their code quality.*

RQ3: *What is the impact of a real-time refactoring environment on the quality of source code?*

H₀: *A real-time refactoring environment has no impact on code quality.*

***H_A:** A real-time refactoring environment impacts positively the code quality.*

***RQ4:** What is the impact of a live refactoring environment in the time needed to reach good programming solutions?*

***H₀:** A live refactoring environment has no impact on the total time needed to reach good programming solutions.*

***H_A:** A live refactoring environment impacts positively the total time needed to reach good programming solutions.*

5. STUDY DESIGN

To answer each research question and validate our hypothesis, we designed a study consisting of a controlled experiment. This experiment consists of several programming or refactoring tasks performed by different experimental using or not our approach or different versions of it.

The different approaches, all the programming tasks, and all the guides needed to perform the experiment will be provided to the participants through a Google Form and email. This form will also have some questions that will help us know the participants' characterization, opinions and suggestions, and usability assessment about the approach used on the experiment.

6. STUDY SETTING/LOCATION

The study will take place at the Faculty of Engineering, University of Porto during different programming classes.

7. STUDY POPULATION AND SETTING

The study will mainly be focused on students from bachelor and master classes related to the area of Software Engineering. They won't need to have specific characteristics. However, we expect that they have some knowledge of programming in Java and using IntelliJ IDE.

We expect to have more than 60 students participating in our study. And they will be divided into two groups. One that will use several versions of the developed approach to refactor code provided by us, and other that won't use it and that will have to refactor the code manually.

8. STUDY OUTCOMES

a. Quantitative

The quantitative outcomes that will result from the study will be divided into different groups:

- **Outcome A: Participants**

On this group, we will focus on knowing the study participants by asking them:

- Their programming experience
- Their experience with Java
- Their experience with data structures
- Their experience with refactoring tools
- Their experience with code metrics' analysis tools
- Their experience using IntelliJ API

These values will be measured on a scale from 1 to 5, where 1 represents poor experience, and 5 represents good experience.

- **Outcome B: Code Quality Metrics**

In this group, we will have outcomes related to the evolution of code quality metrics when applying refactorings to the code. We will measure the following metrics:

- Number of lines of code
- Number of classes
- Number of methods
- Number of long methods
- Lack of cohesion
- Cyclomatic Complexity
- Halstead Length
- Halstead Volume
- Halstead Difficulty
- Halstead Effort
- Halstead Time
- Halstead Bugs Delivered
- Halstead Maintainability

These metrics will be compared between versions of the approach and groups of participants, to verify which where the versions with better results and if participants had better results when using the proposed approach.

• **Outcome C: Action Metrics**

In this group, we will have outcomes related to different actions performed on the code. We will measure:

- The number of refactorings performed on the code
- The number of undo actions performed after refactoring the code

These metrics will help us understanding the behaviours of each participant and if there is a correlation between these kinds of actions and the results obtained by them.

• **Outcome D: Liveness**

In this group, we will only measure the total time needed to solve each programming task. With this metric, we aim to assess the efficiency of our approach by comparing the time measured with and without the proposed approach.

• **Outcome E: Usability and other Evaluation Metrics**

In this group, we will assess several usability aspects. These aspects will be evaluated through a usability questionnaire. We aim to verify:

1. If they would use the approach again
2. If they found the approach complex and difficult to follow
3. If they found the approach difficult to use
4. If they would need more technical support to understand and use the approach
5. If they thought that all the functionalities of the approach were well integrated
6. If they found it difficult to use the approach on the correspondent project
7. If they were inconsistencies in the approach
8. If they learnt how to use the approach quickly
9. If they felt confident using the approach
10. If they would use the approach in the future
11. If they thought that the color scheme was appropriated
12. If they thought that the placement of the suggestions was appropriated
13. If they thought that the number of suggestions was adequate
14. If they thought that the configuration menu was easy to understand
15. If they believe that the approach would reduce the effort needed to create better software systems
16. If they believe that the approach will make it easier for developers to verify if their code has quality or not
17. If they found the approach useful
18. If they believe that using the approach will make it more difficult to maintain large and complex software systems

These values will be measured on a scale from 1 to 5, where 1 represents poor opinion, and 5 represents a good opinion. Questions 2, 3, 4, 6, and 9 aim to assess the perceived ease of use of our approach. Questions 14, 15, 16, and 17 aim to assess the perceived usefulness of the proposed solution. Question 10 aim to assess the intention to use the approach. Other questions will help us assessing more general usability aspects.

Finally, the consistency of our approach will be measured by the number of refactorings applied by each participant compared with the average value inside each group of participants.

- **Outcome F: Other Outcome**

Using the metrics from Outcome F, we will calculate their reliability and significance. Besides, we also aim to compare each result inside each group and also between groups of participants.

b. Qualitative

Our qualitative outcomes will be divided into two groups:

- **Outcome A: Programming Tasks**

For the initial experiment, the participants will have specific programming tasks. For some of them, they will have to answer some questions about the code under analysis. These questions will help us measure their awareness of possible code flaws and possible opportunities to improve the source code.

- **Outcome B: Opinions or Suggestions**

We will take into account the opinions or suggestions provided by each study participant. With it, we aim to know if they like the experiment or if they have valuable recommendations to be implemented on our approach.

9. STUDY PROCEDURES

This section describes exactly what will happen during the study, stating how we are going to recruit the participants, how we are going to divide them, and how the experiment will work.

a. Recruitment of participants

As said, all the study participants will be students from bachelor classes, related to the area of Software Engineering, lectured by the main researchers of the project being studied. Therefore, there is no previous recruitment or selection of the participants before starting the respective study.

b. Study full description

b.1 Initial Experiment

Before starting this study, we will randomly divide the participants into two groups. Participants from Group A will have access to different versions of the developed approach, and group B won't. Group B will only have access to a tool that will help us measuring the evolution of each code quality metric over each programming task. With this division, we aim to verify if a Live Refactoring Environment helps or not to improve the quality of the source code, faster. Afterwards, we will send a form to each participant.

In this form, the participants will have access to a brief description of the experience and why we are carrying it out, as well as why we have divided them into two distinct groups. Then, in that same form, they will have access to the first set of questions that will help us characterize them in terms of their programming experience, experience with refactoring tools, etc.

After these initial questions, the form may vary depending on the group to which each participant belongs. If the participants belong to group A, they will have access to the tool's executable, a guide for using it and a set of programming tasks that should be executed with the tool. In addition, in the end, they will have a set of questions that will allow us to obtain

feedback on the experience itself and also on the usability of the tool. If the participants belong to group B, they will only have access to the set of programming tasks that they must carry out manually, as well as questions that will allow us to assess their opinion regarding the experience carried out.

b.2 Final Experiment

To validate our hypothesis and answer our research questions, we will conduct a final empirical experiment. This experiment won't involve any predefined programming or refactoring tasks, nor had any previously defined experimental groups. Instead, we will only identify our experimental groups when we start analyzing the data collected from the experiment itself. Also, the only goal for the participants is to implement a specific project.

The participants of this empirical experiment will be students from programming Bachelor classes learning how to program in Java. They will work in groups of two or three elements to develop a small Java integrator project that should be implemented in a four-week time frame.

In their projects, the participants will have to apply the SOLID principles, use an MVC architectural pattern, and apply several design patterns to improve their code [5,6]. Also, one of the main goals of this project is to have the students identify code smells and apply the most correct refactorings to mitigate them, improve non-functional attributes, and increase code quality.

After the experiment, we will be defining the experimental groups based on them using or not our tool and in which development stages they started using it. With this type of division, we aim to check if using a Live Refactoring Environment helps converge to better code, faster.

Next, we describe each step with more detail.

- **Characterization of the Participants:**

This section of the experiment form is composed by the following questions:

- Programming experience
- Experience programming in Java
- Experience with data structures
- Experience with refactoring tools
- Experience with code metrics' analysis tools
- Experience using IntelliJ API

- **Programming Tasks for the Initial Experiment:**

In this section, the participants from group A will have access to:

- The tool's executable (we will provide different versions of the tool to different participants)
- A guide explaining how to execute the version of the tool provided to the participants
- A set of programming tasks composed of 3 different tasks that will take up to a maximum of 35 minutes:
 - **Task 1:** Task where the participants will implement code from scratch and will need to refactor their code while doing it or at the end of the programming task;
 - **Task 2:** Task where the participants will review and refactor a long method smell;
 - **Task 3:** Task where the participants will have a class with more than one long method smell and they need to refactor them until the approach doesn't find no more refactoring candidates.

The participants from group B will have access to:

- The code quality metrics tool's executable
- A guide explaining how to use the tool

- A set of programming tasks composed of 3 groups of tasks that will take up to a maximum of 45 minutes:
 - **Task 1:** Task where the participants will implement code from scratch and will need to refactor their code manually while doing it or at the end of the programming task;
 - **Task 2:** Task where the participants will review and refactor manually a long method smell;
 - **Task 3:** Task where the participants will have a class with more than one long method smell and they need to refactor them manually until they think the code is of sufficient quality and the class has no more long method smells.

These tasks will help us measure the final quality of the source code and the total time needed to achieve good programming solutions and assess our approach's usability. With these values, we will be able to measure the efficiency, consistency, usability, and quality of our approach and perform our hypothesis tests and answer our research questions.

- **Feedback on the Initial Experiment:**

In this section, participants of the first experiment will have access to questions that will help us receive feedback about the experiment. These questions can vary between groups.

The participants of group A, will have questions that will help us knowing:

- If participants were satisfied with the experiment;
- If the tool's configuration menu helped them understand the tool itself;
- If the suggestions helped understanding where they should improve their code;
- If the refactoring suggested and applied helped creating better code;
- If a Live Refactoring Environment helped them achieve better code, faster;
- Possible suggestions about the experiment.

The participants of group B, will help us knowing:

1. If participants were satisfied with the experiment;
2. If they think that Live Refactoring Environment would help them achieve better code, faster;
3. If they would like to use a Live Refactoring Environment in the future;
4. Possible suggestions about the experiment and the tool.

In this section, participants from group B will also have access to a video of our tool to know how it works.

- **Usability:**

This section will only be available to the groups that used our tool. There, they will have access to the following questions:

1. If they would like to use the tool frequently;
2. If they found the tool unnecessarily complex;
3. If they thought the tool was easy to use;
4. If they would need the support of a technical person to be able to use the tool;
5. If they found the various functions in the tool were well integrated;
6. If they thought there was too much inconsistency in the tool;
7. If they thought that most people would learn how to use the tool very quickly;
8. If they found the tool very cumbersome to use;
9. If they felt very confident using the tool;
10. If they need to learn a lot of things before they could get going with the tool;
11. If the color scheme used to represent the possible refactoring techniques was adequate;
12. If the placement of the interface was adequate;
13. If the placement of the refactoring suggestions was adequate;
14. If the number of suggestions was adequate;
15. If the configuration menu is easy to understand;
16. If they would you like to see the evolution of the code quality metrics on the tool;
17. If they believe that the approach reduces the effort needed to create better software systems;

18. If they believe that the approach will make it easier for developers to verify if their code has quality or not;
19. If they found the approach useful;
20. If they believe that using the approach will make it more difficult to maintain large and complex software systems;
21. Possible suggestions about the tool usability;

10. DATA HANDLING AND RECORD KEEPING

a. Data Collection and Management Responsibilities

The data collection will be the responsibility of the researchers. They will be responsible for ensuring the accuracy, completeness, legibility, and timeliness of the data collected.

This data will be collected into two different ways:

- Our approach will automatically assess all the data related to the evolution of code quality metrics, the time needed to fulfill any tasks before and after a refactoring is applied to the source code, and several actions performed like undo or scrolling. After measuring each one of these relevant values, they will be automatically saved in a Firebase database;
- All the data related to usability and participants characterization, opinion, and answers to each programming question will be collected using a Google Form.

b. Study Records Retention

The researchers will maintain all the records for a period of two years. All the data collected will be saved locally in an external disk and on a Google Drive folder that only the researchers have access to.

After the two years period, all the records will be erased from their storage units.

11. PUBLICATION & INTELLECTUAL PROPERTY

The researchers involved in the research project will produce a publication synthetizing all work done to implement the respective approach and also the results obtained through the empirical experiment. This publication is expected to be submitted and published in a A or A* conference related to the area of Software Engineering.

12. CONFIDENTIALITY CONSIDERATIONS

All the data collected from the study will be saved in a Firebase database. At the end of each day, all data from that day will be exported to a local device and erased from the online database. Besides, each participant will be identified by a unique ID created using an SHA-256 hashing algorithm to preserve their identity.

13. OUTCOMES AND SIGNIFICANCE

All the outcomes resulting from the study will be relevant. With them, we will be able to answer each one of the research questions.

RQ1: *Which non-intrusive ways can be used to inform developers, in real-time, of possible code smells in their source code?*

To answer this research question, we will take into consideration the quantitative values measured from Outcome E, which aims to assess the usability of our approach, and also the qualitative values provided by each participant as opinions or possible suggestions to improve our solution.

RQ2: *What is the impact of real-time code smell detection on the developer's awareness of code quality?*

To answer this research question, we will compare the evolution of each code quality metric measured in Outcome B, taking into consideration if the participants used or not the proposed approach, and if they used it which was its version. With it, we aim to verify if participants were aware of the code flaws when they used the approach and even when they didn't use it.

Besides, we will also take into account the qualitative outcome from Group A that is related to the answer provided by the participants about possible code flaws and refactoring opportunities present on the code under analysis.

RQ3: *What is the impact of a real-time refactoring environment on the quality of source code?*

To answer this research question, we will compare the evolution of each code quality metric measured in Outcome B and C, taking into consideration if the participants used or not the proposed approach, and which version they used. With it, we aim to verify if participants using specific versions of the approach reach better programming solutions, with more quality.

RQ4: *What is the impact of a live refactoring environment in the time needed to reach good programming solutions?*

To answer this research question, we will compare the total time needed to finish each programming task, taking into consideration if the participants used or not the proposed approach, and which version they used. With it, we aim to verify if participants using the approach finish each task more quickly than the other participants who didn't use the approach. Besides, we also aim to verify which version of the approach had better liveness results.

14. REFERENCES

1. Fowler, M., & Beck, K. (2019). *Refactoring: Improving the design of existing code*. Addison-Wesley.
2. István Kádár, Péter Hegedűs, Rudolf Ferenc, and Tibor Gyimóthy. 2016. A Manually Validated Code Refactoring Dataset and Its Assessment Regarding Software Maintainability. In Proceedings of the “The 12th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2016)”. Association for Computing Machinery, New York, NY, USA, Article 10, 1–4.
3. William F. Opdyke. 1992. Refactoring object-oriented frameworks. Ph.D. Dissertation. University of Illinois at Urbana-Champaign, USA. Order Number: UMI Order No. GAX93-05645.
4. Emerson Murphy-Hill and Andrew P. Black. 2010. An interactive ambient visualization for code smells. In Proceedings of the “5th international symposium on Software visualization (SOFTVIS '10)”. Association for Computing Machinery, New York, NY, USA, 5–14.
5. Robert C. Martin. Agile software development: principles, patterns, and practices. Prentice Hall PTR, Hoboken, New Jersey, USA, 2003.
6. Ethan Marcotte. Responsive design: Patterns and Principles. A Book Apart, New York, NY, 2015.

C.2 Participation Consent Declaration

This section presents the consent declaration used in the first two experiments carried out on this research project. This declaration is based on the *Helsinki Consent* [Ass13].

DECLARATION OF CONSENT

(Based on the Helsinki Declaration)

As part of the *Ph.D.* entitled *A Live Environment for Continuous Software Inspection and Refactoring*, carried out by the student *Sara Filipa Couto Fernandes*, supervised by *Professor Ademar Aguiar* and co-supervised by *Professor André Restivo*, at the *Faculty of Engineering of the University of Porto*, I, _____, declare that I understood the explanation given to me on the study that I will participate in, namely the voluntary nature of my participation, having been allowed to ask the questions that I thought needed.

I also understood that my identity would be maintained confidential.

It was also explained that I could withdraw from the study anytime, without any disadvantages.

Therefore, I consent to participate in the study by answering all the proposed questions.

Porto, ____ / ____ / ____

(Participant)

C.3 Live Extract Method Experiment

In the first experiment, we provided each participant with a guide. After the experiment finished, they had to fill out a specific questionnaire.

C.3.1 Experiment Guide

Here, we present the guide followed by the participants from our initial experiment. This guide was divided into different parts concerning the different experimental groups.

A Live Environment for Inspection and Refactoring of Software Systems - Empirical Experiment

We are working on a **Live Refactoring Environment**. With this approach, software developers will have access to several **Extract Method refactoring** suggestions that aim to improve the quality of their software systems. This refactoring environment was developed for **Java** code on the **IntelliJ IDE**, and it focuses on analyzing specific quality attributes to detect particular code smells. By identifying different smells, this environment selects and applies possible refactoring methods capable of mitigating the respective smells.

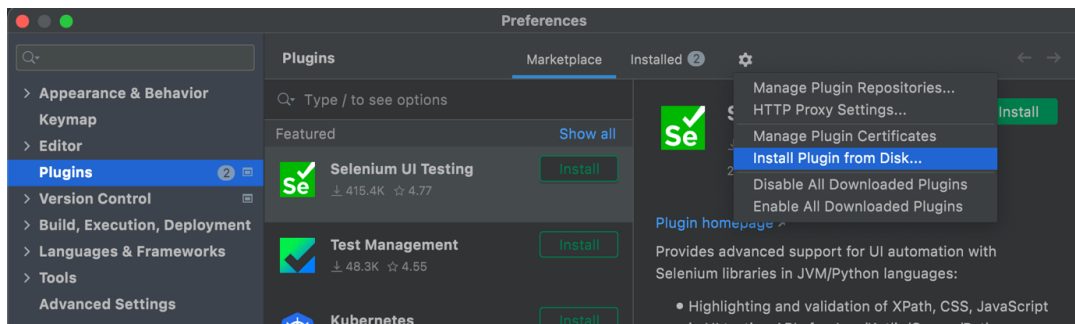
We are doing an empirical experiment that will help us evaluate and validate our project. To do it, we need your help!

This experiment is divided into three different tasks that will correspond to three different projects with different degrees of difficulty. Note that each task is described textually with the help of the respective UML diagrams (these diagrams are also attached to each project).

This experiment is estimated to take no more than **45 minutes**.

Before starting the programming tasks, read the guide related to the tool you will use and watch the sample video that is attached to it.

Please launch the IntelliJ IDE and install the provided tool to start the experiment. To install it, you just need to select your IDE's preferences. There you will find the “**Plugins**” menu, where you must select the option of “**Install Plugin from Disk**” (see the following image). Search the folder where you saved the attached files and select the .zip folder that is inside it. Then, IntelliJ will install the plugin. This description is demonstrated in the sample video.



Note that for each programming task, you can implement or refactor the code manually and undo a refactoring applied to the code when you believe it wasn't the best refactoring to be used on a specific situation.

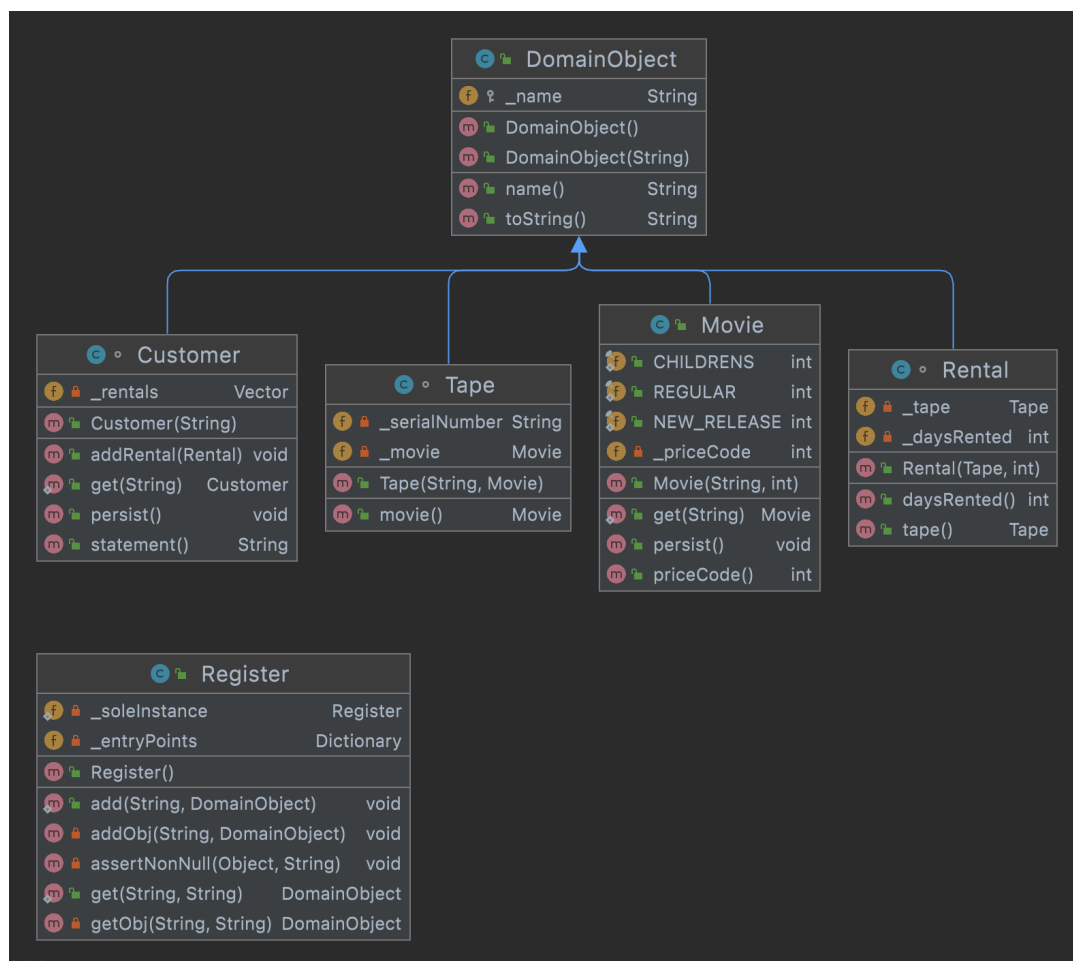
After doing all the programming tasks, you should return to the Goggle Form and answer the remaining questions.

Please read this guide and clarify any doubts you may have...

Group A – Version 1

Programming Task 1

The first programming task is focused on a simple project that represents a movie rental system. The following image represents the UML diagram of that system.



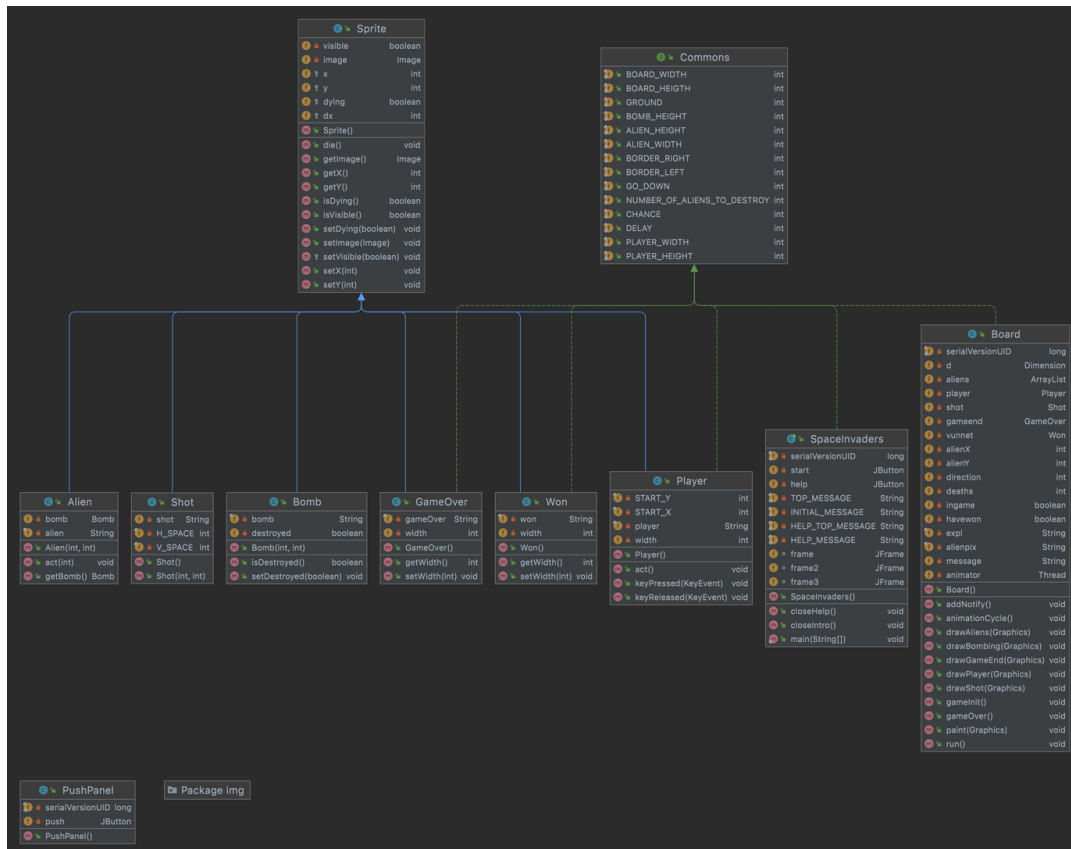
Programming Task: This programming task aims to manually identify, using the available tool, a possible code smell that can be corrected using the Extract Method refactoring.

So, you should follow the next steps:

1. Open the “**MovieRental**” project in your IntelliJ;
2. Analyze the project and try to answer the questions posed on the Google Form;
3. After you have responded to the questions, come back to this guide;
4. Now, start the refactoring tool by clicking on the “**Start Analysis**” option and open the class where you think there is a method where an Extract Method refactoring can be applied;
5. Check if the method you selected in the Google Form is the same indicated by the refactoring tool;
6. If yes, carry out the respective refactoring;
7. If not, and if no refactorings are found in that file, proceed with the same steps but in another file until you find the method that should be refactored.
8. The task is complete when all the refactoring suggested are applied to the code.

Programming Task 2

The second programming task is focused on a medium project that implements the game “Space Invaders”. The following image represents the UML diagram of the game.



Programming Task: This programming task aims to identify, using the available tool, while programming, possible candidate code blocks to be refactored using the Extract Method refactoring.

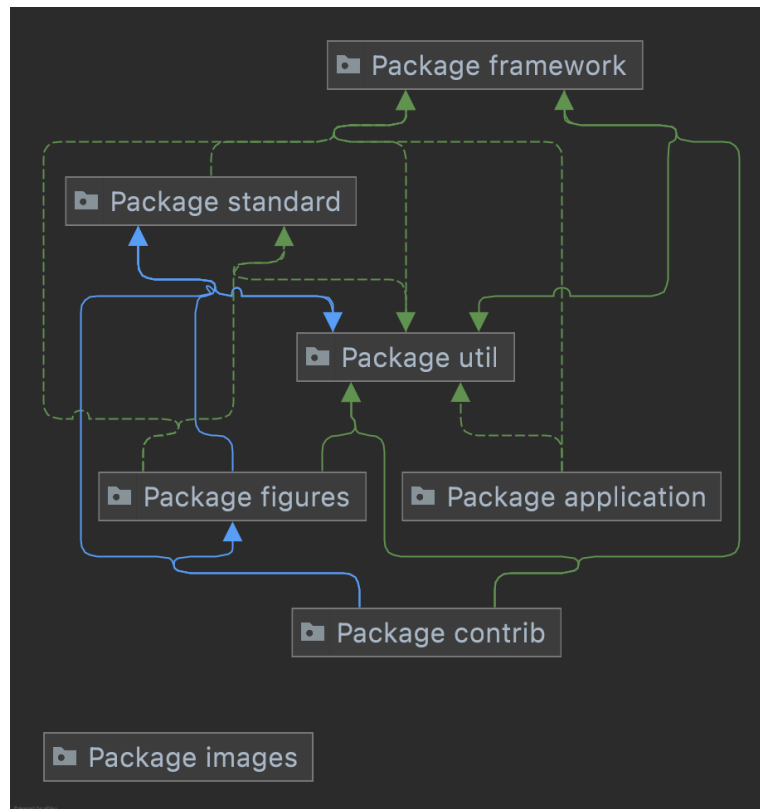
So, you should follow the next steps:

1. Open the “SpaceInvaders” project in your IntelliJ;
2. Analyze the project using the provided diagram;

3. Now, open the Board.java file and start the refactoring tool by clicking on the “**Start Analysis**” option;
4. Focus on the “**animationCycle**” method (class **Board**), which aims to implement the game's animations on the respective board. It will be the method you will have to implement;
5. Within the method itself, you can find comments about the code you must implement;
6. Follow the steps described there;
7. As you are programming, multiple refactoring suggestions may appear and they should be applied either during the implementation of the code or after finishing its development. You can apply all the suggested refactorings or just the ones you think are suitable for your code;
8. The task is completed either when you apply all the suggested refactorings or when you think that the code is good enough.

Programming Task 3

The third and final programming task is focused on a large-scale project that implements the **JHotDraw** system, which is a two-dimensional graphics framework for structured drawing editors. Since the project has several classes and methods, its UML is not easy to understand when inserted into a file like this. In this way, we just create the simplified UML diagram of the system. Attached to this file are more complete UML diagrams representing the packages where the programming task should be applied.



Programming Task: This programming task aims to identify, using the available tool, all code blocks that must be refactored using the Extract Method refactoring.

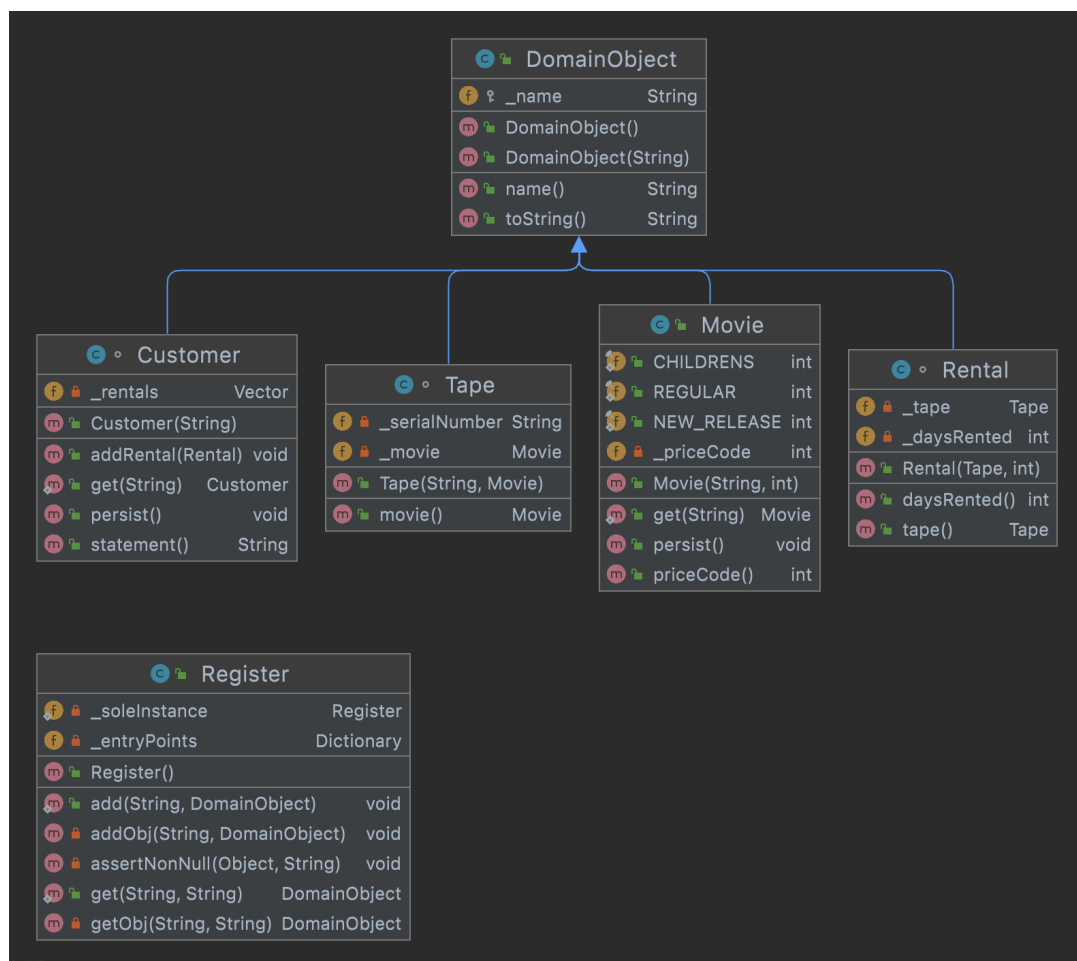
So, you should follow the next steps:

1. Open the “**JHotDraw**” project in your IntelliJ;
2. Analyze the project using the diagram provided to you;
3. Now, open the file **DrawApplication.java** (package **Application**) and start the refactoring tool by clicking on the “**Start Analysis**” option;
4. Then, the tool will suggest multiple Extract Method, which should be applied according to your will and opinion about them. That is, if you think a refactoring with severity 3 is more relevant than a refactoring with severity 4, you should apply it regardless of the color or severity measured by the tool;
5. The task in that file is completed when all possible refactorings have been applied;
6. Then, repeat these steps on **StandardDrawingView.java** (package **standard**).

Group A – Version 2

Programming Task 1

The first programming task is focused on a simple project that represents a movie rental system. The following image represents the UML diagram of that system.



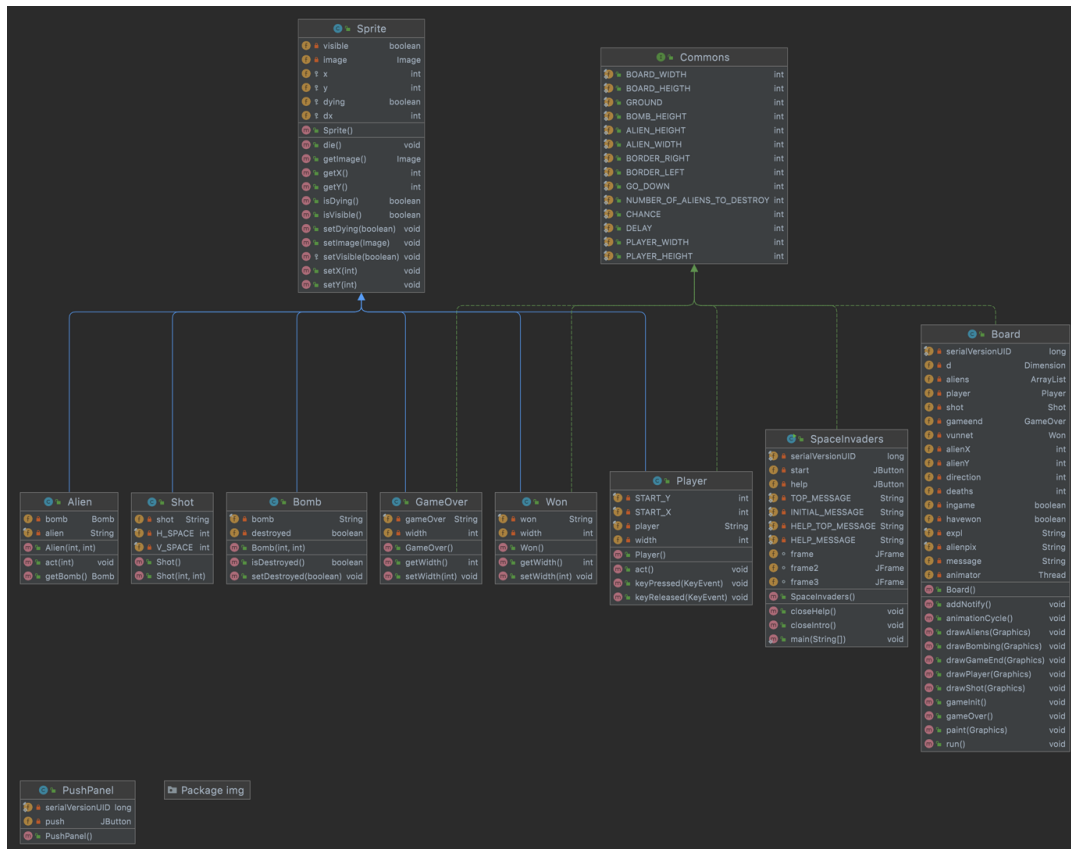
Programming Task: This programming task aims to manually identify, using the available tool, a possible code smell that can be corrected using the Extract Method refactoring.

So, you should follow the next steps:

9. Open the “**MovieRental**” project in your IntelliJ;
10. Analyze the project and try to answer the questions posed on the Google Form;
11. After you have responded to the questions, come back to this guide;
12. Now, start the refactoring tool by clicking on the “**Start Analysis**” option and open the class where you think there is a method where an Extract Method refactoring can be applied;
13. Check if the method you selected in the Google Form is the same indicated by the refactoring tool;
14. If yes, carry out the respective refactoring;
15. If not, and if no refactorings are found in that file, proceed with the same steps but in another file until you find the method that should be refactored.
16. The task is complete when all the refactoring suggested are applied to the code.

Programming Task 2

The second programming task is focused on a medium project that implements the game “Space Invaders”. The following image represents the UML diagram of the game.



Programming Task: This programming task aims to identify, using the available tool, while programming, possible candidate code blocks to be refactored using the Extract Method refactoring.

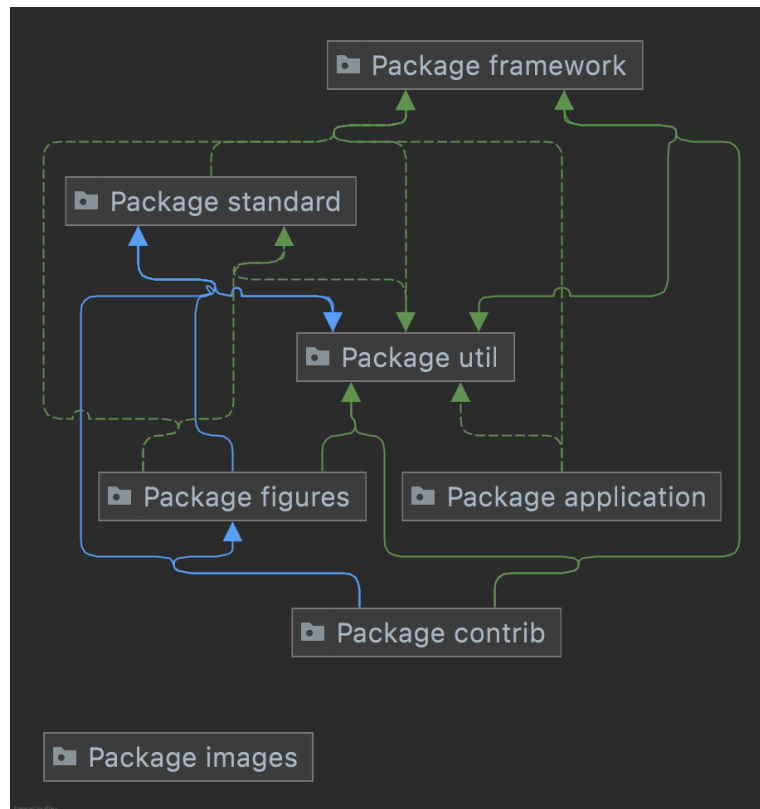
So, you should follow the next steps:

9. Open the “SpaceInvaders” project in your IntelliJ;
10. Analyze the project using the provided diagram;

11. Now, open the Board.java file and start the refactoring tool by clicking on the “**Start Analysis**” option;
12. Focus on the “**animationCycle**” method (class **Board**), which aims to implement the game's animations on the respective board. It will be the method you will have to implement;
13. Within the method itself, you can find comments about the code you must implement;
14. Follow the steps described there;
15. As you are programming, a refactoring suggestion may appear and it should be applied either during the implementation of the code or after finishing its development. You can apply all the suggested refactorings or just the ones you think are suitable for your code;
16. The task is completed either when you apply all the suggested refactorings or when you think that the code is good enough.

Programming Task 3

The third and final programming task is focused on a large-scale project that implements the **JHotDraw** system, which is a two-dimensional graphics framework for structured drawing editors. Since the project has several classes and methods, its UML is not easy to understand when inserted into a file like this. In this way, we just create the simplified UML diagram of the system. Attached to this file are more complete UML diagrams representing the packages where the programming task should be applied.



Programming Task: This programming task aims to identify, using the available tool, all code blocks that must be refactored using the Extract Method refactoring.

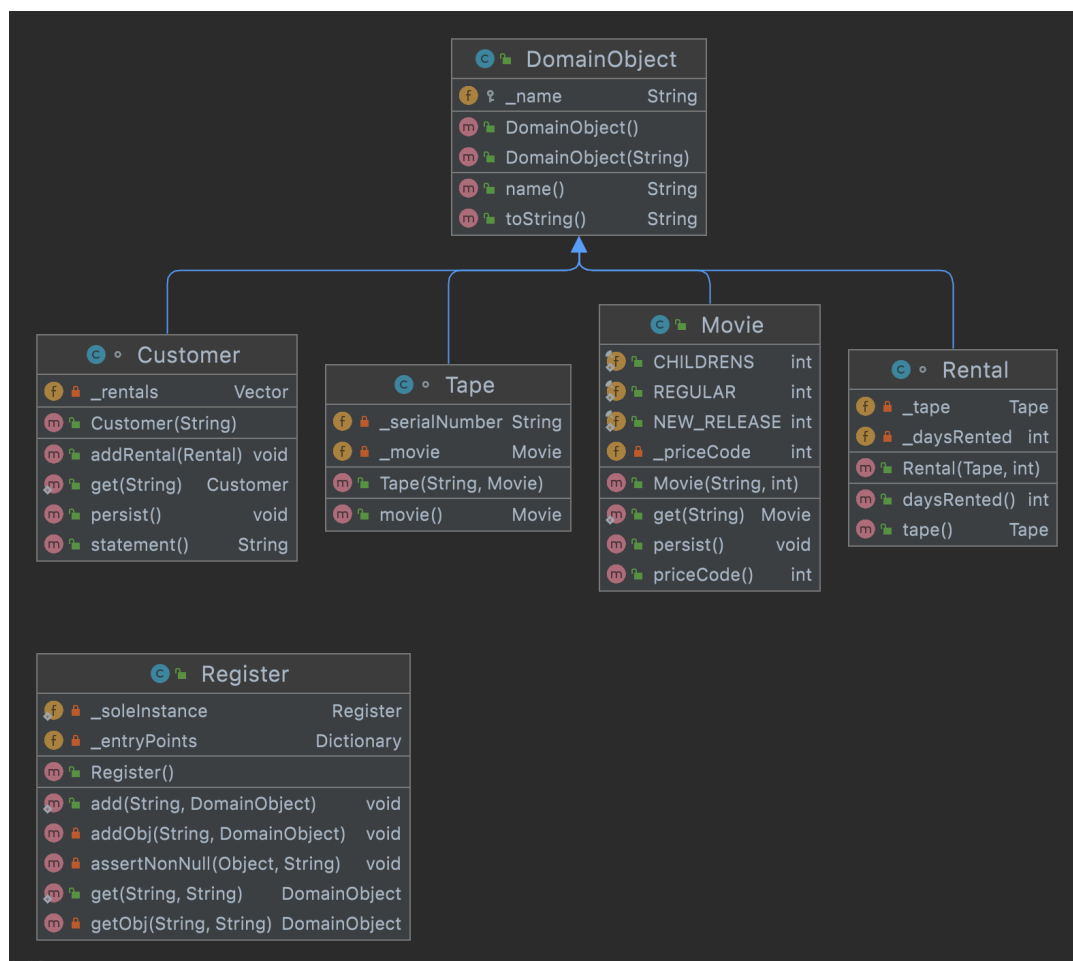
So, you should follow the next steps:

7. Open the “**JHotDraw**” project in your IntelliJ;
8. Analyze the project using the diagram provided to you;
9. Now, open the file **DrawApplication.java** (package **Application**) and start the refactoring tool by clicking on the “**Start Analysis**” option;
10. Then, you will have access to a Extract Method suggestion that represents the refactoring candidate found with more severity;
11. The task in that file is completed when all possible refactorings have been applied;
12. Then, repeat these steps on **StandardDrawingView.java** (package **standard**).

Group B

Programming Task 1

The first programming task is focused on a simple project that represents a movie rental system. The following image represents the UML diagram of that system.



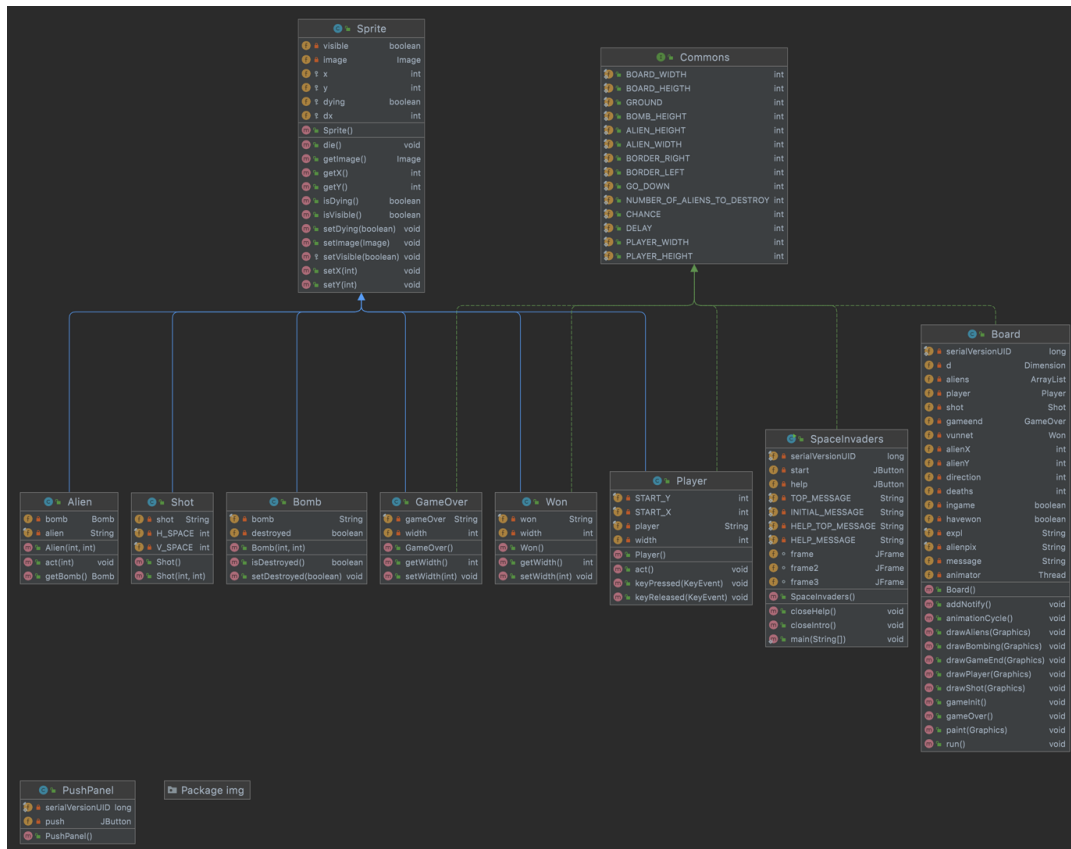
Programming Task: This programming task aims to manually identify, using the available tool, a possible code smell that can be corrected using the Extract Method refactoring.

So, you should follow the next steps:

17. Open the “**MovieRental**” project in your IntelliJ;
18. Analyze the project and try to answer the questions posed on the Google Form;
19. After you have responded to the questions, come back to this guide;
20. Now, start the complementary tool by clicking on the “**Start Experiment**” option and open the class where you think there is a method where an Extract Method refactoring can be applied;
21. Try to manually refactor that method using the Extract Method refactoring.
22. After applying a refactoring, you should click on the option “**Save Metrics**”. Then, you can apply as many refactorings as you want. But don’t forget to save the metrics after each refactoring.
23. The task is complete when you believe that the code as good quality.

Programming Task 2

The second programming task is focused on a medium project that implements the game “Space Invaders”. The following image represents the UML diagram of the game.



Programming Task: This programming task aims to identify, using the available tool, while programming, possible candidate code blocks to be refactored using the Extract Method refactoring.

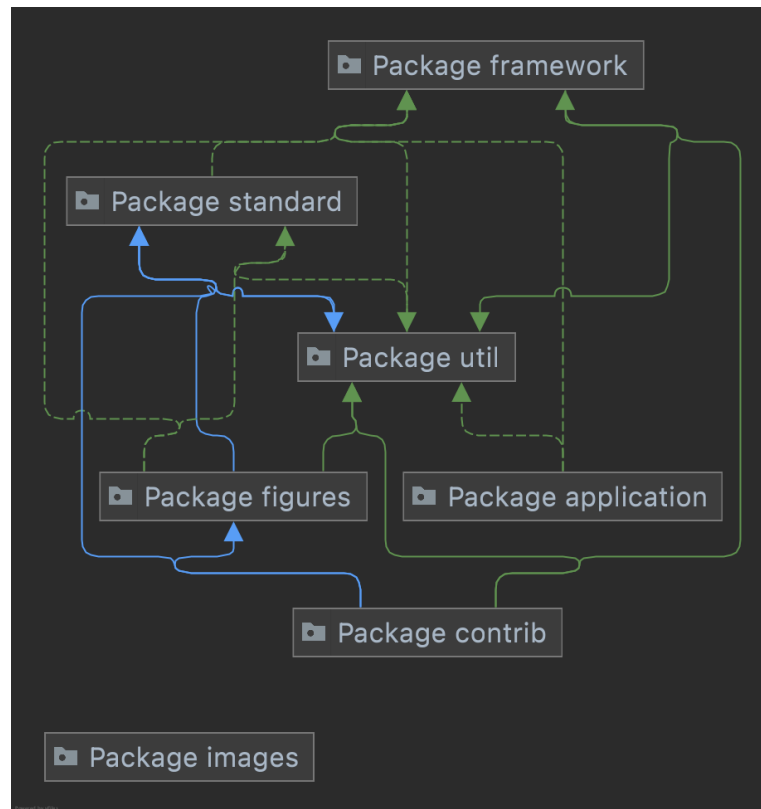
So, you should follow the next steps:

17. Open the “SpaceInvaders” project in your IntelliJ;
18. Analyze the project using the provided diagram;

19. Now, open the Board.java file and start the refactoring tool by clicking on the “**Start Analysis**” option;
20. Focus on the “**animationCycle**” method (class **Board**), which aims to implement the game's animations on the respective board. It will be the method you will have to implement;
21. Within the method itself, you find comments about the code you must implement;
22. Follow the steps described there;
24. After implementing the code, try to refactor it (if you believe the code needs to be refactored). Don't forget to save the metrics after each refactoring applied to the code.
23. The task is completed when you believe that the code has good quality.

Programming Task 3

The third and final programming task is focused on a large-scale project that implements the **JHotDraw** system, which is a two-dimensional graphics framework for structured drawing editors. Since the project has several classes and methods, its UML is not easy to understand when inserted into a file like this. In this way, we just create the simplified UML diagram of the system. Attached to this file are more complete UML diagrams representing the packages where the programming task should be applied.



Programming Task: This programming task aims to identify, using the available tool, all code blocks that must be refactored using the Extract Method refactoring.

So, you should follow the next steps:

13. Open the “**JHotDraw**” project in your IntelliJ;
14. Analyze the project using the diagram provided to you;
15. Now, open the file **DrawApplication.java** (package **Application**) and start the complementary tool by clicking on the “**Start Experiment**” option;
16. There, try to refactor the code implemented on the file (if you believe the code needs to be refactored). Don’t forget to save the metrics after each refactoring applied to the code.
17. The task in that file is completed when you believe that the code has good quality;
18. Then, repeat these steps on **StandardDrawingView.java** (package **standard**).

C.3.2 Experiment Questionnaire

After the initial experiment, its participants had to answer a brief questionnaire. This questionnaire was a Google Form with multiple questions that helped us characterize the participants and their experiment satisfaction and assess our tool's usability and reliability.

A Live Environment for Inspection and Refactoring of Software Systems - Empirical Experiment

We are working on a Live Refactoring Environment. With this approach, software developers will have access to several Extract Method refactoring suggestions that aim to improve the quality of their software systems. This refactoring environment was developed for Java code on the IntelliJ IDE, and it focuses on analyzing specific quality attributes to detect particular code smells. By identifying different smells, this environment selects and applies possible refactoring methods capable of mitigating the respective smells.

We are doing an empirical experiment that will help us evaluate and validate our project. To do it, we need your help!

Please, go further in this form and answer each question accordingly with your opinion...

* Indica uma pergunta obrigatória

Participants Description

Before starting the experiment, we need to know who you are by telling us several aspects of your education and programming experience. Read each question carefully and give the proper answer to your case.

1. Level of Education *

Marcar apenas uma oval.

☐ High School

☐ Bachelor

☐ Master

☐ Ph.D.

☐ Outra: _____

2. Programming Experience *

Marcar apenas uma oval.

	1	2	3	4	5	
Very	☐	☐	☐	☐	☐	Very high

3. Experience programming with Java *

Marcar apenas uma oval.

	1	2	3	4	5	
Very	☐	☐	☐	☐	☐	Very high

4. Experience using different Data Structures *

Marcar apenas uma oval.

	1	2	3	4	5	
Very	☐	☐	☐	☐	☐	Very high

5. Experience using Refactoring tools *

Marcar apenas uma oval.

	1	2	3	4	5	
Very	☐	☐	☐	☐	☐	Very high

6. Experience using Code metrics analysis tools *

Marcar apenas uma oval.

	1	2	3	4	5	
Very	☐	☐	☐	☐	☐	Very high

7. Experience with IntelliJ IDE *

Marcar apenas uma oval.

	1	2	3	4	5	
Very	☐	☐	☐	☐	☐	Very high

8. Are you from Group A? *

Marcar apenas uma oval.

☐	Yes	<i>Avançar para a pergunta 9</i>
☐	No	<i>Avançar para a pergunta 10</i>

Experiment - Group A

Let us know which version of the tool are you using...

9. Will you use Version 1 of the tool? *

Marcar apenas uma oval.

☐	Yes	<i>Avançar para a secção 4 (Programming Tasks (Group A - Version 1))</i>
☐	No	<i>Avançar para a secção 5 (Programming Tasks (Group A - Version 2))</i>

Programming Tasks (Group A - Version 1)

The following link contains the tool, its user guide, and also all the programming tasks that you need to perform: <https://drive.google.com/drive/folders/1SYzndpL7wNhnrDxklJOj7WmO8xgVBTxn?usp=sharing>

Notice that, our approach is only focused on Extract Method Refactorings. An Extract Method Refactoring consists of moving code fragments that could be grouped together from a method to a new one and replacing the old code with a call to the new method.

Avançar para a pergunta 11

Programming Tasks (Group A - Version 2)

The following link contains the tool, its user guide, and also all the programming tasks that you need to perform: <https://drive.google.com/drive/folders/18uxwrGNoy48ODYSbl6lmZ0DQfStasZL-?usp=sharing>

Notice that, our approach is only focused on Extract Method Refactorings. An Extract Method Refactoring consists of moving code fragments that could be grouped together from a method to a new one and replacing the old code with a call to the new method.

Experiment - Group B

Let us know from which group are you...

10. Are you from group A3? *

Marcar apenas uma oval.

- ☐ Yes *Avançar para a secção 8 (Programming Tasks (Group A3))*
- ☐ No *Avançar para a secção 7 (Programming Tasks (Group B))*

Avançar para a pergunta 11

Programming Tasks (Group B)

The following link contains the auxiliary tool, its user guide, and also all the programming tasks that you need to perform:

<https://drive.google.com/drive/folders/1qy2jhZErPhfH5rqYvgtElwXy6G5h2OHW?usp=sharing>

Notice that, our approach is only focused on Extract Method Refactorings. An Extract Method Refactoring consists of moving code fragments that could be grouped together from a method to a new one and replacing the old code with a call to the new method.

Programming Tasks (Group A3)

The following link contains the auxiliary tool, its user guide, and also all the programming tasks that you need to perform:

https://drive.google.com/drive/folders/1ejE_hsl17Yqm2Lx5qHZzMWZKtGjGfy3S?usp=sharing

Notice that, our approach is only focused on Extract Method Refactorings. An Extract Method Refactoring consists of moving code fragments that could be grouped together from a method to a new one and replacing the old code with a call to the new method.

Avançar para a pergunta 20

Experiment Feedback (Group A)

Now, we need some feedback about the experiment. This will help us improve the protocol for future experiments.

11. I'm satisfied with the experiment *

Marcar apenas uma oval.

1 2 3 4 5

Strongly disagree ☐ ☐ ☐ ☐ ☐ Strongly agree

12. The suggestions help understanding where the developer should improve his/hers code *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

13. The refactoring suggested and applied helped create better code *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

14. A "Live Refactoring Environment" helps developers achieve better code, faster *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

15. Suggestions about the experiment (optional)

Avançar para a pergunta 24

Experiment Feedback (Group B)

Now, we need some feedback about the experiment. This will help us improve the protocol for future experiments.

Besides, as you didn't have the chance to use the developed tool now you can see a quick video about it: <https://drive.google.com/file/d/15VX-1ogb7CYiwqWJ1biCWWrt0SHSrlwz/view?usp=sharing>

16. I'm satisfied with the experiment *

Marcar apenas uma oval.

1 2 3 4 5

Stro ☐ ☐ ☐ ☐ ☐ Strongly agree

17. A "Live Refactoring Environment" helps developers achieve better code, faster *

Marcar apenas uma oval.

1 2 3 4 5

Stro ☐ ☐ ☐ ☐ ☐ Strongly agree

18. Would you like to use a "Live Refactoring Environment" *

Marcar apenas uma oval.

☐ Yes

☐ No

19. Suggestions about the experiment (optional)

Experiment Feedback (Group A3)

Now, we need some feedback about the experiment. This will help us improve the protocol for future experiments.

Besides, as you didn't have the chance to use the developed tool now you can see a quick video about it: <https://drive.google.com/file/d/15VX-1ogb7CYiwqWJ1biCWWrt0SHSrlwz/view?usp=sharing>

20. I'm satisfied with the experiment *

Marcar apenas uma oval.

1 2 3 4 5

Stro ☐ ☐ ☐ ☐ ☐ Strongly agree

21. A "Live Refactoring Environment" helps developers achieve better code, faster *

Marcar apenas uma oval.

1 2 3 4 5

Stro ☐ ☐ ☐ ☐ ☐ Strongly agree

22. Would you like to use a "Live Refactoring Environment" that displays the candidates directly in the text editor

Marcar apenas uma oval.

☐ Yes

☐ No

23. Suggestions about the experiment (optional)

Usability Questions

Here you will have some usability questions related to our tool. With your answers, we will be able to improve the tool and make it more usable to developers.

24. I think that I would like to use this tool frequently *

Marcar apenas uma oval.

1 2 3 4 5

Stro ☐ ☐ ☐ ☐ ☐ Strongly agree

25. I found the tool unnecessarily complex *

Marcar apenas uma oval.

1 2 3 4 5

Stro ☐ ☐ ☐ ☐ ☐ Strongly agree

26. I thought the tool was easy to use *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

27. I think that I would need the support of a technical person to be able to use this tool *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

28. I found the various functions in this tool were well integrated *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

29. I thought there was too much inconsistency in this tool *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

30. I would imagine that most people would learn to use this tool very quickly *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

31. I thought it difficult to use the tool on the correspondent project *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

32. I found the tool very cumbersome to use *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

33. I felt very confident using the tool *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

34. I needed to learn a lot of things before I could get going with this tool *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

35. The color scheme used to represent the possible refactoring techniques was adequate *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

36. The placement of the interface was adequate *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

37. The placement of the refactoring suggestions was adequate *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

38. The number of suggestions was adequate *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

39. I believe that the tool reduces the effort needed to create better software systems *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

40. I believe that the tool will make it easier for developers to verify if their code has quality or not

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

41. I found the tool useful *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

42. I believe that using the approach will make it more difficult to maintain large and complex software systems

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

43. Would you like to see the evolution of the code quality metrics on this tool? *

Marcar apenas uma oval.

☐ Yes

☐ No

44. Suggestions about the tool usability (optional)

Este conteúdo não foi criado nem aprovado pela Google.

Google Formulários

C.4 Live Refactoring Experiment

In the second experiment, its participants followed a specific guide. After, they also had to fill out a particular questionnaire.

C.4.1 Experiment Guide

This section presents the guide followed by the participants from our second experiment.

A Live Environment for Inspection and Refactoring of Software Systems - Empirical Experiment

We worked on a **Live Refactoring Environment**. With this approach, software developers have access to several **refactoring suggestions** that aim to improve the quality of their software systems. This refactoring environment was developed for **Java** code on the **IntelliJ IDE**, and it focuses on analyzing specific quality attributes to identify particular refactorings.

Our tool is named **LiveRef**, and it includes the following refactorings:

- Extract Method
- Extract Class
- Extract Variable
- Move Method
- Inheritance to Delegation
- Introduce Parameter Object
- String Comparison

But, you may ask how this tool is different from the other existing refactoring tools. **LiveRef** allows developers to be guided to better code, through refactoring suggestions, while programming. They don't need to click on a button whenever they want to check their refactoring possibilities. Besides, with it, developers can visualize each refactoring in a more user-friendly way than the conventional and well-known refactoring tools.

Basically, this project was developed assuming that *software developers aim to create great software systems, with more quality, faster.*

To validate our beliefs and evaluate our tool, we should perform an empirical experiment with several participants. That's why we need your help!

To help us measure the quality of each project and its evolution, our tool collects several code quality metrics while you program and in each refactoring performed on your code. Then, these values are automatically saved on a Firebase database.

But don't worry! Your identity will always be kept anonymous.

Also, notice that even with this tool, there is a chance that your code quality metrics may get worsen when applying each suggested refactoring. Each refactoring displayed by **LiveRef** is merely a suggestion and not a mandatory action that must be applied to your code.

If you have any question on this tool and how to use it, please send an email to sfcf@fe.up.pt .

User Guide

To start the experiment, you should launch the **IntelliJ IDE** and install our tool **LiveRef**. Please download the current version of **LiveRef** (depending on your IntelliJ version). To install it, you need to select your IDE preferences/settings. There you will find the “**Plugins**” menu, where you must select the option of “**Install Plugin from Disk**” (see the following image). Search the folder where you saved the attached .zip (file **LiveRef.zip**). Then, IntelliJ will install the plugin. Probably, you will need to restart IntelliJ to have full access to **LiveRef**.

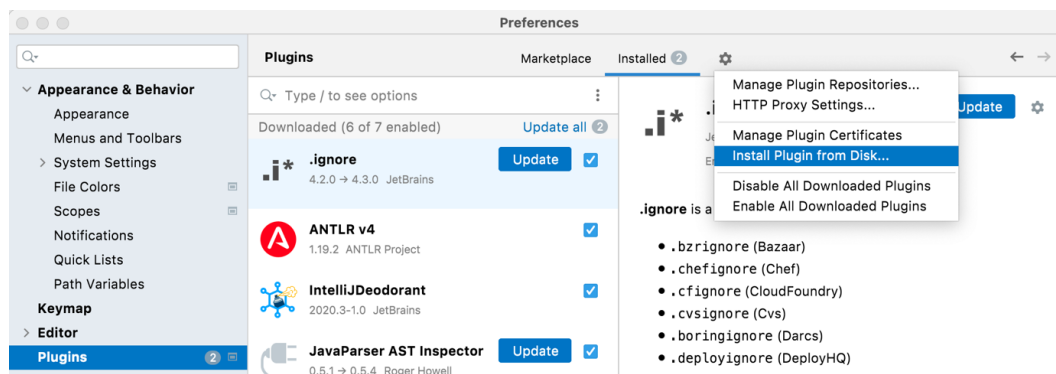


Figure 1 - How to install LiveRef.

After installing the plugin, you can activate the tool. The next steps depend on the IDE version you are using.

- For **IntelliJ 2021**, you should click on the option “**Live Refactoring**” on the main menu;
- For **IntelliJ 2022**, you will have a popup dialog showing when you open your project and a menu named “**Live Refactoring**” inside the “**Tools**” option on the main menu. If you activate the tool through the popup dialog, it starts automatically. If you choose to not activate it using the dialog, you can always start it through the “**Live Refactoring**” menu.

On the “**Live Refactoring**” menu, you will have access to two new options:

1. **Start Analysis** – It activates the tool and starts the analysis to find refactoring candidates;
2. **Configure Tool** – It allows the users to configure the tool and the thresholds for each refactoring.

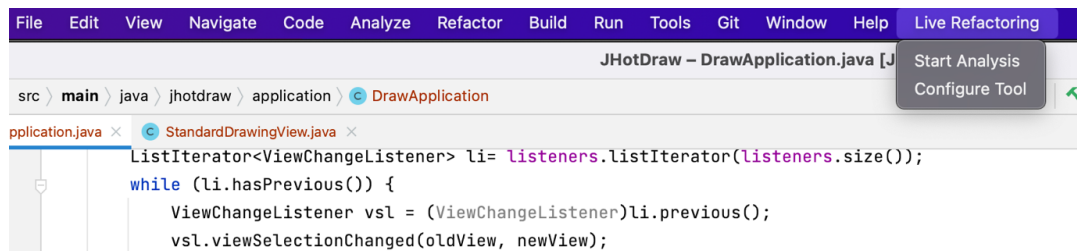


Figure 2 - "Live Refactoring" menu when using IntelliJ 2021.

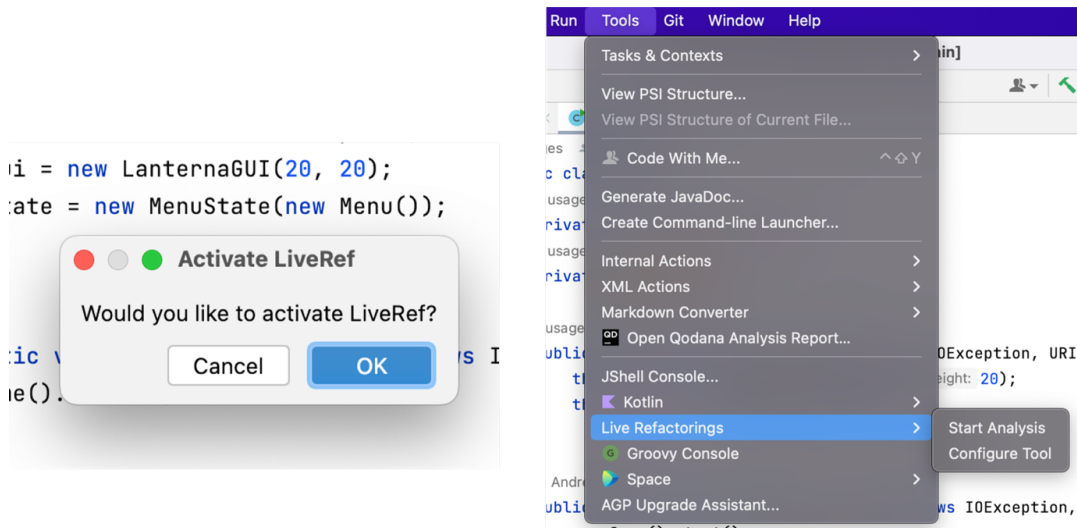


Figure 3 - Popup dialog and "Live Refactoring" menu when using IntelliJ 2022.

Start Analysis

After clicking on “**Start Analysis**” (or after activating the tool through popup dialog), the tool inspects the code and presents the different refactoring candidates. To simplify the suggestion of each refactoring, we display each candidate through different colors ranging from *light green* to *dark red*. The assigned colors were based on the number of candidates per refactoring type. **Light green** represents less impactful candidates and **dark red** highest impactful refactorings.

```

ArrayList<Entity> entities = new ArrayList<>();
entities.addAll(sourceClass.getAttributeList());
entities.addAll(sourceClass.getMethodList());
double[][] jaccardDistanceMatrix = new double[entities.size()][entities.size()];
for (int i = 0; i < jaccardDistanceMatrix.length; i++) {
    for (int j = 0; j < jaccardDistanceMatrix.length; j++) {
        if (i != j) {
            jaccardDistanceMatrix[i][j] = DistanceCalculator.getDistance(entities.get(i));
        } else {
            jaccardDistanceMatrix[i][j] = 0.0;
        }
    }
}
return jaccardDistanceMatrix;
}

```

Figure 4 - Displaying refactorings using multiple colors.

Then, if you click on a color, you will have access to a refactoring menu. This menu lists the refactoring candidates related to the line of code represented by that color. Imagine that there are overlapping refactorings related to that line of code. To simplify, we display the color of just one of those refactorings. But on the menu, the users have access to all refactorings that include that line.

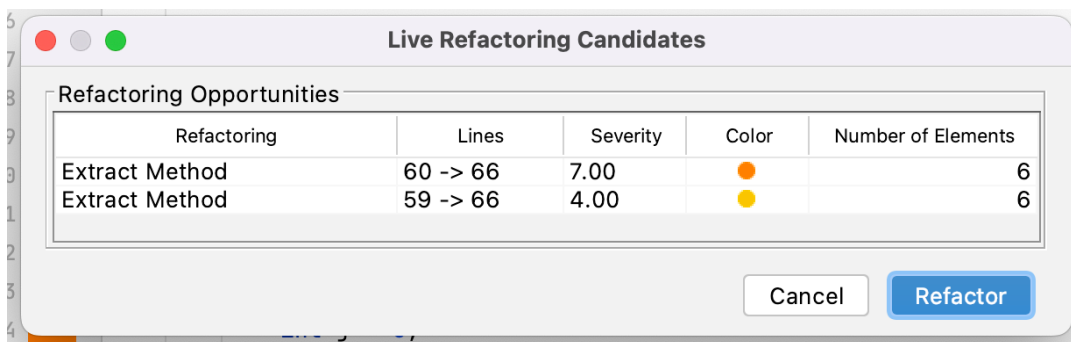


Figure 5 - Refactoring menu listing multiple refactoring candidates.

If you desire to perform one of the refactorings listed on the refactoring menu, you need to select it and click the “**Refactor**” button. Then, the refactoring activity will start.

After applying the refactoring, the tool sends the measured metrics to Firebase, starts a new code inspection, and displays the new refactoring candidates.

Note that the tool may not suggest all refactorings at the same time. As it identifies possible refactoring candidates by their type, it recommends them (e.g., there may be a time-lapse of seconds or less between listing Extract Variable and Extract classes candidates).

If the tool doesn't find any refactoring candidates, a message pops up in the bottom left corner of IntelliJ.

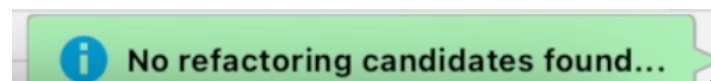


Figure 6 - Message displayed when no refactoring is found.

At any time, you can stop the tool from inspecting your code. To do so, you need to click on the “**Stop Analysis**” option, which is only available after you start the tool.

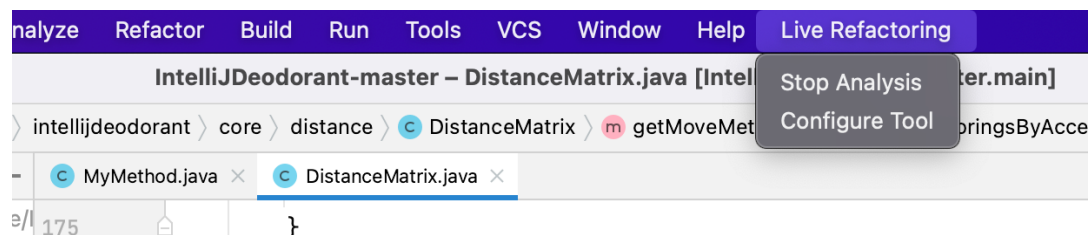


Figure 7 – “Stop Analysis” option displayed when using IntelliJ 2021.

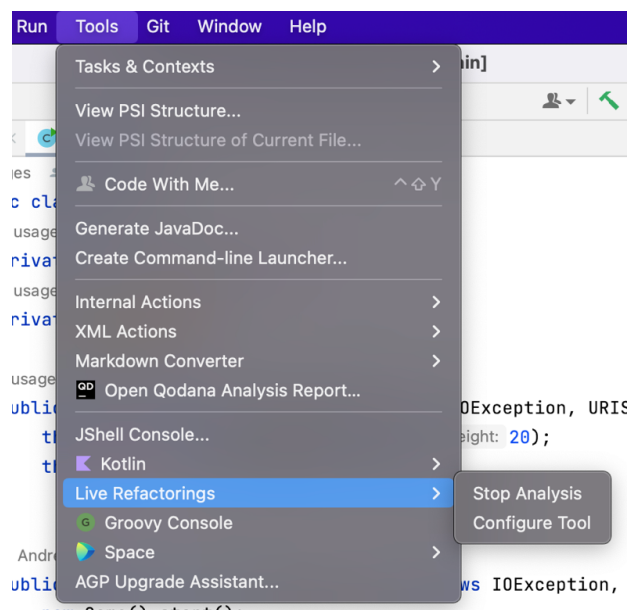


Figure 8 - “Stop Analysis” option displayed when using IntelliJ 2022.

Configure Tool

If you click on the “*Configure Tool*” option, you access the configuration menu. You can select the refactoring to include in the inspection process, change some **thresholds** used to detect or sort the refactoring candidates, **limit the number of refactorings** to be displayed (by refactoring type), or identify if you are color-blind. For color-blind people, the tool uses symbols instead of colors to list the refactoring candidates.

Configure Tool

☐ Extract Method
☐ Extract Variable
☐ Introduce Parameter Object
☐ Inheritance To Delegation

☐ Extract Class
☐ Move Method
☐ String Comparison
☒ All Refactorings

Extract Class

Min. Percentage Original Statements: 70.0%
 Min. Num. Methods to extract: 3.0
 Min. Num. Methods Class: 4.0
 Min. Lack of Cohesion: 0.33
 Min. Num. Foreign Data: 2.0

Extract Method

Min. Percentage Original Methods: 70.0%
 Min. Num. Statements to extract: 3.0

Further Details

Max. Num. of Refactorings: All Refactorings
☐ Color blind

By selecting all available refactorings, you may be decreasing the plugin's performance.

Cancel OK

Extract Class

- **Min. Original Methods:** Minimum percentage of original methods to ensure that the original class maintains a percentage of its original methods, or it will be turned into an empty class;
- **Min. Num. Methods to extract:** Minimum number of methods to extract to ensure that the extracted class contains at least a specific number of methods;
- **Min. Num. Methods Class:** Minimum number of methods of the class being analyzed to ensure that the tool only inspects classes long enough and not simple and small ones;

- **Min. Lack of Cohesion:** Minimum lack of cohesion to ensure that the tool only analyzes low cohesive classes (highest lack of cohesion represents non-cohesive classes). Lack of cohesion should be a value between 0 and 1;
- **Min. Num Foreign Data:** Minimum number of foreign data to ensure that the tool only analyzes classes that use several foreign methods or fields from other classes.

Extract Method

- **Min. Percentage Original Statements:** Maximum percentage of original statements to ensure that the original method maintains a percentage of its original statements, or it will be turned into an empty method;
- **Min. Num. Statements to extract:** Minimum number of statements to extract to ensure that the tool doesn't identify too small and simple Extract method candidates;
- **Min. Num. Lines of Code:** Minimum number of lines of code of the method being analyzed to ensure that the tool only inspects methods long enough and not small ones;
- **Min. Cyclomatic Complexity:** Minimum cyclomatic complexity of the method being analyzed to ensure that the tool doesn't analyze not so complex methods;
- **Min. Halstead Effort:** Minimum Halstead effort of the method being analyzed to ensure that the tool doesn't inspect methods that are easy to understand.

Extract Variable

- **Min. Length of Expressions:** Minimum length of the expressions that should be inspected as possible Extract variable candidates to ensure that the tool doesn't explore small and simple expressions.

Introduce Parameter Object

- **Min. Num. Parameters:** Minimum number of parameters of the methods that should be analyzed to check if they are suitable to be an Introduce Parameter Object refactoring.

Inheritance To Delegation

- ***Max. Percentage inherited Methods:*** Maximum percentage of inherited methods that the subclass being analyzed should have to ensure that the tool only inspects classes that don't inherit enough methods from their superclass;
- ***Max. Percentage Override Methods:*** Maximum percentage of overridden methods that the subclass being analyzed should have to ensure that the tool only inspects classes that don't override enough methods from their superclass.

Included Refactorings:

- **Extract Method:** The *Extract Method* refactoring extracts some code from an existing method to a new one. Its purpose is to improve code by splitting a long and/or complex method into simpler ones;
- **Extract Class:** The *Extract Class* refactoring is applied when a class becomes overweight with too many methods, and its purpose becomes unclear. It involves creating a new class and moving methods and/or data to the new class;
- **Extract Variable:** The *Extract Variable* refactoring consists of placing the result of an expression in a variable to make it easier to understand;
- **Move Method:** The *Move Method* refactoring consists of moving a specific method from its original class to another one in which the method is used more often than its own original class;
- **Inheritance to Delegation:** The *Inheritance to Delegation* refactoring creates a superclass object, delegates methods to it, and gets rid of the inheritance. This refactoring is suitable for solving cases in which the subclass only uses a portion of the superclass methods or fields;
- **Introduce Parameter Object:** The *Introduce Parameter Object* refactoring replaces a long list of parameters of a class with a new object;
- **String Comparison:** The *String Comparison* refactoring allows to replace string comparisons using the operator “==” with method *equals()*.

C.4.2 Experiment Questionnaire

When the second experiment ended, we sent a brief questionnaire to its participants. This questionnaire was a Google Form with multiple questions that helped us characterize the participants and assess our tool's usability and reliability.

A Live Environment for Inspection and Refactoring of Software Systems - Empirical Experiment

Now that you used **LiveRef**, go further in this form and answer each question accordingly with your opinion.

** Indica uma pergunta obrigatória*

Participants Description

1. Programming Experience *

Marcar apenas uma oval.

	1	2	3	4	5	
Very	☐	☐	☐	☐	☐	Very high

2. Experience programming with Java *

Marcar apenas uma oval.

	1	2	3	4	5	
Very	☐	☐	☐	☐	☐	Very high

3. Experience using refactoring tools *

Marcar apenas uma oval.

	1	2	3	4	5	
Very	☐	☐	☐	☐	☐	Very high

4. Experience with IntelliJ IDE *

Marcar apenas uma oval.

	1	2	3	4	5	
Very	☐	☐	☐	☐	☐	Very high

Usability Questions

5. The suggestions help understand the code that should be refactored *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

6. The refactorings suggested and applied helped create better code *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

7. The refactorings suggested and applied helped create better code, faster *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

8. I think that I would like to use this tool in the future *

Marcar apenas uma oval.

1 2 3 4 5

Stro ☐ ☐ ☐ ☐ ☐ Strongly agree

9. I think that I would like to use this tool frequently *

Marcar apenas uma oval.

1 2 3 4 5

Stro ☐ ☐ ☐ ☐ ☐ Strongly agree

10. I found the tool unnecessarily complex *

Marcar apenas uma oval.

1 2 3 4 5

Stro ☐ ☐ ☐ ☐ ☐ Strongly agree

11. I thought the tool was easy to use *

Marcar apenas uma oval.

1 2 3 4 5

Stro ☐ ☐ ☐ ☐ ☐ Strongly agree

12. I found the various functions in this tool were well integrated *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

13. I thought there was too much inconsistency in this tool *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

14. I would imagine that most people would learn to use this tool very quickly *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

15. I felt very confident using the tool *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

16. Do you think the refactorings included on the tool were adequate? *

Marcar apenas uma oval.

☐ Yes

☐ No

17. The color scheme used to represent the possible refactoring techniques was adequate *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

18. The placement of the interface was adequate *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

19. The placement of the refactoring suggestions was adequate *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

20. The number of suggestions was adequate *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

21. I found the tool useful *

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

22. I believe that using the approach will make it easier to maintain large and complex software systems

Marcar apenas uma oval.

	1	2	3	4	5	
Stro	☐	☐	☐	☐	☐	Strongly agree

23. Would you like to see the evolution of the code quality metrics on the tool? *

Marcar apenas uma oval.

☐ Yes

☐ No

24. Suggestions about LiveRef (optional)

Este conteúdo não foi criado nem aprovado pela Google.

Google Formulários

C.5 Automated Experiment

The third experiment consisted of a robotized experiment using a refactoring bot that crawled different projects, inspected them, and automatically refactored them. Therefore, there was no specific experiment guide to follow in this case.

At the end of this experiment, we asked several programming students to fill out six small questionnaires that helped us evaluate if they saw any relevance in refactoring code using specific refactoring techniques.

C.5.1 Questionnaire on Extract Method

The following was the questionnaire the participants had to complete on the *Extract Method* refactoring.

Extract Method

"Extract Method" is a code refactoring technique that involves breaking down a complex section of code into smaller, more manageable methods or functions.

* Indica uma pergunta obrigatória

1. The presented image illustrates an example of "Extract Method".

After applying the respective refactoring, the code has improved, now having higher quality, being less complex, and easier to understand, analyze, and modify.

Before

```

public static EventType<Void> domMutation(Consumer<DomMutationEvent> handler) {
    Require.nonNull( argName: "Handler", handler);

    URL url = CdpEventTypes.class.getResource( name: "/org/openqa/selenium/devtools/mutation-listener.js");
    if (url == null) {
        throw new IllegalStateException("Unable to find helper script");
    }
    String script;
    try {
        script = Resources.toString(url, UTF_8);
    } catch (IOException e) {
        throw new IllegalStateException("Unable to read helper script");
    }

    return new EventType<Void>() {...};
}

```

After

```

public static EventType<Void> domMutation(Consumer<DomMutationEvent> handler) {
    Require.nonNull( argName: "Handler", handler);

    String script = eventMutation();

    return new EventType<Void>() {...};
}

```

New Method

```

private static String eventMutation() {
    URL url = CdpEventTypes.class.getResource( name: "/org/openqa/selenium/devtools/mutation-listener.js");
    if (url == null) {
        throw new IllegalStateException("Unable to find helper script");
    }
    String script;
    try {
        script = Resources.toString(url, UTF_8);
    } catch (IOException e) {
        throw new IllegalStateException("Unable to read helper script");
    }
    return script;
}

```

Marcar apenas uma oval.

1 2 3 4 5

Stro ☐ ☐ ☐ ☐ ☐ Strongly agree

C.5.2 Questionnaire on Extract Class

The following was the questionnaire the participants had to complete on the *Extract Class* refactoring.

Extract Class

"Extract Class" is a code refactoring technique that involves breaking down a class into smaller, more focused classes.

* Indica uma pergunta obrigatória

1. The presented image illustrates an example of "Extract Class".

After applying the respective refactoring, the code has improved, now having higher quality, being less complex, and easier to understand, analyze, and modify.

Before

```

public class BitmapTransitionOptions {
    @NonNull
    public BitmapTransitionOptions crossFade(
        @NonNull DrawableCrossFadeFactory drawableCrossFadeFactory) {
        return transitionUsing(drawableCrossFadeFactory);
    }

    @NonNull
    public static BitmapTransitionOptions withWrapped(
        @NonNull TransitionFactory<Drawable> drawableCrossFadeFactory) {
        return new BitmapTransitionOptions().transitionUsing(drawableCrossFadeFactory);
    }

    @NonNull
    public static BitmapTransitionOptions with(@NonNull TransitionFactory<Bitmap> transitionFactory) {
        return new BitmapTransitionOptions().transition(transitionFactory);
    }

    @NonNull
    public BitmapTransitionOptions crossFade() {
        return crossFade(new DrawableCrossFadeFactory.Builder());
    }

    @NonNull
    public BitmapTransitionOptions transitionUsing(
        @NonNull TransitionFactory<Drawable> drawableCrossFadeFactory) {

```

After

```

public class newClass{
    @NonNull
    public static BitmapTransitionOptions withWrapped(
        @NonNull TransitionFactory<Drawable> drawableCrossFadeFactory) {
        return new BitmapTransitionOptions().transitionUsing(drawableCrossFadeFactory);
    }

    @NonNull
    public static BitmapTransitionOptions with(@NonNull TransitionFactory<Bitmap> transitionFactory) {
        return new BitmapTransitionOptions().transition(transitionFactory);
    }

    @NonNull
    public BitmapTransitionOptions crossFade() {
        return crossFade(new DrawableCrossFadeFactory.Builder());
    }
}

```

Marcar apenas uma oval.

1 2 3 4 5

Stro ☐ ☐ ☐ ☐ ☐ Strongly agree

C.5.3 Questionnaire on Extract Variable

The following was the questionnaire the participants had to complete on the *Extract Variable* refactoring.

Extract Variable

"Extract Variable" is a code refactoring technique used to improve code readability and maintainability by breaking down complex expressions or values into named variables with meaningful names.

* Indica uma pergunta obrigatória

1. The presented image illustrates an example of "Extract Variable".

After applying the respective refactoring, the code has improved, now having higher quality, being less complex, and easier to understand, analyze, and modify.

Before

```
return getPhotoCacheDir(context, DEFAULT_DISK_CACHE_DIR);
```

After

```
File photoCacheDir = getPhotoCacheDir(context, DEFAULT_DISK_CACHE_DIR);
```

```
return photoCacheDir;
```

Marcar apenas uma oval.

1 2 3 4 5

Stro ☐ ☐ ☐ ☐ ☐ Strongly agree

Este conteúdo não foi criado nem aprovado pela Google.

Google Formulários

C.5.4 Questionnaire on Move Method

The following was the questionnaire the participants had to complete on the *Move Method* refactoring.

Move Method

"Move Method" is a code refactoring technique that involves relocating a method from one class to another to enhance code organization and improve maintainability.

* Indica uma pergunta obrigatória

1. The presented image illustrates an example of "Move Method".

After applying the respective refactoring, the code has improved, now having higher quality, being less complex, and easier to understand, analyze, and modify.

Before

```
public class DevTools implements Closeable {
    private static final Logger LOG = Logger.getLogger(DevTools.class.getName());

    private final Domains protocol;
    private final Duration timeout = Duration.ofSeconds(10);
    private final Connection connection;
    private SessionID cdpSession = null;

    public DevTools(Function<DevTools, Domains> protocol, Connection connection) {
        this.connection = Require.nonNull( argName: "WebSocket connection", connection);
        this.protocol = Require.nonNull( argName: "CDP protocol", protocol).apply( t: this);
    }

    public Domains getDomains() { return protocol; }

    @Override
    public void close() {...}

    public void disconnectSession() {...}

    public <X> X send(Command<X> command) {...}

    public <X> void addListener(Event<X> event, Consumer<X> handler) {...}

    public void clearListeners() {...}
}
```

After

```
public class DevTools implements Closeable {
    private static final Logger LOG = Logger.getLogger(DevTools.class.getName());

    private final Domains protocol;
    private final Duration timeout = Duration.ofSeconds(10);
    private final Connection connection;
    private SessionID cdpSession = null;

    public DevTools(Function<DevTools, Domains> protocol, Connection connection) {
        this.connection = Require.nonNull( argName: "WebSocket connection", connection);
        this.protocol = Require.nonNull( argName: "CDP protocol", protocol).apply( t: this);
    }
}
```

```

    }

    public Domains getDomains() { return protocol; }

    @Override
    public void close() {...}

    public <X> X send(Command<X> command) {...}

    public <X> void addListener(Event<X> event, Consumer<X> handler) {...}

    public void clearListeners() {...}

    public void createSessionIfThereIsNotOne() { createSessionIfThereIsNotOne( windowHandle: null); }

```

Target Class

```

public class Connection implements Closeable {

    private static final Logger LOG = Logger.getLogger(Connection.class.getName());
    private static final Json JSON = new Json();
    private static final Executor EXECUTOR =
        Executors.newCachedThreadPool(
            r -> {
                Thread thread = new Thread(r, name: "BiDi Connection");
                thread.setDaemon(true);
                return thread;
            });
    private static final AtomicLong NEXT_ID = new AtomicLong( initialValue: 1L);
    private final WebSocket socket;
    private final Map<Long, Consumer<Either<Throwable, JsonInput>>> methodCallbacks =
        new ConcurrentHashMap<>();
    private final ReadWriteLock callbacksLock = new ReentrantReadWriteLock( fair: true);
    private final HttpClient client;

    public Connection(HttpClient client, String url) {...}

    public void disconnectSession() {...}

    private static class NamedConsumer<T> implements Consumer<X> {...}

```

Marcar apenas uma oval.

1 2 3 4 5

Stro ☐ ☐ ☐ ☐ ☐ Strongly agree

C.5.5 Questionnaire on Introduce Parameter Object

The following was the questionnaire the participants had to complete on the *Introduce Parameter Object* refactoring.

Introduce Parameter Object

"Introduce Parameter Object" is a refactoring technique used in software development to improve code by grouping related parameters into a single object or class.

* Indica uma pergunta obrigatória

1. The presented image illustrates an example of "Introduce Parameter Object".

After applying the respective refactoring, the code has improved, now having higher quality, being less complex, and easier to understand, analyze, and modify.

Before

```
BufferedStartupStep(BufferedStartupStep parent, String name, long id, Instant startTime,
    Consumer<BufferedStartupStep> recorder) {
    this.parent = parent;
    this.name = name;
    this.id = id;
    this.startTime = startTime;
    this.recorder = recorder;
}
```

After

```
BufferedStartupStep(Startup startup) {
    this.parent = startup.getParent();
    this.name = startup.getName();
    this.id = startup.getId();
    this.startTime = startup.getStartTime();
    this.recorder = startup.getRecorder();
}
```

Marcar apenas uma oval.

1 2 3 4 5

Stro ☐ ☐ ☐ ☐ ☐ Strongly agree

Este conteúdo não foi criado nem aprovado pela Google.

Google Formulários

C.5.6 Questionnaire on Replace Inheritance with Delegation

The following was the questionnaire the participants had to complete on the *Replace Inheritance with Delegation* refactoring.

Replace Inheritance with Delegation

"Replace Inheritance with Delegation" is a refactoring technique that involves replacing inheritance relationships between classes with delegation, typically by creating an instance of another class and forwarding calls to it instead of inheriting behaviour directly.

* Indica uma pergunta obrigatória

1. The presented image illustrates an example of "Replace Inheritance with Delegation."

After applying the respective refactoring, the code has improved, now having higher quality, being less complex, and easier to understand, analyze, and modify.

Before

```

public abstract class DeferredScalarObserver<T, R>
    extends DeferredScalarDisposable<R>
    implements Observer<T> {

    private static final long serialVersionUID = -266195175408988651L;

    /** The upstream disposable. */
    protected Disposable upstream;

    /** Creates a DeferredScalarObserver instance and wraps a downstream Observer. ...*/
    public DeferredScalarObserver(Observer<? super R> downstream) { super(downstream); }

    @Override
    public void onSubscribe(Disposable d) {
        if (DisposableHelper.validate(this.upstream, d)) {...}
    }
}

```

After

```

public abstract class DeferredScalarObserver<T, R>
    implements Observer<T> {

    private static final long serialVersionUID = -266195175408988651L;
    protected final MyDeferredScalarDisposable deferredScalarDisposable;

    public void onError(Throwable t) {
        deferredScalarDisposable.value = null;
        error(t);
    }

    @Override
    public void onComplete() {
        R v = deferredScalarDisposable.value;
        if (v != null) {
            deferredScalarDisposable.value = null;
            complete(v);
        } else {
            complete();
        }
    }
}

```

Marcar apenas uma oval.

1 2 3 4 5

Stro ☐ ☐ ☐ ☐ ☐ Strongly agree

