

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Towards a Live Refactoring Recommender Based on Code Smells and Quality Metrics

Sérgio António Dias Salgado



Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Ademar Aguiar

Co-Supervisor: Sara Fernandes

July 22, 2020

Towards a Live Refactoring Recommender Based on Code Smells and Quality Metrics

Sérgio António Dias Salgado

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. João Carlos Viegas Martins Bispo

External Examiner: Prof. Eduardo Martins Guerra

Supervisor: Prof. Ademar Manuel Teixeira de Aguiar

July 22, 2020

Abstract

As software systems evolve throughout their development, they begin to lose structure and the architecture starts to decay, leading to a decrease of code comprehension by current and future developers of the system, which may end up severely crippling the development of new features in it. By applying refactoring techniques, we are improving the internal structure of the system without changing its behavior, allowing for software to remain easy to understand and, more importantly, easy to extend. In the long run, this easiness of development leads to faster programming, better software quality and lower development costs.

Although refactoring and quality metric analysis tools are not necessarily a novelty and are implemented either by default in some mainstream IDEs such as Eclipse and IntelliJ in the form of plugins, or in tools like Visual Studio Code in the form of extensions, little to none of the existent tools unite these two concepts with the emerging concept of liveness, which aims to open new challenges for agile software development, in terms of technical agility.

In order to provide developers with an environment that better fits the live software development philosophy, we propose the development of a refactoring recommendation tool targeting TypeScript and JavaScript systems, in form of a Visual Studio Code extension, which provides the user with live visualization of a wide range of software quality metrics and the possibility of executing optimal refactorings based on extraction, such as Extract Method, Extract Class and Extract Variable (whose conditions are customizable by the user) to the source code, allowing for live observation of the impact of these changes in the values of quality metrics.

To validate the developed tool, a survey on usability and feature set was performed, in order to understand how the tool can impact the workflow of developers, via the information provided by its interface. We tested the correlation between the agility of assessment by the developers and their awareness for taking action. Another part of the validation was to run our tool on multiple documents across various public repositories and versions, to understand if the usage of our tool resulted in a more maintainable code. Hypothesis tests ran both on the correlation between developer awareness and action and whether our refactorings lead to improvements in the source code resulted in some of the null hypothesis not being rejected, thus not validating the main hypothesis in its entirety.

We believe that providing developers, ranging from beginners to experts, with a live environment with these characteristics will allow them to identify problems in the system faster and stay aware of the impact the changes to source code actually have on quality metrics. By providing the possibility of executing refactorings based on these metrics, it allows the programmer to maintain certain quality metrics within limits, leading to a more sustainable software system.

Keywords: agile software development, live software development, refactoring recommendation system, software maintenance

Resumo

À medida que os sistemas de *software* evoluem durante o seu desenvolvimento, estes começam a perder estrutura e a sua arquitetura vai-se degradando, levando a um défice na compreensão do código por atuais e futuros programadores do sistema, podendo levar à incapacidade de desenvolvimento de novas funcionalidades neste. Através da aplicação de técnicas de *refactoring*, estamos a melhorar a estrutura interna do sistema, sem alterar o seu comportamento, permitindo a este se manter simples de compreender e, mais importante, simples de estender. A longo prazo, esta facilidade no desenvolvimento leva a uma programação mais rápida, melhor qualidade de *software* e redução nos custos de desenvolvimento.

Embora ferramentas de *refactoring* e análise de métricas de qualidade não sejam necessariamente uma novidade e estão implementadas quer por defeito em *IDEs* tais como o Eclipse ou o IntelliJ na forma de *plugins*, ou em ferramentas como o *Visual Studio Code* na forma de extensões, poucas ou nenhuma ferramentas existentes unem estes dois conceitos com o conceito emergente de *liveness*, cujo objetivo é abrir novos desafios para o desenvolvimento de *software* ágil, em termos de agilidade técnica.

De modo a fornecer ao programadores um ambiente que melhor se encaixa na filosofia de desenvolvimento *software live*, propomos o desenvolvimento de uma ferramenta de recomendação de *refactoring* com alvo em sistemas em *TypeScript* e *JavaScript*, na forma de uma extensão de *Visual Studio Code*, que fornece ao utilizador visualização ao vivo de uma ampla variedade de métricas de qualidade e a possibilidade de executar *refactorings* ótimos, baseados na extração, como o Extrair Método, Extrair Classe e Extrair Variável (cujas condições são customizáveis pelo utilizador) ao código, permitindo observar ao vivo o impacto destas mudanças nos valores das métricas de qualidade.

Para validar a ferramenta desenvolvida, foi feito um inquérito acerca da sua usabilidade e funcionalidades, de modo a percebermos de que forma esta ferramenta impacta o trabalho dos programadores, através da informação dada pela sua interface. Testamos a correlação entre a agilidade de avaliação dos programadores e a sua consciência para tomar ações. Outra parte da validação passou por correr a ferramenta em vários documentos pertencentes a vários repositórios públicos em diferentes versões, para percebermos se o uso da ferramenta criada leva a um código mais sustentável. Testes de hipóteses feitos à correlação entre percepção dos programadores e a sua ação e se os nossos *refactorings* levam a melhorias no código resultaram em algumas das hipóteses nulas não serem rejeitadas, levando assim a que a hipótese principal não fosse inteiramente validada.

Acreditamos que fornecendo aos programadores, desde iniciantes a especialistas, com um ambiente *live* com estas características permitir-lhes-á identificar problemas no sistema mais rápido e manter-se conscientes do impacto que alterações em código realmente têm nas métricas de qualidade. Ao oferecer a possibilidade de executar *refactorings* baseados nestas métricas, permite ao utilizador manter-se entre limites destas, levando a um desenvolvimento de *software* mais sustentável.

Acknowledgements

First of all, I would like to thank my supervisors, Ademar Aguiar and Sara Fernandes, for all the guidance provided throughout this last year. This would not have been possible if not for your expertise, brilliant ideas and out-of-the-box thinking, which played an essential role for the outcome of this dissertation.

A very special thanks to my research colleague João Barbosa for his support, the productive discussions and, despite the circumstances, his availability to help me overcome whatever obstacle I had encountered that day.

To Matilde, my love, there are no sufficient words to thank you for the unwavering support, caring and love you have given me for the last three years. If not for you, nothing of this would have been possible.

My thanks to my friends on *REDEFINE*, which have provided countless hours of entertainment, *goofs and gaffs* and were always there to hear about my endless rants and frustrations, allowing me to stay sane throughout these years. Miguel Freire, Nuno Silva, Vítor Domingos, Daniel Torre, Cristiano Monteiro, Francisco Mesquita, João Figueiredo, António Gomes and countless others, this one is also for you.

To my family, friends and pets, thank you for all your support and for making me the person I am today. Whilst not family, although as I consider part of it, a special thanks to José Emídio Figueiredo for all the help provided and always pointing me towards the right direction, even through dire times.

To my very special friend Miguel. Seeing you fighting for your life and almost being forced to give up on your dreams has inspired me to be a better version of myself and give just a little bit more every day, thank you for checking up on me throughout these years and I am glad to see you back.

Lastly, to Byron. The world was robbed of one of the good guys too soon. May your kind heart rest in peace knowing that you inspired thousands of people to be kinder and more understanding of each other, myself included. I hope *Everland* turns out to be the legacy you rightfully so deserve.

Sérgio Salgado

“Time’s up, let’s do this.”

Ben Schulz

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation	2
1.3	Problem	2
1.4	Objectives	2
1.5	Dissertation Structure	2
2	Background	5
2.1	Software Engineering	5
2.2	Software Refactoring	6
2.3	Software Quality	6
2.3.1	Code Smells	7
2.3.2	Quality Metrics	7
2.4	Live Programming	8
2.4.1	Levels	9
2.4.2	Criticism	9
3	State of the Art	11
3.1	Live Software Development	11
3.1.1	Live Development Environments	12
3.1.2	Programming Languages	14
3.2	Refactoring Recommendation Systems	15
3.2.1	Refactoring Identification	15
3.2.2	Sequencing Refactorings	17
3.2.3	Tools	21
3.2.4	Live Tools	24
3.3	Quality Metrics Tools	25
3.4	Results and Discussion	28
4	Problem Statement	33
4.1	Open Issues	33
4.2	Research Questions	34
4.3	Proposal	35
4.4	Validation	35
5	Proposed Solution	37
5.1	Context	37
5.2	Usage	39

5.3	Automated Refactoring	39
5.3.1	Extract Method	40
5.3.2	Extract Class	42
5.3.3	Extract Variable	44
5.4	Live Metrics	45
5.4.1	Extract Class Metrics	45
5.4.2	Extract Method Metrics	46
5.4.3	Interface Metrics	47
5.5	Visual Studio Code Extension	50
5.5.1	Commands	50
5.5.2	Settings	51
5.5.3	Events	51
5.6	Summary	52
6	Empirical Validation	53
6.1	Methodology	53
6.1.1	Survey	53
6.1.2	Automated Analysis	54
6.2	Results	54
6.2.1	Survey	55
6.2.2	Automated Analysis	62
6.2.3	Discussion	71
6.3	Threats to Validity	73
6.3.1	Conclusion Validity	74
6.3.2	Internal Validity	74
6.3.3	Construct Validity	75
6.3.4	External Validity	75
7	Conclusions and Future Work	77
7.1	Conclusions	77
7.2	Main Contributions	78
7.3	Future Work	79
	References	81
A	Survey	87

List of Figures

3.1	Circa's introspection capabilities due to its dataflow-based model.	13
3.2	SuperGlue's live characteristics when creating PacMan.	15
3.3	MaxFlow-MinCut algorithm showing results for an Extract Class.	17
3.4	Graph showing different code states when applied a different sequence of refactoring operations.	18
3.5	Vector-based representation of a refactoring sequence.	20
3.6	JDeodorant identification of Feature Envy code smells.	22
3.7	Extract Method refactoring on DNDRefactoring tool.	23
3.8	Evolution over time of selected metrics.	26
3.9	Teamscale's architecture.	27
5.1	LiveRefactoring's user interface within Visual Studio Code.	39
5.2	LiveRefactoring's user interface.	49
5.3	Impact on the interface due to code changes.	49
5.4	Supported commands on the Visual Studio Code's <i>Command Palette</i>	50
6.1	Survey participant's age and education level.	55
6.2	Survey answers to questions TB2, TB3 and TB4.	56
6.3	Survey answers to questions TB1, TB5, TB6 and TB7.	57
6.4	Survey answers to questions regarding the tool's interface.	58
6.5	Survey answers to questions regarding the tool's post-changes interface.	59
6.6	Survey answers to questions regarding the tool's implementation on the Visual Studio Code's UI.	59
6.7	Survey answers to questions regarding the tool's workflow.	60
6.8	Survey answers to questions regarding the tool's refactoring capabilities.	61
6.9	Survey answers to questions regarding the tool's user settings.	62
6.10	Survey answers to the final section.	63
6.11	LiveRefactoring impact on number of statements and cyclomatic complexity metrics.	64
6.12	LiveRefactoring impact on the maintainability index.	65
6.13	LiveRefactoring impact on targeted metrics across five repository versions.	67
6.14	LiveRefactoring impact on the maintainability index across five repository versions of <i>server.ts</i>	68
6.15	LiveRefactoring impact on the selected metrics across ten Extract Method refactorings.	69
6.16	LiveRefactoring's Extract Class impact on the LCOM metric across four files.	70

List of Tables

2.1	Fowler's refactoring catalog.	7
3.1	Code smells identified and refactoring operations supported by an A* approach. .	19
3.2	Live environments comparison.	28
3.3	Refactoring proposals comparison.	29
3.4	Refactoring tools comparison.	30
3.5	Quality metric tools comparison.	30
5.1	Metrics and respective thresholds.	48
6.1	Hypothesis Tests related to the Extract Method refactoring.	66
6.2	T-test on the Extract Method refactoring impact on maintainability.	66
6.3	Hypothesis Tests related to the Extract Class refactoring.	70
6.4	T-test on the Extract Class refactoring impact on the LCOM metric.	71

Abbreviations

IDE	Integrated Development Environment
AST	Abstract Syntax Tree
UI	User Interface
NSGA	Non-dominated Sorting Genetic Algorithm
QMOOD	Quality-Model for Object Oriented Design
JDT	Java Development Tools
JVM	Java Virtual Machine
RMI	Remote Method Invocation
LCOM	Lack of Cohesion of Methods
VSCode	Visual Studio Code

Chapter 1

Introduction

1.1	Context	1
1.2	Motivation	2
1.3	Problem	2
1.4	Objectives	2
1.5	Dissertation Structure	2

In this chapter, the context, motivation, definition of the problem and the main objectives for this study are presented. After a brief overview of the main topics covered in this study in Section 1.1, the motivation leading this dissertation in Section 1.2 follows. The problem addressed and the objectives of this work are next described in Sections 1.3 and 1.4, respectively. Finally, Section 1.5 describes how this dissertation is structured.

1.1 Context

In the context of software, maintenance represents one of the main steps of the Software Development Life-Cycle and concerns for software systems to stay maintainable across different developers which consequentially ensure the system's longevity. However, in systems where this process is not enforced, source code rapidly starts becoming complex and hard to maintain [Fow99], leading to high maintenance costs and inefficiency of developers' work time on the system.

Refactoring is one of the main tools to ensure that a system stays maintainable during its lifetime, by turning, otherwise difficult to read by a developer, source code into a more readable and comprehensible system to the human eye, without interfering with the system's intended behavior [Fow99].

1.2 Motivation

Although software maintenance may pass as an unnecessary waste of time by some people, it is a crucial step during development to ensure the longevity and expansion of a system, by making it more comprehensible to both current and future developers of the system.

With the concept of live programming aiming to tighten the more traditional edit-compile-run feedback loop [Tan13], there is potential for its philosophy to be applied to other areas of software development, more specifically, software maintenance and refactoring [ARC⁺19].

By combining these two concepts, developers may access quality metrics computed in real-time, and apply refactoring techniques tailored to tackle possible code smells commonly characterized by the presence of certain quality attributes.

1.3 Problem

Refactoring is common practice in software development environments, as the need to adjust to new product requirements is of utmost importance. To streamline refactoring, multiple tools exist to aid the developer to create meaningful and impactful refactorings onto a system.

However, current tools operate under the assumption that the developer knows a refactoring is due even before it starts, which is often not the case as developers realize a refactoring is being manually performed while they are in the middle of it and, as a consequence, refactoring tools end up being underused [MHPB12].

This dissertation explores the opportunity of adding liveness to the refactoring process, as we believe its philosophy of continuous execution is a vital aid for developers during the practice of refactoring, via refactoring recommendation systems.

1.4 Objectives

The main objective of this dissertation is to improve the liveness of refactoring recommendation systems and increase user feedback during development via a refactoring recommendation system operating on the analysis of quality metrics and which uses the concept of liveness to continuously process and recommend semi-automatically possible refactoring operations to the user.

An empirical validation of this approach will be performed, to measure whether the existence of a tool with such capabilities ends up proving itself of use for developers on the practice of refactoring.

1.5 Dissertation Structure

For those who do not have a background in Software Engineering, it is recommended to start reading from Chapter 2, where relevant concepts related to the contents of this dissertation are

explained in detail, namely the concepts of software engineering, software refactoring, software quality, live programming and minor concepts related to each of these themes.

May the reader be interested in the bibliographic findings acquired through a literature review related to the topics of this dissertation, Chapter 3 presents the results, divided into the topics of live software development, refactoring recommendation systems and quality metrics tools. These results range from software tools and systems to theoretical implementations and proposals related to liveness in software development.

Chapter 4 details the problem at hand and our respective approach, based on what was found during our literature review, introducing the hypothesis and research questions of this dissertation and the respective found solution and validation methodologies. Chapter 5 explains in great detail how the proposed solution was designed and implemented, with Chapter 6 being dedicated to further detail on the validation methodologies chosen to answer the proposed research questions, our answers to them based on the results obtained and possible threats to validation encountered.

Finally, Chapter 7 summarizes the results obtained throughout this study, reflecting on the lessons learnt, main contributions for the area of software engineering and possible future work deriving from this dissertation.

Chapter 2

Background

2.1	Software Engineering	5
2.2	Software Refactoring	6
2.3	Software Quality	6
2.4	Live Programming	8

In this chapter, background about concepts referred throughout this dissertation is explained, allowing for readers who do not have a background in software engineering to get familiar with these terms and their meaning.

Section 2.1 gives insight about some of the practices performed during software creation and maintenance. Section 2.2 addresses how the practice of refactoring activities works and its importance during software development, followed by Section 2.3 where software quality is discussed and its importance when creating software. Section 2.4 describes an eminent field in software development that has potential to further evolve common software engineering practices and technical agility.

2.1 Software Engineering

Software engineering is a sub-field of engineering related with the creation, evolution, and maintenance of software systems, formally defined by the ISO/IEC/IEEE 24765:2017 Standard as:

“Systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing and documentation of software.” [III17]

The field of software engineering is composed by a wide array of domains and processes, each of utmost relevance and importance during the life cycle of software development, ranging from the analysis of requirements to the software design and its maintenance [BFS14].

The latter of the enumerated processes, Software Maintenance, is composed of a set of practices and operations supporting the evolution of the software system when in the presence of events such as requirement changes or feature introduction. It is a core process during the software development life cycle and is performed with multiple objectives in mind, including fault correction, design improvement, and enhancement implementation [BFS14].

2.2 Software Refactoring

Refactoring is a practice tied with the maintenance of software. It is defined by changes in code which improve its internal structure, while not altering its external behavior [Fow99]. Its main purpose is to oppose the loss of code structure, naturally occurring during software evolution, usually due to unfamiliarity with the system. Developers unfamiliar with the system are often willing to commit to short-term goals, such as the introduction of new features in the system, trading off code structure in the process [Fow99].

The result of source code post-refactoring is easier maintainability and understanding compared to the pre-refactoring code [Fow99]. This maintainability easiness snowballs towards increased development agility, making refactoring a practice of utmost importance during software development. Table 2.1 shows one of the first refactoring catalogs in the literature [Fow99].

One of the fields of study related to refactoring optimization is search-based refactoring. Search-based refactoring is a sub-field of search-based software engineering (SBSE), which focuses on the use of meta-heuristic algorithms, such as genetic and hill-climbing algorithms, to optimize refactoring. Refactoring optimization includes the analysis of identification, execution, and sequencing of refactoring operations in source code.

2.3 Software Quality

Software quality refers to the ability to satisfy the needs, either stated or implied, of software when used under certain conditions [III17]. Eight characteristics describe the quality properties which categorize a quality model: functional stability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability [III1].

The quality of a software artifact can be both externally and internally evaluated, based on whether the software product enables the behavior of a system to satisfy its needs and whether the set of static attributes of said product satisfies the referred needs, respectively [III1].

Table 2.1: Fowler’s refactoring catalog [Fow99].

Refactorings Catalog		
Add Parameter	Inline Method	Replace Conditional with Polymorphism
Change Bidirectional Association to Unidirectional	Inline Temp	Replace Constructor with Factory Method
Change Reference to Value	Introduce Assertion	Replace Data Value with Object
Change Unidirectional Association to Bidirectional	Introduce Explaining Variable	Replace Delegation with Inheritance
Change Value to Reference	Introduce Foreign Method	Replace Error Code with Exception
Collapse Hierarchy	Introduce Local Extension	Replace Exception with Test
Consolidate Conditional Expression	Introduce Null Object	Replace Inheritance with Delegation
Consolidate Duplicate Conditional Fragments	Introduce Parameter Object	Replace Magic Number with Symbolic Constant
Convert Procedural Design to Objects	Move Field	Replace Nested Conditional with Guard Clauses
Decompose Conditional	Move Method	Replace Parameter with Explicit Methods
Duplicate Observed Data	Parameterize Method	Replace Parameter with Method
Encapsulate Collection	Preserve Whole Object	Replace Record with Data Class
Encapsulate Downcast	Pull Down Field	Replace Subclass with Fields
Encapsulate Field	Pull Up Constructor Body	Replace Temp with Query
Extract Class	Pull Up Field	Replace Type Code with Class
Extract Hierarchy	Pull Up Method	Replace Type Code with State/Strategy
Extract Interface	Push Down Method	Replace Type Code with Subclasses
Extract Method	Remove Assignments to Parameters	Self Encapsulate Field
Extract Subclass	Remove Control Flag	Separate Domain from Presentation
Extract Superclass	Remove Middle Man	Separate Query from Modifier
Form Template Method	Remove Parameter	Split Temporary Variable
Hide Delegate	Remove Setting Method	Substitute Algorithm
Hide Method	Rename Method	Tease Apart Inheritance
Inline Class	Replace Array with Object	

2.3.1 Code Smells

A bad smell in code, more commonly known as *code smell*, is defined by a certain structure within code which does not comply with the system’s design principles [KA16] and is usually a symptom of poor design choices [Pan19]. Unlike bugs, code smells do not introduce unwanted behavior or failures in the system. However, they often indicate the presence of eventual problems in the source code which can eventually lead to the introduction of bugs and loss of code structure [KA16].

The presence of bad smells in source code more often than not indicate the need for refactoring [Fow99]. Some code smell catalogs and consequent solving have been purposed throughout the years. For example, Fowler, M. [Fow99] identified twenty-two code smells and the appropriate refactoring operations to fix each one of these bad smells [Fow99] and later, Lanza, M. et al. [LMD06] identified eleven code smells while suggesting options for removal of said smells [LMD06].

2.3.2 Quality Metrics

A quality metric is a form of measure with the purpose of evaluating whether an artifact of source code possesses certain properties. They can be important in order for developers and project managers to monitor and detect the presence of underlying design problems within the project.

Throughout the years, multiple software quality metric suites and catalogs were proposed with object-oriented software design in mind, including:

- **Halstead, Maurice H.** [Hal77] used the number of total and unique operands and operators to calculate metrics such as program *length*, *difficulty* to understand and write code, *effort* and respective *time* to program and the *amount of expected delivered bugs* during implementation [Hal77, Cou19];
- **Chidamber, Shyam R. et al.** [CK91, CK94] proposed the usage of metrics such as *Weighted Methods per Class (WMC)*, *Coupling Between Objects (CBO)*, *Response for a Class (RFC)* and *Lack of Cohesion in Methods (LCOM)* to evaluate object-oriented design [CK91, CK94];
- **Lanza, M. et al.** [LMD06] listed twenty-four metrics including *Locality of Attribute Accesses (LAA)*, *Coupling Intensity (CINT)* and *Access to Foreign Data (ATFD)*, and their respective connection of detecting specific code smells present in the project [LMD06].

Quality metrics are used to describe design properties in a project which in turn can be used, by weighting their values, to compute quality attribute values [BD02] that can be used as fitness functions for refactoring recommenders to operate and recommend near-ideal refactoring.

2.4 Live Programming

Live Programming refers to the ability of programming environments to modify a running system during execution, allowing developers to receive feedback about the impact of these changes instantly [Tan13, ARC⁺19]. First introduced in live development environments such as Smalltalk [ARC⁺19] and visual languages [Tan13, Tan90], it is a concept which aims to break the more traditional *edit, compile, link, run* development cycle, by turning it into a single phase cycle where the system is continuously running, despite edits occurring in the system [Tan13].

Working in a live environment is motivated by the following aspects [Tan13]:

- Minimizing the latency between a programming action and seeing its effect on program execution;
- Allowing performances in which programmer actions control the dynamics of the audience experience in real time;
- Simplifying the 'credit assignment problem' faced by a programmer when some programming actions induce a new run-time behavior (such as a bug);
- Supporting learning, hence the early connections between liveness with visual programming and program visualization.

2.4.1 Levels

Having defined the first four liveness levels back in 1990 [Tan90], Tanimoto added two more levels since [Tan13], making it a total of six possible levels of liveness.

The first level of liveness, the informative level, grants a visual representation of the program, which serves as an aid for the programmer to understand the problem at hand, but is not used in any form by the computer. It is followed by the informative and significant level, where the visual representation is the specification of what the computer needs to execute, essentially being an executable flowchart. The third level of liveness adds responsiveness to the system, (re-)executing the program everytime a change is introduced, where the last level adds liveness as stream-driven updates, continuously updating the display to show results of the changes done to the system [Tan90].

The two recently added levels add tactically and strategically predictive layers, respectively, with tactically predictive meaning the system stays a step ahead of the developer, by predicting the next action to be performed. The last of defined liveness levels, the strategically predictive level, aims to compute a prediction of the desired behavior and, with the aid of a large knowledge base, adds gross functionality to the system [Tan13].

2.4.2 Criticism

Despite its obvious benefits to developers and programming environments, live programming as a concept has gone under a lot of scrutiny in the past. For example, achieving liveness level 4 may be considered secondary in the vast majority of software systems, which Tanimoto refutes, by considering liveness a straightforward evolution over the existent interactive debuggers present in modern IDEs [Tan13].

Application of liveness to systems which either run for small intervals of time, in the order of the milliseconds, or have a relatively large execution time, is also criticized. Editing a section of the code which was already executed and to which the system will not return to will not benefit from any aspect of liveness. Tanimoto suggests the existence of an *auto-repeat* mode, where either the entire system (for the ones running for small intervals) or a section of the code is executed repeatedly until the developer states otherwise [Tan13].

Chapter 3

State of the Art

3.1	Live Software Development	11
3.2	Refactoring Recommendation Systems	15
3.3	Quality Metrics Tools	25
3.4	Results and Discussion	28

In this chapter, the results of a literature review performed on the topics of Live Software Development, Refactoring Identification and Recommendation, and Quality Metric Tools are presented and respective conclusions are discussed.

Section 3.1 describes existent implementations of liveness in development environments and programming languages. Next, in Section 3.2, exposure to current refactoring identification approaches and techniques is given, followed by approaches about sequencing of refactorings and tools which employ said approaches in the formats of environment plugins or extensions and ones who use the concept of liveness. Section 3.3 describes tools which grant insight and, in some cases, exposure of quality metrics in a user-friendly format. Section 3.4 summarizes and discusses the results obtained on each of the topics studied.

3.1 Live Software Development

Although a somewhat novel concept, some tools were already conceived with the application of liveness in mind. Section 3.1.1 describes in detail results found during research related with programming environments using liveness and Section 3.1.2 describes results related with new or adapted programming languages which apply liveness philosophies.

3.1.1 Live Development Environments

Live development environments are designed with the liveness concept in mind, this is, the employment of live programming [Tan13] aspects in development environments. This review aims to pinpoint core aspects found in current live development environments and how differently they interact with developers when compared with traditional environments.

Lemma, Remo et al. [LL13] presented a development environment which is being designed in parallel to the live programming language, created by the same researchers, Moon [LL13].

Programming languages with live characteristics have their IDE designed as an afterthought or are designed to fit into existent IDEs which were created without liveness as one of its objectives, leading to technical constraints that can limit the live experience [LL13].

To combat these shortcomings, the created live programming language, Moon, is having its IDE designed in parallel, with two prototypes developed: the first one using Pharo¹ (a Smalltalk implementation) and the second approach being web-based and written in JavaScript [LL13].

The main features of either implementation of the IDE are an entity visualization, state visualization - where live feedback is received after every change, with the compiler being called after every carriage return - and evolution visualization, allowing for developers to consult an updated visualization of the system's evolution [LL13].

Kanon [OMIA17] is a JavaScript-based live programming environment for data structures. It provides the developers with two connected panels in parallel, one for editing and the other for visualization, allowing for a quick connection between the visual elements and the respective code [OMIA17].

The visualization of the data structures is done using basic shapes and forms such as ovals and arrows [OMIA17]. Modifications to the program trigger its re-execution and are shown to the user by either animating the program's visual representation or by showing a summary about the effects of these changes [OMIA17].

Fisher, Andrew [Fis13] presented Circa, an environment which is tightly connected with a language of the same name [Fis13].

Its dataflow-based model is capable of powerful introspection as, for example, the developer can click anywhere on a rendered scene and the language is able to determine which draw method is related to the clicked position, which is possible due to code execution capable of skipping *side-effecting* functions and the flow-based model allows for display of code sections directly related to the input values of the referred method (Figure 3.1) [Fis13].

This introspection makes clever code manipulation possible, as the user might impose restrictions, referred to as *desires*, to a result of a computation and a solver using a backpropagation algorithm tries to produce a modification to the code which satisfies the restrictions defined [Fis13].

¹Pharo Project - <http://www.pharo.org/>, Last accessed on July 22, 2020

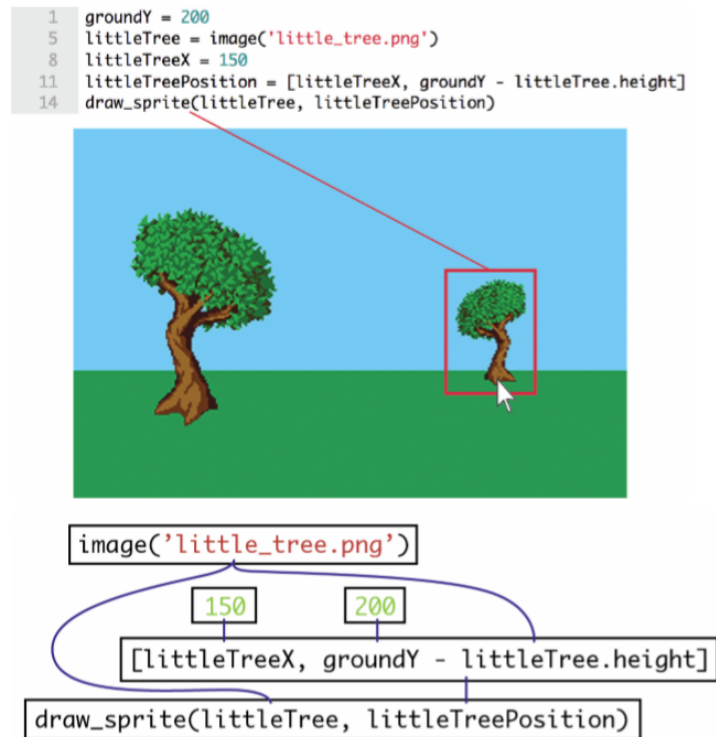


Figure 3.1: Circa's introspection capabilities due to its dataflow-based model [Fis13].

Euclase [OMB13] is a live programming environment which is tied to a visual programming language aimed at interaction designers, who benefit from tools that allow for quick iteration between interface behavior [OMB13].

Liveness in Euclase allows for a greater beginner friendliness, fast evaluation and quick experimentation of the system, characteristics which designers cannot find in current tools aimed at designing how an interface feels [OMB13].

Euclase's primitives are powerful, low in number and simple to comprehend. Their usage as one-way constraints that are updated throughout the system's execution is similar to how a designer thinks about relationships as constraint-based concepts, leading to simplification of interface design [OMB13].

Current live development environments present a novel way to develop software and are characterized by their rich feedback systems and continuous execution, inherited from the live programming philosophy [Tan13].

Tools such as **Moon** [LL13], **Circa** [Fis13] and **Kanon** [OMIA17] provide developers with powerful visual feedback of changes occurring in the system during development. This enhanced feedback allows developers to explore and experiment with the system in a near-instant way, allowing for a quicker iteration towards a more optimal system.

Another relevant aspect found in live environments is the user-friendliness. **Euclase** [OMB13] uses liveness and applies towards interface interaction, a field often unrelated with traditional

programming. Despite not being connected with regular programming, **Euclase** [OMB13] showcases liveness used in a user-friendly format with no programming background required.

3.1.2 Programming Languages

Similar to live development environments, it is important to study designed live programming languages, their impact on software development and the main differences between the more *traditional* programming languages and the ones operating live.

Fisher, Andrew [Fis13] introduced Circa, a dataflow-based language of the same name as its environment, which allows for highly introspectable and understandable code [Fis13].

The dataflow-based model allows for rich visualization of the program and Circa is also capable of tracing input origin and freely re-evaluate code [Fis13].

Code sections related to user interactions on the rendered scene are referred to as intermediate states, and Circa allows for them to be manipulated as first class values and observing the impact of the changes to that state in real-time [Fis13].

Burckhardt, Sebastian et al. [BFdH⁺13] redesigned TouchDevelop², a programming language aimed at UI design and programming, tasks where the traditional development cycle gets tedious and exhausting, as they require multiple iterations [BFdH⁺13].

Changes to the existent language include the addition of live editing, characterized by a continuous recompilation and re-execution of the system during editing which is achievable due to a clear separation between views and models [BFdH⁺13].

Another added feature is bi-directional interaction between code and UI and its direct manipulation, allowing for the user to select code by selecting a UI element and vice-versa, while also allowing for the user to directly edit variable values. This is possible as a mapping between code and the respective UI element is created by the language [BFdH⁺13].

SuperGlue [McD07] is a live programming language based on a reactive data-flow programming model that allows for program building using existent components which are connected through *signals* and form a data-flow graph [McD07].

When a change occurs and leads to valid code, the edit is incorporated into the program in execution (Figure 3.2) and, in order to stay responsive, SuperGlue supports error recovery in the background of the execution [McD07].

The used data-flow based model is supported by the existence of five different constructs: *signals* - data-flow values which ease inter-component communication; *classes* - encapsulate behavior; *connections* - connect two signals so that values stay equal; *extensions* - cause signals to extend classes; *conditions* - gate when connection and extension rules are applied [McD07].

As expected, we found that live programming languages thrive on the fast iteration, powerful visualization and live manipulation they are characterized by.

²TouchDevelop - <https://www.microsoft.com/en-us/research/project/touchdevelop/>, Last accessed on July 22, 2020

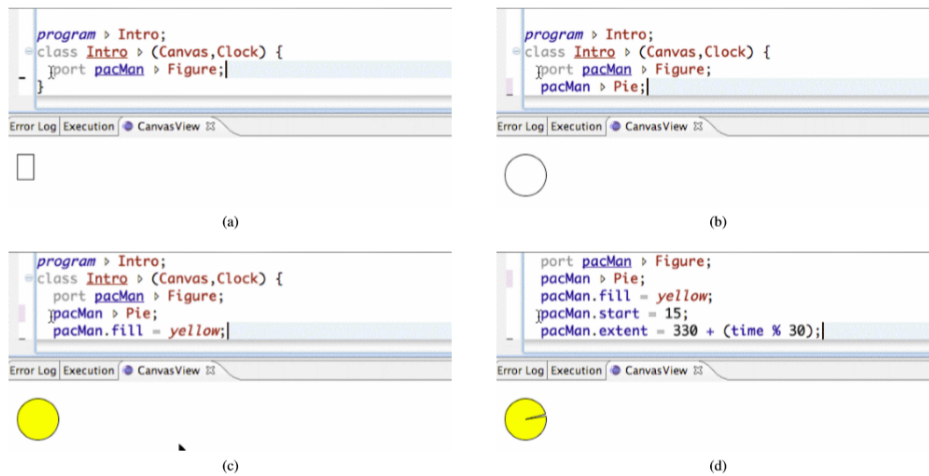


Figure 3.2: SuperGlue's live characteristics when creating PacMan [McD07].

This fast iteration is easily visible on the redesigned **TouchDevelop** [BFdH⁺13], showing the benefits of applying liveness in tasks which would otherwise be considered tedious and exhausting.

All studied live programming languages [Fis13, BFdH⁺13, McD07] allow for a live visualization and value manipulation of the system being developed, with **Circa** [Fis13] and **TouchDevelop** [BFdH⁺13] allowing for a bi-direction interaction between the rendered visualization and the respective code.

3.2 Refactoring Recommendation Systems

Refactoring is common practice during software development and, as such, a vast array of approaches and tools have been crafted with the sole purpose of achieving optimal and effortless refactoring for developers and teams.

Proposals to identify refactoring opportunities are described in Section 3.2.1, while proposals to sequence refactoring operations are presented in Section 3.2.2. Later, in Sections 3.2.3 and 3.2.4, refactoring tools, either implementing liveness characteristics or not, are described.

3.2.1 Refactoring Identification

Identifying refactoring opportunities is the core of any refactoring recommendation system. As such, it is essential to study how refactorings are identified and the main algorithms used to do so.

Bavota, Gabriele et al. [BOD⁺10] proposed a perspective to identify *Extract Class* refactoring opportunities based on the usage of game theory, with the objective of balancing contrasting goals [BOD⁺10].

Using the post-extraction classes being players and the selection methods of the original class being the plays to be made as a metaphor for the game, each new class starts with one of the two

least cohesive methods and play to select methods which are the most related to the ones they already possess, until all methods have their new class assigned [BOD⁺10].

The Nash equilibrium [Nas51] in this game, while not providing solutions as optimal as its Pareto front [Deb01] counterparts, provides a fair compromise between coupling and cohesion, the two contrasting goals in play, achieving a better identification of *Extract Class* opportunities than the Pareto optimum. [BOD⁺10].

Pantiuchina, Jevgenija et al. [PBTP18, Pan19] proposed a tool capable of identifying refactoring opportunities by predicting the need for refactoring with the usage of a code smell predictor - COSP [PBTP18, Pan19].

COSP operates with the objective of preventing the introduction of code smells, more specifically the *God* or *Complex Class* code smells, by identifying problematic classes and sections of code in the need of refactoring operations [PBTP18, Pan19]. It uses a machine learning approach, using quality metrics as predictor variables and taking as an input current, historic and recent code quality trends to classify each class [PBTP18].

To avoid flooding developers with false positives when detecting *smelly* classes, COSP uses a confidence value, alerting developers once a defined threshold is crossed [PBTP18]. Future iterations of COSP aim to employ deep learning to learn about changes made by developers and perform refactoring operations actually performed by developers, using information mined from a large set of refactoring operations [Pan19].

Bavota, Gabriele et al. [BDO11] suggested the identification of *Extract Class* opportunities with measures related to structural and semantic cohesion in mind [BDO11].

The approach takes a class previously selected by a developer and, after extracting information about characteristics such as method calls and attribute references, a weighted graph representation of the class is created, where each node is a method of the class and the weight of each edge represents a value related with how the relationship between the two methods impacts class cohesion [BDO11]. This value is computed using metrics such as the Structural Similarity between Methods and Call-based Dependence between Methods [BDO11].

Once the weighted graph is created, a MaxFlow-MinCut algorithm, shown on Figure 3.3 is used with the objective of dividing the original graph in two sub-graphs, representing the two new classes with improved cohesion between methods [BDO11]. This uses a semi-automatic approach, requiring developer approval or, in case the developer does not agree, changes to what classes are moved can be performed [BDO11].

Tsantalis, Nikolaos et al. [TC09] suggested an approach to identify *Move Method* opportunities using the concept of distance between entities as the core of the approach [TC09].

This concept of distance between entities, which are essentially attributes and metrics, takes into account the accesses between each entity to create sets that are compared in order to identify the degree of similarity between entity sets [TC09]. A list of opportunities for *Move Method* is

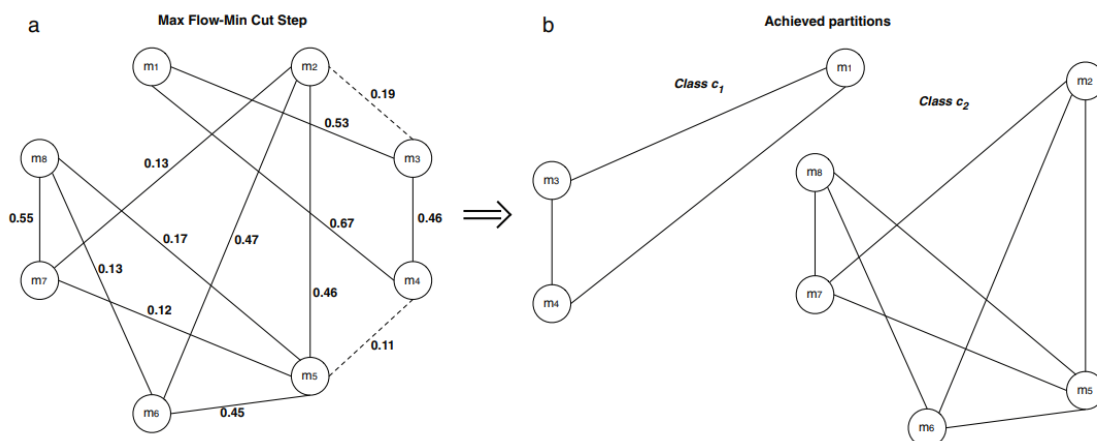


Figure 3.3: MaxFlow-MinCut algorithm showing results for an Extract Class [BDO11].

sorted based on the number of entities a certain method accesses from each target class and the value of the distance between the same method and each target class [TC09].

If a target method preserves all the compilation, behavior-preservation and quality preconditions, it is suggested to the developer to perform the *Move Method* refactoring operation semi-automatically, as this approach is not aware of conceptual and design criteria, requiring developer input for situations as refactoring classes in a *Model-View-Controller* design pattern [TC09].

During this study, we found that the identification of refactoring opportunities can be performed in multiple ways. Despite identifying different kinds of refactorings, all studied approaches rely on quality metrics to identify sections in need of refactoring.

We found that different approaches can be used to identify the same refactoring, as shown by **Bavota, Gabriele et al.** [BOD⁺10, BDO11], who identifies *Extract Class* opportunities using both game theory [BOD⁺10] and a MaxFlow-MinCut algorithm [BDO11].

On approaches taking a more practical format [PBTP18, Pan19, TC09], it is worth noting their interaction with the user. **Pantiuchina, Jevgenija et al.** [PBTP18, Pan19] only inform the user about potential refactorings if the metrics value exceeds a certain threshold [PBTP18]. **Tsantalis, Nikolaos et al.** [TC09] suggests the semi-automated execution once the compilation, behavior-preservation and quality preconditions and checked [TC09].

3.2.2 Sequencing Refactorings

Execution of a single refactoring operation is sometimes not enough and eliminating a certain smell from a system often takes the application of a variety of refactoring operations.

It is important to study how current approaches sequence these refactoring operations, given their impact on the quality metric values and the importance of their ordering.

Meananeatra, Panita [Mea12] proposed the identification of an optimal sequence of refactorings based on the four main criteria developers take into consideration when refactoring

code: number of removed bad smells, system maintainability, size of the refactoring sequence and number of modified system elements [Mea12].

The main focus of this approach is to identify the optimal sequence to remove the *Long Method* code smell using a pool of six refactoring operations [Mea12]. Using the bad code and the developer's objective as an input to the algorithm, it applies refactoring operations iteratively, creating new code states which branch out in a graph, with each path created relating to a sequence of refactoring operations as shown on Figure 3.4 [Mea12].

This work expands on previous work by **Meananeatra, Panita et al.** [MRA11], which computes maintainability metrics to sequentially find new code states, selecting the end state which provides the best maintainability values found [MRA11].

Once the metric value relative to the objective input by the developer is met, the unfolding process stops, outputting the optimal refactoring sequence, which maximizes both the number of removed smells and the maintainability value, while minimizing the number of modified program elements and the size of the sequence, with this last characteristic being further optimized by considering commutative and equivalent paths [Mea12].

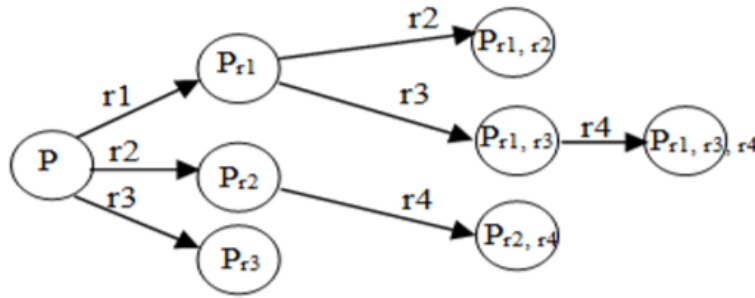


Figure 3.4: Graph showing different code states when applied a different sequence of refactoring operations [Mea12].

Tarwani, Sandhya et al. [TC16b] proposed the usage of a greedy algorithm to identify possible refactoring sequences relying on local optimal solutions to reach a global solution, based on the fact that different refactorings result in different maintainability values and, as such, reaching an optimal solution allows for a higher maintainability value [TC16b].

It operates in two different ways: One-way analysis, resulting in the identification of a single refactoring operation, evaluated based on the most positive impact to metric values; Multi-way analysis, which results in a sequence of refactorings which aim to maximize maintainability values [TC16b].

The multi-way analysis takes combinations of refactoring operations on the same code, creating a path originating from the original code to the final sequence [TC16b]. On each level of the resultant tree, it tries to minimize the metric values (the local optimum) which eventually suggests a possible sequence which maximizes the maintainability values of the system [TC16b].

Chug, Anuradha et al. [CT17] identified optimal refactoring sequences by applying A* algorithm to search for the sequence of refactoring operations to use, after prioritizing system classes using the amount of detected code smells as criteria [CT17].

This approach prioritizes classes in need of refactoring actions by evaluation of their respective Quality Decline Factor (QDF) metric [TC16a], as a higher value of this metric usually corresponds to the class with the highest amount of detected code smells [CT17, TC16a].

Once the classes are sorted by their QDF metric, the most critical class is selected to determine a possible sequence of refactoring operations to apply in order to bring the class to a more maintainable state [CT17].

A* algorithm is then applied, taking both the tree level the current solution being analysed is present and the value of a set of nine metrics is calculated in order to maximize the maintainability attribute [CT17]. Code smells and respective refactoring operations sequenced by this approach are described in Table 3.1.

Table 3.1: Code smells identified and refactoring operations supported by [CT17].

Code Smell	Refactoring Operation
God Class	Extract Class
Long Method	Extract Method
Type Checking	Replace Type Code with State/Strategy
Long Parameter List	Extract Parameter
Dead Code	Inline Class/Remove Parameter
Data Class	Encapsulated Field
Class Hierarchy Problem	Push Down
Dummy Handler	Re-throw with Exception
Nested Try Statement	Extract Method
Careless Cleanup	Throw Exception in Finally Block

Mkaouer, Mohamed Wiem et al. [MKB⁺16] proposed the usage of multiple quality attributes to achieve optimal software refactoring using a multi-objective search-based approach, as this is a practice which usually takes into consideration the optimization of multiple conflicting objectives [MKB⁺16].

To achieve this, it uses the NSGA-III algorithm [DJ14] and takes into account eight objectives to optimize: six QMOOD metrics - reusability, flexibility, understandability, functionality, extendibility and effectiveness - a minimization of the number of refactoring operations performed and a maximization of design coherence [MKB⁺16].

Design coherence is treated as an objective to maximize, as sometimes, despite the refactored design structure being improved, refactoring can lead to design incoherence if not correctly evaluated by the developer [MKB⁺16]. To preserve the semantics design, two measures were

formulated and used towards the objective of preserve design coherence: Vocabulary-based similarity and Dependency-based similarity [MKB⁺16]. Vocabulary-based similarity is based on the assumption that vocabulary used to name code elements in classes reflects a specific domain terminology and dependency-based similarity uses the presumption that a high coupling value between two classes, despite not being recommended, hints that are semantically similar [MKB⁺16].

Solutions presented by this approach use a vector-based representation, where a vector represents a sequence of refactoring operations to perform, with the order of these refactorings corresponding to their respective position in the vector [MKB⁺16]. Figure 3.5 shows how a similar approach using NSGA-II by Mansoor, Usman et al. [MKWD17] represents vector-based refactoring sequences and how mutation occurs [MKWD17].

Initial solutions are created by randomly sequencing a combination of possible refactoring operations throughout the source code, with future generations being created by splitting at random two parent solutions and crossing over their refactoring operations to create the next generation of solutions [MKB⁺16]. For mutations, one or more refactoring operations are replaced with random ones used in the starting pool [MKB⁺16].

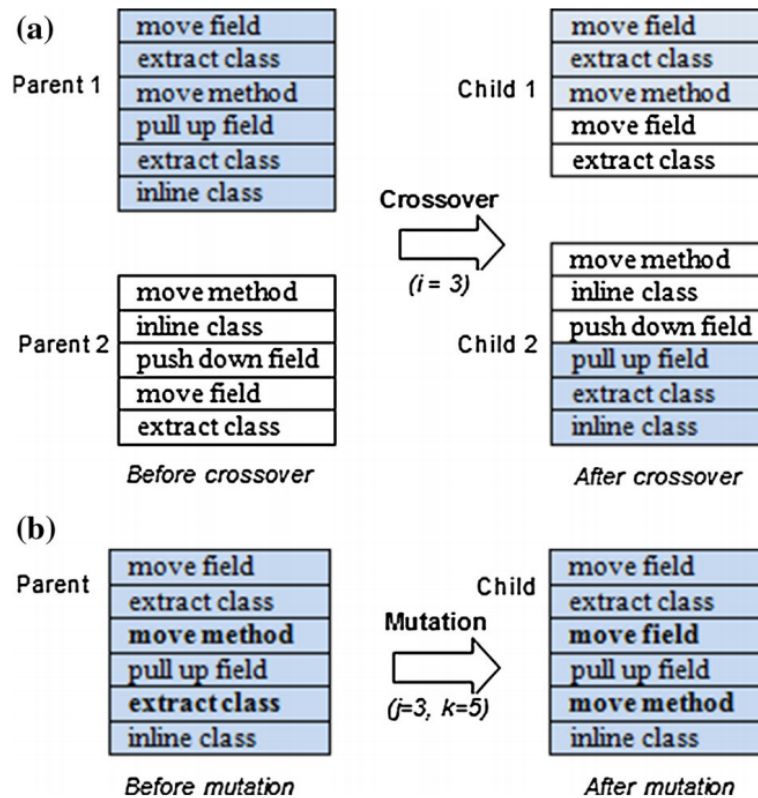


Figure 3.5: Vector-based representation of a refactoring sequence. (a) Crossover. (b) Mutation. [MKWD17]

Harman, Mark et al. [HT07] proposed a search-based approach for refactoring sequencing

which operates upon Java-language systems and is based on a variant hill-climbing approach and aims to optimize a sequence of *Move Method* refactoring operations [HT07].

The first version of this approach uses the CBO³ metric value as the fitness function for the hill-climbing algorithm, with the objective to minimize its value as much as possible [HT07]. The second version combines the SDMP⁴ metric value to the CBO value as the fitness function, as using a single metric as fitness function neglects with values of other important metrics, with the case of optimizing CBO resulting in a small number of bloated classes containing a large amount of methods [HT07].

Refactoring sequencing is essentially an optimization problem, as different refactorings result in different metric values. Multiple approaches were theorized in order to optimize the final values post-refactoring, with the vast majority coming from search-based approaches [HT07, MKB⁺16, TC16b, CT17].

Similarly to the single refactoring identification, all studied approaches use quality metric values or their combination as their fitness function towards optimization. The majority of studied approaches use graphs as the representation [Mea12, TC16b, CT17], with vector-based representation being used in two approaches [HT07, MKB⁺16].

3.2.3 Tools

There exist multiple tools aimed at easing the refactoring practice as a whole. When developing a refactoring recommendation system, it is essential to study how capable current tools are, their main features, user interactions and their shortcomings.

JDeodorant [FTC07, TCC08, FTSC11, TCC18, Nik] is a refactoring plugin for Eclipse focused on the identification and removal of common code smells with the employment of refactoring operations [FTC07, TCC08, FTSC11, TCC18, Nik].

Currently, the tool can correctly identify the presence of five different code smells, more specifically, *Feature Envy* (Figure 3.6), *Type/State Checking*, *Long Method*, *God Class* and *Duplicated Code* [TCC18, Nik]. It uses techniques such as clustering of class attributes and methods in order to identify *Extract Class* refactoring opportunities [FTSC11] and with the help of Eclipse JDT, such as ASTParser, for statement identification, and ASTRewrite, for refactoring execution, this plugin is capable of executing refactorings such as *Extract Class* [FTSC11] and *Replace Conditional with Polymorphism* [TCC08] automatically after user confirmation [FTC07, TCC08, FTSC11, TCC18, Nik].

c-JRefRec [UOII17] is a change-based refactoring recommendation tool aimed to identify *Feature Envy* code smells and fix them by applying the *Move Method* refactoring operation [UOII17]. To do so, it uses static program analysis to find structural and semantic dependencies and detect refactoring opportunities, defining a set of heuristics to use [UOII17].

³CBO - Coupling Between Objects

⁴SDMPC - Standard Deviation of Methods per Class

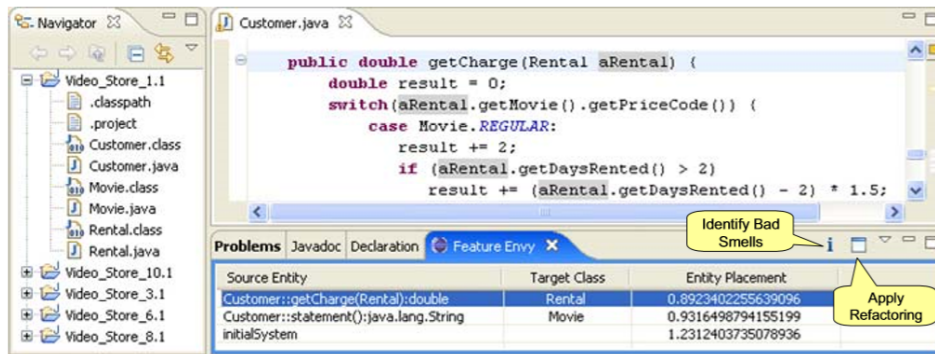


Figure 3.6: JDeodorant identification of Feature Envy code smells [FTC07].

It operates on the Eclipse IDE and takes source code as an input to its operations to analyse current relationships between different classes and methods, creating a dependency graph across the classes and methods on the system which is updated every time the developer operates on the code [UOII17]. A semantic analysis is also used to extract code identifiers in order to identify the *Move Method* candidates [UOII17].

Two interface views are provided by c-JRefRec: the Class State View and the Refactoring Candidates View [UOII17]. Class State View provides four different metrics - number of methods, incoming and outgoing calls to members of the class being analysed, number of classes calling members from the current class and number of classes to which members of the current class are accessing - in order to show coupling and cohesion between classes [UOII17].

Refactoring Candidates View lists the potential *Move Method* candidates, evaluating them according to changes in metrics occurring in case a certain method is selected for refactoring [UOII17]. These metrics include the difference between the incoming and outgoing calls to members of the selected class, difference between the number of classes which use methods or fields of the current class and the difference between the number of classes to which members of the current class are accessing [UOII17].

Code-Imp [MÓ11, ÓTH⁺12] is an automated refactoring tool which operates on Java systems and employs search-based techniques to find refactoring operations to perform on a given source code [MÓ11, ÓTH⁺12]. This tool supports fourteen different refactoring operations across three categories: method-level refactorings, field-level refactorings and class-level refactorings [MÓ11, ÓTH⁺12].

It uses RECODER⁵ to generate an AST relative to the source code being refactored, where the refactoring operations occur [MÓ11, ÓTH⁺12]. Once all refactorings are executed, Code-Imp pretty-prints the AST back into source code, finishing the refactoring process [MÓ11, ÓTH⁺12].

Code-Imp mainly uses hill-climbing as its search algorithm [MÓ11, ÓTH⁺12], more specifically, first-ascent and steepest-ascent hill-climbing [ÓTH⁺12]. In order to evaluate performed refactoring operations, Code-Imp offers a set containing twenty-eight quality metrics,

⁵<https://sourceforge.net/projects/recoder/>, Last accessed on July 22, 2020

with the fitness function being defined by a combination of these metrics using either weighted-sum optimality - sum of metrics with different weights assigned, representing the metric's importance - or pareto optimality - a refactor is only an improvement if it improves one or more metrics without deteriorating another [ÓTH⁺12].

DNDRefactoring [LCJ13] is a refactoring tool built with the objective of streamline and make the refactoring practice more intuitive, by allowing developers to directly manipulate program elements inside the development environment, eliminating the need to navigate through menus and configure tools [LCJ13]. Using the example of performing an *Extract Method* refactoring shown on Figure 3.7, the developer just needs to select the section of a method to extract and drag and drop it to the target class [LCJ13].

By using an effective mapping process to map drag sources and drop targets to a refactoring operations, DNDRefactoring overcomes two common problems identified by the creators [LCJ13]: invocation inconsistencies - as there is no standard for refactoring naming and each IDE has its own naming for the same operation; configuration overload - current tools rely on sometimes complex configuration dialog menus to function properly, even though most developers do not bother with changing default configurations [LCJ13, MHPB12].

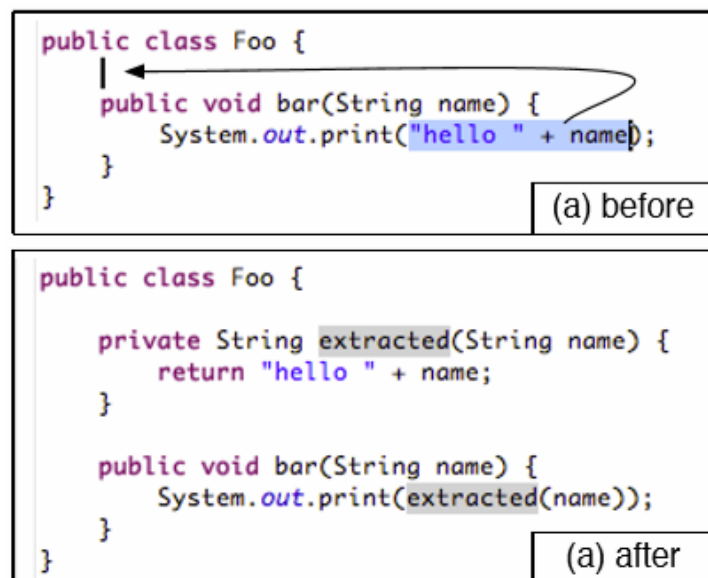


Figure 3.7: Extract Method refactoring on DNDRefactoring tool [LCJ13].

Due to the difficulty of translating certain refactoring operations to drag-and-drop gestures, DNDRefactoring only supports move and extraction-based refactorings [LCJ13]. Another problem arises, as a certain gesture may portray different operations [LCJ13]. Using the example of dragging an expression from inside a method to a class, this gesture can mean the usage of either an *Extract Method* or an *Extract Constant*, with the tool having to default to one of the choices to stay loyal to its design values [LCJ13].

As shown by this study, current refactoring tools are capable of extensive analysis and refactoring execution in very large systems as shown by **JDeodorant** [FTC07, TCC08, FTSC11, TCC18, Nik], **c-JRefRec** [UOII17] and **Code-Imp** [MÓ11, ÓTH⁺12]. However, these tools require the developer to manually execute them and are expected to operate during large periods of time in order to find the best possible refactoring for any given context.

This lengthy execution makes the use of these tools unfeasible during the development phase. Tools such as **DNDRefactoring** [LCJ13] aim to hasten and simplify refactoring related with moving or extracting code using drag-and-drop operations, promoting the refactoring practice during development.

3.2.4 Live Tools

Most refactoring tools and approaches do not explore the liveness concept during their operations, mostly due to how incompatible refactoring optimization and liveness are. However, some tools related with refactoring already operate live.

A study on how liveness is employed in current tools supporting it was performed, with the intent of finding more about their live implementation and supported features.

BeneFactor [GMH11] is a live refactoring tool for Eclipse⁶ that provides automated refactoring to developers who already initiated manual refactoring [GMH11].

This tool has two main components: the refactoring detection module, which operates live on the background, detecting potential manual refactoring behaviors by the developer's part and the code modification module, responsible for finishing the incomplete manual refactoring being performed [GMH11].

The first component of this tool, the refactoring detection, operates live and uses two strategies to fulfill this task: a code-change based approach, which analyses changes performed in the AST, such as node addition, update and deletion, allowing for detection of, for example, variable renaming; an action based approach, which takes into account recent developer actions, such as copy, paste and selection of code and, if matched with common refactoring workflows, assumes the developer is performing manual refactoring [GMH11].

The code modification module allows for safe and automated refactoring completion, without the user needing to undo any changes [GMH11]. It uses the Eclipse Language Tool Kit and consists of reverting the code to a previous state, recovering any partial refactored code, while collecting information about the code being refactored such as original variable names and range of refactored code statements [GMH11]. After this, precondition checks are performed (to be live in the future) and, if successful, the automated refactoring is performed [GMH11].

⁶<https://www.eclipse.org/>, Last accessed on July 22, 2020

Soares, Gustavo et al. [SMHG13] proposed using the functionality of the existent Eclipse refactoring plugin SafeRefactor [SGSM10], to design a new plugin capable of warning the user about behavior changes post-refactoring in a live manner [SMHG13].

SafeRefactor aims to execute refactoring operations safely by using generated unit tests to compare the system after changes and alerting the developer in case any behavior change happens post-refactoring [SGSM10]. This tool requires developers to manually execute it and wait for a considerable amount of time for results, which is not ideal, despite its functionalities being useful to developers [SMHG13].

The proposed live tool solves the performance problem by running the test generator in a separate JVM with binaries of the base version of the system pre-loaded and then starts a communication with the new plugin via RMI [SMHG13]. After that, generated test cases are sent and executed on the modified program via reflection, allowing for results to be compared for behavior changes [SMHG13].

Although there exist refactoring tools operating in a live way [GMH11, SMHG13], they cannot be categorized as traditional refactoring recommendation systems. **BeneFactor** [GMH11] is aimed towards live manual refactoring detection and automatic refactoring completion and **Soares, Gustavo et al.** [SMHG13] approach detects erroneous manual refactoring.

As such, we believe there is opportunity to further implement liveness on refactoring tools, with features similar to traditional refactoring recommendation tools, such as the refactoring suggestion.

3.3 Quality Metrics Tools

Refactoring artifacts studied throughout this dissertation involved the usage of quality metrics in some way. As such, we believe it is only natural to study the capabilities of current tools related with the measurement and evaluation of quality metrics.

Fernandes, Sara [Cou19] designed a Visual Studio Code extension tool which operates in TypeScript and JavaScript systems and incorporates liveness to quality metric visualization, so that developers stay aware of the impact certain changes on the system cause on the values of quality metrics [Cou19]. It also allows Git integration in order to observe the evolution of the selected metrics [Cou19].

It supports the live visualization of any combination picked by the user (Figure 3.8) in a pool of nineteen metrics ranging from simple metrics, such as the number of lines of code, to calculated Halstead metrics [Hal77] such as the implementation effort and the coding time needed to implement an algorithm [Cou19].

Although the main focus of this tool is to provide live metric visualization, it also supports the execution of *Split Variable Declaration* and *Extract Variable* refactoring operations [Cou19].

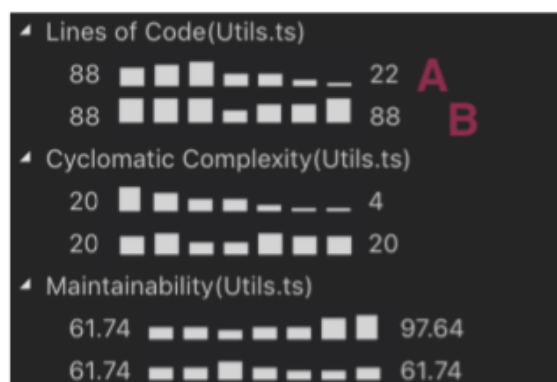


Figure 3.8: Evolution over time of selected metrics [Cou19].

Tech Debt Tracker [Ste19a] is a Visual Studio Code extension that also operates on TypeScript and JavaScript systems only, which mainly aims at helping developers, working alone or as a team, to improve the overall quality of the source code with the usage of quality metrics [Ste19a].

It analyses the code using the following quality metrics: method length, cyclomatic complexity, understandability, parameter count, nesting depth and comment density [Ste19b], displaying to the user interface the results based on pre-defined thresholds [Ste19a, Ste19b]. Based on the results of these metrics, an algorithm evaluates the analysed code and grades it from *A* (the least likely to receive a bug fix in the near future) to *E* (the most likely to receive a bug fix in the near future) [Ste19c].

For development teams, Tech Debt Tracker offers an interface to discuss and prioritize actions to take on a function in order to increase its quality, constructing a prioritized backlog where each developer can consult the voted priorities [Ste19a].

CodeMetrics [Kis16] is a simple Visual Studio Code extension whose main functionality is calculating the complexity of a function, showing it to the user as code lenses above the respective evaluated method [Kis16]. The user can click on the code lens to gather more information about the source of the complexity reported by the extension and evaluation customization is also featured, allowing for the user to decide the amount of complexity a certain node type should be awarded during evaluation [Kis16].

CoreMetrics [Jas18] is a Windows-only Visual Studio Code and .NET Core extension which analyses source code, providing the following metrics for the user: cyclomatic complexity, class coupling, inheritance depth, lines of code and maintainability index [Jas18]. It functions based on threshold values for these metrics, which are freely customizable by the user [Jas18].

MetricsReloaded [Bas04, Had14] is a plugin that works on any IntelliJ-based IDE and offers a wide array of metrics for Java systems, but a limited amount for any other language supported by the IDE being used [Bas04].

Users can customize the range of the analysis, allowing for file, module and project scans and create metric profiles by selecting metrics to show from a vast group, including the Chidamber-Kemerer metrics suite [CK91], complexity and dependency [Had14]. Further customizability is allowed, as the user is capable of creating custom metrics and defining threshold values [Had14].

Output of scans are presented using a table, but the user can also opt to display this information in the forms of distribution diagram, histogram or pie chart, if applicable [Had14].

CodeMR [Cod18] is an IntelliJ-based IDE plugin which operates over Java, Kotlin and Scala projects [Cod18]. It provides code metrics and high-level quality attributes (derived from the combination of code metrics) and a platform for the user to easily visualize them [Cod18].

Offered code metrics are divided into four categories: project metrics, package metrics, class metrics and method metrics [Cod18].

Teamscale [HNJ19] is a code analyser which performs static and dynamic analysis across twenty-six programming languages [HNJ19].

It processes data from version control systems, external analysis tools and bug and issue trackers and the results are available to developers on their IDE plugins as shown on Figure 3.9 [HNJ19]. Managers can select how much technical debt should be allowed by Teamscale, by choosing between no monitoring at all to fully scan until no technical debt is found on the system [HNJ19].

Three different types of analysis are ran by the system: code quality analysis - composed of code structure metrics, code duplication, comment completeness and architecture conformance, test gap analysis and code usage analysis [HNJ19].

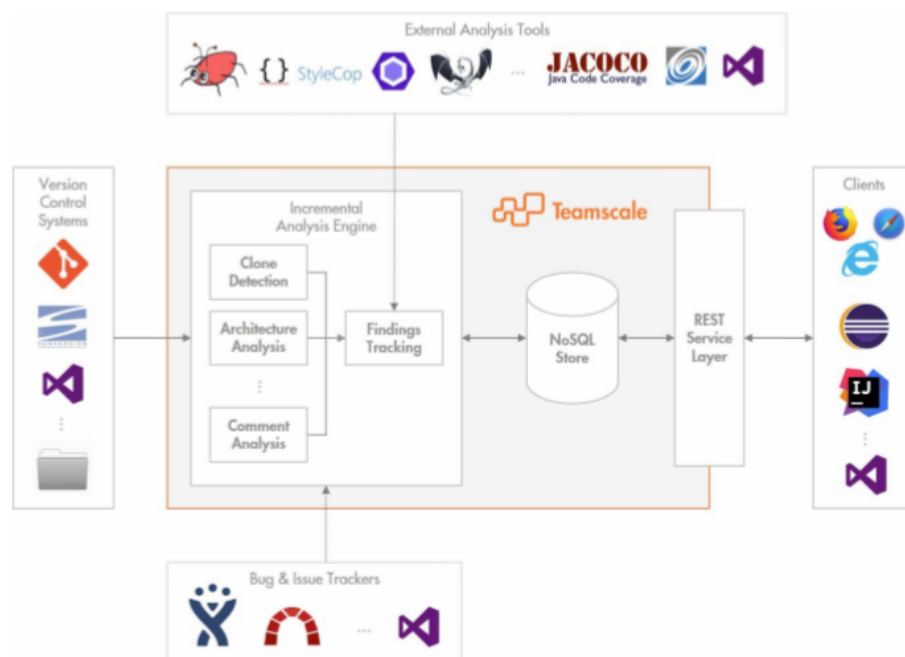


Figure 3.9: Teamscale's architecture [HNJ19].

iPlasma [MMM⁺05] is a quality analysis tool for Java and C++ systems which uses model extractors, quality metrics analysis and code duplication detection which has shown results in *real world* systems like Mozilla and Eclipse [MMM⁺05].

A model of the system being analysed is created as the first step for design analysis, with the goal of extracting all necessary information about a system such as functions and variables (and respective usages), inheritance between classes and call-graphs [MMM⁺05].

After that, the created model is analysed using quality metrics, categorized into four categories: size metrics, complexity metrics, coupling metrics and cohesion metrics [MMM⁺05]. iPlasma also implements structural analysis build on top of the created model, detection strategies which identify deviations from *good* design criteria based on metric thresholds and code duplication detection [MMM⁺05].

As found out through this review, current quality metric tools operate as a plugin or extension of existent IDEs. This makes sense, as most, if not all, software development occurs inside an IDE, and providing quick access to metrics is important.

A wide variety of metrics are supported by most of these tools, with only one [Kis16] supporting a single metric. Most tools calculating multiple metrics disclose if they operate in a live way or not. Only **Fernandes, Sara** [Cou19] discloses operating live and support the measurement of multiple metrics.

Regarding the supported languages, current tools operate on different languages, with the main ones being JavaScript or similar languages [Cou19, Ste19a, Ste19b, Ste19c, Kis16] and Java [Bas04, Had14, Cod18, MMM⁺05].

3.4 Results and Discussion

A quick overview of Tables 3.2 through 3.5 show the results of the literature review performed on the topics of live environments and languages, refactoring proposals and tools and quality metrics tools.

The topic of live software development, and research done on the state of the art on topics such as live development environments and live languages show the potential this novel field of software development and agile can have and its impact on a developer's work environment.

Table 3.2: Live environments comparison.

Name	Reference	Language	Target Language	Features
-	[LL13]	Pharo, JavaScript	Moon [LL13]	Entity, state and evolution visualization
Kanon	[OMIA17]	JavaScript-based	Data structures	Connected parallel panels, visual representation and summary of changes
Circa	[Fis13]	-	Circa [Fis13]	Powerful introspection and code manipulation
Euclase	[OMB13]	-	Undisclosed visual language	Powerful primitives, fast evaluation and beginner friendliness

By analysing Table 3.2 representing the results obtained on the topic of live environments and their respective key features, we can observe the benefits of adding liveness to work environments, whether the respective work areas are directly related to programming or not.

A rich visual overview of the state of both the source and respective execution is core in live environments allows for developers to stay aware of changes done to the system, with features such as introspection, manipulation and beginner friendliness allowing for developers to explore and experiment with the system.

Table 3.3 shows results of the review performed on refactoring approaches and sequencing, where we can observe the wide range of approaches already explored to identify and sequence refactoring opportunities, with search-based algorithms being the most common approach to the refactoring problem.

Table 3.3: Refactoring proposals comparison.

Reference	Used Refactorings	Target Code Smells	Technique	Uses Metrics	Outputs Sequence	Representation
Bavota [BOD ⁺ 10]	Extract Class	-	Game Theory	Yes	Yes	-
Pantiuchina [Pan19, PBTP18]	-	God Class	Random Forest	Yes	No	-
Bavota [BDO11]	Extract Class	Blob Class	MaxFlow-MinCut	Yes	No	Graph
Tsantalis [TC09]	Move Method	Feature Envy	-	Yes	No	-
Meananeatra [MRA11, Mea12]	Six operations, including Replace Temp with Query, Introduce Parameter Object and Extract Method	Long Method	Heuristics	Yes	Yes	Graph
Tarwani [TC16b]	Seven operations, including Extract Method, Move Method, Extract Class and Extract Parameter	Ten smells, including God Class, Long Method, Feature Envy and Long Parameter List	Greedy Algorithm	Yes	Yes	Graph
Chug [CT17]	Nine operations, including Extract Method, Extract Class, Replace Type Code with State/Strategy and Encapsulated Field	Ten smells, including God Class, Long Method, Type Checking, Dead Code and Data Class	A* Algorithm	Yes	Yes	Graph
Mkaouer [MKB ⁺ 16]	Ten operations, including Extract Class, Extract Interface, Inline Class, and Push Down	-	NSGA-III	Yes	Yes	Vector-based
Harman [HT07]	Move Method	-	Hill-Climbing	Yes	Yes	Vector-based

Representation of the solutions presented by these approaches are mostly done using graphs or vector-based representations, with all proposals using metrics to some extent as fitness functions. As expected, approaches which operate over multiple refactoring operations are the ones which generally output refactoring sequences.

Through analysis of Table 3.4, some observations can be done. Most existent refactoring tools do not apply liveness on their operations and operated on mainstream languages such as Java. Also, most tools are, in some way, part of an existent IDE in the form of plugins or extensions.

Another observation that can be done, although there is a small sample size, is that tools which operate live support a lower amount of possible refactoring operations.

Table 3.4: Refactoring tools comparison.

Name	Reference	Supported Refactorings	Languages	Live	Plugin
JDeodorant	[FTC07, TCC08] [FTSC11, TCC18, Nik]	Move Method, Replace Conditional with Polymorphism, Extract Method, Extract Class, Extract Clone, Replace Type Code with State/Strategy	Java	No	Yes
c-JRefRec	[UOIII17]	Move Method	Java	No	Yes
Code-Imp	[MÓ11, ÓTH ⁺ 12]	Push Up/Down Method, Increase/Decrease Method/Field Visibility, Pull Up/Down Field, Extract/Collapse Hierarchy, Make Superclass Abstract/Concrete, Replace Inheritance/Delegation with Delegation/Inheritance	Java	No	-
DNDRefactoring	[LCJ13]	Extract-based and Move-based	Java	No	Yes
BeneFactor	[GMH11]	Rename Method/Variable, Extract Method	Java	Yes	Yes
-	[SMHG13]	-	Java	Yes	Yes

Table 3.5: Quality metric tools comparison.

Name	Reference	Supported Metrics	Languages	Live	Plugin
-	[Cou19]	Nineteen metrics, including Lines of Code, Cyclomatic Complexity, Halstead [Hal77] metrics	TypeScript, JavaScript	Yes	Yes
Tech Debt Tracker	[Ste19a, Ste19b] [Ste19c]	Function Length, Cyclomatic Complexity, Understandability Argument Count, Nesting Depth, Comment Density	TypeScript, JavaScript	-	Yes
CodeMetrics	[Kis16]	Cyclomatic Complexity	TypeScript, JavaScript, Lua	Yes	Yes
CoreMetrics	[Jas18]	Cyclomatic Complexity, Class Coupling, Depth of Inheritance, Lines of Code, Maintainability	-	-	Yes
MetricsReloaded	[Bas04, Had14]	Chidamber-Kemerer [CK94], MOOD, Class Count, Dependency, Complexity	Full Java support, limited for others	-	Yes
CodeMR	[Cod18]	Thirty-seven metrics, including Coupling, Lack of Cohesion, Lines of Code, Complexity, Instability, Depth of Inheritance Tree	Java, Kotlin, Scala	-	Yes
Teamscale	[HNJ19]	-	Twenty-six languages	-	Yes
iPlasma	[MMM ⁺ 05]	Eighty supported metrics, including Lines of Code, Cyclomatic Complexity, Coupling Between Objects, Tight Class Cohesion	Java, C++	-	-

Table 3.5 shows the results found on the literature review performed over quality metric tools and their main characteristics. We concluded that most existent tools available operate as extensions or plugins on the developer IDE and calculate metrics on systems created using mainstream languages such as Java, C++ and JavaScript.

Most of the found tools support a wide array of metrics to present to the user, with some even displaying that information in multiple forms, such as charts. We can also observe the incorporation of live programming philosophies on some of the current tools.

Chapter 4

Problem Statement

4.1	Open Issues	33
4.2	Research Questions	34
4.3	Proposal	35
4.4	Validation	35

This chapter describes the problem statement of this dissertation, with Section 4.1 briefly explaining the found problems with current refactoring tools and approaches.

Section 4.2 details main hypothesis of this study and the research questions proposed, followed by a description of our proposed solution in Section 4.3 and the validation methodologies used to answer our research questions and validate the main hypothesis in Section 4.4.

4.1 Open Issues

Chapter 3 describes in detail existent solutions to the refactoring problem and its sequencing, current applications of liveness in software development tools and how quality metrics tools operate in order to inform the users about the values of metrics on the system being developed.

Based on the literature review analysis, it is possible to identify certain open issues present in current tools and approaches.

Refactoring is still hard. When performed manually, the practice of refactoring is hard and error-prone [GMH11]. Knowing exactly where to apply refactoring and how to apply it on any given context is a difficult task for a human to do. In cases where manual refactoring is wrongly employed, its effect can be counter-productive, further deteriorating the source code. Automatic and semi-automatic refactoring tools attempt at making the refactoring practice easy and painless for both the developer and the system, by applying complex

algorithms to find optimal refactoring operations in a fraction of the time a human would take [TCC18, UOII17, MÓ11, ÓTH⁺12]. However, most of these tools are often used separately from the development process and end up severely underused [MHPB12].

Current recommendation tools lack liveness. Despite tools such as BeneFactor [GMH11] and others [SMHG13] implementing the philosophy of live programming onto refactoring systems, they cannot be portrayed as *refactoring recommendation systems*. Their use of liveness is aimed towards either alerting the user of behavior changes due to poor refactoring [SMHG13] or finishing a refactoring manually started by the user [GMH11], rather than recommending refactoring operations to the developer beforehand.

Current refactoring tools could benefit from enhanced visualization and customization.

Quality metrics are commonly used to evaluate refactoring opportunities as they usually reflect upon the system's status. However, despite current tools being effective at evaluating opportunities, with some even showing the impact of each refactoring on the evaluated metrics, each project is different and the regular developer is not aware of values in the measured metrics considered *safe* in the given context. This can translate in the developer not realizing the true impact a refactoring can have on the system's metrics and to what metrics should the developer be aware of when developing new code.

4.2 Research Questions

Based on the open issues previously stated, the study performed during this dissertation aims to understand if the existence of a live refactoring recommendation system aids developers with the refactoring practice during software development.

The main purpose of incorporating liveness onto refactoring recommendation systems is to offer developers a more efficient way of executing near-optimal refactoring sequences and tackle technical debt during the process of development, without having to wait for the maintenance process to do so.

As such, in this dissertation, we propose the ensuing hypothesis:

“A live refactoring recommendation system operating based on quality metrics is able to aid developers towards a more maintainable code base.”

With the hypothesis defined, this dissertation focuses on answering the following research questions:

RQ1 *“Does providing developers with software visualization tools improves their awareness towards code smells mitigation?”* One of the point of study of this dissertation is to understand the impact of visualizing software metrics on the developer's actions towards mitigating the presence of code smells.

RQ2 *"Does the usage of these tools result in improvements on quality metrics, leading to a more maintainable code base?"* Our study aims to understand how a live refactoring tool can cause developers to act on a given code base in order to improve metric values as to increase the code's maintainability.

RQ3 *"Does early exposure to software metrics and consequent early action by the part of the developer lead to a more maintainable source code on earlier stages of development?"* Another point of study of this dissertation is to understand whether early exposure to software metrics allow developers to detect the presence of problems within the code, and if early action leads to an overall more maintainable code during earlier stages of development.

4.3 Proposal

In order to help us answer these research questions and thus validate our hypothesis, a Visual Studio Code extension was developed, with its main objective being a way to provide developers with live evaluation of quality metrics and refactoring recommendations based on the values of these metrics.

This tool should offer a simple interface for the user to quickly monitor current values of supported metrics during regular development, while using their values to compute near-optimal refactorings that the user can automatically execute effortlessly.

A custom repository crawler was also developed, as a mean to generate data on open-source repositories, for us to answer our research questions. This crawler operates on top of our primary tool and uses the developed API to simulate regular usage of our tool, allowing us to scale our data collection.

More details about the conception of both the Visual Studio Code extension and the development needed towards our automated analysis strategy to validate our theory can be consulted throughout Chapters 5 and 6, respectively.

4.4 Validation

As to validate the hypothesis proposed in this dissertation and provide answers to the formulated research questions, a proposed solution, described throughout Chapter 5, was developed and thus, validated.

Our validation strategy resolves around the employment of two methodologies: survey research and automated analysis. During the survey analysis (Sections 6.1.1 and 6.2.1), we will be conducting a survey which aims to help us understand if our tool is capable of showing the importance of software visualization towards code smell mitigation and whether the tool's refactoring suggestions can prove useful to developers. The automated analysis (Sections 6.1.2

and 6.2.2) provides us with concrete data about the impact of the usage of our tool in *real-life* open-source projects.

By using this strategy, we aim to gather sufficient numerical and non-numerical data to help us answer the proposed research questions and thus validating our hypothesis. We believe that selecting multiple empirical methods as a mean to validate our hypothesis provides a strong basis for mitigating potential weaknesses in each method and minimizing the threats to our validation.

Chapter 5

Proposed Solution

5.1	Context	37
5.2	Usage	39
5.3	Automated Refactoring	39
5.4	Live Metrics	45
5.5	Visual Studio Code Extension	50
5.6	Summary	52

This chapter provides a detailed description of the implementation of the designed tool, aimed at solving the problems found with current existent tools, stated during Chapter 4.

Section 5.1 briefly contextualizes our proposed solution with current trends in computation and software engineering practices and provides a brief overview on how our solution works.

Section 5.2 briefly describes how the tool's features are used by the developer, with Sections 5.3 and 5.4 describe the implemented refactoring and metric features, followed by Section 5.5 contextualizing how our tool fits within the Visual Studio Code's interface, with Section 5.6 briefly summarizing the features of our solution and the contents of this chapter.

5.1 Context

In Chapter 4, we outlined some of the problems found with current refactoring tools developers can use, ranging from their user unfriendliness to a lack of visual feedback on provided changes. We believe that, by providing programmers with a tool capable of filling the gap on most of these issues, both productivity and agility on code base familiarity can be increased.

Computational power translates into a higher efficiency and power on parallel computation, allowing for heavier work loads to be processed in the background. With its increase over the last

few years, coupled with a wide adoption of agile methodologies across the industry, we believe it is the perfect timing to further explore the concept of liveness during the refactoring process.

Our solution proposal consists of a tool capable of **computing refactoring recommendations to the developer and evaluating them based on the values of certain quality metrics, providing near-instant results**. This tool operates upon JavaScript and TypeScript files, as an extension to the Visual Studio Code text editor.

Once the interface command is ran, the user is shown the values of multiple metrics computed live, combined into metrics related to the entire file and metrics relative to each individual method found across all classes in a file. A section describing the found refactoring candidates is also shown, for the user to consult them visually before executing the respective command.

Our choice to approach this problem using a **file-by-file** analysis instead of a full workspace analysis comes with inherent limitations, mainly issues revolving around inheritance across different files. We believe this decision, although limiting in some use cases, is the right choice, considering the utmost importance of maintaining the liveness aspect of this tool. Using this file-by-file approach allows the tool to provide near-instant results for all three supported refactorings, resulting in a higher scalability compared to analysing the entire workspace, which would greatly depend on how large the code base is. Throughout this chapter, we'll discuss the impact of these limitations where relevant.

The choice of this programming language and development platform combination was chosen based on the fact that JavaScript has been the most popular language for eight years in a row across the industry, according to the *2020 Stack Overflow Developer Survey*¹ with a staggering 69.7% of professional developers' votes. TypeScript shows up at number eight in this category, with a 28.5% usage in the professional developers' votes, an increase of 5% when compared to last year's survey². As for the development environment, the 2019 survey shows that Visual Studio Code is the most used environment across different types of development, with the exception of mobile development, with a 50.7% of all respondents' votes.

Implementation details of this tool are described in greater detail on Sections 5.3 through 5.5 and describe aspects related to the tool's supported refactorings, candidate gathering and filtering, usage of both TypeScript's and Visual Studio Code's APIs, limitations and design choices taken throughout development.

This dissertation is being carried out by the Software Engineering group at FEUP, where another dissertation about liveness in refactoring is also being worked on. However, neither of the two dissertations are dependent on one another and, despite both being centered about refactoring, are clearly distinct on the solution approach. This dissertation focuses on the evaluation, and consequent usage, of software metrics as a mean to provide the developer with refactoring suggestions which turn the code more maintainable.

¹<https://insights.stackoverflow.com/survey/2020>, Last accessed on July 22, 2020

²<https://insights.stackoverflow.com/survey/2019>, Last accessed on July 22, 2020

5.2 Usage

One of the main objectives when designing this extension, was to make it as simple and intuitive as possible, while providing powerful tools to help developers working towards more maintainable code.

As such, the implemented features were designed with developer efficiency in mind, with close to no effort required by the programmer to make the extension work. When the developer runs the extension interface command, a new tab is created, showing a webview with information about general file metrics, more specific metrics relative to methods and functions found in the file with found refactoring suggestions shown last. Figure 5.1 shows how our tool's interface fits in the Visual Studio Code's user interface, with the user being capable of assess the metrics quickly, mostly in part due to the color scheme used and the compactness of the interface.

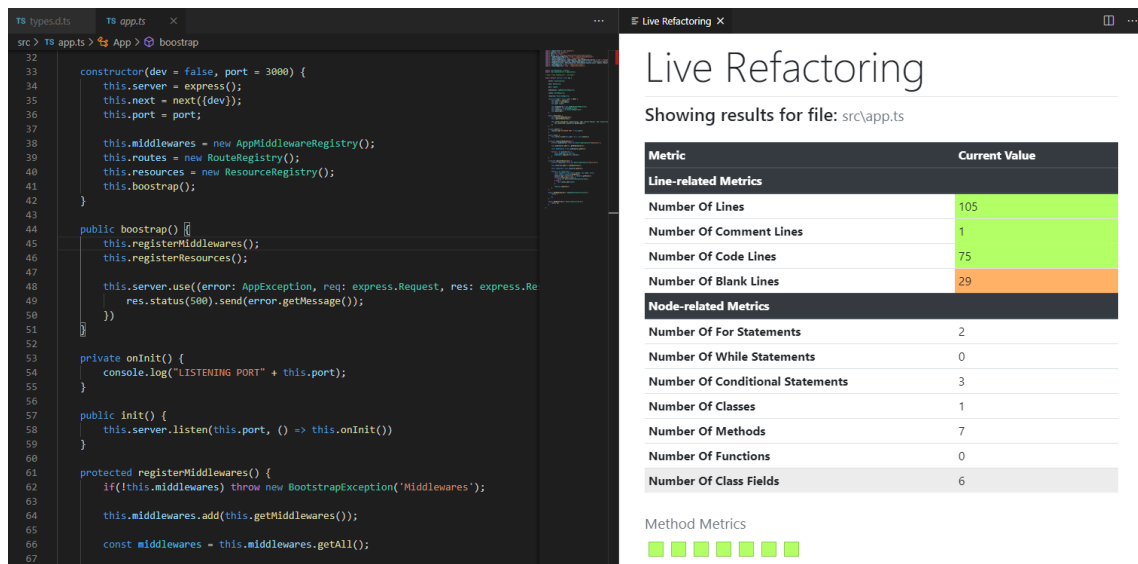


Figure 5.1: LiveRefactoring's user interface within Visual Studio Code.

With the refactoring suggestions requiring user interaction, due to the semi-automated support to empower users with the final decision, our interface provides any extra information required for a refactoring to be executed, where it will be executed and, if applicable, the impact it will have on the metrics. This is done by providing the user with three executable commands, one for each supported refactoring, responsible of executing the best found suggestion on the current state of the file. Once the file is saved, both the metrics and suggestions are recalculated and the interface is updated.

5.3 Automated Refactoring

One of the core modules of this extension, it is responsible to automatically detect possible refactoring operations, evaluate them and semi-automatically execute the best evaluated

refactoring. It is composed of three sub-modules, each responsible for a kind of refactoring supported by this tool: the **Extract Method**, **Extract Class** and **Extract Variable** refactorings.

According to the performed literature review on refactoring tools and tools operating through liveness, current Visual Studio Code extensions which support refactoring operations mostly work similarly to a linter, offering small fixes to source code based on certain predetermined rules as, for example, warn the user about a missing semicolon at the end of a statement.

As such, we believe there is room for improvement for current refactoring tools operating live. These three refactoring operations were chosen to our suite both due to how commonly these are performed by developers and their potential to be performed automatically.

This module of the tool extensively uses the TypeScript compiler API³, as it provides a powerful interface for file analysis through the creation of an AST of a source file. Some of its data structures are used throughout the pseudocode descriptions of created algorithms, including:

- **Node:** The main structure of the created AST, represents a node in the tree. It functions as the superclass for all other node types contained within the AST. Relevant information from this data structure contain its *start* and *end* positions, as to locate the line and character of the node within the text editor;
- **ClassDeclaration:** A subclass of the *Node* superclass, represents a class declaration. Most commonly possesses *PropertyDeclaration*, *Constructor* and *MethodDeclaration* nodes as its children;
- **PropertyDeclaration:** A subclass of the *Node* superclass, represents the declaration of a class field outside of the constructor parameters. Only fields declared this way are considered for the algorithms as declaration inside constructor parameters only differ from regular parameters created with the *Parameter* node due to the presence of *keywords*. Constructor parameters do not count as *PropertyDeclaration*;
- **MethodDeclaration:** A subclass of the *Node* superclass, represents the declaration of a method contained within a class. Usually formed by an *Identifier* (name of method), a list of *Parameter* nodes and a *Block* node, parent of a list of *Statement* nodes containing the method's code.

5.3.1 Extract Method

In order for an Extract Method refactoring to be performed, the first step is to select the fragment of code to be extracted onto the new method. This candidate extraction algorithm operates in two different phases: the first one extracts individual nodes across all methods in each class on a file, with the second phase combining the found nodes resultant of the first phase.

To be considered a potential refactoring candidate, a fragment of code is checked for the following criteria:

³<https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API>, Last accessed on July 22, 2020

Algorithm 1 First phase of the candidate retrieval process (single nodes)

```

1: procedure GETEXTRACTABLEFRAGMENTS(sourceFile)
2:   ranges : Range[]  $\leftarrow$  []
3:   nodes : Node[]  $\leftarrow$  []
4:   for each node  $\in$  sourceFile do
5:     if isStatement(node) & !isSourceFile(node.parent) & !isNodeOnConstructor(node)
6:       then
7:         startingPosition  $\leftarrow$  Position(node.start.line, node.start.character)
8:         finishPosition  $\leftarrow$  Position(node.end.line, node.end.character)
9:         range  $\leftarrow$  Range(startingPosition, finishPosition)
10:        ranges.push(range)
11:        nodes.push(node)
12:      end if
13:    end for
14:  end procedure

```

1. **Statements:** Candidate nodes selected during the first phase of the algorithm, described with detail in Algorithm 1, are required to be one of any of the 20 statement types supported by the TypeScript API;
2. **Node hierarchy:** Selected nodes cannot be direct descendants of a *SourceFile* node. In other words, for a node to be selected, it needs to be contained within another node, such as a function or method nodes;
3. **Statement blocks:** Consecutive statements are joined to create new and more complex Extract Method candidates, which are evaluated individually, simulating a programmer's train of thought when manually selecting fragments to extract;
4. **Filtering:** As expected, many of the extractable fragments in a file and respective combinations result in trivial refactorings. To avoid cases where either the algorithm extracts all the code from the original method, rendering the entire selection process pointless, or the candidate is composed of trivial extractions, filtering needs to be performed. Found candidates are filtered to **not be entirely composed of trivial statements**, such as VariableStatements and filtered by default as to **not contain more than 80%** of the original method's statements and contain a **minimum of one statement**. These settings are highly customizable by the user, allows for programmers to make changes to this tool in order to fit any individual project.

Once the final set of candidates is formed, metrics relevant to the evaluation of Extract Method fragments are calculated, mainly their **cyclomatic complexity**, **number of statements** and the **lack of cohesion of methods** (LCOM), based on an approach to evaluate Extract Method refactorings suggested by Meananeatra, Panita et al. [MRA11]. This acts as a way to predict on how extracting the candidate being analysed reflects on file metrics. By default, our solution

prioritizes candidates which **lower the highest found cyclomatic complexity**, followed by **lowering the highest number of statements** and by **lowering the class' LCOM values**.

The semi-automated refactoring execution is performed with the usage of the Visual Studio Code API, more specifically the *executeCommand()* method. As Visual Studio Code natively supports the Extract Method refactoring on TypeScript source files, we believe reusing this functionality instead of our own custom solution is more efficient, as no time was wasted reinventing the wheel, and ensures **our tool acts naturally for previous users of the built-in command**. Should the native command be updated in future patches, our tool should reflect the same behavior.

Essentially, this module is responsible for carefully selecting the best Extract Method out of all possible solutions. Once the best result is found based on described criteria, our tool simply invokes the Visual Studio Code built-in command for Extract Method.

5.3.2 Extract Class

The second supported refactoring is the Extract Class, which takes one cluttered class and transforms it into two new classes whose methods share more similarities. For this refactoring, as it is not natively supported by Visual Studio Code, we needed to create a custom solution, which could have been done in one of two different ways:

1. Using the TypeScript compiler API to procedurally generate a new post-refactoring AST of the entire source file, resultant of changes imposed by the Extract Class refactoring. Once the new AST is created, we could transform the document by *pretty-printing* the AST to the text editor. This is the **most robust** way to perform this task, however it comes at a far **greater computational cost** and **difficulty of implementation**, as it requires parsing an entire source file and essentially reconstruct it every time an Extract Class is required;
2. Use clever string manipulation to directly manipulate the contents of the document according to the desired refactoring, similarly to how manual editing works. Although this might seem a naive way to perform this task, it requires a **significantly lower computation cost** and a **simpler implementation** when compared to the first option.

In this tool, we tackled this problem by employing the **second approach**, as to achieve as close to full liveness as computationally possible we require the algorithm to operate as fast as possible. As manually performing an Extract Class is, in its nature, a string manipulation process, we believe the trade-off between algorithm robustness and execution time makes the second option more appealing, when applied to our use case.

To create this algorithm, Fowler's Extract Class mechanics [Fow99] were followed, with each step returning the altered class string to be used as a parameter for the next step. Our tool constantly calculates which methods should be extracted, using a method adapted from **Bavota, Gabriele et al. [BDO11]**, where three $M \times M$ matrices (with M being the amount of methods in the class being analysed) containing information about the following metrics, respectively:

Conceptual Similarity between Methods (CSM), Call-based Dependence between Methods (CDM) and Structural Similarity between Methods (SSM).

Once all three matrices are calculated, a final weight matrix is created and values on its first row below a certain threshold represent methods that should be extracted. With the target methods found, the string manipulation algorithm starts by creating a two-way link between classes, updating the contents of the document to reflect the move of methods to a new class, finishing with the creation of the new class. Algorithm 2 illustrates how the contents of the original class are updated to reflect methods moving around the two classes.

Bavota, Gabriele et al. [BDO11] proposal suggests the usage of graph theory in order to further improve the suggestion. A Ford-Fulkerson algorithm, followed by the extraction of a subset of methods dictated by a Max-Flow Min-Cut algorithm [CLRS09] is used on the original approach. However, the employment of graph theory algorithms is too expensive to provide real-time results and, as such, our tool utilizes a more straightforward methodology to filter which methods should be extracted to a new class. Although it is not as precise as the original approach, this rough approach should still be capable of producing acceptable results.

By using the created weight matrix between all methods in a class, we iterate them row by row, checking the values in the matrix against two thresholds: a lower one, whose purpose is to filter out spurious methods, such as getters or setters and an upper one, used to filter out metrics who are not similar enough to others and should then be extracted to a new class. The values chosen for these thresholds are **0.15** for the lower threshold and **0.30** for the upper threshold. This means that methods whose value on the weight matrix are in the interval]0.15; 0.30] should be considered for extraction and methods in the interval]0.30; 1.00] are methods which should not be considered for extraction.

Algorithm 2 Original class being updated to reflect changes

```

1: procedure UPDATEORIGINALCLASS(targetMethods, originalClass)
2:   for each method  $\in$  targetMethods do
3:     originalClass  $\leftarrow$  removeMethod(originalClass, method)
4:     methodCall  $\leftarrow$  "this.".concat(method.name)
5:     updatedMethodCall  $\leftarrow$  "this.newClass.".concat(method.name)
6:     originalClass  $\leftarrow$  originalClass.replace(methodCall, updatedMethodCall)
7:   end for
8:   return originalClass
9: end procedure

```

This refactoring is where our choice of using file-by-file analysis shows its biggest limitation: Extract Class is not supported for classes which implement external interfaces or extend other classes. In order to support it, a full workspace analysis of all classes and interfaces would have to be performed constantly, as to know which methods can and cannot be moved due to inheritance. We believe this option scales poorly with the size of the system and, as such, would not fit towards our goal to provide live analysis to the user.

5.3.3 Extract Variable

The last supported automated refactoring is the Extract Variable, where we take a non-void call expression and extract it to a new variable to be used afterwards on other calls to the same object. The main benefit for using this type of refactoring is to ease code readability at the expense of the creation of a local variable.

One of the first challenges we encountered when designing our approach was how could we distinguish a void call expression, which cannot be extracted to an object, from a non-void call expression, which is what can be extracted. Our choice to use file-by-file analysis does not grant us access to external methods invoked and respective return values.

TypeScript's compiler API provides us with the *createProgram()* method, which allows to compile a TypeScript program from specified files. Once all dependencies for said files are compiled, we can use the provided type checker to filter all call expressions present on that file against their type flags. Type flags provide a vast array of information about the specific node we are visiting, most importantly in the case of call expressions, whether that call returns *void* or not. Algorithm 3 roughly exemplifies how this type checking is performed.

Algorithm 3 Checking return type of found call expressions

```

1: procedure GETNONVOIDCALLS(callExpressions, program)
2:   nonVoidCalls : CallExpression[]  $\leftarrow$  []
3:   for each node  $\in$  program.sourceFile do
4:     for each call  $\in$  callExpressions do
5:       if isCallExpression(node) & (CallExpression)node = call then
6:         nodeType  $\leftarrow$  program.checker.getType(node)
7:         if nodeType.flags  $\neq$  VOID_FLAG_GLOBAL then
8:           nonVoidCalls.push(call)
9:         end if
10:      end if
11:    end for
12:  end for
13:  return nonVoidCalls
14: end procedure

```

With the call expressions filtered out, the tool should avoid to suggest trivial candidates, such as one which are already simple enough to read for the programmer. To solve this, our tool filters out call expressions which do not exceed a certain length. For example, if a call expression's length is 10 characters and the threshold is set at 15 characters, said call does not show up as a suggestion anymore.

As a final step towards the final suggestion, the filtered candidates are sorted in descending order, according to the length of the call expression. The lack of a set of metrics that allow an objective evaluation of this kind of refactoring is the main reason on why we take this approach, which could be further improved with frequency checking of each call as a possible way to remove duplicate code that may exist inside a method.

Similarly to our Extract Method execution, our tool directly uses the built-in command for variable extraction in Visual Studio Code. As such, we ensure its behavior is familiar to users of the original command and, should a new patch occur, our command stays updated.

5.4 Live Metrics

The second main module of our tool, live metrics, has two main responsibilities: measuring the necessary metrics to evaluate candidates and measure general metrics that can prove useful for programmers during their development duties.

In order to evaluate candidates, this module employs algorithms based on an Extract Class evaluation approach by **Bavota, Gabriele et al.** [BDO11] and an Extract Method evaluation approach by **Meananeatra, Panita et al.** [MRA11].

5.4.1 Extract Class Metrics

As briefly referenced in subsection 5.3.2, the developed tool evaluates Extract Class candidates based on previous work done by **Bavota, Gabriele et al.** [BDO11], which uses matrices to compute metrics related to structural and semantic attributes of the class. It uses the following metrics:

1. **Structural Similarity between Methods (SSM)** [GS06]: Ratio between the number of used instance variables shared by two given methods, m_i and m_j , and the total number of used instance variables on the same given methods;

$$SSM(m_i, m_j) = \begin{cases} \frac{|I_i \cap I_j|}{|I_i \cup I_j|} & \text{if } |I_i \cup I_j| \neq 0; \\ 0 & \text{otherwise.} \end{cases} \quad (5.1)$$

2. **Call-based Dependence between Methods (CDM)** [BDO11]: Ratio between the number of calls done by method m_i to m_j and the total amount of incoming calls to m_j throughout the class. It is a commutative metric because the direction of this connection is not relevant;

$$CDM(m_i, m_j) = CDM(m_j, m_i) = \max \{ CDM_{i \rightarrow j}, CDM_{j \rightarrow i} \} \quad (5.2)$$

$$CDM_{i \rightarrow j} = \begin{cases} \frac{calls(m_i, m_j)}{calls_{in}(m_j)} & \text{if } calls_{in} \neq 0; \\ 0 & \text{otherwise.} \end{cases} \quad (5.3)$$

3. **Conceptual Similarity between Methods (CSM)** [MP05]: Cosine of the angle composed between two given methods, m_i and m_j . For any method in a class, a dictionary is created, based on used vocabulary in variable, method call and parameter identifiers, representing a

pool of concepts used by the method. Then, a Latent Semantic Indexing algorithm [BYRN99, DDF⁺90] is used to create a vector based on a method's used vocabulary, which is used to compute CSM.

$$CSM(m_i, m_j) = \frac{\vec{m}_i \cdot \vec{m}_j}{\|\vec{m}_i\| \cdot \|\vec{m}_j\|} \quad (5.4)$$

5.4.2 Extract Method Metrics

The usage of Extract Method refactoring is tightly coupled with the removal of the Long Method code smell [Fow99]. As such, the selection of refactoring candidates to solve Long Methods can, to some extent, be used as well to select Extract Method candidates.

Given this, the developed tool supports and evaluates Extract Method candidates based on an approach by Meananeatra, Panita et al. [MRA11] to select the most suited candidates to remove the Long Method code smell. This approach uses the values of the following metrics:

1. **Complexity of Method (MCX)** [McC76]: Often calculated with the usage of a control flow graph, refers to the number of linearly independent circuits a program can take [McC76]. Other calculations take into account the number of predicates without needing the control flow graph. Lower values of this metric indicate lower complexity and, therefore, higher maintainability;
2. **Lines of Code in Method (LOC)**: Refers to the number of statements contained within a method's code. Lower values indicate lower complexity and greater readability;
3. **Lack of Cohesion of Method (LCOM)** [CK91, CK94, HSCG96]: Takes into account the amount of used instance variables and the class methods to calculate the class' cohesion. Higher values suggest a lower class cohesion and the need for action. Equation 5.5 shows how this calculation is done in our tool [HSCG96].

$$LCOM = \frac{1}{\frac{a}{v} \cdot a - m} \quad (5.5)$$

where:

m = number of methods in class

a = number of methods in a class which access an instance variable

v = number of instance variables

Once the necessary metrics are calculated, **Extract Method candidates are sorted based on which metric we want to optimize and prioritize**. By default, our tool prioritizes minimizing the highest number of statements in class methods, followed by minimizing the highest method complexity and lastly minimizing the class' lack of cohesion of methods.

5.4.3 Interface Metrics

Besides the metrics used to evaluate supported refactorings, our tool provides an extensive suite of metrics for the developer to consult whilst programming. These metrics are mostly based from Fernandes' work [Cou19], as we believe it presents a solid suite of useful metrics for developers be aware of. Metrics shown to the user can be divided into four categories:

1. **Line-related Metrics:** metrics which are connected to the kind of lines present on the file. Includes metrics such as number of total lines, number of comment lines, number of code lines and number of blank lines;
2. **Node-related Metrics:** metrics referring to the amount of nodes and its respective kind, present in a file. Extensively uses TypeScript's compiler API⁴ to analyse the constructed AST. Supported metrics include number of methods, number of classes and cyclomatic complexity;
3. **Halstead Metrics:** metrics directly inferred from Halstead's work [Hal77]. Supported metrics on this category include program length, vocabulary, volume, difficulty to write/understand, effort and time required to program and the number of bugs delivered. Halstead's volume is later used to compute the maintainability index [OHA92, CALO94], with Equation 5.6 being used in our tool[Mic].

$$MI = (171 - 5.2 \cdot \log(V) - 0.23 \cdot (CC) - 16.2 \cdot \log(LOC)) \cdot \frac{100}{171} \quad (5.6)$$

where:

V = Halstead Volume
 CC = Method's Cyclomatic Complexity
 LOC = Method's Lines of Code

4. **Method Metrics:** metrics which are also used by our tool to evaluate the Extract Method refactoring, based on work presented by Meananeatra [MRA11] on the subject. Includes number of statements, cyclomatic complexity and lack of cohesion of methods.

As to allow quick visualization and simplify the interpretation of the values of these metrics, our tool checks the metric values and compares most of them against predetermined thresholds, coloring the respective table cell with either green, yellow, orange or red, according to the severity of the calculated value. Table 5.1 shows the metrics which are being checked against a threshold, references consulted to determine threshold values (if applicable) and the threshold value.

Selecting threshold values for our tool is not a trivial task, as different programming languages might have different paradigms of programming, each developer has its own

⁴<https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API>, Last accessed on July 22, 2020

programming patterns and each project fits into different contexts. As such, pinpointing threshold values which fit any JavaScript or TypeScript project is prone to extensive experimentation. One solution could be providing full customization of these values for the user, however this can become counter-productive once the developer decides to manually tailor the threshold values to unfeasible values, which remove the need for the developer to worry about them at all.

Table 5.1: Metrics and respective thresholds.

Metric	Reference	Threshold value
Number of Lines	[McC]	1000
Number of Comment Lines	[McC]	1:4:1 ratio between comment, code and blank lines, respectively
Number of Code Lines	[McC]	1:4:1 ratio between comment, code and blank lines, respectively
Number of Blank Lines	[McC]	1:4:1 ratio between comment, code and blank lines, respectively
Halstead Length	[McC]	300
Halstead Volume	[McC]	1500
Halstead Difficulty	[McC]	30
Halstead Effort	[McC]	45000
Halstead Level	[McC]	0.6
Halstead Time	[McC]	2100
Halstead Bugs Delivered	[McC]	0.6
Maintainability Index	[Mic]	0-9: low 10-19: moderate 20-100: good
Method's Complexity	[HGW11]	5
Method's Number of Statements	[HGW11]	12
Lack of Cohesion of Methods	[HSCG96]	0.5

Figure 5.2 shows the compact view of our user interface. The first section shows some metrics regarding the contents of the source file, with cells containing thresholds to check against being colored according to their value. The *Method Metrics* and *Function Metrics* sections of the interface are collapsible and show metrics relevant for method and function evaluation (see Figure 5.3 for an expanded view of a method metric), with colored squares serving as preview for each metric status. Our idea behind this concept was to provide a way for the user to quickly evaluate both methods and functions, without the need to extend the interface, similarly to how unit testing frameworks work. Note that, for the file shown, only one method contains values which should be monitored by the developer.

Once changes are done to the source code, they are also reflected on the metric values. In order for users to consult how metrics looked like before changes were done, our tool provides

5.5 Visual Studio Code Extension

The developed tool bundles the modules presented in Sections 5.3 and 5.4 and allows for user interaction with them through custom executable commands, a simple webview-based interface and options customization.

While one of our first ideas was to automatically execute an entire sequence of refactoring operations, some limitations within the Visual Studio Code's API do not allow for such functionalities. These limitations were mainly found around the lack of customization for virtual documents⁵, where most of this sequencing and metric optimization would have occurred, and the lack of automated, fail-proof, execution of a chain of commands.

5.5.1 Commands

Command creation is possible through the usage of Visual Studio Code's API and the tool's interface is created using Visual Studio Code's Webview API⁶. This extension supports the execution of four commands, shown in Figure 5.4:

1. **Live Refactoring: Extract Method:** Once executed, this command automatically performs the best evaluated Extract Method candidate for the current selected file, based on the metrics results discussed in Subsection 5.3.1;
2. **Live Refactoring: Extract Class:** Once executed, this command automatically performs the best evaluated Extract Class candidate for the current selected file, based on the metrics results discussed in Subsection 5.3.2;
3. **Live Refactoring: Extract Variable:** Once executed, this command automatically performs the best evaluated Extract Variable candidate for the current selected file, based on the metrics results discussed in Subsection 5.3.3;
4. **Live Refactoring: View User Interface:** A new interface column is created and a new tab shows a webview, allowing the user to consult metrics information and our top suggestions for each of our supported refactorings and their relative impact on some metrics. Shown metrics are described in more detail in Section 5.4.

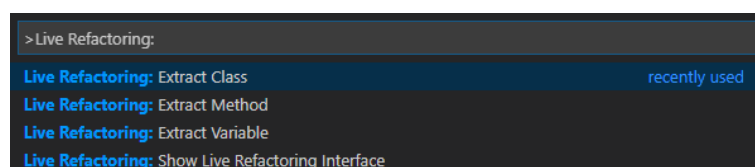


Figure 5.4: Supported commands on the Visual Studio Code's *Command Palette*.

⁵<https://code.visualstudio.com/api/extension-guides/virtual-documents>, Last accessed on July 22, 2020

⁶<https://code.visualstudio.com/api/extension-guides/webview>, Last accessed on July 22, 2020

5.5.2 Settings

Despite most of this tool operating automatically, it allows for some user customization which impacts how some candidate evaluation is performed and provides a way for users to tailor the tool as best as possible to their respective context. Possible customization includes:

- **Minimum number of extracted methods:** Users can customize the minimum number of methods an Extract Class candidate can contain. Both minimum and default values for this setting is **two methods**, with no maximum value, as it depends on the number of methods a class contains;
- **Maximum original method percentage:** Users can customize the maximum percentage of statements an Extract Method candidate can extract from its original method. This number is bound within 0% and 100% of extraction of the original method, with values close to both ends representing redundant values which often provide pointless results. Default value is **80 %**;
- **Minimum number of statements:** Users can customize the minimum amount of statements an Extract Method candidate should extract to a new method. Minimum value is one statement, with the default being **three statements**;
- **Minimum call expression length:** Users can customize the minimum call expression length for an Extract Variable candidate to be considered. Minimum value is a length of one character, with the default value being **twelve characters**.

More options regarding interface customization are also provided to the user, allowing for line-related, node-related and Halstead [Hal77, Cou19] metrics to be filtered in or out of the shown interface.

5.5.3 Events

Visual Studio Code's API allows for developers to subscribe certain events happening in the editor, as to trigger code execution once said events occur. Our tool uses this API feature to simulate continuous execution in order to achieve liveness. However, choosing which events to subscribe is crucial to ensure that a live-like experience is achieved, while not overwhelming the user with unnecessary updates and a constant barrage of incoming information.

Given this, two of the events that provide the best live results are the *onDidChangeTextDocument* and *onDidChangeTextEditorSelection* events, which trigger every time a document is changed and the current selection is changed, respectively. Despite this high level of liveness, **these events can be unnecessarily intrusive** on the developer's train of thought and, therefore, **counter-productive**.

To tackle this issue, our tool subscribes events which, although not as live as the ones previously stated, are still considered within liveness bounds while not being as intrusive as said events.

Subscribed events of our tool are *onWillSaveTextDocument* and *onDidChangeActiveTextEditor*, which trigger when the current document is saved or when the active text editor changes, which occurs when, for example, the user opens a document on the editor, respectively.

We believe these events correspond to the point in time when the user is done with a task and is either moving onto a new task or ready to iterate on the recently finished task, with the right amount of intrusion for the user to not be overwhelmed and the workflow to not be interrupted.

5.6 Summary

This chapter describes our approach to tackle the problems we found with current tools, outlined in the previous chapter. The proposed solution comes in the form of a Visual Studio Code extension that provides developers with semi-refactoring execution of extraction-based refactorings, such as Extract Method, Extract Class and Extract Variable.

Further on this chapter, we detail the techniques used to execute said refactorings, both the quality metrics used to sort the refactoring opportunities and the general metrics provided to the user via a webview interface, followed by explaining the usage of Visual Studio Code's API to enhance our extension.

Chapter 6

Empirical Validation

6.1	Methodology	53
6.2	Results	54
6.3	Threats to Validity	73

In order to validate our developed tool and answer the proposed research questions, we adopted the following empirical methods: survey research and automated analysis. Section [6.1](#) describes our approach in using each of said methods, with the respective results being discussed in Section [6.2](#). Section [6.3](#) discusses the main threats to our validation.

6.1 Methodology

This section provides more detail about the used methodologies to validate our hypothesis and answer the proposed research questions. Section [6.1.1](#) describes the conducted survey, its main objectives and how data retrieved from it helps us answering the research questions. Section [6.1.2](#) describes how we evaluated our tool’s results when used on files from open-source projects, with varying degrees of complexity.

6.1.1 Survey

To understand how capable our tool is at providing useful and quick information to programmers via its interface, we ran a survey, available in [Appendix A](#), focused on studying our tool’s usability and feature set. The main goal for this survey is to understand the impact it has on a developer’s workflow, this is, if our tool can trigger developer action onto the code, once the user understands the extent of technical debt present on the file. Another goal for this survey is to understand if the feature set implemented proves useful and accessible to the user.

Most of the questions present on this survey are answered resorting to the Likert psychometric scale [Lik32], with the exceptions to this rule being questions relative to participant profiling and technical background. The scale used ranges from 1 to 5, depending on how the user agrees with a given statement, with 1 being labeled 'Strongly disagree' and 5 labeled 'Strongly agree'.

Results obtained from this survey should grant us a better understanding on whether our developed tool achieves its purpose of providing developers with a metrics visualization systems and if the features we provide to tackle any technical debt encountered have the quality to do so. Interpretation of gathered data should help us answer **RQ1** and **RQ3**.

6.1.2 Automated Analysis

The last of our methodologies is an experimental approach, which consists on an automated usage of our tool on open-source repositories. This allows for experimentation and evaluation of our tool's performance in real projects, with varying degrees of complexity, while gathering a large amount of numerical data we can use to answer our research questions and further support our validation.

For this, a custom repository crawler was developed. This crawler reads from a JSON object describing the repositories to be analysed, using information such as the repository's URL, versions of the repository to be analysed (as we aim to also examine different versions of the same file) and the files of interest (files on which the tool will be tested) spread through the specified repository versions.

Our goal when adopting this approach of validation is to understand whether our tool can provide the developer with meaningful refactorings which take into account the values of a set of predefined metrics (see Section 5.4), and improves them after each iteration of refactoring. Our choice of analysing different versions of the same file, across different repository commits, ties into our need to understand whether developers using our tool are able to detect the presence of possible problems lying in the code and how our tool's suggestions differ across multiple versions of a file.

Data resultant from the analysis of our tool's impact in the quality metrics of selected files is of great importance, as conclusions taken are essential for us to answer **RQ2** and **RQ3**.

6.2 Results

This section describes in greater detail the results obtained in all the validation methodologies used, which were described in Section 6.1: survey analysis and automated analysis.

Section 6.2.1 gives more details about how the survey was conducted, the number of participants in our study, the questions performed and the results obtained, with Section 6.2.2 presenting the numerical data retrieved from the usage of our tool in open-source repositories.

6.2.1 Survey

As explained in Section 6.1.1, one of our means to validate our hypothesis and answer the proposed research questions was to conduct a survey on the usability, user interaction and the importance of our tool's features. This survey was answered by a group of 31 participants. Sections 6.2.1.1 through 6.2.1.6 detail the questions done to the participants and discuss their respective answers.

6.2.1.1 Participant's Profile

In order to profile our participants, the introductory section gathers information about their age and education level. Figure 6.1 shows how both the age and education levels are spread among our survey's participants.

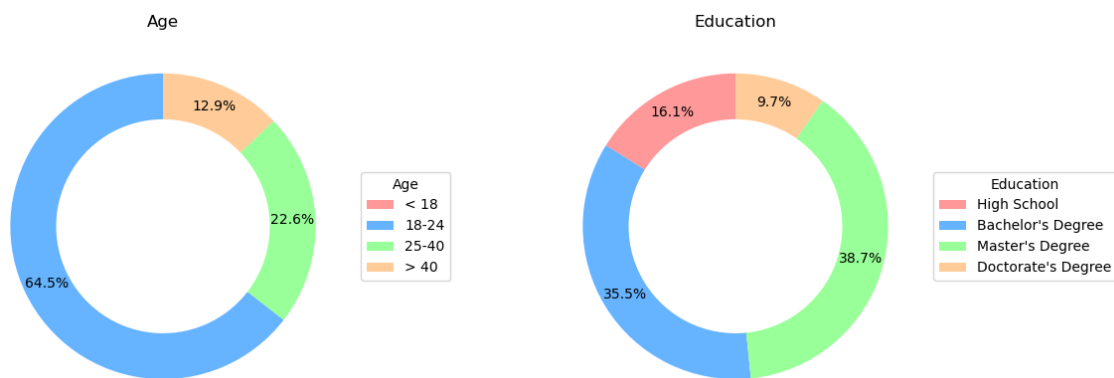


Figure 6.1: Survey participant's age (left) and education level (right).

As it can be observed, our survey reached a highly diverse group of respondents, with the most common answer being in the age bracket of 18 to 24 years old with a bachelor's degree.

6.2.1.2 Technical Background

To further contextualize our participants in our survey, we also gathered information about their technological background. For this, questions asked revolved around their programming experience (in years), familiarity with certain software engineering and programming terms, experience in both JavaScript and TypeScript and previous usage of Visual Studio Code, refactoring recommendation systems and software metric analysis tools. Questions asked in this section are identified as TB1 through TB7:

TB1: I have experience in programming for...;

TB2: I am familiar with programming terms, such as classes, methods and data structures;

TB3: I am familiar with software development terms, such as refactoring, debugging and unit testing;

TB4: I am experienced in JavaScript or TypeScript;

TB5: I have used Visual Studio Code before;

TB6: I have used refactoring tools or extensions before;

TB7: I have used software metric analysis tools before.

Figures 6.2 and 6.3 show graphical representations of the answers for this section of the survey. Participants of this survey are mostly familiar with most programming and development terms that the survey may refer to, with all respondents having some degree of programming experience. This suggests a software engineering experience, which indicates that all participants fit within the target demographic for this survey.

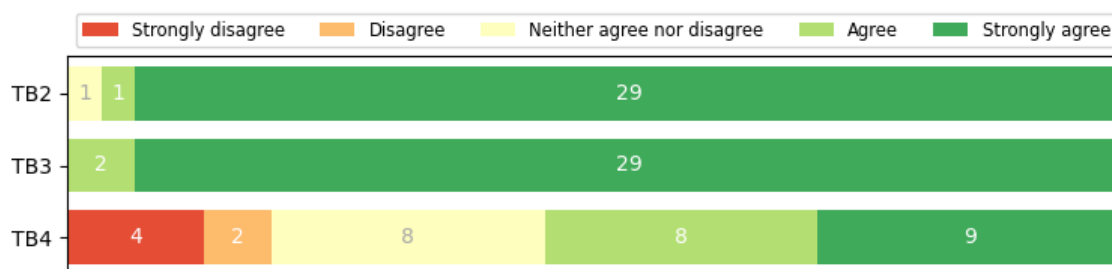


Figure 6.2: Survey answers to questions TB2, TB3 and TB4.

On the topic of relevant technologies and tools used in the developed tool, all participants have experience with using Visual Studio Code, while most of our participants already had any experience with both refactoring and metric analysis tools. On the topic of relevant programming languages for our tool, more than half the participants (17 out of 31) consider themselves experienced with JavaScript or TypeScript.

6.2.1.3 Interface

On this section of the survey, we present the participant with pictures of our interface report and how it changes when the programmer alters the source code, in which we aim to understand the first impressions a user might have when using the tool. This section is divided into questions about the general interface, post-changes interface and Visual Studio Code-related interface. Questions asked in this section are identified as I1 through I10 for questions relative to the general interface:

I1: Most of the metric values on this file are within healthy limits;

I2: The color scheme used allows me to quickly identify problematic methods without spending time looking through them individually;

I3: Despite the long list of metrics, I am able to process the information fairly quickly;

I4: The first section of the interface, file-related metrics, is emphasized enough and its information is of easy interpretation;

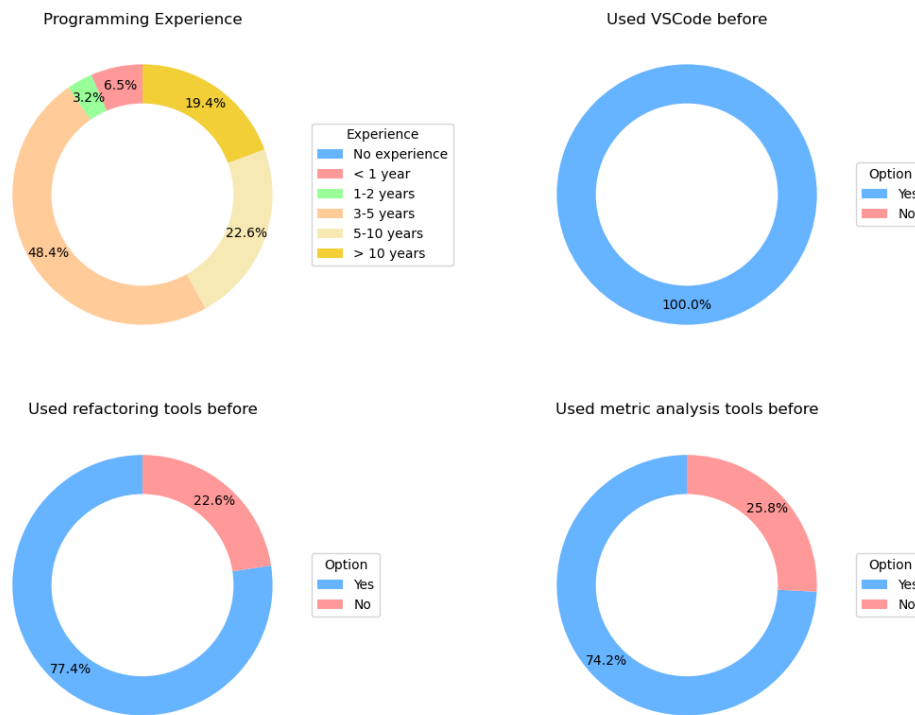


Figure 6.3: Survey answers to questions TB1 (top-left), TB5 (top-right), TB6 (bottom-left) and TB7 (bottom-right).

- I5:** The second section of the interface, method-related metrics, is emphasized enough and its information is of easy interpretation;
- I6:** The third section of the interface, refactoring suggestions, is emphasized enough and its information is of easy interpretation;
- I7:** The amount of information presented is overwhelming;
- I8:** The information given is enough for me to know if any action is needed on the code;
- I9:** The preview colored squares on the 'Method Metrics' section allows for even faster analysis of methods;
- I10:** This report format of presentation fits the purpose of this tool.

Figure 6.4 shows the distribution of answers relative to the interface questions, I1 through I10. The results obtained suggest that our approach to the visual part of our system is mostly well perceived by our target end-user, as answers to questions I1-I3 and I8-I10 indicate that our tool's interface is capable of transmitting information in a simple to read format. Answers to questions I1, I3 and I8 in particular, suggest that our tool provides a way for developers to quickly evaluate the current state of the metrics on the file being worked on and whether any action should be taken or not.

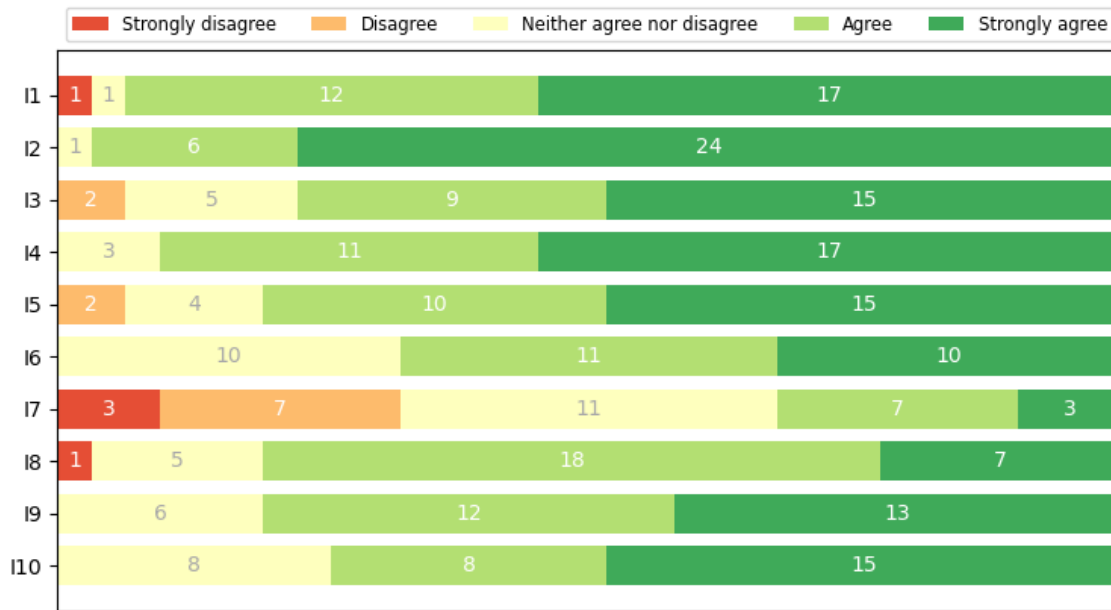


Figure 6.4: Survey answers to questions regarding the tool's interface.

However, based on answers to questions I4-I7, the current interface could use some improvements on the way information is presented to the user, as some participants believe that two of the report sections could be presented better and a significant portion of the respondents agreed that the amount of information presented on the report is overwhelming, which could lead some developers to spend more time than intended to interpreting the information displayed.

To test a possible existence of correlation between the agility of assessment of the metrics and the awareness of requiring action. As such, we calculated the mean results for questions I2, I3 and I9 to figure the evaluation of the participants to the agility of assessment our tool provides and the answers to question I8 which consider if the participant is aware that action is needed on the code. For this, we ran a correlation test between the mean results of I2, I3 and I9 against the results of I8, using **Kendall's tau-b** correlation tests [Ken38, Ken45], with the null hypothesis being *there is independence between the agility of assessment and the awareness of action*. The resultant *p-value* of approximately 0.015, below the used significance value of 0.05 makes us reject the null hypothesis.

Following the general interface questions, the next subsection covered how the post-changes interface is perceived by the user, with questions PCI1 and PCI2:

PCI1: I can clearly understand the values on the left side correspond to old values, while the ones on the right correspond to the newest ones;

PCI2: Only showing the two more recent values of each metric for a programming session is enough.

Figure 6.5 shows the distributions of answers to questions PCI1 and PCI2, shown to the user once a post-changes interface screen is presented. Answers to question PCI1 show that the way

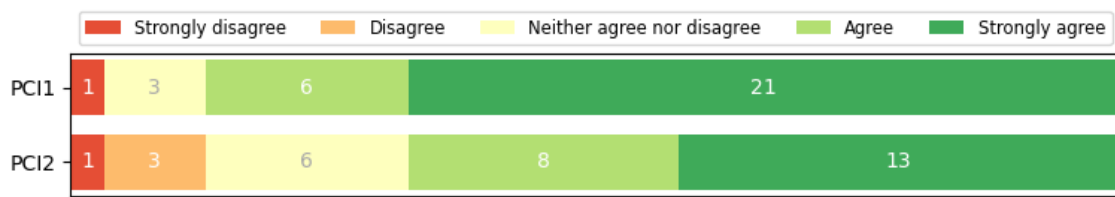


Figure 6.5: Survey answers to questions regarding the tool's post-changes interface.

our interface shows changes in metrics is clear to the user (27 of 31 participants agreed), but PCI2 shows that only comparing the two most recent values could not be enough for some users.

After the post-changes questions, the participant is exposed to the Visual Studio Code's interface, where common contribution points for new interface elements are shown and how our tool fits into this interface. Figure 6.6 shows the answers obtained for questions VSCI1, VSCI2 and VSCI3:

VSCI1: The webview interface is distracting;

VSCI2: I would prefer for the information given by the tool to be shown in other contribution points, eliminating the need for a webview;

VSCI3: Extensions whose features are scattered throughout the VSCode UI can be confusing to use.

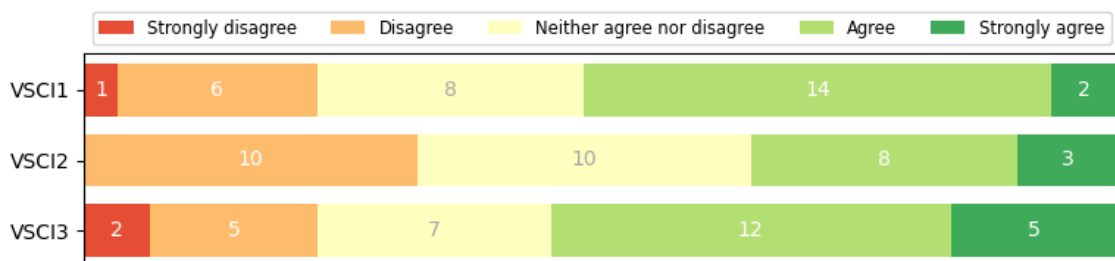


Figure 6.6: Survey answers to questions regarding the tool's implementation on the Visual Studio Code's UI.

As observed on the results obtained in Figure 6.6, most participants believe that the webview format we used for our interface is distracting for the user. However, most respondents believe that extensions with features present in multiple contribution points of the Visual Studio Code UI can be confusing to use and a considerable amount of answers to VSCI2 disagree with the information given by the tool to be spread through the contribution points.

A possible interpretation of the inconsistency seen in the answers for these questions is that our webview approach is a possible correct approach for the interface, due to its capability of keeping all required information in a single section of the interface, without resorting to fragment our interface into smaller parts and insert them into the embedded Visual Studio Code UI, which

could pose a problem, considering the answers to VSCI3. However, as suggested by questions VSCI1 and VSCI2, it should be improved, as to not be as much of a distraction as the current version is.

6.2.1.4 Workflow

Once the respondent is contextualized about how our tool fits in the programming environment, a video of the tool's usage and workflow is shown. The point of study in this section is to understand how clear are the instructions given by the tool, if the processing time for all the metrics and refactoring suggestions is acceptable and whether the webview still proves a distraction for the user, after watching the tool being used. Questions in this section are identified as WF1 through WF5:

WF1: The workflow to execute the top Extract Method suggestion looked simple;

WF2: Instructions presented on the interface on how to execute a refactoring are clear;

WF3: The processing time of post-changes metrics and refactoring suggestions (around five seconds for this file) is too high;

WF4: The webview is distracting for the programmer;

WF5: I would prefer a button in the interface to run the refactoring, instead of manually executing a command.

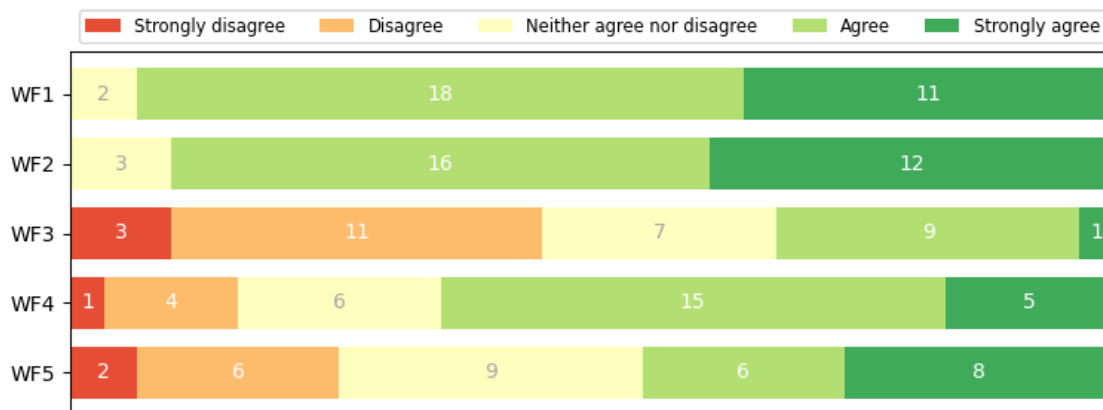


Figure 6.7: Survey answers to questions regarding the tool's workflow.

Results shown in Figure 6.7 indicate that the usage of our tool is simple and clear. However, the webview interface is still seen as a distraction, which suggests that, despite the tool providing useful information (see Section 6.2.1.3), the format in which it is presented could be improved.

Question WF3, in particular, shows a wide spread of responses, as a significant portion of the answers suggest that our tool takes too long for analyse supported metrics and evaluate all possible

refactorings, which is not ideal. This distribution of answers could be tied to multiple factors, including tool inefficiency by our part, lack of direct comparison between our system and existent ones with similar purposes or unfamiliarity of the computational cost of the used algorithms by the respondents.

6.2.1.5 Refactoring

Another main functionality of our tool is the refactoring module, responsible for semi-automatically execute refactoring operations on the source code. Respondents are questioned about whether our choice of supported refactorings is correct and how programmers can benefit from a tool which analyses in real-time and executes the best found refactoring.

RF1: All three supported refactorings (Extract Method, Extract Class and Extract Variable) are refactorings I use regularly;

RF2: A tool which automatically finds the best refactoring of each kind, for any file and in close to real time, is something I see value in;

RF3: Having semi-automated execution of the best found refactorings is useful, i.e. the user triggers the execution, but execution is performed automatically;

RF4: Having a preview of what the refactoring is going to change before executing is useful.

Figure 6.8 shows how the refactoring module of our tool is perceived by our respondents, with a vast majority recognizing the importance of optimal refactoring finding, semi-automated execution and refactoring preview. However, our selection of supported operations is divisive, as answers to question RF1 are distributed almost evenly through the *Agree* and *Disagree* options.

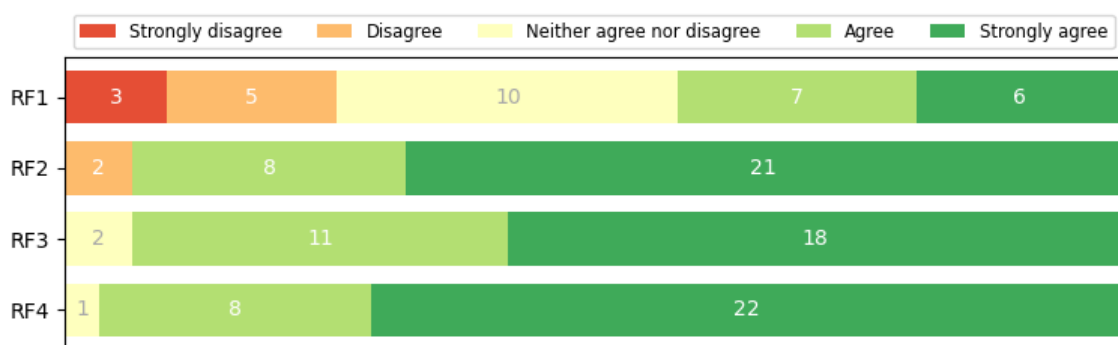


Figure 6.8: Survey answers to questions regarding the tool's refactoring capabilities.

Apart from semi-automated refactoring execution, our tool also allows for users to customize the algorithms of finding and evaluating refactorings, as we believe that every project has its own context and no one is more familiar with it than its own developers. Survey participants are questioned about the clarity of each option and how useful customization on refactoring tools is.

UC1: Offering user customization for refactoring tools is useful;

UC2: The description of each option and its impact on the suggestions are clear.

As shown by Figure 6.9, the feature of user customization on refactoring suggestion systems is considered very useful by the respondents (30 out of the 31 participants agreed), with the way such options and their purpose are presented to the user being considered clear by the vast majority of the responses.

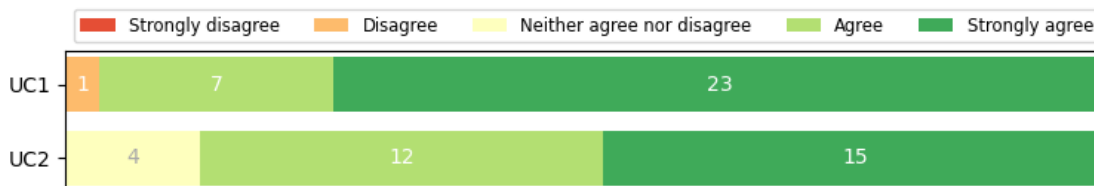


Figure 6.9: Survey answers to questions regarding the tool's user settings.

6.2.1.6 Final Remarks

The final section of the survey requires respondents to take into account all the information and features exposed throughout the survey and its purpose is to understand the usefulness of the tool as a whole, and which of the features implemented were their favorite and which were not as impressive.

FR1: This tool's features are simple and fast to understand;

FR2: This tool can positively impact my development workflow;

FR3: I would use this tool.

The final balance of our tool shows a very positive result, as seen by Figure 6.10. The majority of participants understood how the features and the tool as a whole worked, recognizing its impact on development workflow. Most participants (22 out of 31) would consider using this tool for their work, with the two answers disagreeing with this statement being participants which did not regularly use the supported refactorings (both answering *Strongly disagree* on question RF1). This could suggest that the consideration of using our tool could be tightly connected to the kind of supported refactorings.

6.2.2 Automated Analysis

The automated analysis ran our tool through twenty-two files, spread between five open-source repositories, ranging from a small independent project to large-scale open-source projects such as Microsoft's *vscode*¹, passing through TypeScript showcase projects, like *dojo*². This choice of

¹<https://github.com/Microsoft/vscode/>, Last accessed on July 22, 2020

²<https://github.com/dojo/intern-only-dojo/>, Last accessed on July 22, 2020

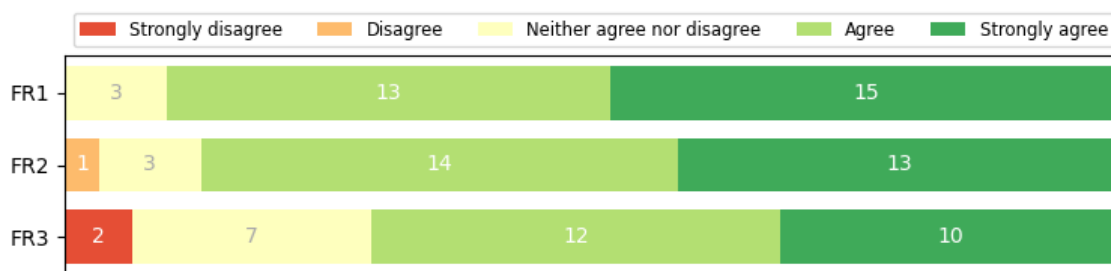


Figure 6.10: Survey answers to the final section.

projects was done as to portray how our tool can work on projects with varying scopes and fit various workloads, while we progressively gained confidence in its results during tests on larger files.

While the chosen files were not entirely randomly selected, we believe that the ones chosen grant a solid representation on how our tool operates in *real-life* situations. Our selection criteria for these files was rather simple, as we targeted files which contain some amount of logic, while being conservative about the size of the file, due to obvious limitations on live processing a code file. To further contextualize on the size of the chosen files, the median and mean number of **lines of code** present in these files being 119 and 144 lines of code, respectively.

Despite our tool providing execution of three refactoring operations, only two of them operate using the analysis of quality metrics: Extract Method and Extract Class. As such, our experiments on the selected files will be focused on analysing the metrics before and after the aforementioned refactorings. The metrics used by these refactorings are discussed in greater detail in Section 5.4.

No mixing of refactorings will be performed during these experiments, in order to test the impact a single kind of refactoring has on the original files. In other words, when evaluating our approach to the Extract Method refactoring, only Extract Methods will be executed, starting at the original code state to the final state of our experiment. This is to restrict the interference one refactoring might cause to the targeted metrics of another, for example, an Extract Method changing the LCOM value of a class.

6.2.2.1 Extract Method

Starting off with the Extract Method refactoring, our tool's objective is to try to find possible refactorings whose execution allows for metric values to be more evenly leveled among methods. By default, our system prioritizes extracting fragments with the highest amount of cyclomatic complexity, followed by the fragment's number of statements and then by its impact on the LCOM metric. As LCOM is a metric operating on an entire class cohesion, we found that Extract Method suggestions did not have much impact on drastically changing this metric, which is most likely tied to the fact it takes into account instance variables, which is something the Extract Method refactoring cannot change without external action by the developer. Given this, results shown in this section will not show changes on the LCOM metric.

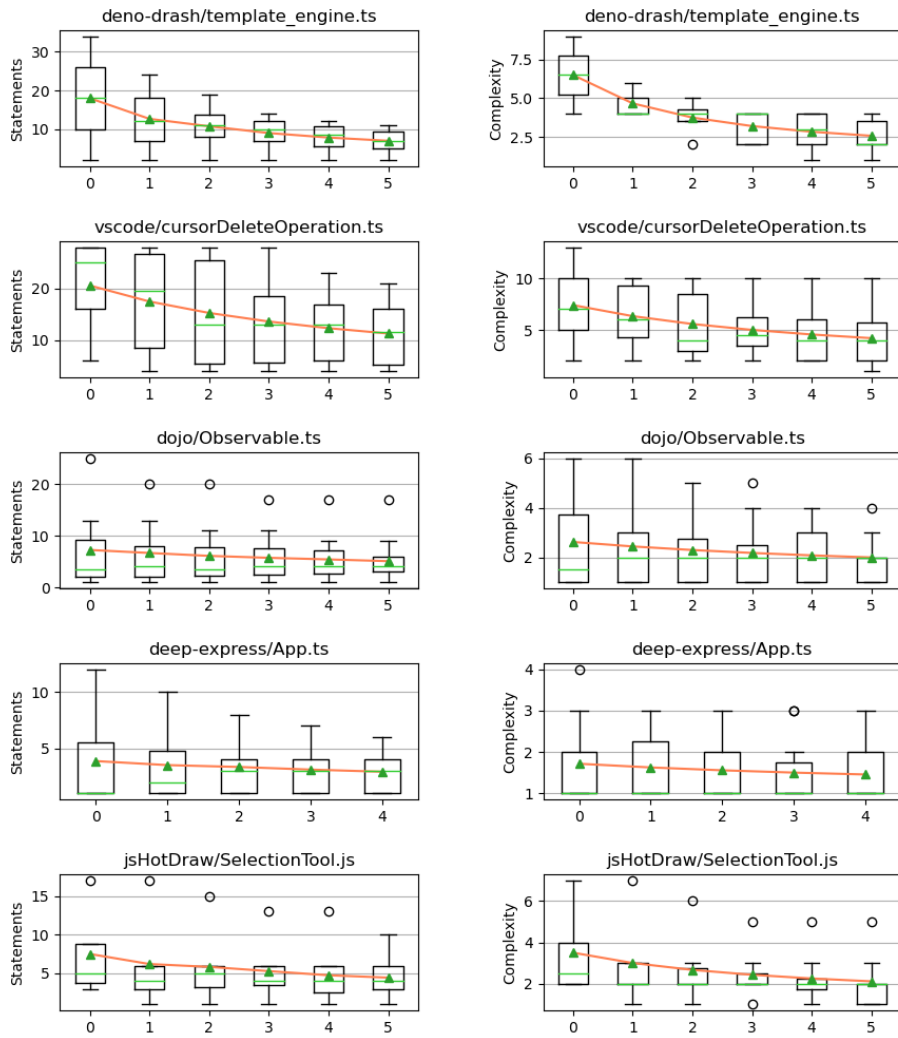


Figure 6.11: LiveRefactoring's impact on number of statements (left) and cyclomatic complexity (right) metrics.

On the 22 files analysed, our tool found 188 possible Extract Methods which passed the filtering of a minimum of three statements and a maximum of 80% of the original method's statements. We then executed the five best possible Extract Methods on every studied file, however, in files which did not have five suggestions, we executed as many as possible.

The experiment ran utilized the default user settings for our tool (see Section 5.5.2 for more details about the user settings) and prioritizes minimizing the highest number of cyclomatic complexity found in all methods and functions of the file, followed by minimizing the highest number of statements in a given method or function, with minimizing the values of LCOM being prioritized last.

Figure 6.11 shows the results our tool obtained on five of the analysed files, one per repository. The downward trend on the metrics we are trying to minimize is clear across all files, which suggests that our tool is capable of finding and extracting methods according to the defined criteria,

with the goal to minimize the number of statements and the cyclomatic complexity of methods in the file.

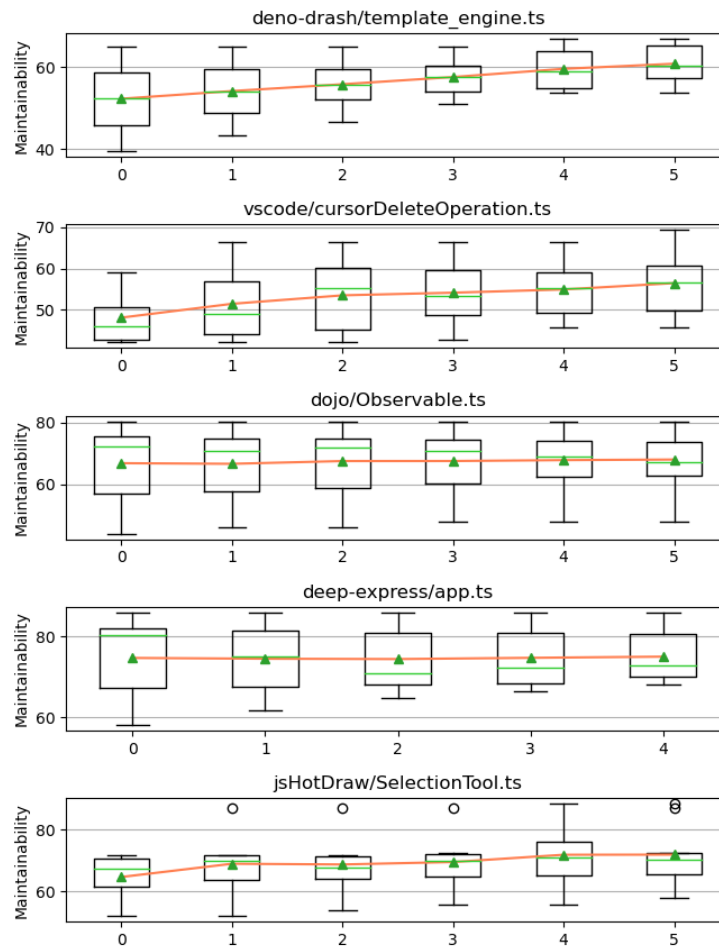


Figure 6.12: LiveRefactoring's impact on the maintainability index.

To test if the downward trend we observed on the selected metrics translates into the process of improving the overall maintainability of the source code analysed, our experiment also tracked the trend of other metrics, including the maintainability index [OHA92, CALO94, Mic], which is also used to evaluate, as the name suggests, how maintainable a certain code structure is. As such, Figure 6.12 shows how the refactoring suggestions impact the value of this metric, across the same files presented in Figure 6.11. As higher maintainability values usually indicate the presence of code with higher quality, Figure 6.12 shows that our refactorings turned the existent code into a more maintainable one across all files.

By observing both figures and the metric trends, it is possible to establish a visual correlation between a lower number of statements and cyclomatic complexity values and a higher maintainability index. However, further statistical testing must be performed.

A **paired (or matched) samples upper-tailed t-test** was conducted to evaluate whether our Extract Method approach lead towards a more maintainable code, comparing the initial

maintainability values, before action of our tool, with the final maintainability values, after action of our tool. The conducted t-test is directed at testing the hypothesis in Table 6.1.

Null Hypothesis	Alternative Hypothesis
H_0 : Extract Methods used by the tool did not improve the code's maintainability. $\mu_d \leq 0$	H_a : Extract Methods used by the tool improved the code's maintainability. $\mu_d > 0$

Table 6.1: Hypothesis Tests related to the Extract Method refactoring.

With our tool working towards improving the maintainability index metric and with the objective being comparing the values of the same sample in different points in time, we chose to run an upper-tailed paired samples t-test, where the obtained t-value should help us reject or accept the null hypothesis - the Extract Methods performed by our tool do not improve the code's maintainability.

One of the assumptions of a paired samples t-test is the normality of the sample data. To verify this assumption, we used the **Shapiro-Wilk** normality test [SW65] to test the normality of our data, with the null hypothesis meaning that *the sample comes from a normally distributed population*. Table 6.2 shows the obtained *p-value* of the Shapiro-Wilk test and the performed t-test.

Shapiro-Wilk *p-values* of 0.0726 and 0.1951 for the before and after samples, respectively, are slightly higher than the chosen significance value of 0.05, which shows that we cannot reject the null hypothesis of this test on both samples. We can also verify that the *p-value* of the t-test is significantly lower than the significance value chosen of 0.05, thus, meaning that **there is enough evidence to reject the null hypothesis H_0** .

Sample	Mean value	Std. Deviation	Sig. Value	DF	Shapiro-Wilk(p)	t-test(p)
Before	67.758	11.739	0.05	21	0.0726	0.003968
After	70.764	8.136			0.1951	

Table 6.2: T-test on the Extract Method refactoring impact on maintainability.

Given this, based on the evidence obtained from the analysis of quality metrics calculated by our tool, we are convinced that Extract Method suggestions provided by our tool result in an overall more maintainable code.

Another point of study on this refactoring is whether our tool can detect and act on similar problems in multiple versions of the same file, which could indicate that early usage of this tool could aid developers on detecting and preventing the persistence of code smells on earlier versions of a file. As such, we conducted another experiment using one of the studied repositories, *deno-drash*³ to analyse how our tool performed at detecting problems in three different files: *server.ts*, *http_service.ts* and *string_service.ts* across multiple versions. These files were selected due to

³<https://github.com/drashland/deno-drash/>, Last accessed on July 22, 2020

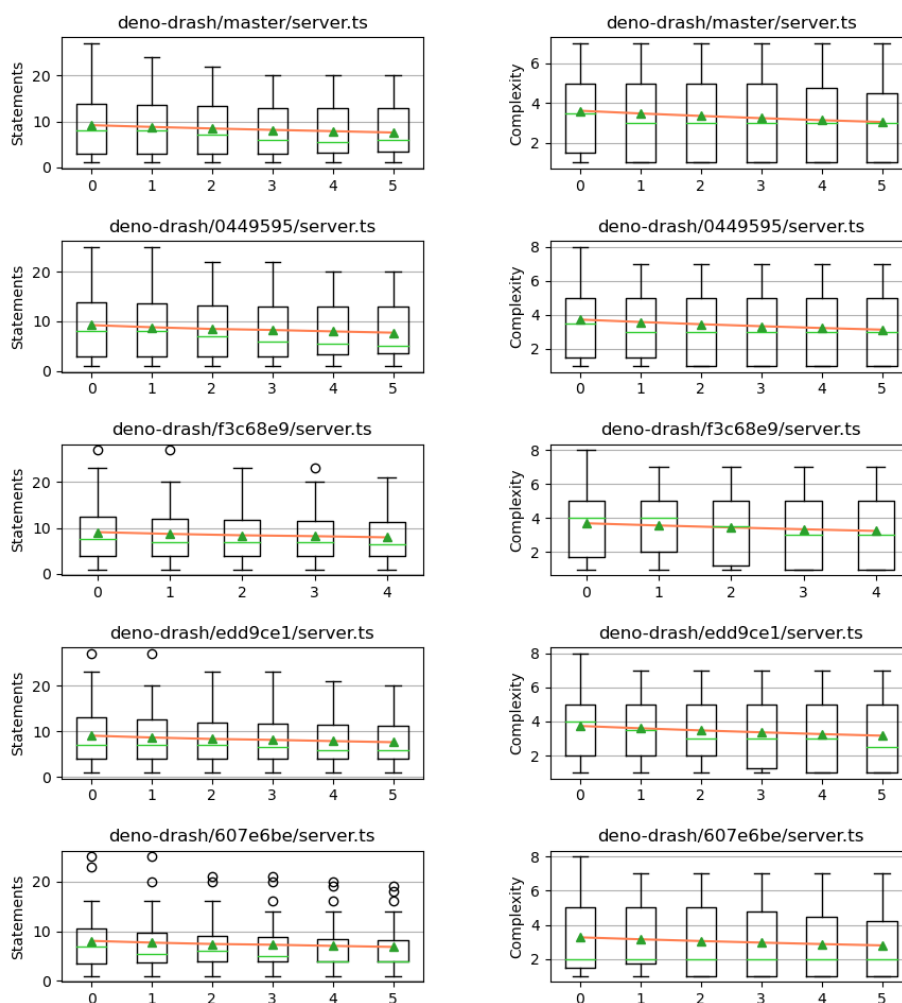


Figure 6.13: LiveRefactoring's impact on targeted metrics across five repository versions.

previous analysis showing their potential to carry problems and due to being files targeted by multiple changes across the repository's history. To find good candidate versions of the repository, we tried to select versions with multiple commits in between, as we believe it provides files with substantial difference between them.

Figure 6.13 shows the evolution of the targeted metrics (number of statements and cyclomatic complexity) on the *server.ts* file across five versions, spread between more than 50 commits done on the file across four months of development. We can observe that our tool was capable of finding similar improvements on all five versions of the file. Figure 6.14 shows the extent of how the tool impacted the maintainability index on all methods across the five selected versions of *server.ts*. As expected, the tool improved the maintainability index on all five versions, similarly to what was observed on the previous experiment (see Figure 6.12). However, these improvements are only in the single-digit margin, indicating residual improvements, which could be explained by the number of Extract Method executed - five - relative to the number of methods present in the file (18 to 20 methods across selected versions before any refactoring occurred)

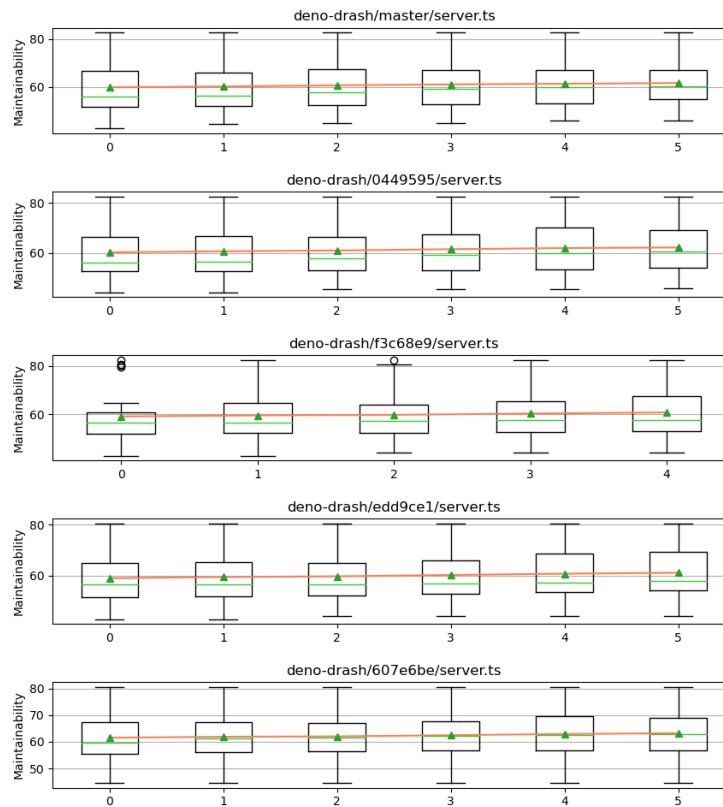


Figure 6.14: LiveRefactoring’s impact on the maintainability index across five repository versions of *server.ts*.

This may suggest that early usage of our tool could have benefits for developers and can result in healthier metrics for files across their lifetime if used early on, as results found after five Extract Methods across different versions of the studied files, show that both the targeted metrics and the metrics used to evaluate our approach trend towards better and healthier values across the board.

From the obtained results in this last experiment, we figured that one more study could be done on our tool’s approach to the Extract Method refactoring. This is, if the single-digit improvements found on our second experiment are in some way related to the number of Extract Methods performed, when compared to the number of methods a file has before the refactoring begins. Figure 6.15 shows the evolution of the previously studied metrics in *deno-drash*’s *http_service.ts* file, on commit *f9953e3* when, instead of five Extract Methods executed, we execute ten of them.

As seen in Figure 6.15, the metrics can be further improved when more Extract Methods are performed. However, we cannot execute as many Extract Methods as there exist, as dozens of small methods being extracted might actually turn the code more bloated than the one we started with, which ends up being counter-productive towards a more maintainable code base. Our user customization makes our tool very versatile to tackle this problem, as developers can specifically tailor the size of suggested Extract Methods, given their project’s context, allowing them to find a balance between the number of refactorings performed, to the benefit gained from them.

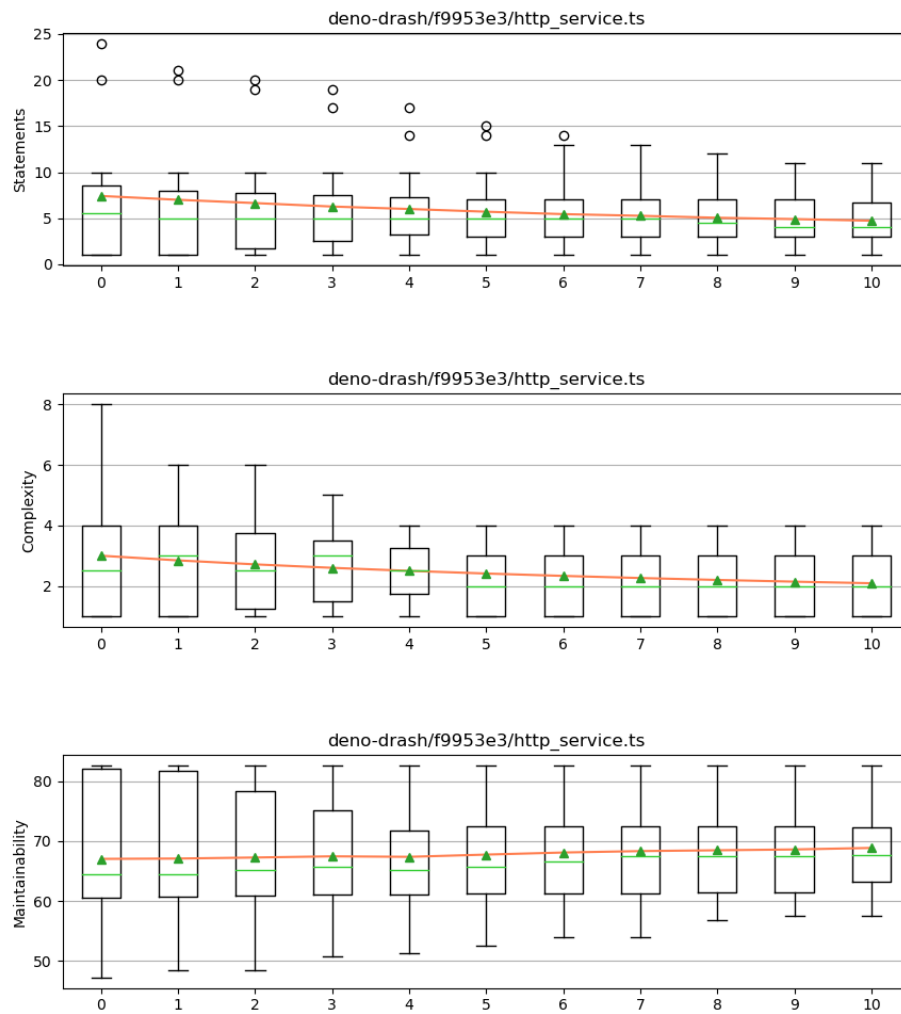


Figure 6.15: LiveRefactoring’s impact on the selected metrics across ten Extract Method refactorings.

6.2.2.2 Extract Class

Another of our tool’s objectives is to provide automated Extract Class refactorings using a modified metrics-based approach (see Section 5.3.2 for details of implementation). However, due to the need for our solution to perform in close to real time, some compromises had to be made, which could make our approach lose some fidelity, when compared to the original [BDO11].

In order to evaluate the effectiveness of our modifications and answer the related research questions regarding improvements on the source file, we studied the Extract Class refactorings our tool suggests in each of the twenty-two analysed files and its impact on the LCOM metric. Extract Class refactorings resulting in lower values of LCOM across all classes in the file should represent an improvement on the metrics, while higher values should indicate a deterioration of the LCOM metric.

Out of all the analysed files, our tool only found four files with possible Extract Class refactorings, on which we proceeded to execute the highest amount of Extract Class refactorings

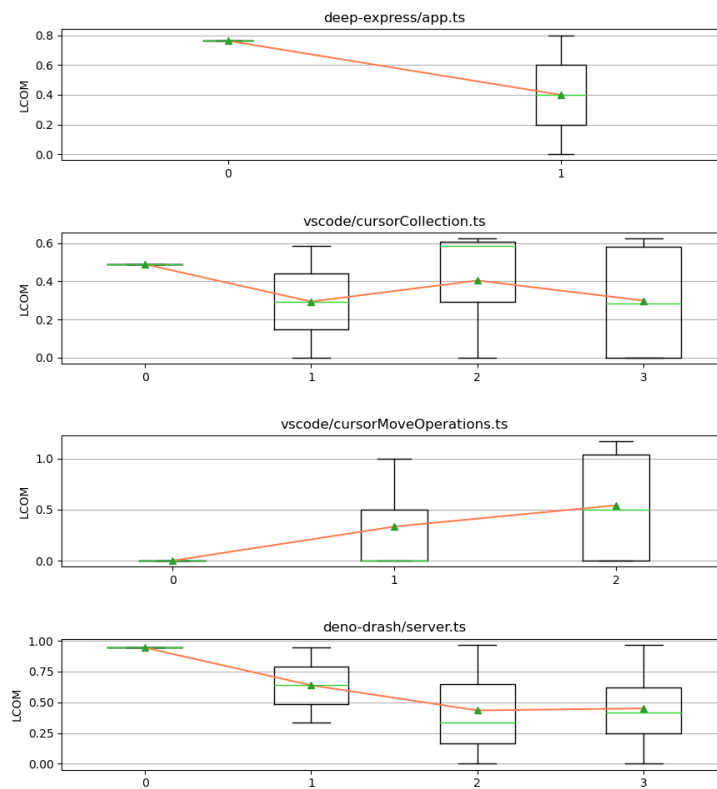


Figure 6.16: LiveRefactoring's Extract Class impact on the LCOM metric across four files.

as possible, up to a maximum of five, similarly to how the Extract Method experiment was conducted. Despite our tool filtering for Extract Class opportunities starting at two extracted methods, with the selected files containing a moderate amount of logic onto them, the amount of Extract Class suggestions was close to none.

To evaluate whether or not our Extract Class approach leads towards a reduction of the LCOM values across all classes in a file, we conducted a t-test. Similarly to the Extract Method statistical test, the performed t-test was a **paired sample one-tailed t-test**, with the difference of this one being **lower-tailed**, since we want to test the decrease of a variable and not the opposite. Table 6.3 shows the null and alternative hypothesis being tested during this t-test.

Null Hypothesis	Alternative Hypothesis
H_0 : Extract Class used by the tool do not improve the LCOM values across all classes in a file. $\mu_d \geq 0$	H_a : Extract Class used by the tool improve the LCOM values across all classes in a file. $\mu_d < 0$

Table 6.3: Hypothesis Tests related to the Extract Class refactoring.

Table 6.4 shows the results obtained with the performed t-test, where it is possible to observe that the obtained *p-value* is significantly higher than the defined significance value of 0.05, leading

to the conclusion that **there is not sufficient evidence to reject the null hypothesis H_0** . The Shapiro-Wilk normality test could not be done on such a small sample and, as such, we will assume the sample used follows a normal distribution, which is further discussed as a threat to validity.

Sample	Mean value	Std. Deviation	Sig. Value	DF	Shapiro-Wilk(p)	t-test(p)
Before	0.549	0.411	0.05	3	-	0.312
After	0.423	0.101			-	

Table 6.4: T-test on the Extract Class refactoring impact on the LCOM metric.

Figure 6.16 shows how the value of LCOM is impacted when an Extract Class is executed on each respective file. In a first look of this graph, we can compare initial and final values of LCOM for the studied files and observe that in three out of the four files, the values of LCOM trend towards more favorable ones. However, the LCOM values followed an opposite trend across all Extract Class refactorings performed on *vscode's cursorMoveOperations.ts*, which indicates a substantially decay of the LCOM metric after every refactoring performed. Upon closer inspection of the file at hand, *cursorMoveOperations.ts*⁴, the targeted class for the refactoring, *MoveOperations*, is a class without any fields, composed of only static methods. As such, any Extract Class performed on this class would inevitably increase the value of LCOM, due to the creation of a link between the new and old classes which serves no purpose on a class with only static methods.

Despite existing a strongly plausible explanation for the only unfavorable growth of LCOM values observed in the targeted files and considerable improvements on the LCOM metrics for the other, more regular files, we believe the data and evidence we gathered shows some potential in our modified approach to identify Extract Class opportunities however, it is not enough to claim the effectiveness of our Extract Class towards a more maintainable code base.

6.2.3 Discussion

This work was done with the main objective of validating the following hypothesis:

"A live refactoring recommendation system operating based on quality metrics is capable to aid developers towards a more maintainable code base."

The obtained results, explained throughout this chapter, should help us answer the proposed research questions:

RQ1 *"Does providing developers with software visualization tools improves their awareness towards code smells mitigation?"*

⁴<https://github.com/microsoft/vscode/blob/master/src/vs/editor/common/controller/cursorMoveOperations.ts>, Last accessed on July 22, 2020

The answer to **RQ1** lies within the conducted survey (see Section 6.2.1), where we asked participants about the usability of our tool and its possible impact when used during development tasks. During the *Interface* section of the survey (see Section 6.2.1.3), **93.6%** of participants were able to correctly assess the metrics of a file they were unfamiliar with, during the first seconds of contact with both our tool and the evaluated file, as seen in answers to question *I1*. Answers to questions *I3* and *I7* show us that, despite the amount of information being considered overwhelming by a significant portion (**32.3%**) of the respondents, the information could be quickly processed according to **77.4%** of the participants. Question *I2* suggests that the used color scheme was key for users to assess the information rapidly, as **96.8%** of participants agreed on its role in the interface. On whether this awareness can translate to developer action (question *I8*), **80.7%** of respondents agreed that provided information was enough to understand if any action was needed on the code, which suggests there could be causality between having access to information and the incentive to take action.

We can also establish a correlation between the correct assessment of metrics and whether the information provided by our tool is enough to consider taking action as, out of the 29 participants which agreed with question *I1*, 24 of them considered the information given enough for action to be taken (**82.8%**). With an existent correlation between the agility of assessment and the awareness for action being needed, we believe that **there is significant evidence suggesting that our tool is capable of allowing developers for a quick assessment of metrics and identification of problematic portions of the code**, such as methods or functions, in files the programmer is not necessarily familiar with.

RQ2 *"Does the usage of these tools result in improvements on quality metrics, leading to a more maintainable code base?"*

Our final research question, **RQ2**, was answered by studying how the developed tool can improve code files towards a greater maintainability. To study this, we selected five open-source repositories with varying degrees of complexity and executed refactorings on selected files. This process and its results are described in greater detail during Section 6.2.2. In order to isolate the effect each refactoring type had on the metrics of a file, we conducted studies separately with each refactoring type (Extract Variable was excluded due to insignificant impact on metrics) on the original files, where we executed, at most, 5 refactorings of each type. As we executed each kind of refactoring separately, we also studied their impact separately, with Extract Method being studied in Section 6.2.2.1 and Extract Class in Section 6.2.2.2.

Results obtained for Extract Method, shown in Figures 6.11 and 6.12, indicate that our tool, operating on minimizing the maximum value of statements and cyclomatic complexity found in all methods and functions across a file, is capable of leading other metrics towards values which suggest a higher maintainability. The results of the Extract Class refactoring, shown in Figure 6.16, suggest that the value of the measured metric trends towards values

considered *better* across most files. However, the amount of Extract Class refactorings performed represents a small sample size and, as such, we cannot conclude whether our Extract Class approach leads towards a more maintainable code.

We can observe that results obtained with our Extract Method approach provide **substantial evidence that post-refactoring code is more maintainable** than its pre-refactoring counterpart. However, results show that our Extract Class approach suggests **is not conclusive** about the maintainability of post-refactoring code.

RQ3 *"Does early exposure to software metrics and consequent early action by the part of the developer lead to a more maintainable source code on earlier stages of development?"*

To answer **RQ3**, we studied the evolution of selected metrics across multiple versions of the same file, a process detailed throughout the final parts of Section 6.2.2.1, where we execute multiple Extract Methods sequentially on each of the selected versions and analyse whether these metrics trend towards more maintainable values.

Results obtained for some files in *deno-drash* repository, a subset of the analysed files in **RQ2**, shown in Figures 6.13 through 6.15, indicate that our Extract Method approach leads to more maintainable code overall, albeit without significant margins, across most versions of a file, with final results between versions converging to similar values.

With the results observed in **RQ1** showing significant evidence of greater developer awareness when using a visualization tool and answers to question 17 showing most respondents considered that provided information was enough for action to be taken, we believe **there is evidence supporting the statement** that early action leads towards higher maintainability early on the development process.

The results obtained in the selected research points indicate that, although there exists significant evidence pointing towards an effectiveness in using quality metrics to guide a live refactoring tool to reach a more maintainable source code, **the main hypothesis of this dissertation could not be entirely validated**. While both the correlation between metric visualization and awareness of action and the effectiveness of our Extract Method approach leading to a more maintainable code being validated, there was not enough evidence to reject the null hypothesis related to our Extract Class approach pointing towards higher maintainability.

6.3 Threats to Validity

As with any research project, it is important to identify and mitigate possible validity threats which may hinder the validation process. As such, this section is dedicated to identify and describe how validity threats we encountered were mitigated and how can they impact the conclusions taken from the results of our study.

6.3.1 Conclusion Validity

Threats to the validity of our conclusions are ones which hinder the ability to draw conclusions considered correct about the relations between treatment and outcomes of experiments [WRH⁺12].

Reliability of measures. The external metric used to evaluate our Extract Class approach, LCOM, is one which has been subject to multiple revisions throughout time [CK91, CK94, HSCG96] and it is one metric whose relevance and usefulness in quality measurement has been put into question [HSCG96, BBM96]. Considering the background of this metric, despite results on our Extract Class pointing towards improvements on the LCOM metric in most files studied, which in turn suggests an overall improvement on the code, any conclusion taken from our results should take into consideration the history of the metric used to evaluate them. Future studies should look to expand on the metrics used to validate our tool's approach.

Assumption of normality. The sample size of our experiment on the Extract Class method is very small, as it is composed of four documents. As such, the used normality test, Shapiro-Wilk's, cannot provide useful results to determine the normality of the sample, leading to an assumption that the samples for this experiment followed a normal distribution. Future experiments should expand on the sample size provided for this experiment.

6.3.2 Internal Validity

Threats to internal validity are ones which affect a conclusion about the causal relationship between treatment and outcome, without the researcher's knowledge [WRH⁺12].

Participant selection. Participants of our survey were volunteers and, as such, are subjects whose motivation is considered to be greater than the whole population [WRH⁺12]. This variability in motivation could influence the answers which would otherwise be given.

File sample characteristics. The files for analysis were selected based on a simple criteria: files should have some logic within them. This was done in order to filter out possible files containing trivial code, such as, for example, a single abstract class. Another point of selection for these files, were files whose amount of code resulted in reasonable processing times, particularly due to the cost of our Extract Class algorithm. Despite the file length selection criteria not influencing the obtained results, it can influence the interpretation of said results. This is, the more code a file has, the less percentage of this code is altered by a refactoring, which results in a less noticeable gain across all code in the file.

Besides this criteria, files were selected randomly, as to not further influence results obtained. As such, interpretation of results should take into consideration how the chosen files were selected.

Environment. The environment in which the survey was performed was not controlled, due to the restrictions imposed by the current circumstances. As such, participants may have had access to resources which can influence their answers. Despite the weight of this threat being close to insignificant, future studies should have more control in the environment they are performed on.

6.3.3 Construct Validity

Threats to this type of validity concerns about the generalization of the results of the experiments to the theory behind those experiments and can be related either to design or social factors [WRH⁺12].

Hypothesis guessing. Participants of our survey, destined to evaluate the usability and feature set of our tool, could have tried to guess the hypothesis being tested, based on the type of questions asked. Despite the intentions behind the survey not being directly disclosed to the participants, such behavior could influence their answers to the proposed questions.

6.3.4 External Validity

External validity is a type of validity which refers to whether results found in our study are capable of holding when tested against populations with different characteristics in different contexts and conditions [WRH⁺12].

Survey sample size. The conducted survey about the tool's usability and feature set counted with about 31 participants, which could be considered a small sample size, with reduced statistical relevance. As such, interpretation of the obtained results should take into consideration the number of participants.

File sample size. Our automated analysis examined the usage of our tool on 22 TypeScript and JavaScript code files. This is considered a small sample size and generalization of results obtained should consider the amount of files analysed. To mitigate this threat, the chosen files are contained within five different open-source repositories of varying structure and complexity, ranging from small, personal *pet projects* to large, industry-scale repositories. Although we believe possible threats caused our relative small file sample size can be mitigated due to the variation of the selected repositories, future experiments should target a greater number of files contained in a larger number of repositories.

Chapter 7

Conclusions and Future Work

7.1	Conclusions	77
7.2	Main Contributions	78
7.3	Future Work	79

This chapter serves as a retrospective about all the work performed during this dissertation, the conclusions obtained, the main contributions this work has done towards scientific progress and future work that can enhance this project in its current state. Section 7.1 summarizes the main conclusions taken from this work, including the answers for the research questions proposed. Section 7.2 details how this work has contributed to the field in question and Section 7.3 describes possible future work to be done in order to enrich what was obtained during this dissertation.

7.1 Conclusions

Refactoring suggestion systems are important tools which help developers tackle some of the technical debt that naturally creeps onto software systems during their development. After a literature review on liveness in software development, refactoring candidate identification, refactoring sequencing, current refactoring tools and live tools, we identified some problems on how current tools could use some improvements about the way results are shown to the user and how easy they are to use.

Our solution for mitigating this problem comes in the form of a Visual Studio Code extension which incorporates the concept of liveness to provide instant feedback to developers about the changes on metrics during development. These metrics are also used to compute refactoring suggestions which target specific metrics and aim to lead them towards values considered more *optimal*.

The following hypothesis was considered:

“A live refactoring recommendation system operating based on quality metrics is able to aid developers towards a more maintainable code base.”

In order to validate said hypothesis, three research questions were proposed which lead our validation methodologies:

RQ1 *"Does providing developers with software visualization tools improves their awareness towards code smells mitigation?"*

Results obtained through a survey suggest that our tool's interface allows developers to quickly identify problematic methods and quickly assess the value of the metrics for any file, even if its contents are not familiar to the programmer. In turn, this agility in assessment serves as significant evidence to claim that **visualization tools lead towards an increase in awareness** to tackle code smells.

RQ2 *"Does the usage of these tools result in improvements on quality metrics, leading to a more maintainable code base?"*

We studied if the tool's supported refactorings which impact metrics, Extract Method and Extract Class, can result in improvements of quality metrics. Results for our Extract Method approach **heavily suggest that it leads towards a more maintainable code**, while results obtained on our Extract Class approach were **not conclusive**.

RQ3 *"Does early exposure to software metrics and consequent early action by the part of the developer lead to a more maintainable source code on earlier stages of development?"*

Studies performed on multiple versions of a subset of files analysed in **RQ2** using the Extract Method refactoring shows that **there is evidence** suggesting that early action leads to more maintainable code from earlier stages of development, as final values converge to similar metric values.

7.2 Main Contributions

Over the duration of this dissertation, our work was directed towards the creation of a refactoring suggestion system, which operates on the analysis and usage of quality metrics to provide developers with a tool which helps tackling the presence of technical debt on the code. Beyond the proposed solution, our work contributed to the software engineering field in the following manners:

Literature review. One of the first steps taken towards understanding how should we approach this problem, was by reporting on what previous researchers have accomplished on the topics of liveness in software development, how refactoring recommendation systems identify candidates, schedule the execution of refactorings, existent refactoring tools and how liveness is incorporated on refactoring tools that support it and existent tools created to evaluate quality metrics in software systems.

Visual Studio Code extension. The proposed solution for the problem at hand, was to design and develop a Visual Studio Code extension. This tool must be able to operate live and offer the developer with an analysis of some quality metrics which help the developer identify the presence of code smells, using them to compute refactoring suggestions which tackle said code smells, for the user to execute semi-automatically.

Custom repository crawler. One of the validation methodologies we used, required the development of a simple repository crawler, capable of executing refactorings on pre-determined repositories and respective documents of interest, through versions of repositories previously specified.

Survey study. Another of the validation methodologies applied, was the creation of a research survey on the developed extension, in order for subjects with background in software engineering to evaluate the feature set, the usability and the accessibility of our tool. We believe results obtained from this survey are promising and could lead to more research being done on the usability of live tools in the future.

7.3 Future Work

As with every software engineering project, current features present on the tool could be improved upon and more experiments could be engineered in order to fully understand how an extension such as ours can improve a developer's workflow and a system's evolution. Here is some future work that can be performed, based on what we learnt throughout its development and based on feedback received from survey participants:

Improvements to interface. Although we believe that our tool provides useful information regarding the detection of code smells by a developer's standpoint, supported by the results obtained in our survey (see Section 6.2.1), some features on the interface could definitely use some improvements to make the information provided more clear, as also pointed by the results of the survey. Also, more interface customization could be done, such as a dark mode option, as to allow users to personalize the tool to their preferences.

Refactoring sequencing. One of the first ideas we envisioned for our tool was the ability for proposing refactoring sequences to the user. However, we found some limitations regarding the implementation of such search algorithms within the Visual Studio Code's API and concerns about the computational cost of employing such algorithms in a tool supposed to be *live*.

Experimentation. With the concept of liveness emerging on software engineering tools and how tightly it is connected to user experience, we believe our tool could be subject to more testing, regarding both its usability and its usefulness. For example, one could evaluate how a project evolves when developers start using our tool starting from early parts of its conception, compared to what the project would look like otherwise.

Our work explored the employment of liveness in the scope of the software engineering process of refactoring. Despite not pioneering live refactoring, we believe the work performed serves as a step forward in the live refactoring field and future research on the area can benefit from the lessons learnt throughout this dissertation and further explore both the interaction between developers and their development environments and the usage of more efficient algorithms to reach better and faster solutions.

References

- [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 Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*, Programming '19, New York, NY, USA, 2019. Association for Computing Machinery.
- [Bas04] Bas Leijdekkers. MetricsReloaded, 2004. <https://plugins.jetbrains.com/plugin/93-metricsreloaded>, Last accessed on 22-07-2020.
- [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.
- [BD02] 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.
- [BDO11] Gabriele Bavota, Andrea De Lucia, and Rocco Oliveto. Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software*, 84(3):397–414, mar 2011.
- [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. *ACM SIGPLAN Notices*, 48(6):95, jun 2013.
- [BFS14] Pierre Bourque, Richard E Fairley, and IEEE Computer Society. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. IEEE Computer Society Press, Washington, DC, USA, 3rd edition, 2014.
- [BOD⁺10] Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, Giuliano Antoniol, and Yann-Gael Gueheneuc. Playing with refactoring: Identifying extract class opportunities through game theory. *2010 IEEE International Conference on Software Maintenance*, pages 1–5, sep 2010.
- [BYRN99] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., USA, 1999.
- [CALO94] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, August 1994.
- [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.
- [CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- [Cod18] CodeMR. CodeMR, 2018. <https://plugins.jetbrains.com/plugin/10811-codemr>, Last accessed on 22-07-2020.
- [Cou19] Sara Filipa Couto Fernandes. Supporting Software Development through Live Metrics Visualization. Dissertation, Faculdade de Engenharia da Universidade do Porto, 2019.
- [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.
- [DDF⁺90] S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer, and R.A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science* 41, pages 391–407, 1990.
- [Deb01] Kalyanmoy Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, Inc., USA, 2001.
- [DJ14] Kalyanmoy Deb and Himanshu Jain. An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints. *IEEE Transactions on Evolutionary Computation*, 18(4):577–601, aug 2014.
- [Fis13] Andrew Fischer. Introducing Circa: A dataflow-based language for live coding. *2013 1st International Workshop on Live Programming (LIVE)*, pages 5–8, may 2013.
- [Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.
- [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.
- [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.
- [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.

- [Had14] Hadi Hariri. Touring Plugins: Software Metrics, 2014. <https://blog.jetbrains.com/idea/2014/09/touring-plugins-issue-1/>, Last accessed on 22-07-2020.
- [Hal77] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., USA, 1977.
- [HGW11] Steffen Herbold, Jens Grabowski, and Stephan Waack. Calculation and optimization of thresholds for sets of software metrics. *Empirical Softw. Engg.*, 16(6):812–841, December 2011.
- [HNJ19] Roman Haas, Rainer Niedermayr, and Elmar Juergens. Teamscale: Tackle Technical Debt and Control the Quality of Your Software. *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 55–56, may 2019.
- [HSCG96] Brian Henderson-Sellers, Larry L. Constantine, and Ian M. Graham. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object Oriented Systems*, 3:143–158, 1996.
- [HT07] Mark Harman and Laurence Tratt. Pareto optimal search based refactoring at the design level. *Proceedings of the 9th annual conference on Genetic and evolutionary computation - GECCO '07*, page 1106, 2007.
- [II11] ISO and IEC. ISO/IEC 25010:2011 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models, 2011.
- [III17] ISO, IEC, and IEEE. Iso/Iec/Ieee 24748-4:2016. *ISO/IEC/IEEE 24765:2017(E)*, 2016:1–522, 2017.
- [Jas18] Jasper Koning. CoreMetrics, 2018. <https://marketplace.visualstudio.com/items?itemName=jasper.coremetrics>, Last accessed on 22-07-2020.
- [KA16] Yahya Khrishe and Mohammad Alshayeb. An empirical study on the effect of the order of applying software refactoring. *2016 7th International Conference on Computer Science and Information Technology (CSIT)*, pages 1–4, jul 2016.
- [Ken38] M. G. Kendall. A New Measure of Rank Correlation. *Biometrika*, 30(1-2):81–93, 06 1938.
- [Ken45] M. G. Kendall. The treatment of ties in ranking problems. *Biometrika*, 33(3):239–251, 1945.
- [Kis16] Kiss Tamás. CodeMetrics, 2016. <https://marketplace.visualstudio.com/items?itemName=kisstkondoros.vscode-codemetrics>, Last accessed on 22-07-2020.
- [LCJ13] Yun Young Lee, Nicholas Chen, and Ralph E. Johnson. Drag-and-drop refactoring: Intuitive and efficient program transformation. *2013 35th International Conference on Software Engineering (ICSE)*, pages 23–32, may 2013.
- [Lik32] R Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22 140:55, 1932.

- [LL13] Remo Lemma and Michele Lanza. Co-evolution as the key for live programming. *2013 1st International Workshop on Live Programming (LIVE)*, pages 9–10, may 2013.
- [LMD06] Michele Lanza, Radu Marinescu, and Stéphane Ducasse. *Object-Oriented Metrics in Practice*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
- [McC] McCabe Software. All Metrics Thresholds in McCabe IQ. <http://www.mccabe.com/pdf/McCabe%20IQ%20Metrics.pdf>, Last accessed on 22-07-2020.
- [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. *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications - OOPSLA '07*, 42(10):623, 2007.
- [Mea12] Panita Meananeatra. Identifying refactoring sequences for improving software maintainability. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, page 406, 2012.
- [MHPB12] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering*, 38(1):5–18, jan 2012.
- [Mic] Microsoft. Code Metrics – Maintainability Index. <https://docs.microsoft.com/pt-pt/archive/blogs/zainnab/code-metrics-maintainability-index>, Last accessed on 22-07-2020.
- [MKB⁺16] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Mel O Cinneide, and Kalyanmoy Deb. *On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach*, volume 21. dec 2016.
- [MKWD17] Usman Mansoor, Marouane Kessentini, Manuel Wimmer, and Kalyanmoy Deb. Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm. *Software Quality Journal*, 25(2):473–501, jun 2017.
- [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.
- [MÓ11] Iman Hemati Moghadam and Mel Ó Cinnéide. Code-Imp. *Proceeding of the 4th workshop on Refactoring tools - WRT '11*, page 41, 2011.
- [MP05] A. Marcus and D. Poshyvanyk. The conceptual cohesion of classes. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 133–142, 2005.
- [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.
- [Nas51] John Nash. Non-Cooperative Games. *Annals of Mathematics*, 54(2):286–295, 1951.
- [Nik] Nikolaos Tsantalis. JDeodorant. <https://github.com/tsantalis/JDeodorant>, Last accessed on 22-07-2020.
- [OHA92] P. Oman, J. Hagemester, and D. Ash. A definition and taxonomy for software maintainability. *Moscow, ID, USA, Tech. Rep*, pages 91–08, 1992.
- [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)*, pages 15–18, may 2013.
- [OMIA17] Akio Oka, Hidehiko Masuhara, Tomoki Imai, and Tomoyuki Aotani. Live Data Structure Programming. *Proceedings of the International Conference on the Art, Science, and Engineering of Programming - Programming '17*, Part F1296:1–7, 2017.
- [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.
- [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.
- [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.
- [SGSM10] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. Making Program Refactoring Safer. *IEEE Software*, 27(4):52–57, jul 2010.
- [SMHG13] Gustavo Soares, Emerson Murphy-Hill, and Rohit Gheyi. Live feedback on behavioral changes. *2013 1st International Workshop on Live Programming (LIVE)*, pages 23–26, may 2013.
- [Ste19a] Stepsize. Tech Debt Tracker, 2019. <https://marketplace.visualstudio.com/items?itemName=Stepsize.tech-debt-tracker>, Last accessed on 22-07-2020.
- [Ste19b] Stepsize. Tech Debt Tracker: The Code Metrics, 2019. <https://www.notion.so/Tech-Debt-Tracker-The-Code-Metrics-738c12fe245b4064bdc951ea6b5b1403>, Last accessed on 22-07-2020.
- [Ste19c] Stepsize. Tech Debt Tracker: The Debt Ratings, 2019. <https://www.notion.so/Tech-Debt-Tracker-The-Debt-Ratings-764fb1ea2af64446b14323992181981b>, Last accessed on 22-07-2020.

- [SW65] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples)[†]. *Biometrika*, 52(3-4):591–611, 12 1965.
- [Tan90] Steven L. Tanimoto. VIVA: A visual language for image processing. *Journal of Visual Languages and Computing*, 1(2):127–139, jun 1990.
- [Tan13] Steven L. Tanimoto. A perspective on the evolution of live programming. *2013 1st International Workshop on Live Programming (LIVE)*, pages 31–34, may 2013.
- [TC09] N. Tsantalis and A. Chatzigeorgiou. Identification of Move Method Refactoring Opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, may 2009.
- [TC16a] Sandhya Tarwani and Anuradha Chug. Prioritization of code restructuring for severely affected classes under release time constraints. *2016 1st India International Conference on Information Processing (IICIP)*, pages 1–6, aug 2016.
- [TC16b] 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.
- [UOII17] Naoya Ujihara, Ali Ouni, Takashi Ishio, and Katsuro Inoue. c-JRefRec: Change-based identification of Move Method refactoring opportunities. *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 482–486, feb 2017.
- [WRH⁺12] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Björn Regnell, and Anders Wessln. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

Appendix A

Survey

This appendix is a copy of the survey filled by volunteers related to the software engineering area. Results to this survey are shown in Section [6.2.1](#).

Live Refactoring

This survey aims to evaluate two refactoring tools: "Extract Method Finder" and "Semi-automated Extractor", developed in the context of two master thesis done at FEUP: "Towards a Smart Recommender for Code Refactoring" and "Towards a Live Refactoring Recommender Based on Code Smells and Quality Metrics", respectively.

Both tools were built as a VS Code extension for JavaScript and TypeScript. They're recommendation systems that allows developers to visualize, select and apply refactoring suggestions.

No extensive knowledge about refactoring is required to answer this survey. However, basic programming experience is recommended.

All answers are anonymous and will be used exclusively for academic purposes. If you're interested and consent, your participation will be acknowledged in our work.

Thank you for your collaboration,
João Barbosa & Sérgio Salgado

(Estimated completion time: 15-30 mins)

* Required

Participant's
Profile
(Optional)

This section will only be used to better understand the distributions among the participants. No filling is required if you don't want to share your personal information.

1. Age

Mark only one oval.

- ☐ < 18
- ☐ 18 - 24
- ☐ 25 - 40
- ☐ > 40

2. Education

Highest completed degree of education. If currently enrolled in one, choose the highest one completed.

Mark only one oval.

- ☐ High school
- ☐ Bachelor's degree
- ☐ Master's degree
- ☐ Doctorate's degree
- ☐ Other:

Technical Background

Tell us a bit more about yourself, regarding knowledge in software development and refactoring systems.

3. Programming Experience *

Mark only one oval.

- ☐ No experience
- ☐ < 1 year
- ☐ 1 - 2 years
- ☐ 3 - 5 years
- ☐ 5 - 10 years
- ☐ > 10 years

4. I am familiar with programming terms, such as classes, methods, data structures, etc. *

Mark only one oval.

1 2 3 4 5

Highly disagree ☐ ☐ ☐ ☐ ☐ Highly agree

5. I am familiar with software development terms, such as refactoring, debugging, unit testing, etc *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

6. I am experienced in JavaScript or TypeScript. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

7. I have used Visual Studio Code before. *

Mark only one oval.

☐ Yes

☐ No

8. I have used refactoring tools or extensions before. *

Mark only one oval.

☐ Yes

☐ No

9. I have used software metric analysis tools before. *

Mark only one oval.

☐ Yes

☐ No

**Part 1:
Extract
Method
Finder**

The first part of this survey aims to evaluate the "Extract Method Finder".

The application focuses on the Extract Method refactoring. We recommend this simple 2-min read if you want to learn or need a refresh about this technique:
<https://refactoring.guru/extract-method>.

**1.1.
Visualization**

This section focuses on the first component of the tool: visualizing refactoring recommendations.

The tool is always active and analyzing the source code as it is modified, providing real-time updates to the developer.

Suggestions are shown as colored blocks, to the left of the editor's line number. Each suggestion uses a different color.

Colors follow a gradient from green to red and are assigned according to the suggestion's severity: closer to red means the refactoring is more evident. If there are no suggestions, only a green color is shown, spanning from the beginning to the end of the block.

This section evaluates three main aspects:

- Continuous Feedback
- Multiple vs. Single Suggestion
- Refactoring Availability

Continuous Feedback

The animated image below shows a method being implemented. Our tool updates its suggestions in real-time as changes are being made to the code.

```

examples > TS UtilsTS.ts > UtilsTS > levenshtein
1  class UtilsTS {
2      levenshtein(a: string, b: string) {
3          |
4      }
5  }
6

```

10. 1.1.1. I can quickly locate the different refactoring suggestions. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

11. 1.1.2. The color scheme allows me to quickly identify the most prominent suggestions. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

12. 1.1.3. Continuous feedback next to line numbers is too distracting for my regular development activity. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

13. 1.1.4. How frequently should suggestions be updated? *

Mark only one oval.

- ☐ Once I stop typing.
- ☐ Once I change to another line.
- ☐ At fixed rates (e.g. every 1 second).
- ☐ Continuously.
- ☐ Other: _____

Multiple vs. Single Suggestion

Consider the two images below. The first image shows a more traditional refactoring recommender, with a single suggestion. The second image shows all suggestions available in the method 'levenshtein', of the class 'UtilsTS'.

Single suggestion

```

TS UtilsTS.ts ×
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     }
}

```

Multiple suggestions

```

TS UtilsTS.ts ×
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     }
}

```

14. 1.1.5. I see benefit in having multiple suggestions available at the same time, since it raises my awareness on the options I have. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

15. 1.1.6. I see benefit in having multiple suggestions available at the same time, since it helps me better understand the tool's reasoning. *

For instance, consider line 18 (second image). Do you agree with the assignment? If not, can you understand why/how it may have reached a different outcome?

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

Refactoring Availability

On the animation below, we move our cursor from top to down, across all lines of code. Our tool only shows feedback in certain blocks.

```

1
2  const DEFAULT = 20;
3
4  enum PrintMedia {
5    Newspaper,
6    Newsletter,
7    Magazine,
8    Book,
9  }
10
11 function getMedia(mediaName: string): PrintMedia {
12   if (mediaName === "Forbes" || mediaName === "Outlook") {
13     return PrintMedia.Magazine;
14   }
15 }
16
17 interface Department {
18   printName(): void;
19   printMeeting(): void;
20 }
21
22 class Accounting implements Department {
23   public name: string;
24   public meet: { weekDay: string; hour: string };
25
26   constructor() {
27     this.name = "Accounting and Auditing";
28     this.meet = {
29       weekDay: "Monday",
30       hour: "10am",
31     };
32   }
33
34   printName(): void {
35     console.log("Department name: " + this.name);
36   }
37
38   printMeeting(): void {
39     console.log(
40       `The Accounting Department meets each ${this.meet.weekDay} at ${this.meet.hour}.`
41     );
42   }
43 }
44

```

16. 1.1.7. I am able to quickly understand when refactoring feedback is available. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

17. 1.1.8. In which data structures was our tool giving feedback? *

Check all that apply.

- ☐ Global variables
- ☐ Enums
- ☐ Functions
- ☐ Interfaces
- ☐ Class declarations
- ☐ Class attributes
- ☐ Class constructors
- ☐ Class methods

18. Additional remarks

Write here any additional feedback that wasn't addressed in the previous questions.

1.2.
Selection
and
Application

This section is concerned with the selection and application of refactoring suggestions provided by the tool.

Both actions are available through a command reachable by VS Code's Command Palette, which opens a new selection menu.

For each suggestion, the developer is able to see if it is automatically extractable and the number of lines it contains. Also, code lines of the suggestion in focus will be highlighted in the editor.

After a suggestion is accepted, the developer is prompted to write the name of the new extracted method.

```

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
23    // ...
24    return matrix[bn][an];
25  }
26 }
27

```

19. 1.2.1. Selecting refactoring suggestions through the VS Code's Command Palette is easy and intuitive. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

20. 1.2.2. Highlighting a suggestion allows me to better perceive the lines of code it refers to. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

21. 1.2.3. I find usefulness in having a semi-automated refactoring tool. *

Semi-automated means it modifies the code automatically, but first a suggestion must be selected by the developer.

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

22. Additional remarks

Write here any additional feedback that wasn't addressed in the previous questions.

1.3. Code Quality History

Our tool provides a second command, in VS Code's Command Palette, which allows developers to analyze the file's evolution in terms of code quality.

The command opens a webview displaying a multi-line chart. The chart updates every time the file is saved.

The chart displays a subset of software metrics, picked according to their relevance for Extract Method refactoring. Each software metric has a label and is represented by a unique color.

The developer is also able to hover recorded instances, which displays the absolute values of these metrics (colored according to the label).

(Note: knowledge about code quality metrics is not necessary to answer this section. In short, metrics are numerical values that are indicative of the software's overall complexity and/or quality. If you want to know more about the metrics we use, you can check https://www.verifysoft.com/en_halstead_metrics.html)

```

1 class Utils {
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
23    // ...
24    return matrix[bn][an];
25  }
26 }
27
```

23. 1.3.1. Software metrics are a good indicative of overall code complexity and quality. *

Mark only one oval.

1 2 3 4 5

Highly disagree ☐ ☐ ☐ ☐ ☐ Highly agree

24. 1.3.2. I find value in having an historical record for code quality. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

25. 1.3.3. I find value in being able to see code quality evolution as it is being modified. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

26. 1.3.4. I find value in being able to see absolute values for the code quality metrics. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

27. Additional remarks

Write here any additional feedback that wasn't addressed in the previous questions.

1.4. Final Remarks

This section of the survey is aimed at better understanding your findings on the tool's overall experience and usability, as well as possible improvements to be made.

28. 1.4.1. The tool's features were simple to use and easy to understand. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

29. 1.4.2. This tool can positively impact my development workflow. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

30. 1.4.3. I would consider using this tool. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

31. 1.4.4. Based on what you saw on this survey, which features would you say were the most important? *

Check all that apply.

- ☐ Live feedback on refactoring needs.
- ☐ Refactoring severity based on color gradient.
- ☐ Multiple refactoring suggestions.
- ☐ Semi-automated refactoring application.
- ☐ Code quality history analysis.

Other: ☐ _____

32. 1.4.5. Based on what you saw on this survey, which features could be improved?

*

Check all that apply.

- ☐ Live feedback on refactoring needs.
- ☐ Refactoring severity based on color gradient.
- ☐ Multiple refactoring suggestions.
- ☐ Semi-automated refactoring application.
- ☐ Code quality history analysis.

Other: ☐ _____

33. Additional remarks

Write here any additional feedback that wasn't addressed in the previous questions.

Part 2:
Semi-
automated
Extractor

This section of the survey showcases our second tool, presented below, which consists of a refactoring tool that supports the execution of three refactorings based on extraction: Extract Method, Extract Class and Extract Variable. These refactorings are executed based on background live analysis of quality metrics, whose values can indicate the presence of 'code smells' lying within the source code.

A code smell is a characteristic in code, which usually indicates the presence of a deeper problem. If you need to refresh your memory about what a code smell is, we recommend reading the following:

What is a code smell? - <https://martinfowler.com/bliki/CodeSmell.html>
Types of code smells. - <https://sourcemaking.com/refactoring/smells>

2.1. Interface

This section is dedicated to questions relative to the tool's interface and respective usability. The images below show the tool's interface presenting a report about the results retrieved from the analysis of the 'Observable.ts' source code file, in both the compressed and expanded view. If you wish to view the file, although not needed to answer the questions, it is located at: <https://github.com/dojo/intern-only-dojo/blob/master/src/Observable.ts>

The interface is divided into three sections:

1. File-related metrics: area where line, node and Halstead metrics (not shown in the picture, see below for more information) are shown. This section shows metrics calculated across the entire code file.
2. Method metrics: area where metrics relative to each individual method contained in the file is shown. Small colored squares reflect the metric values on each method, without the need to expand this area of the interface.
3. Refactoring suggestions: area which shows the amount of suggestions found for each kind of supported refactoring, with more information about the top suggestion when clicked.

Halstead metrics identify measurements of software properties and the relationships between them and are inferred using the amount of operands and operators in a given code structure. Although knowledge about them is not needed to answer the following questions, you can read more about them at <https://www.geeksforgeeks.org/software-engineering-halsteads-software-metrics/>

Background colors are coded to show the severity of each metric, ranging from red (for critical values), to green (near-optimal/optimal values).

If needed, full resolution pictures are provided at:

- Compressed View: <https://i.imgur.com/tupGifx.png>
- Expanded View: <https://i.imgur.com/VC4yLZ3.png>

Interface compressed report

Live Refactoring

Showing results for file: src\middlewares\middleware.ts

Metric	Current Value
Line-related Metrics	
Number Of Lines	153
Number Of Comment Lines	0
Number Of Code Lines	119
Number Of Blank Lines	34
Node-related Metrics	
Number Of For Statements	3
Number Of While Statements	0
Number Of Conditional Statements	10
Number Of Classes	6
Number Of Methods	15
Number Of Functions	0
Number Of Class Fields	0

Method Metrics



Function Metrics

Refactoring Suggestions

Extract Method Suggestions (14 suggestions)
Extract Class Suggestions (0 suggestions)
Extract Variable Suggestions (12 suggestions)

Interface expanded report

Live Refactoring

Showing results for file: src\middlewares\middleware.ts

Metric	Current Value
Line-related Metrics	
Number Of Lines	153
Number Of Comment Lines	0
Number Of Code Lines	119
Number Of Blank Lines	34
Node-related Metrics	
Number Of For Statements	3
Number Of While Statements	0
Number Of Conditional Statements	10
Number Of Classes	6
Number Of Methods	15
Number Of Functions	0
Number Of Class Fields	0

Method Metrics



Middleware.bypassCrawlers	
Maintainability	85.83
Method Complexity	1.00
Number of Statements	1.00
LCOM	0.00
Middleware.getExceptionResources	
Maintainability	85.83
Method Complexity	1.00
Number of Statements	1.00
LCOM	0.00
Middleware.getHandler	
Maintainability	82.24
Method Complexity	1.00
Number of Statements	1.00
LCOM	0.00
Middleware.getName	
Maintainability	94.66
Method Complexity	1.00
Number of Statements	0.00
LCOM	0.00
MandatoryQueryParams.getName	
Maintainability	85.83
Method Complexity	1.00
Number of Statements	1.00
LCOM	0.00
ValidateParams.getName	
Maintainability	85.83
Method Complexity	1.00
Number of Statements	1.00
LCOM	0.00
NamespaceValidator.getName	
Maintainability	85.83
Method Complexity	1.00
Number of Statements	1.00
LCOM	0.00
Middleware.handler	
Maintainability	89.41
Method Complexity	1.00
Number of Statements	0.00

LCOM	0.00
MandatoryQueryParams.handler	
Maintainability	60.18
Method Complexity	4.00
Number of Statements	9.00
LCOM	0.00
ValidateParams.handler	
Maintainability	52.39
Method Complexity	7.00
Number of Statements	19.00
LCOM	0.00
NamespaceValidator.handler	
Maintainability	64.95
Method Complexity	3.00
Number of Statements	7.00
LCOM	0.00
Middleware.isProd	
Maintainability	84.25
Method Complexity	1.00
Number of Statements	1.00
LCOM	0.00
Middleware.register	
Maintainability	81.11
Method Complexity	1.00
Number of Statements	1.00
LCOM	0.00
AppMiddleware.register	
Maintainability	58.25
Method Complexity	4.00
Number of Statements	11.00
LCOM	0.00
ResourceMiddleware.register	
Maintainability	79.77
Method Complexity	1.00
Number of Statements	1.00
LCOM	1.00

Function Metrics

Refactoring Suggestions

Extract Method Suggestions (14 suggestions)

In order to execute the following refactor operation automatically, please execute the following command: **Live Refactoring: Extract Method**

Our algorithm considers the extraction of the following fragment in method *handler* to be the most valuable of all found candidates:

```
const queryParams = Object.entries(this.resource.getQueryParams());
const query = req.query;
for(let [n, p] of queryParams) {
  if(n in query) {
    const param = new p(query[n] as string)
    if(!param.isValid()) throw new InvalidParamException(n);
    params[n] = param;
  }
}
```

This fragment to be extracted has the following metrics:

- **Method Complexity** (method-wide metric, the lower, the better): **4.00**
- **Number of Statements** (method-wide metric, the lower, the better): **8.00**
- **Lack of Cohesion in Methods** (class-wide metric, the lower, the better): **0.00**

and changes the highest values of these metrics on all methods in class *ValidateParams* to the following:

- **Method Complexity** (method-wide metric, the lower, the better): **4.00**
- **Number of Statements** (method-wide metric, the lower, the better): **11.00**
- **Lack of Cohesion in Methods** (class-wide metric, the lower, the better): **0.00**

Extract Class Suggestions (0 suggestions)

Extract Variable Suggestions (12 suggestions)

38. 2.1.5. The second section of the interface, method-related metrics, is emphasized enough and its information is of easy interpretation. *

Mark only one oval.

1 2 3 4 5

Highly disagree ☐ ☐ ☐ ☐ ☐ Highly agree

39. 2.1.6. The third section of the interface, refactoring suggestions, is emphasized enough and its information is of easy interpretation. *

Mark only one oval.

1 2 3 4 5

Highly disagree ☐ ☐ ☐ ☐ ☐ Highly agree

40. 2.1.7. The amount of information presented is overwhelming. *

Mark only one oval.

1 2 3 4 5

Highly disagree ☐ ☐ ☐ ☐ ☐ Highly agree

41. 2.1.8. The information given is enough for me to know if any action is needed on the code. *

Mark only one oval.

1 2 3 4 5

Highly disagree ☐ ☐ ☐ ☐ ☐ Highly agree

42. 2.1.9. The preview colored squares on the 'Method Metrics' section allows for even faster analysis of methods. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

43. 2.1.10. This report format of presentation fits the purpose of this tool. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

Post-Changes Interface

When changes in the file occur, the tool displays the two more recent values, with the most recent being on the right, storing them for the current section.

The image below shows the impact of an Extract Method refactoring on the metric values for the same file as the previous image.

If necessary, the full resolution image is available here: <https://i.imgur.com/FR7v7r7.png>

Post-changes impact on the interface report

Metric	Value Before	Current Value
ValidateParams.handler		
Maintainability	52.39	57.95
Method Complexity	7.00	4.00
Number of Statements	19.00	12.00
LCOM	0.00	0.00

44. 2.1.11. I can clearly understand the values on the left side correspond to old values, while the ones on the right correspond to the newest ones. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

45. 2.1.12. Only showing the two more recent values of each metric for a programming session is enough. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

2.2. Visual Studio Code

This section contextualizes how our tool is incorporated into the VSCode UI.

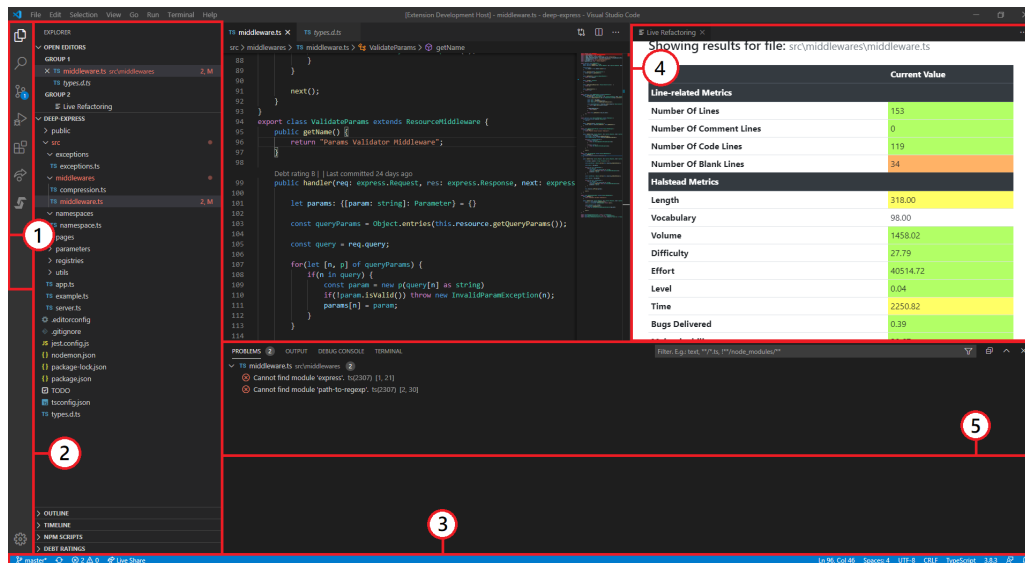
The image below shows five contribution points in the VSCode UI, where implementing UI fragments is the most common. Their nomenclature is as follows:

- 1 - Tree View Container
- 2 - Tree View
- 3 - Status Bar
- 4 - Webview
- 5 - Diagnostics Report

If necessary, the full resolution image is available here:

<https://i.imgur.com/8HOMRtV.png>

VSCode main interface



46. 2.2.1. The webview interface is distracting. *

Mark only one oval.

1 2 3 4 5

Highly disagree ☐ ☐ ☐ ☐ ☐ Highly agree

47. 2.2.2. I would prefer for the information given by the tool to be shown in other contribution points, eliminating the need for a webview. *

Mark only one oval.

1 2 3 4 5

Highly disagree ☐ ☐ ☐ ☐ ☐ Highly agree

48. 2.2.3. Extensions whose features are scattered throughout the VSCode UI can be confusing to use. *

Mark only one oval.

1 2 3 4 5

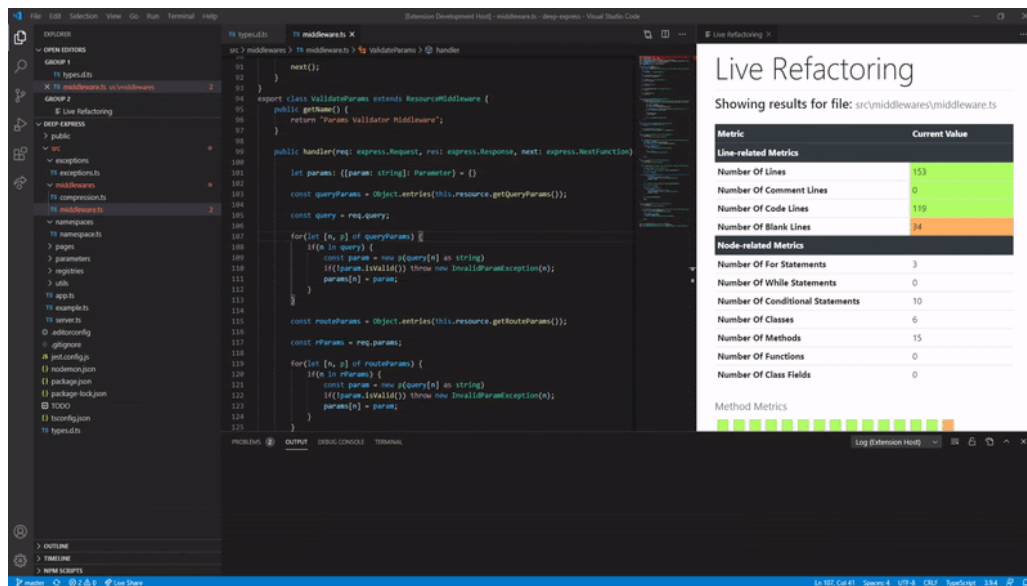
Highly disagree ☐ ☐ ☐ ☐ ☐ Highly agree

Workflow

The video below shows the usage of the tool, from the moment metrics are quickly scanned, to the assessment and execution of the top Extract Method suggestion. Apart from Extract Method, our tool directly supports the Extract Class and Extract Variable refactorings.

The file shown, 'middleware.ts', contains 6 classes, 14 methods across them, with around 150 lines of code.

Workflow on the Semi-automated Extractor



53. 2.2.8. I would prefer a button in the interface to run the refactoring, instead of manually executing a command. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

2.3. Refactoring

Refactoring is a common software engineering practice, where the programmer changes the internal aspect of a system, without changing its external behavior.

As you may have noticed from the previous video, apart from a metrics report, our tool supports the automated execution of three refactorings: Extract Method, Extract Class and Extract Variable. These refactorings are automatically evaluated according to the values of a specific set of quality metrics (some not shown on the report).

54. 2.3.1. All three supported refactorings (Extract Method, Extract Class and Extract Variable) are refactorings I use regularly. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

55. 2.3.2. A tool which automatically finds the best refactoring of each kind, for any file and in close to real time, is something I see value in. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

56. 2.3.3. Having semi-automated execution of the best found refactorings is useful, i.e. the user triggers the execution, but execution is performed automatically. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

57. 2.3.4. Having a preview of what the refactoring is going to change before executing is useful. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

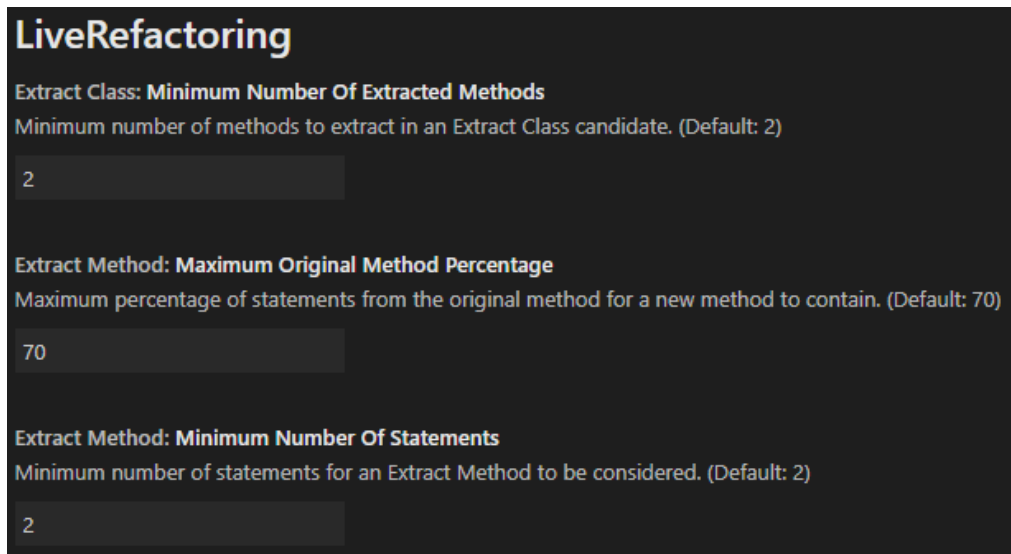
Customization

As each project's context is different from one another, our tool allows programmers to customize the refactoring suggestions and the information shown on the interface report.

The image below shows some of the options the user can customize to alter the tool's behavior.

If necessary, the full resolution image is available here: <https://i.imgur.com/ALrKnai.png>

Options menu



LiveRefactoring

Extract Class: Minimum Number Of Extracted Methods
Minimum number of methods to extract in an Extract Class candidate. (Default: 2)

2

Extract Method: Maximum Original Method Percentage
Maximum percentage of statements from the original method for a new method to contain. (Default: 70)

70

Extract Method: Minimum Number Of Statements
Minimum number of statements for an Extract Method to be considered. (Default: 2)

2

58. 2.3.5. Offering user customization for refactoring tools is useful. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

59. 2.3.6. The description of each option and its impact on the suggestions is clear. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

2.4. Final remarks

The final section of the survey aims to understand your opinion on this tool and which changes should be made to make it better.

60. 2.4.1. This tool's features are simple and fast to understand. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

61. 2.4.2. This tool can positively impact my development workflow. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

62. 2.4.3. I would use this tool. *

Mark only one oval.

	1	2	3	4	5	
Highly disagree	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Highly agree

63. 2.4.4. Based on what you saw on this survey, what would you say were the best features of this tool? *

Check all that apply.

- ☐ Metrics and refactoring report (Webview interface)
- ☐ Comparison between old and new metrics
- ☐ Live computation of metrics
- ☐ Live evaluation of refactoring opportunities
- ☐ Semi-automated refactoring execution
- ☐ Extension customization

Other: ☐ _____

64. 2.4.5. Based on what you saw on this survey, what would you say were the features which could be improved upon? *

Check all that apply.

- ☐ Metrics and refactoring report (Webview interface)
- ☐ Comparison between old and new metrics
- ☐ Live computation of metrics
- ☐ Live evaluation of refactoring opportunities
- ☐ Semi-automated refactoring execution
- ☐ Extension customization

Other: ☐ _____

All done :)

Thank you for your time and valuable feedback!

And remember, refactor early and continually.



This content is neither created nor endorsed by Google.

Google Forms

