Supporting Software Development through Live Metrics Visualization

Sara Filipa Couto Fernandes

Integrated Master in Informatics and Computing Engineering

Supervisor: André Restivo
Co-Supervisor: Ademar Aguiar

July 20, 2019
Supporting Software Development through Live Metrics Visualization

Sara Filipa Couto Fernandes

Integrated Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: João Carlos Pascoal Faria
External Examiner: Luís Manuel Borges Gouveia
Supervisor: André Monteiro de Oliveira Restivo
July 20, 2019
Abstract

Software systems are increasingly complex and extensive, making their understanding often difficult, costly and time-consuming. When we need to implement or change a feature in a system, it is often necessary to restructure it before changing it, doing refactoring, without modifying its external behaviors, improving not only its structure but also its comprehension and maintainability.

Sometimes it is difficult to know the state or quality of a software system, while it is being programmed since there are not many tools that allow the analysis of metrics in real-time. Moreover, when it comes to improving a program by doing refactoring there are also not many tools dedicated to assessing the quality of the final result. Some IDEs such as Eclipse, IntelliJ or Visual Studio Code already ensure that it is possible to analyze different software metrics, through several plugins, and also to suggest different refactoring methods or plausible modifications. However, in most cases, it is necessary to compile and run the program to verify the evolution of the respective metrics and, likewise, to infer if the changes achieved the desired goal, giving late feedback to programmers, thus making the development time longer than expected.

In order to mitigate the described problem, we proposed the development of a Visual Studio Code extension, that could give live feedback to a programmer through the analysis, in real and execution-time, of several software metrics, providing valuable information, visually, regarding the selected metrics, and also suggesting some refactoring methods and modifications that might benefit the program. With this, the developer would be able to check the evolution of the system metrics, while programming it, reducing his development time, but improving his code’s quality and his convergence to a good solution. Also, to get an overview of the software system, this extension should give information about the different software metrics in each Git commit previously executed.

To validate the described extension, we carried out a controlled experiment with programmers, where they had to test the tool with a TypeScript software system. They were divided into two groups where one of the groups used the developed tool and the other not. In this way, it was possible to verify not only that the tool was easy to use and to understand, but also that the participants who used the tool achieved better results regarding the development time, code’s quality or their solution’s convergence more consistently. However, through several hypothesis-test we could not reject the null hypothesis, thus not validating completely the research questions and the main hypothesis that supports this dissertation. Despite this, we verified that the participants showed enthusiasm in using the tool and also showed willingness to reuse it in the future.

However, we consider that this project had a very innovative aspect, since it joins several aspects like live programming, quality metrics analysis, refactoring and software, and information visualization, building a tool that benefits programmers of systems of medium-to-high complexity and dimension. The tool allows immediate feedback about the software under analysis, in an easy way to understand it, enabling also the developer to know the state of the current system and its evolution over each change done.
Resumo

Os sistemas de software estão cada vez mais complexos e extensos, tornando por vezes a sua compreensão mais difícil, custosa e morosa. Quando é necessário adicionar ou alterar uma feature num sistema é, frequentemente, preciso reestruturá-lo antes de o modificar, fazendo refactoring, não alterando os seus comportamentos externos, mas melhorando não só a sua estrutura como também a sua compreensão e adaptabilidade.

Por vezes, é complicado compreender o estado ou a qualidade de um sistema de software, à medida que este é programado, visto existirem poucas ferramentas que permitem a análise de métricas em tempo-real ou que permitem avaliar a qualidade final de um sistema após um refactoring. Alguns IDEs como o Eclipse, IntelliJ ou Visual Studio Code já garantem que seja possível analisar diferentes métricas através de diversos plugins e, também, sugerir diferentes tipos de refactorings ou modificações plausíveis. Contudo, maioria, é necessário compilar e executar o programa para verificar a evolução dessas métricas e, do mesmo modo, inferir se as alterações efetuadas foram as mais corretas, dando feedback tardio aos programadores, tornando assim o tempo de desenvolvimento mais longo do que o expectável.

De modo a mitigar este problema, foi proposta a criação de uma extensão, para Visual Studio Code, que dá live feedback ao programador através de análise, em tempo real e de execução, de diversas métricas de software, fornecendo visualmente informação valiosa sobre as mesmas, sugerindo ainda alguns métodos de refactoring e alterações que podem beneficiar o sistema. Com isto, o programador pode verificar a evolução das métricas, enquanto programa, reduzindo assim o tempo de desenvolvimento, melhorando a qualidade do seu código e a sua convergência para uma solução melhor. Para além disso, para obter uma visão geral do software, esta extensão fornece também, informações sobre as diferentes métricas em cada Git commit, previamente, executado.

Para validar a ferramenta descrita, efetuou-se uma experiência controlada com programadores, em que estes testaram a ferramenta com um sistema de software em TypeScript, tendo sido divididos em dois grupos, um que em que se utilizou a ferramenta respetiva e outro não. Desta forma, foi possível verificar não só que a ferramenta era fácil de usar e compreender, como também que em média se obteve melhores resultados relativos ao tempo de desenvolvimento, qualidade de código e convergência para uma melhor solução, de forma mais consistente, quando se utiliza a ferramenta criada. Contudo, através de alguns testes de hipóteses não se conseguiu rejeitar as hipóteses nulas, não se conseguindo validar completamente quitar as research questions quer a hipótese principal que sustentam esta dissertação. Apesar disso, verificou-se que os participantes mostraram entusiasmo em utilizar a ferramenta, mostrando ainda vontade em reutilizá-la no futuro.

Apesar disto, considera-se que este projeto tem uma vertente bastante inovadora, visto que agrupa diversos aspectos como live programming, análise de métricas de qualidade, refactoring e visualização de informação, construindo uma ferramenta que beneficia os programadores, de sistemas de média a grande complexidade e dimensão. Esta ferramenta permite, assim, dar feedback imediato sobre o sistema sob análise, de uma forma fácil de compreender, permitindo igualmente dar conhecimento do seu estado atual e da sua evolução ao longo das alterações efetuadas.
Acknowledgements

Above all, I would like to express my special thanks to the Faculty of Engineering of the University of Porto, namely, to the teachers and non-professors with whom I had the privilege of meeting.

A special thanks to my Supervisor, André Restivo, and to my Co-Supervisor, Ademar Aguiar, for sharing their knowledge, for helping me and for always be available during my existential crises. I should also thank Professor Hugo Sereno Ferreira for being like a Pseudo Co-Supervisor, who helped me during the development of my dissertation, since without his genius ideas, none of this would be possible, and also Professor Filipe Correia who was always ready to give me new ideas and solutions for all problems that I have encountered during this project. To all of them, an extra special thanks for encouraging my creativity and ability to create memes and nicknames.

Besides, I should also thank João Pedro Dias for all the advice and ideas that he shared with me. In the same way, I must thank Diogo Campos and Mário Gustavo Fernandes for being able to live with my mood swings and for always being available to help me, anytime, any hour of the day.

To all my family, especially my mother and father, thanks for always supporting my decisions and dreams and, also, to my brother for often making me distract from my strenuous study moments.

To my friends and colleagues who stayed by my side during these five years, thanks for the friendship, motivation, mutual help, constant laughter and for putting up with my craziness.

Sara Filipa Couto Fernandes
“Where there is great power there is great responsibility.”

Winston Churchill
Contents

1 Introduction ........................................ 1
   1.1 Context ...................................... 2
   1.2 Motivation ................................... 3
   1.3 Problem ...................................... 3
   1.4 Objectives ................................... 3
   1.5 Report Structure .............................. 4

2 Background ........................................ 7
   2.1 Software Engineering .......................... 7
      2.1.1 SWEBOK Guide .......................... 7
      2.1.2 Challenges ................................ 9
   2.2 Live Programming .............................. 10
      2.2.1 Motivation ................................ 10
      2.2.2 Liveness Levels .......................... 10
      2.2.3 Problems with Live Programming ....... 11
   2.3 Refactoring .................................... 12
   2.4 Quality Analysis ............................... 12
   2.5 Software Visualization ....................... 13

3 State of the Art .................................... 17
   3.1 Live Software Development ...................... 18
      3.1.1 Live IDEs .................................. 18
      3.1.2 Live Languages ................................ 20
      3.1.3 Live Tools .................................. 21
      3.1.4 Discussion .................................. 25
   3.2 Refactoring ..................................... 26
      3.2.1 Refactoring Tools .......................... 26
      3.2.2 Refactoring Proposals ...................... 34
      3.2.3 Discussion .................................. 35
   3.3 Live Refactoring ............................... 36
      3.3.1 Discussion .................................. 38
   3.4 Quality Metrics Analysis ....................... 39
      3.4.1 Quality Metrics Tools ...................... 39
      3.4.2 Quality Metrics Proposals .................. 41
      3.4.3 Discussion .................................. 41
   3.5 Software Visualization ....................... 42
      3.5.1 Metaphors .................................. 42
      3.5.2 Visualization Tools ....................... 43
### CONTENTS

3.5.3 Discussion .......................... 45
3.6 Visual Studio Code Tools ................. 45
  3.6.1 Software Metric’s Analysis .............. 45
  3.6.2 Refactoring .......................... 48
  3.6.3 Discussion .......................... 53
3.7 Summary ................................ 54
  3.7.1 Systematic Literature Review Results .... 54
  3.7.2 Visual Studio Code Tools ............... 56

4 Problem Statement ....................... 59
  4.1 Current Issues .......................... 59
  4.2 Assumptions ............................ 61
  4.3 Hypothesis .............................. 61
  4.4 Research Questions ...................... 62
  4.5 Proposal ................................ 62
  4.6 Validation Methodology ................... 63
  4.7 Summary ................................ 64

5 Proposed Solution ....................... 65
  5.1 Contextualization ......................... 65
  5.2 Objectives .............................. 66
  5.3 Requirements ........................... 66
    5.3.1 Functional Requirements ............... 66
    5.3.2 Non-Functional Requirements .......... 68
  5.4 Implementation’s Proposal ............... 68
  5.5 Architecture ............................ 69
  5.6 Case Study ............................. 71
  5.7 Summary ................................ 73

6 Live Static Metrics Analysis .......... 75
  6.1 Software Metrics ........................ 75
  6.2 Metrics Detection and Calculation ...... 77
  6.3 Live Analysis ........................... 83
  6.4 Git Integration .......................... 84
  6.5 Software Metrics Visualization ........ 85
  6.6 Summary ................................ 88

7 Live Refactorings and Quick Actions Suggestion 89
  7.1 Refactorings ............................. 89
  7.2 Quick Actions ........................... 91
  7.3 Suggestions Visualization ............... 94
  7.4 Summary ................................ 95

8 Empirical Evaluation ................... 97
  8.1 Objectives .............................. 97
  8.2 Guidelines ............................. 98
  8.3 Experience Planning ...................... 99
  8.4 Testing Tasks ........................... 100
  8.5 Tasks’ Evaluation ....................... 102
## CONTENTS

8.5.1 Participants’ Characterization ........................................... 102  
8.5.2 Task 1 ........................................................................ 105  
8.5.3 Task 2 ........................................................................ 109  
8.5.4 Task 3 ........................................................................ 112  

8.6 Usability Evaluation ................................................................. 115  

8.7 Research Question’s Validation ............................................... 117  
8.7.1 RQ1 - Code Quality ......................................................... 118  
8.7.2 RQ2 - Solution’s Convergence .......................................... 121  
8.7.3 RQ3 - Development Time ................................................ 124  

8.8 Validation Threats ................................................................. 126  

8.9 Discussion ........................................................................ 128  

8.10 Summary .......................................................................... 132  

9 Conclusions and Future Work .................................................. 133  
9.1 Summary ........................................................................ 133  
9.2 Main Contributions .............................................................. 135  
9.3 Main Difficulties .................................................................. 137  
9.4 Future Work ...................................................................... 139  

References ............................................................................. 141  

A Participation Consent Declaration ........................................... 151  

B Questionnaire Applied to the Controlled Experience ................. 153
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Software development cycle</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Liveness levels proposed by Steven L. Tanimoto</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>Flowchart obtained by Haibt</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>Flowchart with documentation integrated</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>Mockup that displays the properties of an object in Euclase</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>Circa’s execution</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>Moon code developed on the two IDE versions</td>
<td>20</td>
</tr>
<tr>
<td>3.4</td>
<td>MTBD overview</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>LiveMTBD’s editing history</td>
<td>22</td>
</tr>
<tr>
<td>3.6</td>
<td>Kaluta and Quokka architecture</td>
<td>23</td>
</tr>
<tr>
<td>3.7</td>
<td>Quokka dynamic view</td>
<td>24</td>
</tr>
<tr>
<td>3.8</td>
<td>NaturalMash interface</td>
<td>24</td>
</tr>
<tr>
<td>3.9</td>
<td>Display of a live object</td>
<td>25</td>
</tr>
<tr>
<td>3.10</td>
<td>ExtC visualizer</td>
<td>27</td>
</tr>
<tr>
<td>3.11</td>
<td>Drag-and-drop actions</td>
<td>27</td>
</tr>
<tr>
<td>3.12</td>
<td>Class State View and Refactoring Candidates View</td>
<td>29</td>
</tr>
<tr>
<td>3.13</td>
<td>Code smells analysis in pipes environment</td>
<td>31</td>
</tr>
<tr>
<td>3.14</td>
<td>Wrangler code clone detection</td>
<td>32</td>
</tr>
<tr>
<td>3.15</td>
<td>Code cloning examples</td>
<td>33</td>
</tr>
<tr>
<td>3.16</td>
<td>Code clone detection</td>
<td>34</td>
</tr>
<tr>
<td>3.17</td>
<td>Wizard for the Pull Up refactoring</td>
<td>35</td>
</tr>
<tr>
<td>3.18</td>
<td>Refactoring roles as subclass</td>
<td>36</td>
</tr>
<tr>
<td>3.19</td>
<td>Ad Hoc Refactorings</td>
<td>37</td>
</tr>
<tr>
<td>3.20</td>
<td>Slicing proposed model</td>
<td>38</td>
</tr>
<tr>
<td>3.21</td>
<td>Kobold interface</td>
<td>39</td>
</tr>
<tr>
<td>3.22</td>
<td>LAMBDAFICATOR being performed in batch mode</td>
<td>40</td>
</tr>
<tr>
<td>3.23</td>
<td>LAMBDAFICATOR being performed in quick hint mode</td>
<td>41</td>
</tr>
<tr>
<td>3.24</td>
<td>Mockup of the plugin sketched</td>
<td>42</td>
</tr>
<tr>
<td>3.25</td>
<td>SQUANER results obtained via web</td>
<td>43</td>
</tr>
<tr>
<td>3.26</td>
<td>SQUANER results obtained via email</td>
<td>44</td>
</tr>
<tr>
<td>3.27</td>
<td>Data access visualization via city methaphor</td>
<td>45</td>
</tr>
<tr>
<td>3.28</td>
<td>City metaphor applied in CodeCity</td>
<td>46</td>
</tr>
<tr>
<td>3.29</td>
<td>CodeMetrics’ cyclomatic complexity report</td>
<td>46</td>
</tr>
<tr>
<td>3.30</td>
<td>Code Time daily reports</td>
<td>47</td>
</tr>
<tr>
<td>3.31</td>
<td>SonarLint execution example</td>
<td>48</td>
</tr>
<tr>
<td>3.32</td>
<td>Extract Expression refactoring</td>
<td>50</td>
</tr>
<tr>
<td>3.33</td>
<td>Toogle inline HTML and CSS refactoring</td>
<td>50</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

3.34 Conversion of CSS to object style ........................................ 53
5.1 Tool architecture described by a package diagram .......................... 71
5.2 Architecture of the project used as case study ............................... 72
5.3 Tool developed being executed with the described case study ............. 73
6.1 Live software metrics’ menu ................................................ 86
6.2 Selected metric’s quick pick menu ........................................... 86
6.3 Metric’s evolution overview ................................................... 87
6.4 Metric’s deselection quick pick menu ....................................... 87
6.5 Metric’s evolution chart ....................................................... 87
7.1 Refactoring and quick actions suggestion’s activation ....................... 94
7.2 Refactoring or quick actions’ suggestion using the VSCode “Bulb Linting Provider” 95
7.3 “Problem Panel” with the description of each refactoring or quick action suggested 95
8.1 Gender of the participants .................................................... 103
8.2 Programming’s practice and software system’s understanding ............ 103
8.3 Participants’ knowledge about TypeScript .................................. 104
8.4 Participants’ usage of refactoring tools .................................... 104
8.5 Verification if the participants already used software metrics or refactoring tools . 105
8.6 Checking if the participants were able to understood the Task 1’s metrics 106
8.7 Completion of Task 1 ......................................................... 106
8.8 Task 1 execution’s times ..................................................... 106
8.9 Checking if the participants were able to achieve the Task 1’s goals easily 107
8.10 Tool’s help performing Task 1 ............................................... 108
8.11 Checking if the tool could help performing Task 1, easily and faster ...... 108
8.12 Checking if the participants were able to understood the Task 2’s metrics 109
8.13 Completion of Task 2 ......................................................... 110
8.14 Task 2 execution’s times ..................................................... 110
8.15 Checking if the participants were able to achieve the Task 2’s goals easily 110
8.16 Tool’s help performing Task 2 ............................................... 111
8.17 Checking if the tool could help performing Task 2, easily and faster ...... 112
8.18 Checking if the participants were able to understood the Task 3’s metrics 113
8.19 Completion of Task 3 ......................................................... 113
8.20 Task 3 execution’s times ..................................................... 113
8.21 Checking if the participants were able to achieve the Task 3’s goals easily 114
8.22 Checking if the tool could help performing Task 3, easily and faster ...... 115
8.23 Tool comprehension’s evaluation .......................................... 116
8.24 System’s Usability Scale ..................................................... 116
8.25 Group Statistics’ results example .......................................... 118
8.26 Independent Samples Test’ results example ................................ 118
8.27 Maintainability convergence collected from Task 1 and Task 2 ........... 122
8.28 Maintainability convergence collected from Task 3 ........................ 122
8.29 Times collected by task ..................................................... 125
## List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Refactorings categorized by Martin Fowler</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Some examples of quality metrics</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>Smalltalk, Haskell and VDM-SL comparison</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>SOMETHINGit overview</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>Drag-and-drop refactorings within a Java editor</td>
<td>28</td>
</tr>
<tr>
<td>3.4</td>
<td>Drag-and-drop refactorings within Project Explorer</td>
<td>28</td>
</tr>
<tr>
<td>3.5</td>
<td>Embedded code smells</td>
<td>30</td>
</tr>
<tr>
<td>3.6</td>
<td>Some design patterns, anti-patterns and code smells analyzed by SQUANER</td>
<td>40</td>
</tr>
<tr>
<td>3.7</td>
<td>Live programming tools comparison</td>
<td>55</td>
</tr>
<tr>
<td>3.8</td>
<td>Refactoring tools comparison</td>
<td>55</td>
</tr>
<tr>
<td>3.9</td>
<td>VSCode software metrics tools</td>
<td>56</td>
</tr>
<tr>
<td>3.10</td>
<td>VSCode refactoring tools</td>
<td>57</td>
</tr>
<tr>
<td>4.1</td>
<td>Hypothesis Tests taking into account the different research questions</td>
<td>63</td>
</tr>
<tr>
<td>5.1</td>
<td>System’s user stories</td>
<td>67</td>
</tr>
<tr>
<td>5.2</td>
<td>System’s non-functional requirements</td>
<td>68</td>
</tr>
<tr>
<td>8.1</td>
<td>Task 1’s average time, respective quartiles and deviation values</td>
<td>107</td>
</tr>
<tr>
<td>8.2</td>
<td>Task 2’s average time, respective quartiles and deviation values</td>
<td>110</td>
</tr>
<tr>
<td>8.3</td>
<td>Task 3’s average time, respective quartiles and deviation values</td>
<td>114</td>
</tr>
<tr>
<td>8.4</td>
<td>SUS grading scale</td>
<td>117</td>
</tr>
<tr>
<td>8.5</td>
<td>Hypothesis Tests related to the research question RQ1</td>
<td>119</td>
</tr>
<tr>
<td>8.6</td>
<td>T-Test’s group statistics about task’s lines of code</td>
<td>119</td>
</tr>
<tr>
<td>8.7</td>
<td>T-Test’s about task’s lines of code</td>
<td>120</td>
</tr>
<tr>
<td>8.8</td>
<td>T-Test’s group statistics about task’s maintainability</td>
<td>120</td>
</tr>
<tr>
<td>8.9</td>
<td>T-Test’s about task’s maintainability</td>
<td>121</td>
</tr>
<tr>
<td>8.10</td>
<td>Hypothesis Tests related to the research question RQ2</td>
<td>121</td>
</tr>
<tr>
<td>8.11</td>
<td>T-Test’s group statistics about task’s maintainability to validate RQ2</td>
<td>123</td>
</tr>
<tr>
<td>8.12</td>
<td>T-Test’s about task’s maintainability</td>
<td>124</td>
</tr>
<tr>
<td>8.13</td>
<td>Hypothesis Tests related to the research question RQ3</td>
<td>124</td>
</tr>
<tr>
<td>8.14</td>
<td>T-Test’s group statistics about task’s execution’s time</td>
<td>125</td>
</tr>
<tr>
<td>8.15</td>
<td>T-Test’s about task’s execution’s time</td>
<td>126</td>
</tr>
</tbody>
</table>
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveSD</td>
<td>Live Software Development</td>
</tr>
<tr>
<td>Infoviz</td>
<td>Information Visualization</td>
</tr>
<tr>
<td>UML</td>
<td>UML Unified Modeling Language</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>RQ</td>
<td>Research Question</td>
</tr>
<tr>
<td>WoS</td>
<td>Web of Science</td>
</tr>
<tr>
<td>MTBD</td>
<td>Model Transformation By Demonstration</td>
</tr>
<tr>
<td>WYSIWYG</td>
<td>What You See Is What You Get</td>
</tr>
<tr>
<td>PbD</td>
<td>Programming by Demonstration</td>
</tr>
<tr>
<td>DNDRefactoring</td>
<td>Drag-and-Drop Refactoring</td>
</tr>
<tr>
<td>ROP</td>
<td>Role Object Pattern</td>
</tr>
<tr>
<td>SQUANER</td>
<td>Software Quality Analyzer</td>
</tr>
<tr>
<td>SVN</td>
<td>Subversion</td>
</tr>
<tr>
<td>CDSBP</td>
<td>Class Data Should Be Private</td>
</tr>
<tr>
<td>LPL</td>
<td>Long Parameter List</td>
</tr>
<tr>
<td>RPB</td>
<td>Refused Parent Bequest</td>
</tr>
<tr>
<td>SG</td>
<td>Speculative Generality</td>
</tr>
<tr>
<td>CSWRs</td>
<td>Client-Side Web Refactorings</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
</tr>
<tr>
<td>VSCode</td>
<td>Visual Studio Code</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>SUS</td>
<td>System’s Usability Scale</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Context .................................................. 2
1.2 Motivation ............................................... 3
1.3 Problem .................................................. 3
1.4 Objectives ............................................... 3
1.5 Report Structure ....................................... 4

As programmers write software, it is often necessary to add or change certain functionalities to build the best possible system. The inherent complexity of these systems makes them difficult to understand, and, to improve their comprehensibility, it is often necessary to change the program without modifying its structure, behaviors or objectives [LF95, ND86, MS06].

In this way, changing or adding features in a software system, without knowing the state of that system, may be one of the most difficult tasks software developers have to face. This happens, first of all, because they have to effectively understand the program, but also because they have to change it to include the new desired features without affecting the original functionalities and structure. Many times, programmers think that they are improving a system by reducing his code or simplifying a certain expression, however that does not always happen, because they do not know whether or not the modifications they are making are the best ones or if they are changing the system’s results through that alterations.

Besides, it is often necessary to restructure software systems, without changing their behavior, to effectively improve their structure. This restructuring is called refactoring [MG18, FdR99, MCSW07, SLZ08].

One way is to analyze certain metrics that assess the quality of code, inferring if the modifications performed by a developer have the correct impact on the structure of the respective software. There are already several tools, for different IDEs, that do just that. However, they only allow programmers to check the results of the analysis performed after the code is compiled, manually,
rather than in real-time, when it is done automatically, and therefore there is no instantaneous feedback, since even though there is an analysis of the software metrics [AI 12, SS08], it is only done after the end of the development and not in programming time [MS06].

These main difficulties can be reduced by applying the concept of liveness [ARC+19, Han03] to the analysis of software metrics, giving almost immediate feedback to a developer. Besides, letting developers know how their metrics evolve over different Git commits, might improve their comprehension about the system in general, not just as they are developing it. Furthermore, this information can be presented using information or software visualization techniques to show the metrics calculated.

1.1 Context

Nowadays, there are already some advances in programming that allow for almost immediate feedback when software features are changed. These aspects are also very present in science fiction, where we see characters such as Tony Stark, Bruce Banner or Zuri, from films such as Iron Man\(^1\), Avengers\(^2\) or Black Panther\(^3\), to build diversified systems through virtual environments allowing to live programming their software systems, immediately seeing the result of the implemented features. Many liveness cases performed by these characters were refactoring actions in order to improve their systems without changing their behaviors or goals [FdR99, MGAN17].

This presence in science fiction together with the proof of the results achieved through programming associated with liveness or through refactoring methods and the constant growth of technology, allow for more and more projects about this fields in software engineering.

Increasingly, there is a need to improve a system to reduce the costs associated with its maintenance or modifications, thus reducing the total development time of a software system [MS06, MCSW07, CAGA11]. This requires to diminish or finish the edit-compile-run cycle.

In this sense, this work deals with the live analysis of software metrics, in a programming environment capable of calculating several metrics and also search for refactoring opportunities in real-time, giving immediate and complete feedback to developers about their code, showing the metrics’ evolution through via information visualization techniques, like charts and also showing some suggestions for code improvements, to do certain refactoring, while programming. Furthermore, this project will also allow to analyze and to verify the evolution of software metrics during the Git commits carried out during the software system’s development. These verified values will also be represented by charts that will complement the collected information about the project as it is being programmed. In this way, a software developer can get full and immediate feedback, in an easy way to understanding it, reducing the complexity of his system, increasing its comprehension, maintainability, adaptability and, consequently, its quality and development time.

---

1 Iron Man IMDb, [https://www.imdb.com/title/tt0371746/](https://www.imdb.com/title/tt0371746/), Last access on 16 January, 2019
2 Avengers IMDb, [https://www.imdb.com/title/tt0848228/](https://www.imdb.com/title/tt0848228/), Last access on 16 January, 2019
1.2 Motivation

Programmers, software developers, and software researchers often come across complexity in their software systems. With this large complexity, when they need to change or add a feature to their systems, the comprehension’s effort and the cost and time taken to do the modifications needed, increase [MG18, MS06, CAGA11].

In order to reduce the complexity and increase the quality of the code, enhancing its comprehension and adaptability, it is necessary to be informed about the system structure and status through metrics, for instance, to be able to restructure the system, refactoring it, without changing its behaviors [FdR99]. However, it is also necessary to shorten the cycle of editing, compiling and executing the respective system, so that feedback can be received quickly as it is being programmed, being necessary to include liveness characteristics in the programming actions, having a real-time analysis of the quality of a program through its software metrics.

In addition to these problems, there is still the warning caused by the lack of intuitiveness and usability of the forms that represent the information gathered in terms of the software metrics analyzed.

1.3 Problem

This dissertation seeks to explore whether a tool that includes liveness features and the analysis of software metrics helps to increase the comprehension, maintainability and, consequently, quality of software systems.

Thus, different solutions within the scientific community that approach or explore this main context, such as studies or tools, were researched for. The presence of a vast literature related to the themes under analysis, as well as related to subjects inherent and relevant to the main research, such as the concept of refactoring and software visualization, demonstrates the growing interest in studying solutions to similar problems or even this main problem.

Chapter 4 (p. 59) explores, in more detail, the problem statement, describing the main issues found on the current literature and identifying not only the main hypothesis of this dissertation but also the respective research questions and a solution’s proposal that is described minutely in Chapter 5 (p. 65).

1.4 Objectives

The purpose of this research is to increase and improve the feedback given to software developers relatively to their software systems while programming, checking if it influences the form and the results of their implementations.

Thus, the main research questions explore whether it is possible to improve the code’s quality, creating a good solution and reducing the development time when taking into account the live feedback that they receive about their software metrics.
Introduction

In this way, some solutions and implementations related to the focus of this research were investigated, extending the research field to the adjacent concepts such as information or software visualization and also the potential of Git integration, to be able to improve the feedback given to a developer.

At the end of this research, the project developed should allow to analyze and visualize a group of software metrics and refactoring methods and quick actions’ suggestion that can be executed, as an extra to the main goal, while programming. In addition, it should allow the Git integration, by analyzing the metrics and their evolution throughout the project’s Git commits, giving more complete feedback to developers as they are programming. Thus, a developer can compare the current metrics with the evolution of the system throughout its commits.

1.5 Report Structure

First, this chapter presents an introduction to the research carried out, addressing the context, motivation and its main objectives. Then, Chapter 2 (p. 7), Background, summarizes the concepts that support this research project, namely the topics of Software Engineering, Live Programming, Refactoring, Quality analysis, and Software Visualization, as well as all the sub-themes inherent to them.

Chapter 3 (p. 17), State of the Art, briefly summarizes all the most significant and important areas that are related to the study of live refactoring of software systems, referring the bibliographic findings made through a systematic literature review. In this chapter is also presented a detailed review regarding live programming, refactoring, quality metrics analysis and software visualization, addressing either tools, projects or discoveries related to the topics mentioned. Furthermore, in this chapter, there is a brief analysis of the different software metrics analysis and refactoring suggestions tools provided by the Visual Studio Code.

Chapter 4 (p. 59), Problem Statement, presents the issues that current solutions have, briefly describing a solution proposal to solve them and to reach the main objective of this research project, taking into account some assumptions. Also, it also describes the hypothesis and research questions that support this dissertation.

Then, Chapter 5 (p. 65), Proposed Solution, contextualizes this research project, presenting with more detail the proposed solution already introduced in Chapter 4 (p. 59), describing its objectives, its functional and non-functional requirements, its possible implementation’s steps and architecture, as well as the case study used to validate the implemented solution.

Chapter 6 (p. 75), Live Static Metrics Analysis, contextualizes, in a succinct way, the strategies used to create a live static metric analysis tool that improves the structure and quality of a software system, increasing its comprehension, maintainability, and adaptability.

Chapter 7 (p. 89), Live Refactorings and Quick Actions Suggestion, summarizes the strategies used to create the extras of the tool created, namely the live refactorings and quick actions’ suggestions, that helps to increase a system’s comprehension, maintainability and adaptability.
Introduction

Besides, Chapter 8 (p. 97), Evaluation, presents the evaluation protocol used to validate the tool created and also all the results obtained through them which were analyzed in detail.

Finally, Chapter 9 (p. 133), Conclusions and Future Work, presents the conclusions drawn by the author, as well as the work to be developed in the future.
Introduction
Chapter 2

Background

2.1 Software Engineering .............................................. 7
2.2 Live Programming .................................................. 10
2.3 Refactoring .......................................................... 12
2.4 Quality Analysis ...................................................... 12
2.5 Software Visualization ............................................. 13

In this chapter, we briefly describe and contextualize the most relevant topics approached in this dissertation. We start by presenting the concept of Software Engineering (cf. Section 2.1, p. 7) and then we delve into more specific topics including Live Programming (cf. Section 2.2, p. 10), Refactoring (cf. Section 2.3, p. 12), Quality Analysis (cf. Section 2.4, p. 12) and Software Visualization (cf. Section 2.5, p. 13).

2.1 Software Engineering

Software engineering is an engineering discipline that is concerned with all aspects of software production. The process starts from the early stages of system specification, when the features and behavior of the system are defined, to maintaining the system after it has gone into use [Som10].

2.1.1 SWEBOK Guide

The Guide to the “Software Engineering Body of Knowledge” (SWEBOK Guide) highlights some software engineering subtopics. It should be noted that all subtopics described are relevant steps in the development cycle (cf. Figure 2.1, p. 8) used to build comprehensible software systems and maintainable [SBF14]:

In this chapter, we briefly describe and contextualize the most relevant topics approached in this dissertation. We start by presenting the concept of Software Engineering (cf. Section 2.1, p. 7) and then we delve into more specific topics including Live Programming (cf. Section 2.2, p. 10), Refactoring (cf. Section 2.3, p. 12), Quality Analysis (cf. Section 2.4, p. 12) and Software Visualization (cf. Section 2.5, p. 13).
**Background**

![Software development cycle](image)

**Figure 2.1:** Software development cycle.

**Software requirements.** The requirements area is related to the elicitation, analysis, specification, and validation of the specified requirements as well as the validation and management of these requirements throughout their life cycle.

**Design.** The design activity is part of a software system life-cycle, in which the requirements of such software are transformed into a description or representation of the internal structure, that the program must have according to its requirements describing in this way its architecture and the organization of its components and interfaces.

**Development.** The construction or development of software is related to the creation, more detailed, of the different functionalities of a system, namely through the code, verification, and validation of the same. The verification and validation of the system are done through the development of unit tests, integration tests and debugging actions. The fundamentals of construction reflect the intention to reduce complexity, change’s anticipation, verification, and reuse.

**Testing.** Software testing consists of the dynamic checking of the behaviors of a software system and whether these correspond to what is expected. Normally each behavior corresponds to a set of test cases that can be executed before (using test-driven-development) or after the implementation of each feature. There are several types of tests that can be performed, from the unit tests, where each component is tested as an isolated unit, to the integration tests, that tests the integration of all the system’s components, using various tools and frameworks such as JUnit\(^1\), Protractor\(^2\) or FitNesse\(^3\).

**Maintenance.** A software system must have the ability to be altered and evolved. Many systems to be developed have errors, faults, and defects since it is expected that every 10 lines of code implemented there is 1 bug or error and, therefore, it is necessary to maintain them.

---

\(^1\)JUnit, [https://junit.org/junit5/](https://junit.org/junit5/), Last access on 22 January, 2019

\(^2\)Protractor, [https://www.protractortest.org/](https://www.protractortest.org/), Last access on 22 January, 2019

\(^3\)FitNesse, [http://docs.fitnesse.org/FrontPage](http://docs.fitnesse.org/FrontPage), Last access on 22 January, 2019
Background

Besides, there is sometimes a need to change the system as new specifications or new details are included in the software program. To do this, it is necessary to modify the existing software, maintaining it without altering its integrity.

2.1.2 Challenges

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

Software engineering has a highly collaborative nature, however, this type of engineering does not come without its own set of problems and challenges [CMMDN12]:

Geographical distribution of the team. Often software engineering allows different software project development teams to be located geographically in different locations due to several factors that can affect the success of a project such as cultural differences, timezone differences, working time, communication, technical skills, among others. Also, each activity of the software implementation process has particular needs related to the distributed software development environment and it is difficult to achieve such needs with a distinct geographic distribution [CMMDN12].

Establishment of requirements. The task of setting requirements is one of the most challenging tasks in implementing a software system since these requirements provide knowledge regarding the dominance of the system for programmers and software developers and these requirements can be obtained in a formal and/or informal way. With this set of requirements, developers can understand what the customer’s needs are, so if mistaken requirements are met, the software can fail, because it isn’t under the customer’s wishes [CMMDN12].

Customer/User engagement. Increasingly, customers and users of the software system being developed are increasingly being involved in this implementation as it increases the customer’s trust in the project being created. Since the software is built according to the customer’s choice, it turns out to be more involved in the software. This is complicated to obtain because there are still teams that aren’t accustomed to this involvement of elements external to the project, but that have importance in the same, because they will be the end users of the system to be created [BT14, PM12].

Estimating cost and time of development. Typically, before software system development begins, software engineers or programmers try to estimate the cost and time required to implement the project, according to the requirements and complexity of the software in question. In this case, the challenge is to build the software successfully within the budget and latency stipulated. This phase is, therefore, a complicated phase, as there is a need to estimate the
cost and time of each task, before implementing it and sometimes there may be errors in the estimation that can harm the project as a whole [BT14].

Testing. The testing phase of software allows providing information about a software system’s reliability and quality. With the tests developed it is possible to evaluate the system by checking whether it meets the requirements specified by the client or not, since each test is used to execute the project and to discover any errors, failures or lack of requirements [BT14].

Quality. To obtain quality, software needs to meet the requirements established by its customer. So, if a program has implemented all the specifications established in an accurate, flexible and portable way, it has high quality for its customer. However, sometimes, this is hard to achieve [BT14].

2.2 Live Programming

Live Programming consists of the possibility that a programmer can see the results of his software system as the program is being programmed. Through this live development, we can reduce the edit-compile-run cycle, thus reducing the time required to build the system, since the implemented changes are noticed immediately [Gol04]. This concept can also appear at different stages of the software development cycle [SBF14] and not only in the implementation step by itself, when the user only develops code. Sometimes liveness can be applied in a debug stage [Dom18] and also in combined steps such as the implementation and maintenance stages [Lou18, Ama18], to give more complete feedback to developers.

2.2.1 Motivation

The use of liveness associated with programming and software engineering is motivated by the following four points [Tan13]:

1. Decreasing the latency between software development and verifying the result of that development;

2. Enhancing developers real-time experience;

3. Simplifying the analysis and correction of errors, bugs or failures;

4. Improving the comprehension of a software system by making software visualization more live.

2.2.2 Liveness Levels

In 1990, Steven L. Tanimoto proposed a hierarchy with four different liveness levels [Tan90]. Level one was only concerned with the use of ancillary information methods, such as flowcharts. In level two, programmers develop their software system, ask for a response and shortly afterward
Background

they can check the results obtained through it. At level three, the computer waits for the developer to program and, shortly thereafter, shows the results obtained. In level four, the computer does not expect the programmer to develop his system since it is running the program even though it is still being created, showing the programming results immediately [Tan90].

Many current programs already take into account the first four levels of liveness. However, in 2013, Tanimoto presented a hierarchy with six levels [Tan13], the four levels described before plus an extra two levels. This new hierarchy is represented by Figure 2.2.

![Figure 2.2: Liveness levels proposed by Steven L. Tanimoto. [Tan13]](image)

In the first of these two new levels, level five of liveness, the computer not only executes the software while it is being programmed but also foresees multiple next actions of the programmer, executing one or more versions resulting from these predictions, staying a step ahead of the software developer. This level five can be called “Tactically Predictive”, once it discovers different versions of a program that a developer may be interested in. In the last level of the liveness hierarchy, level six, predictions of the next actions by the programmer are strategically analyzed, in order to cover, for example, all the behaviors of a large software unit [Tan13].

2.2.3 Problems with Live Programming

There are some criticisms related to using liveness when programmers are developing software systems. The first one is related to programs that run in a short period and then end. Even if a system runs over a long time, a change made to the system may not be immediately visible, since execution may already have passed through that code location. Finally, another criticism pointed out is that live programming requires a lot of computational effort, spending immense resources on a system, since editing, compiling and executing it happen simultaneously [Tan13].
All the criticisms pointed out have a solution. The first problem can be solved by editing the execution settings of a program using the auto-repeat mode, so that the system can run countless times until it will be stopped by the programmer, verifying all the new changes implemented in software. A programmer can also use breakpoints, “start section” or “end section” locations where the code has changed so that the respective modification is guaranteed to be executed. The last problem can be solved through new and more modern computers that allow to edit and run a program in parallel without apparent difficulties [Tan13].

2.3 Refactoring

Increasingly, software systems are more complex and difficult to understand, reducing their maintainability. Changing or adding a feature to a system that has little or no capacity to be maintained or adapted increases the total time needed to develop a program and also increases its development cost [MS06, MCSW07].

In order to change a software system, it is necessary to restructure it, doing refactoring, without changing its current behaviors and objectives [FdR99].

Thus, refactoring is the process of modifying or cleaning a software system or program, improving its internal structure and increasing its quality, without changing its behaviors, thus creating a cleaner and self-explanatory program [FdR99, MHB08, CAGA11].

Through refactoring methods we can move fields from one class to another, extract code from a method by creating a new method or move methods within the hierarchy, to improve a software system [FdR99].

There are different methods of refactoring that change a software system for better. Table 2.1 (p. 13) summarizes some of these refactoring types as categorised by Martin Fowler [FdR99].

2.4 Quality Analysis

The quality of a software system relates to the fact that in the end, it agrees with its requirements [ISO, SS08, Al 12]. At the quality level of a project, eight characteristics should be taken into account when developing software [ISO]. These characteristics are related to functional adequacy, effectiveness, compatibility, usability, reliability, safety, maintenance and portability [ISO, Al 12], which will be described below.

Functional Suitability. Degree to which a software system, under certain conditions, is according to the specified requirements and objectives.

Performance Efficiency. Performance of a software system compared to the number of resources used.

Compatibility. Degree to which one system can exchange information with another system, performing its functions while sharing its hardware and/or software.
Table 2.1: Refactorings categorized by Fowler, Martin. [FdR99]

<table>
<thead>
<tr>
<th>Refactorings Catalog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change function declaration Move statements into function Replace exception with pre-check</td>
</tr>
<tr>
<td>Change reference to value Parameterize function Replace function with command</td>
</tr>
<tr>
<td>Change value to reference Preserve whole object Replace inline code with function call</td>
</tr>
<tr>
<td>Collapse hierarchy Pull up constructor body Replace loop with pipeline</td>
</tr>
<tr>
<td>Combine functions into class Pull up field Replace magic literal</td>
</tr>
<tr>
<td>Combine functions into transform Pull up method Replace nested conditional with guard clauses</td>
</tr>
<tr>
<td>Consolidate conditional expression Push down field Replace parameter with query</td>
</tr>
<tr>
<td>Decompose conditional Push down method Replace primitive with object</td>
</tr>
<tr>
<td>Encapsulate collection Remove dead code Replace query with parameter</td>
</tr>
<tr>
<td>Encapsulate record Remove flag argument Replace subclass with delegate</td>
</tr>
<tr>
<td>Encapsulate variable Remove middle man Replace superclass with delegate</td>
</tr>
<tr>
<td>Extract class Remove setting method Replace temp with query</td>
</tr>
<tr>
<td>Hide delegate Remove subclass Replace type code with subclasses</td>
</tr>
<tr>
<td>Inline class Rename field Return modified value</td>
</tr>
<tr>
<td>Inline function Rename variable Separate query from modifier</td>
</tr>
<tr>
<td>Inline variable Replace command with function Slide statements</td>
</tr>
<tr>
<td>Introduce assertion Replace conditional with polymorphism Split loop</td>
</tr>
<tr>
<td>Introduce parameter object Replace constructor with factory function Split phase</td>
</tr>
<tr>
<td>Introduce special case Replace control flag with break Split variable</td>
</tr>
<tr>
<td>Move field Replace derived variable with query Substitute algorithm</td>
</tr>
</tbody>
</table>

**Usability.** Degree to which a user can use the software system, understanding his objectives and managing to execute everything as expected.

**Reliability.** Degree that verifies if a software system performs correctly, under specified conditions for a certain amount of time.

**Security.** Degree if the system’s information and data are protected from external threats or illegal data access, taking into account the different authorization levels.

**Maintainability.** Effectiveness and efficiency in which a program can be adapted to meet new requirements and new environments in which it is inserted.

**Portability.** Effectiveness and efficiency in which a program can be transferred between hardware or software components.

There are still some metrics that allow analyzing a system statically, evaluating its quality, according to CodeMR[^4]. This metrics are presented in Table 2.2 (p. 14).

### 2.5 Software Visualization

Software visualization is an extension of information visualization that began to gain prominence in the eighteenth century, through the introduction of representative charts or diagrams of data related to scientific studies. So, since a software program is a set of lines of code and data, it can be represented using information visualization methods [Die07].

Background

Table 2.2: Some quality metrics that can be analyzed to see if a software system is well designed and structured.

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

More specifically, software visualization is used as a form of reverse engineering, since Price et al. [PBS93], in 1993, defined it as such. However, despite being an old approach, it is still used by several researchers in this main field [WLR11, FLC+02], since it is considered an interesting way that allows the analysis of the structure, behavior, and evolution of a software system [Die07].

The first steps in the foundation of software visualization date back years before 1980, through the construction of diagrams, as already mentioned. It was in 1959 when Haibt [Hai59] decided to investigate further on this topic and developed a system that allowed the creation of flowcharts of Fortran or Assembly’s software systems (cf. Figure 2.3).

![Flowchart obtained by Haibt. [Hai59]](image)

With this initial impulse given by Haibt [Hai59], three years later, Knuth [Knu63], implemented a similar system but, this time, he also integrated the documentation of the source code to generate the respective flowcharts (cf. Figure 2.4, p. 15).
Years later, in 1980, the software visualization investigation changed its course and began to focus on the behavior of the program and not only on its source code. This way, Baecker [Bae98], in 1981, created the movie “Sorting Out Sorting”\(^5\) that gave visual information to programmers regarding algorithms such as Quick-Sort, Bubble-Sort or Shell-Sort, thus facilitating the comprehension of such algorithms.

Since then, much research is being done encompassing other aspects such as 3D visualization, like Reiss [Rei95], in 1995, or Young and Munro [YM98a], in 1998. Furthermore, software visualization started to be coupled with components or stages of software engineering, namely the refactoring step, assisting the refactoring suggestions that many tools or IDEs already provide [MHB10].

Then, through these visual representations, it became possible to tell several stories in a more perceptible way to those who are analyzing or reading them [FLC02, HCL05].

---

\(^5\) Sorting Out Sorting IMDb, https://www.imdb.com/title/tt0210301/, Last access on 13 June, 2019
Background
Chapter 3

State of the Art

This chapter discusses the different and most relevant tools and studies about the concepts related to the main objective of this research.

Initially, the state-of-the-art of each concept is analyzed, taking into account the tools, studies or proposed solutions collected through a systematic literature review, including the concepts of liveness (cf. Section 3.1, p. 18) and refactoring (cf. Section 3.2, p. 26). Then, we analyze some tools or studies focused on program’s quality (cf. Section 3.4, p. 39) and software visualization (cf. Section 3.5, p. 42), that were collected in the early stages of the systematic literature review done as complementary articles to the main study. Finally, we report some tools available in the Visual Studio Code Marketplace (cf. Section 3.6, p. 45) that are related to the concepts of refactoring and software metrics’ analysis, to improve a software system’s quality.

At the end of this chapter, (cf. Section 3.7, p. 54), there is a summary of the topics covered as well as a brief analysis of the various tools described and the conclusions drawn from this analysis.
3.1 Live Software Development

Liveness can be applied in software systems in different ways, namely through programming languages, development environments or tools. In the next subsections, we describe some of the uses that already exist related to liveness or live programming.

3.1.1 Live IDEs

**Euclase.** Live integrated development environment that allows the implementation of interactive Web Applications. Through this IDE, a user can immediately see the results of his changes in a program to increase his comprehension, increasing the speed of software evaluation and decreasing the total development time needed to create the program (cf. Figure 3.1). Euclase combines constraints and finite state machines in order to specify the different software’s behaviors [OMB13].

![Figure 3.1: Mockup that displays the properties of an object in Euclase. [OMB13]](image)

This IDE was designed to allow a programmer to develop his system in a few lines of code as possible, to be user-friendly so that everyone can understand the development environment and to allow a live development of a software system. For instance having the expression X:=Y, the user can immediately check that the value of parameter X is equal to the value of Y and if we change the Y’s value, the value of X will be equal to the value of Y after the respective modification [OMB13].

In addition to what has already been mentioned, Euclase was developed using HTML and JavaScript. As a constraint solver, a tool named ConstraintJS [OMB12] was used which solves constraints in JavaScript code.

Euclase had to overcome several challenges while it was developed, dealing with problems such as changing variable references, since not only the reference value can be changed but also its reference, while the user is modifying the source code. Besides, during the implementation of Euclase, it was necessary to deal with finite state machine transitions since event listeners should be kept as synchronous as the value of a variable is modified [OMB13].

Euclase also considers performance, trying to make every modification evaluated as quickly as possible. However, when too many changes are inserted at the same time, they are only displayed after the last change is implemented so that if there are 100 modifications, for instance, the program is not executed 100 times and only once when the hundredth change is introduced [OMB13].

**Circa.** This is a language and live platform that creates and manipulates software code, to define live programs. Circa is based on back-propagation algorithms that allow a programmer to manipulate his program, expressing his desires and goals related to the results of his software.
State of the Art

Through this IDE, a user can click on the drawing so that he can inspect and verify the code so that he can change it. So when there is a click on the scene the first thing that is done is a check of which was the call that is associated with the position of the mouse. For example, in Figure 3.2 the mouse is clicked on the smaller tree and when this click exists it is verified the call to the `draw_sprite()` method that is associated with the mouse position. In this case, the IDE observes all the calls made to the `draw_sprite()` method and checks the position of each sprite, compared to the respective position of the mouse [Fis13].

![Figure 3.2: Circa’s execution. [Fis13]](image)

The Circa IDE allows to visualize the developed code through its textual form, but also through a diagram since it is considered that a programmer can correctly understand a system using either a textual form or a visual form of its code. After visualizing the filtered code, a user may want to modify it directly through the diagram presented, which is only possible, because the stack contains links to the previously compiled source code [Fis13].

In this way, the Circa IDE allows the verification and edition of a software system in real-time since it allows a developer to discover the code behind of a certain result obtained, through a click and, also, because it allows modifying the code through its visual form [Fis13].

**Moon.** This IDE is still being developed to support the Moon language, having native support to give live feedback to programmers [LL13].

So far two different prototypes have been developed. The first (option (A)) is directly integrated with Pharo1, a Smalltalk’s open-source implementation, used to create the first compiler version for Moon. The second prototype (option (B)) is a web-based environment, dissociated from the compiler, developed through JavaScript, using HTML5 for its front-end. Regardless of the IDE version, it immediately reacts to the program that is being developed, providing visual feedback relative to the individual entities, system states, and their evolution [LL13]. This two Moon prototypes can be verified in Figure 3.3 (p. 20).

Although this IDE is not yet finalized, it is expected that the components of a program can be represented through visual forms, in a human-perceptible format, in which a programmer can
State of the Art

manipulate these visual forms, creating representations of entities and then creating new components. So, with this, a user can create the desired format by creating code while mixing shapes and components [LL13].

Regarding the feedback that can be given about the state of a system, in the first prototype developed using Pharo, parts of the code that are not compiled are underlined to warn the programmer. In the case of the second prototype, colors are used to distinguish the different possible states (green for correctly compiled code, yellow for modified code and red for errors) [LL13].

![Figure 3.3: Moon code developed on the two IDE versions. [LL13]](image)

3.1.2 Live Languages

Circa [Fis13] and Moon [LL13] are development environments and also programming languages used by these IDEs.

The language Circa is a simple and intuitive language, very similar to other languages that already exist, such as Java, JavaScript, C#, among others, allowing a programmer to understand what he is developing. The language Circa together with its IDE allows doing the live development of software systems, being able to edit code through a visual representation [Fis13]. Examples of implementation of this language can be found in (cf. Figure 3.2, p. 19).

The Moon language as its development environment is still being developed and the first phase of its creation was related to implementing different numerical primitives and then several mathematical operations. In a second phase was added the possibility of a programmer to create his own functions, as in a typical programming language. Thus, it was possible to make components that could use other components, being able to perform more complex mathematical operations. This language along with its IDE allows a user to immediately receive the results of his programming, namely, mathematical operations, receiving feedback from his software system [LL13]. Examples of this language can be found in Figure 3.3.
3.1.3 Live Tools

**SomethingGit.** Smalltalk live library that associates dynamic Smalltalk and static Haskell (GHCi) [HHPJW07] and VDM-SL (VDMJ) [FL98, AL99] code. Smalltalk is a prominent programming language, development environment or library that allows doing pair programming which is a concept used in eXtreme Programming [ONY13].

SOMETHINGGit allows a programmer to create, even if incomplete, using live programming, mathematically sound programs. With this library, a programmer can develop a Haskell code system among Smalltalk code and also evaluate the Haskell code specified through a GUI and another system can execute a VDM function of the Smalltalk code. Thus, SOMETHINGGit allows developing code in Haskell, VDM-SL or Smalltalk, providing interactive and flexible interfaces to APIs and GUIs for both Haskell and VSM-SL [ONY13]. These three languages are distinct, as can be seen in the Table 3.1. Moreover, Table 3.2 describes SOMETHINGGit operating environment.

Table 3.1: Smalltalk, Haskell and VDM-SL comparison. [ONY13]

<table>
<thead>
<tr>
<th>Name</th>
<th>Smalltalk</th>
<th>Haskell</th>
<th>VDM-SL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Style</td>
<td>Dynamic environment</td>
<td>Compiler + REPL interpreter</td>
<td>IDE with a REPL interpreter</td>
</tr>
<tr>
<td>Paradigm</td>
<td>Object-Oriented Programming</td>
<td>Functional Programming</td>
<td>Formal Methods</td>
</tr>
<tr>
<td>Type system</td>
<td>Dynamic typing</td>
<td>Static typing</td>
<td>Static typing</td>
</tr>
<tr>
<td>UI</td>
<td>GUI</td>
<td>CUI</td>
<td>GUI (CUI interpreter)</td>
</tr>
<tr>
<td>Moto</td>
<td>Dynamism</td>
<td>Purity</td>
<td>Rigor</td>
</tr>
<tr>
<td>Code base</td>
<td>Image</td>
<td>File</td>
<td>File</td>
</tr>
<tr>
<td>Runtime</td>
<td>IDE = Runtime</td>
<td>Binary code</td>
<td>Animation on IDE</td>
</tr>
</tbody>
</table>

Table 3.2: SOMETHINGGit overview. [ONY13]

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Linux or Mac OS X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smalltalk System</td>
<td>Squeak 4.3 or higher ; Pharo 1.4 or higher</td>
</tr>
<tr>
<td>Required Smalltalk Packages</td>
<td>OSProcess</td>
</tr>
<tr>
<td>Haskell interpreter</td>
<td>GHCi 7.4.1 or higher</td>
</tr>
<tr>
<td>VDM-SL interpreter</td>
<td>VDMJ 2.0.1</td>
</tr>
</tbody>
</table>

**LiveMTBD.** Eclipse plugin that allows the live transformation of a model. By using the Model Transformation By Demonstration (MTBD) technique, LiveMTBD also allows the user to specify the desired transformation, a centralized repository to assist in the sharing of transformations and the live model transformation engine that suggests the application of certain transformations during model-edit time [SGW+11].

The central idea of MTBD is to ask a user to use a concrete template and apply transformations and changes directly on the template provided, rather than manually writing the template transformation rules. These transformations are recorded and are automatically inferred in a pattern that summarizes all that process (cf. Figure 3.4, p. 22) [SGW+11].
State of the Art

The LiveMTBD tool thus consists of 3 new components applied to the original MTBD (cf. Figure 3.4). These new components include new competencies to improve the specification, sharing and re-owning of a model working pattern, taking into account the original MTBD framework [SGW+11].

![MTBD overview with new components implemented by LiveMTBD](image)

Figure 3.4: MTBD overview with new components implemented by LiveMTBD. [SGW+11]

In this way, a live demo was implemented in which the recording engine writes all editions applied to the model, in the editor. If a user needs to check a particular modification applied, he can go back to the recording view and check all operations performed for a particular modification (cf. Figure 3.5) [SGW+11].

Moreover, since transformation patterns are also automatically stored in a centralized pattern repository chosen by the user, it is possible for other users to have access to these patterns, so that they can use them, allowing a collaboration live environment between users. In this way, users can learn from each other [SGW+11].

![Checking the editing history with LiveMTBD](image)

Figure 3.5: Checking the editing history with LiveMTBD. [SGW+11]

**Quokka.** A live tool that allows interactions visualization in Kaluta emulations (enterprise software emulator capable of representing large-scale environments). This tool can increase the comprehension of emulations in general [HSHV12]. The architecture of Quokka and Kaluta can be analyzed in Figure 3.6 (p. 23).
Kaluta is more specifically a prototype of an emulator for an enterprise software environment, representing large-scale, heterogeneous environments, using only a single physical host. Through this prototype, a user can specify the behavior of an endpoint type, such as SOAP Web Services or BitTorrent peers. Kaluta’s architecture has two main components: the engine and the networking interface. The first one is intended for concurrent execution of several endpoints specified by the user while the network interface must communicate between the external software systems and the models or endpoints being executed by the engine. Also, this interface allows to not only visualize the Quokka data but also allows to check a set of improvements that can be made, based on live emulation events that have occurred [HSHV12].

Figure 3.6: Kaluta and Quokka architecture. [HSHV12]

Quokka allows for dynamic and static visualizations of the Kaluta live emulations that were recorded. Dynamic visualization lets a user see the emulation endpoints and their interactions with external software systems (cf. Figure 3.7, p. 24). External systems are represented by circles in the center of the dynamic view and endpoints are shown by rectangles. The open communication channels are visualized through circles in the middle of the endpoint’s rectangles. The transmission of messages is represented by directed triangles depending on the direction of the message [HSHV12].

In short, Quokka has a high-level dynamic view to visualize live emulations, allowing a user to see what interaction is occurring at that time and in which endpoints are occurring, but also what interactions have occurred in the past. Thus, this tool benefits a QA team that tries to understand the behavior of a software system. In addition, Quokka provides additional graphics that summarize key emulation features that enable the QA team to more easily identify scalability and performance issues [HSHV12].

**NaturalMash.** Live mashup tool based on live programming. This tool can update the output of the mashup as the data sources are live updated. A mashup is a lightweight type of Web Applications composed of content and functionality, which can be reused and which are present on the Web [AP12, AP13].
NaturalMash [AP12] is based on the natural programming language in which What You See Is What You Get (WYSIWYG) since it is easier to understand and learn. This tool has an intuitive environment composed by 4 main elements as the ones that are represented in Figure 3.8: (i) visual field, where the modifiable output perspective is shown, (ii) text field, where the natural mashup language is found and can be edited, (iii) component dock, where a list of mashup components that are being modified is presented and (iv) stack, with the component library, created mashups and so on [AP13].

This tool also supports synchronized interactions with multiple perspectives, since component dock, text field and visual field are always synchronized in all the user interactions (eg, when there is mouse over an icon of a component in the dock, it is highlighted its corresponding widgets in the WYSIWYG output and corresponding description in the text field). NaturalMash also allows to use Programming by Demonstration (PbD) [CHK+93] in the visual field, getting suggestions in the text field (e.g., click event) [AP13].

Biegel, Benjamin et al. [BLD15] introduces a new tool that represents Java objects that can
be live visualized and manipulated. This tool is a Web Application, which runs separately, for example, on a tablet, allowing a user to modify the state of an object or invoke methods and observe the evolution of the system. Thus it is permanently linked to an IDE, visually representing the instances of all classes and their attributes. When updating an object, the tool changes the source code, while maintaining the current state of the object, thus allowing manipulation and observation of the current states of each object (cf. Figure 3.9) [BLD15].

In this case, a plugin for IntelliJ IDE was developed to control the communication between the respective IDE and the Web Application. This plugin developed uses the Java reflection API, instantiating new objects, manipulating its state and invoking its methods. Thus, when an object is changed or added in the Web Application, it is propagated to the source code, immediately [BLD15].

![Figure 3.9: Display of class “Motorcycle” live object and the interface which allows to create new objects. [BLD15]](image)

### 3.1.4 Discussion

Live programming allows a programmer to check the results of his software system as he programs, reducing software development time by reducing or eliminating the edit-compile-run cycle. Moreover, once the execution is carried out as it is being programmed it is possible to obtain feedback much more quickly, being able to improve it earlier, reducing the cost associated to the changes that would be necessary to do at the end of the system development.

There are several tools, languages or development environments that allow a programmer to make live software development, according to the basic principles of liveness. However, what
already exists does not allow a software developer to be able to receive complete feedback about bugs or code smells, for instance.

In addition, it has also been found that many of these live tools, languages or IDEs are still in a prototyping phase and have as main objective to increase the programmer’s comprehension about his software system, not yet integrating the new levels of hierarchy (level 5 and 6) proposed by Tanimoto, Steven L. [Tan13].

3.2 Refactoring

As already described, refactoring consists of restructuring a software system without modifying its behaviors already implemented. Nowadays there are enumerated refactoring tools or proposals. Some of them will be listed below.

3.2.1 Refactoring Tools

ChEOPSJ. Tool which analyzes refactoring masking in Java systems, working in Eclipse background. This tool checks if some refactoring methods are masked by other types of restructurings, using a change-based approach [SPDZ15].

ChEOPSJ records silently the actions of a developer and registers all the changes made while the developer is working. This record is then analyzed to see if any refactoring has been “masked” by another type of change. It uses FAMIX model [Dem] associated with Groove [Ren04]. FAMIX captures object-oriented programming languages, defining entities that represent packages, classes, methods, attributes, invocations, among others [Dem, SPDZ15]. This tool is capable of analyzing 11 different refactoring types which are listed below [SPDZ15].

1. PullUpMethod;
2. PullUpField;
3. PushDownMethod;
4. PushDownField;
5. MoveClass;
6. MoveMethod;
7. MoveField;
8. RenamePackage;
9. RenameClass;
10. RenameMethod;
11. RenameField.

ChEOPSJ was compared to RefFinder [KGLR10], an existing tool that allows the reconstruction of refactorings executed in a software system and it was found that RefFinder failed to reconstruct some refactoring methods, especially when there were combinations of Extract Method and Move Method, whereas ChEOPSJ could detect and reconstruct these refactorings [SPDZ15].

ExtC. Eclipse plugin that helps a programmer to understand the structure and the interactions within a Java class and consequently of its project, evaluating the cohesion, using techniques and algorithms of clustering [CAGA11]. Clustering allows identifying data’s subsets representing coherent concepts [JMF99]. In this case, these data clustered are the members of the classes of the software system, grouping them according to their affinity. With the use of Extract Class
refactoring, it is possible to create simpler classes, but with better-defined objects. This tool only uses the Extract Class refactoring method [CAGA11].

By better understanding its program, a programmer can make better decisions about the refactorings he should apply. To collect the ExtC metrics, namely, the system cohesion, it is used the Eclipse Metrics2 plugin\(^1\), and then the data is saved to a database. Graphs are drawn using the JUNG [OB03] framework [CAGA11].

ExtC also provides a GUI with different views of the different classes of the software system being analyzed (cf. Figure 3.10). The metric view provides the tabular display of the metrics analyzed in each class. Dependency graph view allows a user to explore relationships between members of each class. The agglomerated clustering view and the “betweenness” clustering view allow visualizing the clustering algorithms used and how they are applied [CAGA11].

![Figure 3.10: ExtC visualizer. [CAGA11]](image)

**DNDRefactoring.** Tool that allows to directly manipulate a software system through drag-and-drop gestures (cf. Figure 3.11), eliminating the need to use interfaces or dialog boxes. This tool supports 12 of 23 types of Java refactoring in the Eclipse IDE. Thus, it is not necessary to know the names of the refactorings nor to know specifically what they do. With the drag-and-drop movements, the process becomes more intuitive [LCJ13].

![Figure 3.11: Drag-and-drop actions (a) Extract Method example. [LCJ13]](image)

\(^1\)Eclipse Metrics2 Plugin - [http://metrics2.sourceforge.net](http://metrics2.sourceforge.net), Last access on 20 January, 2019
State of the Art

With this kind of refactorings, we can drag a source and drop into the desired target, invoking and configuring the refactoring methods through a simple step and movement. Thus, with this type of gesture, it is not necessary to know in advance what refactorings exist, what their names and objectives are since many of the nomenclatures given to refactorings by Martin Fowler are not used by different IDEs. For example, Fowler listed the Extract Method method while in the NetBeans IDE this method is called Introduce Method. Furthermore, Eclipse joins the Extract Superclass together with the Pull Up methods, since they work in the hierarchy of a system, but IntelliJ joins Extract Superclass with other extract-based restructurings and Pull Up refactorings in another collection [LCJ13].

Also, DNDRefactoring, for the Eclipse IDE, supports the execution of refactorings using drag-and-drop either in the Java code editor (cf. Table 3.3, p. 28) or in the Package Explorer and Outline views (cf. Table 3.4, p. 28), where the project hierarchy and all source files are checked.

Table 3.3: Drag-and-drop refactorings within a Java editor. [LCJ13]

<table>
<thead>
<tr>
<th>Drag Source</th>
<th>Drop Target</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local variable</td>
<td>Declaring type</td>
<td>Promote local variable to field</td>
</tr>
<tr>
<td>Expression inside method</td>
<td>Same method between current method signature brackets</td>
<td>Extract temp variable</td>
</tr>
<tr>
<td></td>
<td>Declaring type</td>
<td>Introduce parameter</td>
</tr>
<tr>
<td>Statements in method</td>
<td>Declaring type</td>
<td>Extract method</td>
</tr>
<tr>
<td>Non-static method</td>
<td>Field variable in declaring type</td>
<td>Move instance method to field type</td>
</tr>
<tr>
<td></td>
<td>Argument type in current method signature</td>
<td>Move method to argument type</td>
</tr>
<tr>
<td>Static method of field</td>
<td>Another type in current editor</td>
<td>Move member to target type</td>
</tr>
<tr>
<td></td>
<td>Field variable in declaring type</td>
<td>Move member to target type</td>
</tr>
<tr>
<td></td>
<td>Local variable type in declaring type</td>
<td>Move member to local variable type</td>
</tr>
<tr>
<td>Anonymous class</td>
<td>Declaring type</td>
<td>Convert anonymous to nested type</td>
</tr>
</tbody>
</table>

Table 3.4: Drag-and-drop refactorings within Project Explorer. [LCJ13]

<table>
<thead>
<tr>
<th>Drag Source</th>
<th>Drop Target</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-static Method</td>
<td>Type of field variable in declaring type</td>
<td>Move instance method to target field type</td>
</tr>
<tr>
<td></td>
<td>Type</td>
<td>Pull-up, Push-down or Move method to target type</td>
</tr>
<tr>
<td>Nested Type</td>
<td>Package</td>
<td>Move nested type to new file + Move type to target package</td>
</tr>
<tr>
<td>Anonymous Type</td>
<td>Type</td>
<td>Convert anonymous to nested type</td>
</tr>
<tr>
<td></td>
<td>Package</td>
<td>Convert anonymous to nested type + Move nested type to new file + Move type to target package</td>
</tr>
<tr>
<td>Field</td>
<td>Type</td>
<td>Pull-up, Push-down or Move field to target type</td>
</tr>
<tr>
<td>Static Members</td>
<td>Another type declared in current editor</td>
<td>Move members to target type</td>
</tr>
<tr>
<td></td>
<td>Type of field variable in declaring type</td>
<td>Move members to target field type</td>
</tr>
<tr>
<td></td>
<td>Type of local variable in declaring type</td>
<td>Move members to local variable type</td>
</tr>
<tr>
<td>Nested Type</td>
<td>Package</td>
<td>Move nested type to new file + Move type to target package</td>
</tr>
<tr>
<td>Non-static fields</td>
<td>Package</td>
<td>Extract data class + Move type to target package</td>
</tr>
<tr>
<td>Non-static methods</td>
<td>Package</td>
<td>Extract interface</td>
</tr>
<tr>
<td>Static and non-static methods</td>
<td>Package</td>
<td>Extract super class</td>
</tr>
</tbody>
</table>

c-JRefRec. Eclipse tool that identifies Move Method refactoring opportunities in Java code to correct Feature Envy smells. Feature Envy smell is a classical code smell that occurs when a method is “more interested in a class rather than the one it actually is in”. More specifically,
these smells are checked when a method of a certain class overused attributes or data from another class [FdR99]. The suggestion of this refactoring type is based on 4 heuristics analyzed statistically and semantically [UOII17]:

1. Number of methods defined in a class, excluding abstract methods;
2. Total number of incoming or outgoing edges connected between members of a class;
3. Number of classes using methods or fields of a given class;
4. Number of classes whose methods or fields are used by methods of a given class.

c-JRefRec allows a user to access two views: Class State View and Refactoring Candidates View (cf. Figure 3.12). The first one, the Class State View, checks the current state of a class by referring to the cohesion that exists in that class after the changes have been made. The second one, the Refactoring Candidates View, presents Move Method refactoring opportunities [UOII17].

WebScent. Tool that detects embedded code smells in Web Applications, namely in PHP server code. Embedded code smells violate basic principles like modularity or concerns’ separation, causing great maintenance efforts [NNN+12].

First, it detects these types of code smells in the created code and then locates them on the server-side, mapping it between the client-side snippets and their location on the server. This tool identifies and detects 6 different problems such as mixed code for HTML, CSS or JavaScript, duplicated JS code on the server-side and HTML errors that can create embedded smells [NNN+12]. These code smells are listed in the Table 3.5 (p. 30).

Stolee, K. T. et al. [SE11] created a refactoring tool that focuses on identifying code smells for pipe-like mashups on Yahoo! pipes environment. This tool analyzes code smells and suggests refactorings that can solve these smells, to reduce the complexity of a program by increasing its abstraction, using an interface to view the status of pipes and which refactoring methods can be applied, taking into account the verified code smells (cf. Figure 3.13, p. 31) [SE11].
With this tool, a programmer can find multiple code smells related to a pipe environment, such as the ones listed below [SE11].

**Laziness Smells** - This type of code smell is related to the smell “Lazy Class”, which identifies classes that do not do enough actions [FdR99]. In this case, “Laziness Smells” identifies pipes containing modules or fields that do not collaborate to the pipe’s output and, consequently, to the system output.

**Redundancy Smells** - Duplicate code is one of the worst code smells, often introduced into a software system by even experienced programmers. The “Redundancy Smells” consists of strings, modules or sequences of modules that are duplicated in a pipe. With this duplication, the code becomes more complex and difficult to understand and maintain.

**Environmental Smells** - This code smell is related to the identification of pipes that were not updated when the external environments were modified. This can occur when a module or field exists that is no longer sustained by pipes’ proper language or references an invalid external element.

**Population-Based Smells** - This type of code smell does not focus individually on each pipe, but rather checks the repository where the system is inserted, to discover patterns in the different modules that were used and reused by the pipes. These pipes can be more complicated to understand, making maintenance difficult.

**Li, Huiqing** [LT11] created a tool that detects similar code or cloned code in the context of Erlang programs [CT09]. This tool detects and incrementally eliminates clones, making it possible to use them interactively in large programs. The detection and elimination of this code smell are integrated with Wrangler [LTOT08], which is an existing refactoring tool for Erlang systems and which uses the analysis of ASTs (cf. Figure 3.14, p. 32) [LT11].

This tool can be integrated into 2 distinct IDEs, Emacs or Eclipse, but can also be executed through the command line. To be able to execute it is necessary to specify 5 parameters if not, default values are used. These parameters are the following [LT11]:

- Minimum number of expressions included in a code clone;
Figure 3.13: Code smells analysis in pipes environment. [SE11]

- Maximum number of new parameters of the least-general common abstracted method;
- Minimum number of parameters of a cloned class;
- Similarity score limit.

The following steps are followed to remove a code clone [LT11]:

1. Copy-paste the cloned class anti-unifier into an Erlang component;
2. If necessary, rename variables names;
3. Re-order the method parameters, if necessary;
4. Rename a method of a class with an appropriate name;
5. Enforce fold expressions against the definition of a function relative to the new method.

In this way, this tool can do refactorings such as Renaming, Reordering the parameters of a function and Folding expressions against the definition of a method [LT11].

Fenske, Wolfram et al. [FMS+ 17] developed a tool for FeatureIDE\(^2\) that uses the detection of code clones to identify clone variants and variant-preserving refactorings to remove these similarities between features. This tool defines 4 different types of code clones that can be visualized in Figure 3.15 (p. 33) [FMS+17].

\(^2\)FeatureIDE - http://fosd.net/fide, Last access on 20 January, 2019
State of the Art

In Figure 3.15 the 3 main types of code clones, which occur more easily, are presented. Clones of Type-1 are shown in red and represent exact copies of other code fragments, in yellow are presented the Type-2 clones that assume all Type-1 clones, but also all renames made and in green there are the Type-3 code clones that represent statements that may have been added, removed or modified compared to the original code. Type-4 clones, which are the rarest, are called “Functional Clones”, implement similar functionalities sharing little or no textual common elements [FMS+17].

These code clones are identified textually through quick hint mode (cf. Figure 3.16, p. 34) and are handled using refactorings like Rename or Pull Up To Common Feature (cf. Figure 3.17, p. 35) based on the Eclipse refactoring framework, to perform this kind of restructurings [FMS+17].

Steimann, Friedrich et al. [SS11] implemented an Eclipse refactoring tool for Java programming language, which uses lightweight role object pattern for code restructuring [SS11].

Role objects represent objects with different behaviors depending on the context in which they are inserted. This tool previews of all changes that can be executed and in case the respective refactoring cannot be implemented, a list with the violated preconditions is also visualized [SS11].

In order to perform refactoring through ROP (Role Object Pattern) [SS11] it is necessary to follow the following steps (cf. Figure 3.18, p. 36):

1. Rename the entity with the name of the core component class, using Rename Class refactoring [FdR99] (in this case, Person is renamed to PersonCore);

2. Use Extract Interface refactoring [FdR99] to extract the public interface from the core class (PersonCore Class, in this case) to build a new interface with the component name (Person
component), using this interface instead of the core class. In this way, we move the class Person to PersonCore and then we turn to Person again;

3. Add a new abstract class between the entity class and its sub-classes and role classes, giving it the nomenclature of the component role class (PersonRole). The first 3 steps can be visualized in Figure 3.18 (p. 36) b);

4. Override the inheritance relationship inserted by the new component role class by a delegation, using a refactoring method called “Replace Inheritance with Delegation” [FdR99];

5. Insert the collection roles to the component core class, adding also all necessary methods or functions to the role management of the component core and core classes;

6. Add builders to role classes using only the states that should be accepted.

Steimann, Friedrich et al. [SvP12] also created a framework that works with refactorings without names, where the user selects from a list of actions, the one he wants to perform or is the best for his system.

These suggestions are not presented through the names of refactoring like Move Method, Pull Up, among others, but rather through a description of the action that can be performed, so that the user does not need to know in advance what are the refactorings that exist, thus making ad hoc refactorings (cf. Figure 3.19, p. 37) [SvP12].

The concept of refactorings without names was applied over a constraint-based refactoring language and framework REFACOLA [SKvP11]. The definition of a refactoring through REFACOLA is carried out through [SvP12]:

- Specification of the language that identifies the constituent elements of a program, as well as all its properties;
- A set of rules and restrictions that identify refactoring and allow its use;
- Specification of actions that characterize a refactoring. For example, in the case of the Pull Up Field refactoring, the location of a field that is pulled up is changed.
3.2.2 Refactoring Proposals

Vimala, S. and Bavota, Gabriele et al. [VNB+12, BOD+10] proposed the use of game theory for the evaluation of refactorings, to restructure the system, increasing its perception, understanding, maintenance, and adaptability.

Through the theory of games, it is possible to evaluate if refactoring is better than others, to get the best possible results through the implementation of this refactoring [VNB+12, BOD+10].

Through this approach, weights are given to several factors that constitute a method of refactoring and some values are also given to aspects of the software system that are modified, such as the number of classes or methods. Once these weights are given, game theory is used to know what is most advantageous and what makes the code better [VNB+12, BOD+10].

Meananeatra, Panita [Mea12] proposed a methodology that allows creating an optimum sequence of refactorings to be executed, to make the system the best possible, increasing its quality and comprehension. To do so, four criteria are used to select the optimal solution:

1. Number of bad code smells removed;
2. Maintainability;
3. Size of the sequence created;
4. Number of system’s elements that have changed.

The sequence that includes the four criteria described produces code without code smells with high adaptability [Mea12]. In this proposal 6 different refactoring methods are used: Replace Temp with Query, Introduce Parameter Object, Preserve whole Object, Extract Method, Decomposed Conditional and Replace Method with Method Object and four quality features that enable the software system’s maintainability, to remove long and complex methods and other bad smells to create code with more simple and with more quality [Mea12].

AlAbwaini, Nour et al. [AAJ+18] proposed the use of slicing techniques decomposition and refactoring methods to identify “live” and “dead” code, without causing any errors or bugs.
“Dead” code is code that is not executed and that does not bring value to the software system in which it is inserted. In this case, experiments were carried out and it was proved that this model succeeded in identifying and removing “dead” code, although it is still necessary to improve the accuracy in identifying the “live” code [AAJ⁺18].

The slicing of the code is done by objectives of the different parts or slices of the program. For example, if a function has sum, average, minimum and maximum calculation functions, this program is divided into 4 slices each with its well-defined functionality that will be merged removing the “dead” code [AAJ⁺18]. The proposed model is described in Figure 3.20 (p. 38).

This proposal still does not work perfectly when a system is very complex, failing to properly do code slicing by eliminating the “dead” code, just leaving the code that is “alive” in the software system [AAJ⁺18].

### 3.2.3 Discussion

Refactoring a software system modifies a program, its internal structure, and design to improve its quality, comprehension, maintainability, and adaptability, without changing its main behaviors or goals [FdR99].

There are already several ways to perform different refactoring methods, and the simplest way to perform a refactoring is through tools that make restructuring suggestions to a programmer, giving system feedback and helping him to decide which changes to perform, without having to carry out an extensive analysis of his program.

Most of the tools analyzed, through the systematic literature review, were developed for the Eclipse IDE, for programs developed in Java. It is only through the tool introduced by Steimann, Friedrich et al. [SvP12] that a programmer does not need to have any previous knowledge about the different refactoring methods that exist since this tool allows the execution of refactoring through the description of the action that will be performed and not through its nomenclature.

It has further been found that the various tools and proposals presented do not specifically analyze a method or groups of refactoring methods. Despite this, most of the collected publications
analyze types like Move Method, Extract Class or Pull Up refactorings. In addition, most of the tools analyze more than one type of refactoring, having a fairly wide range of restructurings that can be performed.

Furthermore, it was found that the refactoring tools or proposals analyzed, take into account the analysis of quality metrics of a software system, as they allow to know if a system is in good condition or not. Tools like ExtC [CAGA11] use external plugins or tools, such as Eclipse Metrics\(^3\) to analyze and verify different software metrics.

It was also possible to verify that many of the refactoring suggestions are made through the quick hint mode of the IDE in which the software system is inserted [FGLD13, FMS+17] or through interfaces in which it is necessary for the programmer to indicate previously that he wants to refactor his program [SE11, UOII17].

Finally, all proposals or refactoring tools serve their purpose, but the feedback given to the programmer is not a 100% complete and immediate, since it is necessary for the developer to finish programming or indicate that he wants to do refactoring of his program and then the suggestions and analyzes of refactorings to be implemented are initiated.

### 3.3 Live Refactoring

Live and refactoring are topics related to the application of liveness or live programming to the evaluation of refactoring methods that can be executed in a software system to increase its quality. Next, the tools found, through the systematic literature review, are presented.

**Kobold.** This tool allows to apply Client-Side Web Refactorings (CSWRs) to a Web Application. This is used to capturing user interactions so that code smells can be detected in real-time. When a user logs in to his Kobold account he has access to all captured code smells and their suggested CSWRs to solve them (cf. Figure 3.21, p. 39) [GGGR18].

So, with this tool, we can create private versions of a Web Applications we are executing by combining CSWRs to create these versions without requiring the cloned sandbox environment for each of the versions. Thus through different versions, it is possible to solve in different ways a certain smell related to usability [GGGR18].

The refactorings executed through this tool change the Document Object Model (DOM) of web pages to increase their usability, adaptability, and comprehension [GGGR18].

**LAMBDAFICATOR.** Tool that automates 2 different types of refactorings in Java lambda expressions. The first one modifies anonymous inner classes to create lambda expressions (AnonymousToLambda) and the second one changes collections loops generating operations with lambda expressions (ForLoopToFunctional) [FGLD13].

This tool gives two types of workflow options through batch or the Quick Hint mode of the code editor that is being used. In the batch mode (cf. Figure 3.22, p. 40), a programmer can invoke automatically the refactoring that he wants to execute, via the NetBeans IDE. LAMBDAFICATOR applies the refactorings to all project files or generates a preview that lists the valid transformations that can occur. Through this mode, several refactorings can be executed in fast seconds [FGLD13].

The quick hint mode (cf. Figure 3.23, p. 41) checks in real-time editor mode for refactoring possibilities. If this tool finds some piece of code that can be refactored, this code is underlined and it is presenting a hint in the sidebar, indicating that a lambda refactoring is available. In this way, a programmer can perform refactoring without leaving his workflow and the code editor that is being used [FGLD13].

**Soares, Gustavo et al.** [SMHG13] sketched a new plugin (cf. Figure 3.24, p. 42) that can be used in any IDE and can give live feedback regarding the behavioral change of a Java program.

This plugin would help a programmer distinguish a refactoring from a behavioral change in programming time, since it uses SafeRefactor\(^4\) for the system to be continuously compared to previous versions, giving feedback to the developer regarding changes made to his code informing whether the same was refactorings or not, helping the developer to confirm whether the changes were the best to be made and how they affected his software. SafeRefactor, therefore, generates tests for the code before and after a change, running both versions and comparing whether the results obtained are the same or not. If they are the same, it turns out that the system behaviors have not changed, implementing successfully a refactoring method, increasing the programmer’s

confidence about his system. If the results are not the same, it means that there were changes in the behavior of the software system, meaning that refactoring was not performed correctly, because the behavior of the program was not maintained and its outputs were changed [SMHG13].

The execution of this tool is sequenced in Figure 3.24 (p. 42). In this way, while programming if there is a behavioral change and not a refactoring, this change is highlighted by drawing the attention of the programmer through quick hint mode. Making mouse hover in the hint is explained to the developer what was the change that he made and what were the tests that prove that a behavioral change occurred and not a refactoring as he wanted [SMHG13].

3.3.1 Discussion

Live refactoring refers to the mixture of live software development theme with the refactoring concept. Through this combination, a programmer can perform real-time refactoring of his system as he programs, reducing the cost of maintaining and adapting it, making his code easier to understand.

Although it has been verified that there is enough research on live programming and refactoring methods, there is still no great focus in the area of live refactoring, since only three tools about this topic were found through the systematic literature review done [GGGR18, FGLD13, SMHG13].

Since few tools have been found for live refactoring, it turns out that this topic can still be extensively explored so that a developer can understand the best tools and approaches to be developed or used, related to this main theme.
State of the Art

3.4 Quality Metrics Analysis

Many quality metrics analysis methods are already used by refactoring tools that have been analyzed in the previous section (cf. Section 3.2.1, p. 26). Even so, there are still tools or proposals that aim to analyze this kind of metrics.

3.4.1 Quality Metrics Tools

The ExtC tool presented by Cassell, Keith et al. [CAGA11] uses the Eclipse Metrics2 plugin\(^5\) to evaluate the quality of a software system through a static analysis of its metrics.

The WebScent tool presented by Nguyen, Hung Viet et al. [NNN+12] analyzes metrics that allow identifying embedded code smells which present problems in a Web Application. In order to analyze the degree of maintenance of a system with and without code smells, this tool uses some metrics such as the number of bugs that the system had throughout its life cycle, as well as the average number of changes that a file suffered.

**c-JRefRec** [UOII17] uses the analysis of metrics such as number of methods of a class, number of uses of methods or fields of a class or how many times a method uses other methods belonging to another class, in order to evaluate the cohesion and coupling between classes of a system software, to identify possibilities of Move Method refactorings.

**SQUANER.** (Software QUality ANalyzER) is a framework that allows monitoring the evolution of the quality of an object-oriented system. This tool is connected to the SVN of the project and every time it is committed, its source code is analyzed, verifying design patterns, code smells, anti-patterns (cf. Table 3.6, p. 40) and quality, predicting also faults, through a previous analysis

---

\(^5\)Eclipse Metrics2 Plugin - [http://metrics2.sourceforge.net](http://metrics2.sourceforge.net), Last access on 20 January, 2019
of several quality metrics of a software system such as number of parameters of a class, number of classes, number of lines of a method or class, among others [HKA10].

SQUANER also evaluates some quality characteristics such as reusability, flexibility, comprehension, functionality, extensibility and effectiveness, and some structural design properties like encapsulation, coupling, polymorphism, data abstraction, and hierarchies. After this extensive verification, feedback is given to the programmer and some instructions are given on how he can improve his system, thus reducing the cost associated with late changes to a software system. The feedback in this case is given via web page (cf. Figure 3.25, p. 43) or via email (cf. Figure 3.26, p. 44) [HKA10].

Table 3.6: Some design patterns, anti-patterns and code smells analyzed by SQUANER. [HKA10]

<table>
<thead>
<tr>
<th>Design Patterns</th>
<th>Anti-Patterns</th>
<th>Code Smells</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Factory</td>
<td>AntiSingleton</td>
<td>AbstractClass</td>
</tr>
<tr>
<td>Adapter</td>
<td>Blob</td>
<td>ChildClass</td>
</tr>
<tr>
<td>Command</td>
<td>ClassDataShouldBePrivate (CDSBP)</td>
<td>ClassGlobalVariable</td>
</tr>
<tr>
<td>Composite</td>
<td>ComplexClass</td>
<td>ClassOneMethod</td>
</tr>
<tr>
<td>Decorator</td>
<td>LargeClass</td>
<td>ComplexClassOnly</td>
</tr>
<tr>
<td>Factory Method</td>
<td>LazyClass</td>
<td>ControllerClass</td>
</tr>
<tr>
<td>Observer</td>
<td>LongMethod</td>
<td>DataClass</td>
</tr>
<tr>
<td>Prototype</td>
<td>LongParameterList (LPL)</td>
<td>FewMethods</td>
</tr>
<tr>
<td>Singleton</td>
<td>MessageChains</td>
<td>FieldMethods</td>
</tr>
<tr>
<td>State</td>
<td>RefusedParentBequest (RPB)</td>
<td>FieldPrivate</td>
</tr>
<tr>
<td>Strategy</td>
<td>SpaghettiCode</td>
<td>FieldPublic</td>
</tr>
<tr>
<td>Template Method</td>
<td>SpeculativeGenerality (SG)</td>
<td>FunctionClass</td>
</tr>
<tr>
<td>Visitor</td>
<td>SwissArmyKnife</td>
<td>HasChildren</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LargeClass</td>
</tr>
</tbody>
</table>

**Code City.** Code city is a visualization tool for a software system. However, to develop the visual form of a given program, it is needed to analyze the different metrics of a given project [WLR11].

This tool will be discussed in more detail in the section about software visualization (cf. Section 3.5, p. 42), however regarding the analysis of the different metrics that constitute a software system and that can verify the quality of the same, it turns out that Code City analyzes different metrics such as number of packages, number of classes of each package, number of functions of each class as well as the number of parameters of the same, so that it is able to create the respective visualization that will help the programmer to better understand his program [WLR11].
State of the Art

![Figure 3.23: LAMBDAFICATOR being performed in quick hint mode. [FGLD13]](image)

### 3.4.2 Quality Metrics Proposals

**Yano, Keisuke [YM17]** proposed the use of software clustering to detect problematic characteristics of a software system. With the use of this technique, the quality of a system can be analyzed, understanding its data utilization.

When checking usage and data access, we can verify metrics such as the number of invocations, checking whether these are or not excessive, causing the quality of the code to decrease. This proposal uses the city metaphor, namely the CodeCity tool, to verify the data access (cf. Figure 3.27, p. 45) [YM17].

The buildings are interconnected allowing to know how they communicate with each other. These connections are represented by different colors showing if there are more or less amount of data accessed in these interactions [YM17].

### 3.4.3 Discussion

A software system consists of several metrics that characterize it. These metrics allow a developer to be aware of the quality of a software program.

Many of the analyzed tools are refactoring suggestion tools in which the analysis of these quality metrics is made to verify what the state of the system, verifying not only its quality but also its capacity of adaptation and maintenance.

Although most refactoring tools check different software quality metrics, through external plugins such as Eclipse Metrics2 [6], the SQUANER tool [HKA10] does not suggest or aims to suggest refactorings as the remaining tools, through the analysis of the respective metrics, but it allows a programmer to know the quality of his system, assuming that it may be necessary to modify its code, if it verifies that the quality analysis of the same indicates that the software could be developed in a better way. Also, the Code City tool [WLR11], whose visualization of the city that represents the software system is created through the different metrics and characteristics of a program, allows a developer to become aware of the state of his software, being able to change it if necessary.

---

3.5 Software Visualization

The software visualization is present in several tools that allow not only the development of live software but also in some tools that allow to suggest or to execute different types of refactoring. Besides, a visual metaphor is often used to represent a software system.

3.5.1 Metaphors

Visual metaphors allow a user, through a context with which he is familiar, to better understand a particular visualization [AD07, YM98b]. These metaphors can be applied to software visualization and the most popular of them is the city metaphor, already explored in other scientific researches [AD07, WLR11, VNP17, KM00].
City Metaphor. Through the city metaphor a developer can understand a software system, where the respective program is represented visually by a city, which contains districts and different buildings with different characteristics [WLR11, Ama18].

Software packages are represented visually by the city districts and their classes as the buildings that are contained within a district. The appearance of buildings is implemented taking into account several software metrics, which represent the physical properties of the city’s artifacts. These metrics are essentially reflected in the dimensions and colors of the different artifacts that are being represented by the metaphor. Figure 3.28 (p. 46) represents the city metaphor with all aforementioned aspects to being visualized [WLR11].

This metaphor is applied by Wettel et al. in the CodeCity tool, which allows the comprehension, analysis, and evaluation of the quality of a software system [WLR11].

Through this tool, Wettel et al. concluded that the city metaphor is an efficient way to represent and visualize a software system and that both less and more experienced programmers can understand the purpose of these visualizations with a better comprehension of the software system under analysis [WLR11].

3.5.2 Visualization Tools

Many of the previously reviewed tools that allow live programming, refactoring or quality metrics analysis have visual representations other than textual elements passed through quick hint mode [FGLD13, FMS+17], for example.

Other tools like Quokka [HSHV12] and the tool introduced by [BLD15] allow the user to have a visualization of what goes on in their software system, through both visual forms and interfaces, which in this case allow the programmer to understand how the components of a program interact between themselves or to change his system easily.

ExtC [CAGA11] allows, through clustering methods, to create graphs that visually represent the software and its states, giving feedback to a programmer, thus knowing which refactorings can be executed to improve the system [SE11].
Stolec, K. T. et al. [SE11] tool which visually represents pipes, allows a programmer to view the state of the development environment using pipes. Also, along with this visualization, there is an interface through which he can know in advance which refactorings can be performed, depending on the verified code smells.

Yano, Keisuke et al. [YM17] present the visualization of the software using the city metaphor, using the Code City tool, which will be explained in detail below. Through this visualization, a programmer can get a sense of the data flow and how they are accessed by the different classes.

**CodeCity.** This tool is already presented in both Section 3.4.1 (p. 39) and Section 3.5.1 (p. 42). Code City is a tool that allows a 3D visualization of a software system, based on the city metaphor (cf. Figure 3.28, p. 46), aiming to increase the comprehension of the system, in a more intuitive and user-friendly way [WLR11].

The districts visualized represent the packages of a system, while the buildings represent the different classes of each package. The color given to a district is related to the nesting level of that package, while the color of each building is related to the number of lines of code in that class. The number of attributes of a class defines the length and width of the building that represents it, while the number of methods defines the height of it. In this way, the different metrics of the software are also analyzed [WLR11].

Analyzing the visualization created through the city metaphor, developers can obtain information about the static and dynamic aspects of a software system. By analyzing the size, density, and properties of different buildings, it is also possible to identify areas of the program that are not as high quality and which may require refactoring [WLR11].
3.5.3 Discussion

Software visualization allows a programmer to understand better his system in a more user-friendly way, what is happening in his software system and how it is structured since it is possible to synthesize almost all the information related to the program in a certain visual form. Thus, the main purpose of visualizing software is to transmit effectively all the relevant information about a system [VNP17].

Many of the software visualizations use visual metaphors so that it can create a familiar environment for a programmer. The city metaphor is perhaps the most widely and well-known metaphor used. Through this metaphor, someone can visualize a software system through the form of a city, with districts and buildings, with diverse properties and characteristics that distinguish them [AD07, WLR11, VNP17, KM00].

Many tools analyzed did not allow visualization of the software system under analysis, only allowing the verification of refactoring suggestions through the quick hint mode, in which the suggestion is made verbatim or through interfaces. However, the Code City tool, although not directly related to refactoring or refactoring suggestions, allows the analysis of the city that represents the system to verify the most critical areas of the software that need to be improved.

3.6 Visual Studio Code Tools

We also analyzed the different software metrics analysis and refactoring suggestions tools available in the Visual Studio Code Marketplace\(^7\). All tools found and analyzed are described in detail below throughout the Section 3.6.1 and Section 3.6.2 (p. 48).

3.6.1 Software Metric’s Analysis

**CodeMetrics.** This extension allows to calculate the cyclomatic complexity of functions developed in TypeScript, JavaScript or Lua [Tama, Tamb].

\(^7\)https://marketplace.visualstudio.com, Last access on 15 May, 2019
Cyclomatic complexity is the number of independent paths that a piece of code, function, or file has and that can be evaluated as true or false. This metric is calculated by following the next steps [Tama, Tamb]:

1. Creates the active source file’s AST;
2. Analyzes each node of the created AST, analyzing the number of possible paths that can create;
3. Sums the cyclomatic complexity of all the nodes that belong to a certain method and calculated the maximum of this complexity for the nodes that belong to a class.

The calculated values are presented next to the methods or functions analyzed (cf. Figure 3.29). Furthermore, these values are also represented with a color component indicative of their severity (for example, a function with high complexity will have this result displayed in red as a form of warning) [Tama, Tamb].

Code Time. Visual Studio Code extension that gives daily and weekly reports to a software developer, regarding the time he has been programming his software system [Sofa, Sofb].

State of the Art
State of the Art

After installing this plugin, a programmer has access to real-time metrics related to the time that he was programming during that day (cf. Figure 3.30). In addition to the daily reports, he also has access to weekly reports sent via email [Sofa, Sofb].

So, through this tool, a developer has access to relevant information that helps to realize what his best moments are, during the day, to program. Besides, Code Time can also be integrated with Google Calendar allowing a developer to manage his day around the results presented by this extension [Sofa, Sofb].

CoreMetrics. Visual Studio Code and .Net Core extension that analyzes various software metrics allowing a programmer to check and correct problematic zones of his software system. Although it is a fairly complete plugin, this is built on top of the roslyn-analyzer and it can be only used in the Windows operating system, since roslyn-analyzer files are only referring to the Windows environment [Kona, Konb].

The different software metrics that the CoreMetrics plugin allows to calculate refer to [Kona, Konb]:

- **Cyclomatic Complexity** - Checks the number of possible paths in the software system’s flow (conditions or loops statements) [Kona, Konb];
- **Class Coupling** - Class coupled in a method [Kona, Konb];
- Depth of Inheritance;
- Lines of Code;
- **Maintainability Index** - Metric calculated taking into account the Halstead Volume [Hal77] that takes into account the total number of operators and operands used and also the unique operators and operand, respectively. The Maintainability Index can be between 0-100, being 100 the best result possible [Kona, Konb].

Figure 3.30: Code Time daily reports.
**State of the Art**

**SonarLint.** Visual Studio Code plugin that allows to detach and correct some problems that a program developed in JavaScript, TypeScript, PHP or Python, can have [Sona, Sonb].

SonarLint evaluates the quality of a software system as a developer deploys it, allowing the system to be executed as quickly as possible, before committing or releasing a system. All the problems that SonarLint finds are listed and described in the “Problem Panel”, which is usually located below the text editor (cf. Figure 3.31) [Sona, Sonb].

This way, with this tool a programmer has access to the good programming rules of JavaScript, TypeScript, Python and PHP code, benefiting from code analyzers such as SonarJS, SonarTS, SonarPython and SonarPHP [Sona, Sonb].

![Figure 3.31: SonarLint execution example.](image)

**DeepScan.** This extension is intended to detect bugs and quality problems in software systems developed in JavaScript, TypeScript, React and Vue.js. Although it allows the analysis of systems developed in several languages, this tool is more specific to JavaScript/TypeScript, but it can also be used with React and Vue.js since it allows to analyze the rules characteristic of good programming in React and Vue.js [Deea, Deeb].

DeepScan is very similar to the SonarLinter extension [Sona, Sonb], since it reports the different problems encountered in the “Problem Panel” as a programmer is developing and updating documents of type *.js, *.jsx, *.mjs, *.ts, *.tsx, and *.vue. Furthermore, it also highlights the issues in the text editor as well as describing them in the problems panel [Deea, Deeb].

### 3.6.2 Refactoring

**JS Refactor.** Automatic refactoring tool that can be used in the Windows, Linux, and Mac OSX operating systems and it allows to improve a software system and a developer’s programming experience. This Visual Studio Code extension can be used on systems developed in JavaScript/ECMAScript, Vue or HTML and supports an extensive list of refactorings and actions that can be performed in a program [Steb, Stea].

This plugin allows executing refactorings like Extract Method, Extract Variable, Inline Variable or Rename Variable [FdR99] (alias of VSCode internal command). Also, it allows utilities such as
State of the Art

Convert To Arrow Function, Convert To Function Declaration, Convert To Function Expression, among others, or wrap selection options/snippets such as Arrow Function or Async Function [Steb, Stea].

The different refactorings supported by this extension [Steb, Stea] are listed and described below.

**Extract Method** - Creates a new function with the original selected code as its body [FdR99].

**Extract Variable** - Creates a new constant and assigned variable declaration and replaces the original one with this new constant variable [FdR99].

**Inline Variable** - Replaces all references to a variable with its initialization [FdR99].

**SCSS Refactoring.** Visual Studio Code extension that enables a developer to perform Extract Variable refactorings on SCSS files, on Windows or MacOSX operating systems [Zakb, Zaka]. Through this type of refactoring, this tool extracts the selected code to a new variable named automatically, taking into account the context in which it is inserted [FdR99, Zakb, Zaka]. An example of SCSS Refactoring, being performed, can be verified as follows:

**Before:**

1. .foo { background-color: #f8f8f8; }

**After:**

1. $foo-bg-color: #f8f8f8;
2. .foo { background-color: $foo-bg-color; }

Listing 3.1: Extract Variable in a SCSS file

**glean.** This tool allows to perform refactoring of software systems developed in React, JavaScript or TypeScript, varying its features vary depending on the program’s type. In a React project a developer can extract JSX into a new component, convert class components to functional components and vice versa, JSX wrap with conditional and move code between files. In a JavaScript or Typescript system, it is possible to extract code between files and to export code [Wixa, Wixb].

Through the glean plugin, the refactoring suggestions are given through the Visual Studio Code’s "Bulb Quick Actions Suggestions" [Wixa, Wixb].

**Super Sharp.** Tool that allows implementing some refactorings in a C# system. With this extension, a programmer can add injected dependencies from anywhere inside a class and move classes, enumerations, interfaces or structures from one file to another one [Cra].

To perform these refactorings it is necessary for the user to underline the code where he intends to carry out the restructuring or to select the desired option from those presented by “Bulb
Quick Action List”, not being a live tool that allows executing refactorings without the developer’s external actions [Cra].

**Vscode-Fsharp-Refactor.** Visual Studio Code’s tool that allows performing refactorings in a F# software system [Manb, Mana].

Vscode-Fsharp-Refactor allows a developer to apply refactorings such as Extract Expression (cf. Figure 3.32), Extract Lambdas, Extract String, that are similar to the Extract Variable refactoring, or Inline Binding which is like the Inline Variable refactoring [FdR99, Manb, Mana].

![Figure 3.32: Extract Expression refactoring being executed.](image)

**NG-Refactor.** Extension that allows applying refactorings in AngularJS applications. This plugin implements several refactorings, however only refactorings such as Toggle inline HTML and Toggle inline CSS, which allow toggling between an inline template and external template files or style sheets (cf. Figure 3.33), don’t present any limitations [Picb, Pica].

![Figure 3.33: Toggle inline HTML and CSS refactoring being performed.](image)

**refactorix.** Visual Studio Code extension that implements different refactorings and quick actions on TypeScript systems [Oetb, Oeta].

The refactorings provided by this extension consist of:
**State of the Art**

**Split Variable Declaration.** Splits the initialization part of a variable declaration after a developer selects the code that he wants to split. If the variable’s type is not indicated, it will be initialized with a generic type “type” [FdR99, Oetb, Oeta].

**Before:**

```javascript
1  const x = false;
2  const y: number = 1;
```

**After:**

```javascript
1  let x: type;
2  x = false;
3
4  let y: number;
5  y = 1;
```

Listing 3.2: Split Variable Declaration execution

**Extract Variable.** Replaces the selected text with a constant variable declaration, operating on text rather than its AST [Oetb, Oeta, FdR99].

**Before:**

```javascript
1  var x = false;
```

**After:**

```javascript
1  const var_name = false;
2  var x = var_name;
```

Listing 3.3: Extract Variable execution

Besides, with this extension, a developer can also have access to a set of quick actions such as:

**Toggle access modifier.** Toggles a property and its get and set method access between private, protected, public and no access modifier [Oetb, Oeta].

**Property to getter/setter.** Converts a property to getter/setter. To execute this quick action place the cursor on a property and invoke the respective command [Oetb, Oeta].

**Before:**
State of the Art

```java
class Color {
  rgb: string;
}
```

**After:**

```java
class Color {
  private _rgb: string;
  get rgb(): string {
    return this._rgb;
  }
  set rgb(value: string) {
    this._rgb = value;
  }
}
```

Listing 3.4: Getter and Setter implementation

**Interpolate.** Surrounds the selected part of a string literal with $ and converts the literal to backticks [Oetb, Oeta].

**Before:**

```javascript
'my name is refactorix.'
```

**After:**

```javascript
'my name is ${refactorix}.'
```

Listing 3.5: String interpolation

**Add/remove semicolons.** Adds or removes the semicolons of all statements in the active document, if necessary [Oetb, Oeta].

**Arrow function toggle single statement block to expression and vice versa.** Toggles between an arrow function’s single statement block and expression [Oetb, Oeta].

**Before:**

```javascript
() => 0;
```

**After:**
State of the Art

```javascript
() => { return 0; };
```

Listing 3.6: Arrow function toggle between single statement block and expression

**VSCode OBCSS.** This Visual Studio Code extension can be used on React projects allowing to simplify the world of CSS syntax and JS style object syntax, especially when it is necessary to make changes between the syntax. Thus, the refactorings that this plugin supports are the conversion of CSS to object style (cf. Figure 3.34) and the conversion of an object style to CSS [Uhea, Uheb].

![Figure 3.34: Conversion of CSS to object style.](image)

**Elegant PHP.** Visual Studio Code refactoring tool that only performs restructurings, not including any types of quick actions. This plugin aims to improve the workflow of a programmer as it develops its software system and also his programming experience. With this extension, a developer has access to Extract To Method, Extract Variable and Inline Variable refactorings [Kos].

### 3.6.3 Discussion

Through this research, it has been found that practically all the tools found in the Visual Studio Code Marketplace for analyzing software metrics are effectively live tools since they allow a developer to check the results of these tools in real-time, as they are programming and that they can also be used with several programming languages.

Furthermore, it has been found that tools such as SonarLint [Sona, Sonb] and DeepScan [Deea, Deeb] are tools that perform background analysis of their software metrics, not presenting them directly to programmers, only giving suggestions of improvements obtained by pre-calculating some metrics while the system is being developed.

It was also found that the CoreMetrics tool [Kona, Konb] analyzes a number of key metrics like cyclomatic complexity, maintainability index, number of bugs delivered, among others, allowing
a programmer to become aware of the state of his software system, unlike the CodeMetrics ex-
tension [Tama, Tamb], which only analyzes the cyclomatic complexity of functions and methods. However, the CoreMetrics tool [Kona, Konb], although being a very complete tool, it can only be
used in the Windows operating system, which does not occur in the other extensions.

Relatively to the refactoring tools available on Visual Studio Code, it was found that none
of them is actually a live tool, since they do not allow software developer to receive refactoring
suggestions as it is developing his system, because these extensions require to have previous
knowledge of the refactorings that exist, what their objectives are and how to execute them through
these extensions, to be executed in full. Furthermore, it was found that some of these tools have
restrictions on the operating systems in which they can be used.

However, in spite of these disadvantages presented by the analyzed refactoring tools, these
include a large set of refactorings, restructurings and quick actions that can be performed in a
software system, being able to be used in programs developed in multiple programming languages
and not restricting the programmer to use only one or a reduced set of them.

3.7 Summary

The introduction of liveness or live programming in the life of developers has great poten-
tiality and relevance since it allows to decrease the time of programming because while they are
programming, they can see the results of their actions.

Furthermore, the use of refactoring methods coupled with quality metrics analysis allows a
software developer to receive feedback from their program to improve and adapt it so that they
can understand it better and therefore can change it more rapidly when necessary.

That way, the area of Software Engineering encompasses all other topics, being one of the
bases of this scientific investigation.

To analyze the implementation of a live software metrics analysis system, an investigation
was made of the existing literature on the main topics where this dissertation is inserted, namely
through a systematic literature review, aiming to respond to a set of research questions related to
the main research question presented in Section 4.3 (p. 61).

In addition, different tools available for the Visual Studio Code were also researched to analyze
the relevance and innovation of the proposed solution in comparison with the extensions that are
already operational to be used in this IDE.

3.7.1 Systematic Literature Review Results

The findings, obtained through the systematic literature review done, indicate that the topic of
live refactoring is still unexplored, but with great potential and importance, since the subtopics of
live programming and refactoring are already investigated separately and there are already several
relevant contributions to this programming area, with the purpose of helping a developer. In addi-
tion, it has been found that more and more quality metrics are used so that refactoring suggestions
can be created, as well as using software visualization methods to complement these suggestions.
Thus, it was concluded that there is still room for research on this great theme that is live refactoring, grouping the 4 subtopics of live software development, refactoring, quality metrics analysis and software visualization, being this subject relevant and necessary to improve programmers’ software comprehension, in order to reduce the total development or restructuring time.

Since one of the objectives was to verify how live programming was embedded in the current development of software systems and also the refactoring tools that already exist, next there are 2 comparative tables of tools that allow doing live software development (cf. Table 3.7, p. 55) and refactoring suggestions (cf. Table 3.8, p. 55). For this, the target language of the different tools was analyzed and also if they allow the visualization of software through visual forms, if they analyze quality metrics and if they were developed for some specific IDE. Also, it was verified whether the refactoring tools were also live tools (cf. Table 3.8, p. 55). Blank spaces identify unanswered points.

Table 3.7: Live programming tools comparison.

<table>
<thead>
<tr>
<th>Name</th>
<th>Reference</th>
<th>Target Language</th>
<th>Software Visualization</th>
<th>Quality metrics</th>
<th>Plugin IDE</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveMTBD</td>
<td>[SGW+11]</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>Quokka</td>
<td>[HSHV12]</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NaturalMash</td>
<td>[AP13]</td>
<td>-</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SOMETHINGit</td>
<td>[ONY13]</td>
<td>Smalltalk, Haskell, VDM-SL</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>-</td>
<td>[BLD15]</td>
<td>Java</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Through the analysis of the previous tables, some similarities between the tools were verified. Most live programming tools (cf. Table 3.7, p. 55) have been developed as plugins for different IDEs, allowing visualization of software through visual forms, allowing a user to understand a system, correctly and in a user-friendly way. However, these tools do not perform real-time analysis of software metrics.

As for the refactoring tools (cf. Table 3.8, p. 55), it was found that most of the tools were also developed as plugins for different IDEs, namely for Java programs. Almost all of these tools perform quality metrics analysis, but they do not allow to visualize the software through a visual...
method or metaphor. Besides, it was found that only three of these tools were also live tools, creating real-time refactoring suggestions.

### 3.7.2 Visual Studio Code Tools

Since one of the goals of this analysis was to check the software metrics analysis and refactoring suggestions tools that currently exist in the Visual Studio Code Marketplace, two tables have been created to summarize all the information gathered through the extensions analyzed previously (cf. Table 3.9, p. 56) and (cf. Table 3.10, p. 57). These tables list the different tools, their target languages, as well as their live character and operating systems where they can be used. Additionally, Table 3.9 shows the software metrics that each analysis tool calculates and Table 3.10 (p. 57) presents all the refactoring methods that each extension supports.

With the summary done in Table 3.9 we can verify that many software metric tools don’t mention which metrics they support or focus on, being only the CodeMetrics [Tama, Tamb], and CoreMetrics [Kona, Konb] extensions the ones that effectively mention them, describing the different values that they calculate, having in common the measurement of the cyclomatic complexity of functions or methods. Moreover, it was also found that most of the extensions analyze JavaScript or TypeScript programs and that practically all of them are also live tools because they analyze metrics as the software is being developed.

<table>
<thead>
<tr>
<th>Name</th>
<th>Reference</th>
<th>Target Language</th>
<th>Software Metrics</th>
<th>Live</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Time</td>
<td>[Sofa, Sofb]</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>CodeMetrics</td>
<td>[Tama, Tamb]</td>
<td>JavaScript, TypeScript, Lua</td>
<td>Cyclomatic Complexity</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>CoreMetrics</td>
<td>[Kona, Konb]</td>
<td>-</td>
<td>Cyclomatic Complexity, Class Coupling, Depth of Inheritance, Lines of Code, Maintainability Index, Halstead Volume</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>SonarLint</td>
<td>[Sona, Sonb]</td>
<td>JavaScript, TypeScript, PHP, Python</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>DeepScan</td>
<td>[Deea, Deeb]</td>
<td>JavaScript, TypeScript, React, Vue.js</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
</tr>
</tbody>
</table>

Through the analysis of Table 3.10 (p. 57) it turns out that the refactoring tools available on the Visual Studio Code Marketplace aren’t live extensions, because the refactoring suggestion is not made as the system is being programmed, since it is necessary to do external actions (eg. select code and be the user the one who decides which refactoring should be executed, needing to know in advance which refactoring exists and what are their objectives). It was also verified that these plugins focus on several programming languages and in different types of refactorings, being the Extract type refactorings the most supported.

Furthermore, both Table 3.9 and Table 3.10 (p. 57) allow us to verify that most of the tools make no mention in their description of the operating systems in which they can be used, assuming that these can be performed on all existing operating systems.
In this way, it is verified that there are already several software metrics analysis and refactoring tools on Visual Studio Code Marketplace, but none specifically focuses on solving the problem presented in the Section 1.3 (p. 3).

Table 3.10: VSCode refactoring tools.

<table>
<thead>
<tr>
<th>Name</th>
<th>Reference</th>
<th>Target Language</th>
<th>Refactorings</th>
<th>Live</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td>JS Refactor</td>
<td>[Steb, Stea]</td>
<td>JavaScript, ECMAScript, Vue, HTML</td>
<td>Extract Method, Extract Variable, Inline Variable, Rename Variable</td>
<td>No</td>
<td>Windows, Linux, Mac OSX</td>
</tr>
<tr>
<td>SCSS Refactoring</td>
<td>[Zakb, Zaka]</td>
<td>SCSS</td>
<td>Extract Variable</td>
<td>No</td>
<td>Windows, Mac OSX</td>
</tr>
<tr>
<td>glean</td>
<td>[Wixa, Wixb]</td>
<td>React, JavaScript, TypeScript</td>
<td>Extract JSX, Convert Components Move Code</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>Super Sharp</td>
<td>[Cra]</td>
<td>C#</td>
<td>Inject Dependencies, Move Classes, Move Enumerations, Interfaces, Structures</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>Vscode-Fsharp-Refactor</td>
<td>[Manb, Mana]</td>
<td>F#</td>
<td>Extract Expressions, Lambdas, Strings Inline Binding</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>NG Refactor</td>
<td>[Picb, Picb]</td>
<td>AngularJS</td>
<td>Toggle inline HTML and CSS</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>refactorx</td>
<td>[Oetb, Oeta]</td>
<td>TypeScript</td>
<td>Split Variable, Extract variable</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>VSCode OBCSS</td>
<td>[Uhea, Uheb]</td>
<td>React</td>
<td>CSS to Obj Style and vice-versa</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>Elegant Php</td>
<td>[Kos]</td>
<td>PHP</td>
<td>Extract Method and Variable, Inline Variable</td>
<td>No</td>
<td>-</td>
</tr>
</tbody>
</table>
State of the Art
Chapter 4

Problem Statement

This chapter starts by describing, more accurately, the problem that exists within the current solutions related to the live analysis of software’s metrics (cf. Section 4.1, p. 59) to develop software with higher quality, comprehensibility, and maintainability. In addition, it also analyzes a possible proposal (cf. Section 4.5, p. 62) that can be adopted taking into account the issues collected and described, analyzing all the assumptions (cf. Section 4.2, p. 61), hypothesis (cf. Section 4.3, p. 61), related research questions (cf. Section 4.4, p. 62) and also the validation methodology used (cf. Section 4.6, p. 63).

4.1 Current Issues

Throughout Chapter 3 (p. 17) several solutions have been presented that are intended to facilitate software development, to reduce the developer’s comprehension and maintenance effort. However, none of these solutions was complete, since they did not encompass the different concepts and aspects important to achieve the desired goal. In this way, the main issues found were:

Low level of live feedback. Most software metrics tools either were not live enough or were not live at all since the programmer needed to compile or pre-run, manually, his program to
Problem Statement

verify the evolution of his software metrics. For example, all of the Visual Studio Code software metrics analysis tools that were described were live tools, which analyze the metrics during software development, but they did not give real-time feedback to the programmer. Furthermore, other tools like ExtC [CAGA11] or WebSent [NNN+12] were refactoring and software metrics analysis tools that did not allow the developer to get feedback about his system as he was programming.

Analysis of a reduced number of software metrics. Many of the software metrics tools looked at a small number of software metrics, often focusing only on simple metrics such as the number of lines of code or even the cyclomatic complexity of a system like the CodeMetrics tool [Tama, Tamb]. Also, many of these tools did not describe specifically which metrics they analyze like the SonarLint [Sona, Sonb] or DeepScan tools [Deea, Deeb]. Even the Visual Studio Code tool that included more software metrics, the CoreMetrics extension [Kona, Konb], had the problem of just being able to be executed in the Windows operating system.

Does not allow the direct visualization of the metrics’ evolution. Many of the tools did not allow direct visualization of the evolution of the software metrics, only giving quality reports to programmers, without explaining the results they calculated. This might also be due to the lack of liveness, since in the existing tools many times the metrics were not calculated as the system was being implemented, which causes their visualization to be done later, not showing their true evolution while programming, but rather their final results. For example, the SQUANER [HKA10] tool allows to see the evolution of software metrics, but these are not visualized directly in the IDE, while programming since they could be visualized through the tool’s website. Moreover, the CodeMetrics tool [Tama, Tamb] allows the visualization of the value of the cyclomatic complexity of each function directly in the VSCode IDE, using even visual elements like different colors to represent the values calculated but does not show their evolution, but rather the last calculated cyclomatic values.

Does not allow to compare the project with other Git versions. Often, a programmer is developing a software system with a team of other programmers, so he needs to know beforehand if the code he is developing has a good quality compared to the one that was already developed by himself or others and that is stored in a Git repository, but none of the tools analyzed allows Git integration. Furthermore, without having the possibility to verify the evolution of the system metrics after a Git commit, we couldn’t check if the changes made improve or not the program effectively. This way, these tools do not show the evolution of software metrics throughout the different Git commits made, not giving full feedback to the programmer.

Through these current problems, we were able to make some assumptions (cf. Section 4.2, p. 61) and to present a hypothesis that tried to mitigate the described issues (cf. Section 4.3, p. 61), also describing the respective research questions that we took into account during this dissertation.
(cf. Section 4.4, p. 62). Moreover, through these issues, we were able to do a brief description of the proposed solution (cf. Section 4.5, p. 62), which will be analyzed, in more detail, in (cf. Chapter 5, p. 65).

4.2 Assumptions

To solve all the problems listed above we had to take into account some important assumptions which make the solution proposed both relevant and valid (cf. Section 4.5, p. 62).

The first consideration that should be taken into account is whether or not a programmer values the information given by the evolution of their software metrics. Many programmers are focused on developing a product that meets the customer or manager’s requirements, as quickly as possible, tending to create systems with poor structure and quality. However, in this project, it will be taken into account that most developers, besides intending to fulfill all the proposed objectives, also try to develop a system with quality and therefore have in count the software metrics of their systems, doing refactoring and modifying them to be better, more understandable and in the future more adaptable.

Besides, we assume that developers have deadlines to deliver certain features and that they intend to do so as soon as possible, but maintaining code quality throughout the whole process; so that they avoid having to perform major refactorings at the end of the development process. This will be taken into account as our tool will give live feedback about the evolution of the different software metrics as the developer is programming, as well as at each Git commit, and shorten the total development time of a software system.

4.3 Hypothesis

This dissertation is based on a hypothesis that serves as a basis for its development and implementation. The respective hypothesis is:

"When developers have constant exposure to software metrics, it influences the form and final results of their solutions."

This statement can be subdivided into different aspects, which must be clearly defined, to justify the main statement. First, it is necessary to justify why the target audience is software developers. This happens, because they are those who already have prior knowledge about programming and who deal daily with the development and maintenance of different software systems.

Then, the constant exposure to software metrics, in real-time, is related to the awareness of the state of the implemented code, as well as its quality (verified through the evolution of a set of metrics). Thus, by having this constant awareness about the state and quality of a program, while programming, and not after finishing the development cycle of the software system, a developer can make better decisions about his implementation.
Problem Statement

With this better development, the programmer can better control the form and final results of his solution, because he can create a system with better structure and quality, with the expected behaviors, but with a greater comprehension and adaptability’s capacity.

4.4 Research Questions

Taking into account the statement made previously, the main research question of this dissertation are:

RQ1 “When developers have constant exposure to software metrics, do they reach better solutions?” We aim to analyze how the constant knowledge of each software metric’s evolution can help a software developer improve his system quality and consequently its comprehension and maintainability.

RQ2 “When developers have constant exposure to software metrics, do they converge faster to a good solution?” This question aims to verify if a developer can converge to a good solution, well structured and with quality through constant knowledge of different software metrics.

RQ3 “When developers have constant exposure to software metrics, is the development time reduced?” This question aims to check if it is possible to reduce the development time, knowing in advance the evolution of software metrics while programming, since all refactoring actions could be done during development and not after.

In this way, we can identify some hypotheses that will allow the creation of a hypothesis test (cf. Section 8.7, p. 117) based on the research questions described (cf. Table 4.1, p. 63). The null hypotheses (H0) present the cases where the tool created does not add any value or advantage to its users regarding code quality and development time. The alternative hypotheses (H1) present the cases in which the tool developed is relevant to its different users, taking into account the different research questions described.

4.5 Proposal

The goal of this project was to develop a plugin for Visual Studio Code, able to analyze software metrics in a live way, presenting its evolution through bar or line charts so that developers could understand the evolution of their programming actions.

In addition, this tool should allow the integration with Git, so the project under analysis should be saved in a Git repository. The analysis of the metrics of each previously committed version should be presented similarly, and together with the project metrics, inside the IDE. Finally, as an extra, it should also be able to analyze and suggest some refactorings and quick actions that could improve not only the program’s software metrics but also its structure and appearance.
Table 4.1: Hypothesis Tests taking into account the different research questions

<table>
<thead>
<tr>
<th>RQ1 - Code Quality</th>
<th>Null Hypothesis</th>
<th>Alternative Hypothesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>H0: Developers did not reach better solutions when constantly expose to their software metrics.</td>
<td>H1: Developers reached better solutions when constantly expose to their software metrics.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RQ2 - Solution's Convergence</th>
<th>Null Hypothesis</th>
<th>Alternative Hypothesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>H0: Developers did not converge faster to a good solution when constantly expose to their software metrics.</td>
<td>H1: Developers converged faster to a good solution when constantly expose to their software metrics.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RQ3 - Development Time</th>
<th>Null Hypothesis</th>
<th>Alternative Hypothesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>H0: Developers did not reduce their development time when constantly expose to their software metrics.</td>
<td>H1: Developers reduced their development time when constantly expose to their software metrics.</td>
<td></td>
</tr>
</tbody>
</table>

In Chapter 5 (p. 65), a more detailed description of the proposed solution will be made, given the analysis of the state of the art made in Chapter 3 (p. 17) and current issues described in Section 4.1 (p. 59).

4.6 Validation Methodology

In this dissertation, we will follow the engineering research method [THP93], through which a proposed solution described in Chapter 5 (p. 65), related to the main hypothesis (cf. Section 4.3, p. 61) will be developed and, then, validated.

For that, a validation protocol will be defined taking into account a case study that will be used by different participants (cf. Section 5.6, p. 71), in a controlled experiment, with common objectives, following a previously established script. Through this protocol, we will define different guidelines regarding what it will be intended to validate (cf. Section 8.2, p. 98), as well as a whole experience planning involving the participants, location, duration, tasks that will be performed, among other aspects (cf. Section 8.3, p. 99).

After the definition of the respective protocol, and after the end of the idealized controlled experiment, a set of data will be analyzed allowing to draw some conclusions regarding the respective participants as well as the different tasks performed (cf. Section 8.4, p. 100). Through this data, we hope to draw some conclusions about the usability of the proposed solution (cf. Section 8.6, p. 115).

Then, the most important and relevant data related to the validation of the respective solution will be analyzed through the execution of hypothesis tests, using Independent-Samples t-tests [KK15], for each one of the respective research questions listed in Section 4.4 (p. 62), trying to reject each null hypotheses created, validating or not the research questions (cf. Section 4.4, p. 62) and, consequently, the hypothesis enumerated in Section 4.3 (p. 61).
4.7 Summary

In this chapter we explored all the current issues that the tools discussed in Chapter 3 (p. 17) presented. So, the goal of this research project was not to develop something similar to what already exists, but rather something that could cope with the various issues; a tool for software developers to use in Visual Studio Code that helps them improve their programming experience.

Taking into account all the issues that have been collected, we proposed a solution that can solve all these problems (cf. Section 4.5, p. 62). This solution will be further explored in Chapter 5 (p. 65). With this solution, a programmer can receive more complete and immediate feedback, helping him, while programming, through the information given about the evolution of his system’s metrics.

Besides, we also described the main hypothesis (cf. Section 4.3, p. 61) that supports this dissertation, as well as all the research questions (cf. Section 4.4, p. 62), complemented by possible hypothesis tests that can be done to validate them.
Chapter 5

Proposed Solution

5.1 Contextualization

This dissertation is part of the research in live software development, carried out by the Software Engineering group of the Faculty of Engineering of the University of Porto.

Currently, there are three other dissertations related to the concept of liveness, but none of them is dependent on this research project, in the same way, that this project is independent of the rest. The development of this project will encompass other subjects apart from the liveness topic.
Proposed Solution

Therefore, this dissertation is responsible for creating a tool that will contain a mechanism for detecting and analyzing software metrics — while a developer is programming — giving him visual information regarding the evolution of the calculated metrics. Also, it will allow the user to compare the current metrics with those calculated in each previous Git commit.

Finally, and as an extra, a mechanism to detect and analyze some refactoring and quick actions’ opportunities will be developed. These actions will be suggested to the user using the suggestion module of Visual Studio Code.

5.2 Objectives

The main objective of this dissertation is to increase and improve the feedback given to software developers relatively to their software systems’ code. Furthermore, through this feedback, we can reduce the effort required to understand and to modify a software system.

The focus of this dissertation is the application of the concept of liveness, mainly with the analysis of software metrics, allowing the creation of a live feedback system that gives a developer a view of the evolution of his system while he is programming. Additionally, combining refactoring and quick action’s suggestions also allows a developer to more quickly check the evolution of the metrics while executing the respective suggestions.

All these objectives, described below, were obtained through an extensive analysis of both the different advantages of several tools and the problems they present, having defined a set of research questions involving the issues collected, in Section 4.4 (p. 62).

Therefore, at the end of this dissertation, the project developed should allow to analyze and visualize a group of software metrics and refactoring methods and quick actions’ suggestions that can be executed, in real-time. Besides, it should allow Git integration, by analyzing the metrics and their evolution throughout the project’s Git commits, giving more complete feedback to developers as they are programming. Thus, a developer can compare the evolution of each metric throughout its commits. Achieving all these objectives, we can respond positively to each of the research questions listed in Section 4.4 (p. 62), proving the justifying that bases this dissertation.

5.3 Requirements

In this section, all the functional (cf. Section 5.3.1, p. 66) and non-functional requirements (cf. Section 5.3.2, p. 68) of the proposed tool will be presented.

5.3.1 Functional Requirements

This tool will have only one type of user that is the software’s programmer. Table 5.1 (p. 67) describes the different user stories for the developer, demonstrating all interactions the extension should be able to accomplish and that the main used could make.
### Table 5.1: System’s user stories.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>US01</td>
<td>Metric Choice</td>
<td>As a developer, I want to choose which metrics to analyze so that I can focus in what I feel that is important to my software system.</td>
</tr>
<tr>
<td>US02</td>
<td>Metric Visualization</td>
<td>As a developer, I want to visualize the evolution of the metrics previously selected so that I can know the current state of the project.</td>
</tr>
<tr>
<td>US03</td>
<td>Live Visualization</td>
<td>As a developer, I want to visualize the evolution of the metrics selected, while programming, so that I can receive immediate feedback.</td>
</tr>
<tr>
<td>US04</td>
<td>Git Metric Analysis</td>
<td>As a developer, I want to be able to analyze the evolution of different metrics over different Git versions of the project under analysis.</td>
</tr>
<tr>
<td>US05</td>
<td>Read Description</td>
<td>As a developer, I want to be able to have access to more information about each metric so that I can know what each one of them represents.</td>
</tr>
<tr>
<td>US06</td>
<td>Refactoring Suggestions</td>
<td>As a developer, I want to be able to receive refactoring suggestions, while programming, so that I can know what structural improvements I can make.</td>
</tr>
<tr>
<td>US07</td>
<td>Refactoring Execution</td>
<td>As a developer, I want to be able to execute the refactoring suggested so that I can improve my software system without changing its behaviors.</td>
</tr>
<tr>
<td>US08</td>
<td>Quick Action Suggestions</td>
<td>As a developer, I want to be able to receive quick actions suggestions, while programming, so that I can know what are the changes I can make.</td>
</tr>
<tr>
<td>US09</td>
<td>Quick Action Execution</td>
<td>As a developer, I want to be able to execute the quick actions suggested so that I can change my software system more easily and faster.</td>
</tr>
</tbody>
</table>

With all of these interactions listed, the user will have at his disposal a set of functionalities that will allow him to fully experience the tool developed, to achieve the main objective of this research project, described in Section 5.2 (p. 66).
5.3.2 Non-Functional Requirements

Nonfunctional requirements set constraints on how the functional requirements should be accomplished (cf. Table 5.2, p. 68).

Table 5.2: System’s non-functional requirements.

<table>
<thead>
<tr>
<th>Non-Functional Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Performance</td>
<td>The analysis of either software metrics or refactoring and quick actions opportunities should be done as soon as the user starts the extension as well as as they program or change the files that are active, to receive the correct feedback taking into account the context in which he is.</td>
</tr>
<tr>
<td>Code Change Performance</td>
<td>Whenever changes are made on the project’s source code, the system waits 10 seconds after the last modification to initiate a new analysis, so that the user can have feedback about the changes he made through the new analysis of software metrics or actions that can be performed in his program (sense of cause-effect while programming).</td>
</tr>
<tr>
<td>Active File Change Performance</td>
<td>Whenever a developer changes the active file, the software metrics and even the suggested actions should immediately be changed to match the context in which the developer is programming.</td>
</tr>
</tbody>
</table>

In this case, the greatest concern for achieving the desired goals is the conjunction of the liveness concept with performance aspects, so that the user can receive feedback as quickly as possible without any noticeable delays.

One of the biggest concerns is the propagation of changes as quickly as possible so that the feedback given to the user is made as he develops his system. Therefore, it is necessary that the analysis of either software metrics or refactoring or other actions opportunities are done quickly after he stops typing so that he has a sense of cause and effect about his programming actions. Since it is not expected that the user wants to receive feedback about the status of his software with each line of code or each character inserted or modified in his program, the tool waits a configurable number of seconds after the last change made to show the evolution of the system regarding its metrics and to suggest some actions that can be performed to modifying it.

Besides, the calculated metrics need to be changed whenever a programmer changes his context. In this way, whenever the developer changes the file where he is programming, the software metrics, refactorings, and quick action suggestions are changed to match the developer and system’s new context.

5.4 Implementation’s Proposal

In this sense, this dissertation addresses a set of topics and steps to be considered to achieve the desired result. These are described below:
Proposed Solution

- Application of the liveness concept allowing a developer to receive immediate feedback as he is programming his software system;
- Creation and analysis of an AST that represents a file (*.ts or *.js) of a software system developed in TypeScript or JavaScript;
- Calculation of a set of software metrics according to the results of the AST analysis;
- Git integration, analyzing all the commits made previously, creating an AST for each file present in each version stored in the Git project’s repository;
- Calculation of the software metrics of each system committed, through the AST created for each file;
- Visualization of the metrics calculated by the tool, using a menu present in the “Activity Bar” of the Visual Studio Code;
- Interaction with the respective menu that makes available the different metrics to visualize the evolution of each analyzed value;
- Interaction with the option that activates the refactorings and quick actions analysis, to search for refactoring and quick actions opportunities that come from the previous analysis of the AST created;
- Refactoring and quick action suggestions that can be applied to the respective software, through the suggestion system of Visual Studio Code (“Bulb Linting” system).

In this way, all the points presented previously allow to create a proposal that reduces the developer’s effort, improving not only the development of his software system but also his programming experience.

5.5 Architecture

The tool described will be composed of eight main packages, each with its own focus. The created system activity will start in the package “AST” and it will be propagated throughout the remaining packages. Figure 5.1 (p. 71) represents a high-level package diagram of the tool created.

The objectives of each package consist of:

**AST.** This package should analyze the AST of each TypeScript or JavaScript file. To do this, it initially will use the FileSystem package to get the contents of each file and the Git package to get the different files stored in the project’s Git repository. Also, it will use the CodeMetrics and Refactoring packages’ to check each AST and therefore detecting and calculating some software metrics and refactoring and quick modifications’ opportunities. Finally, it will use the Utils package to use common functionalities and also to initiate the process that creates both the metrics visualization systems and the linting suggestion.
Proposed Solution

**FileSystem.** This package is a secondary package that will read the content of each project’s file that will be further analyzed.

**Git.** The package Git will connect the tool to the project’s Git repository. Initially, it will copy the project to a temporary folder where we will subsequently check each committed version, in order to analyze the different software metrics throughout each version. This package will use the CodeMetrics package to check the AST of each file and, then, to calculate the values of each metric. Like it happens in CodeMetrics, this package will use some features implemented in Utils.

**CodeMetrics.** This package will create different data structures where we will store the different metrics, whether analyzed by file, project or Git version. Furthermore, it will contain some mathematical formulas to calculate several metrics, taking into account the results obtained previously through the AST.

**Refactoring.** This package will create a data structure where we will store the different refactoring and quick actions’ type, and their respective position in the file under analysis, to create the correct suggestion. Besides, it will contain one other package inside called RefactoringExtension where we will implement the code to execute the respective refactorings and quick actions detected previously.

**Linting.** The Linting package will create the refactoring and quick actions’ suggestion system using the Visual Studio Code “Bulb Linting Provider” where the suggestions are given through the bulb icon that appears near the code and also in the text editor’s “Problem Panel”. To do so, it will use the data about the refactorings and quick actions detected, stored through the Refactoring package, to know which of them need to be suggested. Furthermore, it will use some auxiliary functions that will be implemented in Utils. This package will also implement some of the liveness concepts allowing to update the refactorings and actions suggested regarding the source code’s modifications and also the currently active text editor.

**Visualization.** This package will create the most detailed visualization of the metrics selected for analysis, using the CodeMetrics package for more information about each metric and also the Git package to get information on the calculated metrics of each Git version checked. In this package will be also implemented some of the liveness concepts that allow updating the different metrics when a developer modifies his source code or changes his active text editor.

**Utils.** This package will serve as a basis for other packages since it hosts several features and auxiliary functions. Although it will be an auxiliary package, it will use packages such as AST, CodeMetrics, Linting and Visualization for the implementation of certain details, namely those related to the visual construction of the software metrics’ evolution or to the refactoring and quick action’s suggestion.
5.6 Case Study

The presented proposal and consequently the tool developed should be tested and evaluated to validate its relevance, take into account the main hypothesis described in Section 4.3 (p. 61) and its research questions (cf. Section 4.4, p. 62). Therefore, several projects, with great complexity and dimension, that were developed in TypeScript were analyzed. However, virtually all systems found either were fairly well implemented or it did not allow new users to easily add or modify features without a prior knowledge about it, like the VSCode’s API\(^1\) or the TypeScript’s\(^2\) system. Thus, we developed our own project to be used as a case study, since in this way we were able to force it to have certain aspects, with a certain structure and a certain number of critical metrics, to test the main solution proposed. Therefore, we created the “Supermarket” project that is a TypeScript project, which consists of a system that mimics the operation of a supermarket chain. In this way, this project is composed by several classes and functions, each one with its own purposes. The

\(^1\)Visual Studio Code, https://github.com/Microsoft/vscode, Last access on 8 June, 2019
\(^2\)TypeScript, https://github.com/Microsoft/TypeScript, Last access on 8 June, 2019
Proposed Solution

The architecture of this project is described in Figure 5.2.

Figure 5.2: Architecture of the project used as case study.

This project was chosen as a case study because its functionalities are familiar to everybody. Since it was necessary to have some complexity for better validation, we developed some features and functions with a weak structure and a reduced level of comprehension and maintainability.

Figure 5.3 (p. 73) shows the use of the tool with the case study developed, describing some of its software metrics.

Furthermore, “Supermarket”, besides containing the implementation of the expected behaviors related to a supermarket and a chain of supermarkets, also contains different unit tests developed using the Mocha test framework, in order to guarantee that the features developed are well implemented and that the improvements done in the system are effectively valid refactorings, which do not change the expected behavior.

Thus, this case study was selected to be used in a controlled experiment, as presented in Chapter 8 (p. 97), to evaluate and validate the tool developed and its relevance.

---

3Mocha Test Framework, https://mochajs.org, Last access on 7 June, 2019
5.7 Summary

This chapter contextualized the research project developed by the software engineering group regarding live software development (cf. Section 5.1, p. 65).

In addition, we identified the main objectives (cf. Section 5.2, p. 66) as well as all the different requirements (cf. Section 5.3, p. 66), functional and non-functional, also describing the architecture that will be adopted (cf. Section 5.5, p. 69). Finally, we presented the case study (cf. Section 5.6, p. 71) that aims to facilitate the evaluation of the proposed solution, taking into account the intended objectives and its hypothesis (cf. Section 4.3, p. 61) and respective research questions (cf. Section 4.4, p. 62).
Proposed Solution
Chapter 6

Live Static Metrics Analysis

6.1 Software Metrics

This dissertation focused on the development of a Visual Studio Code’s extension that does a live analysis of software metrics, while a developer is programming. This chapter explains the implementation strategies of the main components of this tool, describing the software metrics supported by this tool (cf. Section 6.1, p. 75), how their analysis and calculation was done (cf. Section 6.2, p. 77), how we integrate the liveness concept with it (cf. Section 6.3, p. 83) and how we did the Git Integration (cf. Section 6.4, p. 84). Furthermore, the methodology created for the visualization of the different metrics will be also described in (cf. Section 6.5, p. 85).

6.1 Software Metrics

One of the objectives of this dissertation was to analyze different software metrics, with value and relevance for software developers. In this way, the metrics supported by this project and that were evaluated and analyzed are:

Total of Lines. Total number of lines including code, commented and blank lines.

Lines of Code. Total number of code lines not including blank or commented lines.
Live Static Metrics Analysis

**Comment Lines.** Total number of commented lines. This metrics analysis single line comments and also multi-line comments.

**Blank Lines.** Total number of blank lines.

**Classes.** Total number of classes implemented.

**Classes’ Properties.** Total number of classes’ properties implemented.

**Functions.** Total number of functions developed.

**Conditions.** Checks the number of possible branches that result from different condition expressions, depending on whether they are evaluated in true or false.

**Loops.** Calculates the total number of loops.

**Cyclomatic Complexity.** Metric that checks the number of independent paths in a piece of code.

**Operands + Operators.** Calculates the total number of operators and operands in the source code. This value is calculated using the Equation 6.1 (p. 81).

**Vocabulary.** Calculates the total number of distinct operators and operands in the source code. This metric is calculated using the Equation 6.2 (p. 81).

**Volume.** Size of an algorithm implementation calculated using the Equation 6.3 (p. 81).

**Difficulty.** Calculates, through the Equation 6.4 (p. 82), the difficulty in understanding or writing a software program.

**Effort.** Calculates, using the Equation 6.5 (p. 82), the effort to perceive or write a software program.

**Time.** Calculates the time required to understand a program, using the Equation 6.6 (p. 82).

**Level.** Calculates the program’s level through the Equation 6.7 (p. 82).

**Bugs Delivered.** Estimates, through the Equation 6.8 (p. 82), the number of bugs that can exist in an implementation.

**Maintainability.** Calculates the degree or index of maintainability of a software system, using the Equation 6.10 (p. 83).

Metrics such as the total number of lines in a file, number of lines of code or blank lines give information to a programmer about the dimensions of his code, showing that files with a large number of lines may be harder to change or understand. The number of commented lines allows a developer to know if a certain file is too commented, making that file a document and not a code’s file. Also, it lets him know if there are few comments in his project.
The number of classes, class’ properties, conditions and loops show whether the software system is complex, as the number of lines in each file, showing that the larger the value, the greater the difficulty in changing or understanding it will be.

The cyclomatic complexity allows analyzing the number of independent paths that exist in a software system, in this case, that exist in each code file implemented. This metric is relevant to a software developer because it allows limiting the complexity of his system as it is being programmed, as well as to analyze the structuredness of his program. Furthermore, a programmer through this metric can know in advance how many software tests he needs to do to cover all possible paths.

We also calculated the number of operators or operands used, in general or in single occurrences and the volume of an implementation, because they allow a developer to verify if he uses a large number of operands and operators and whether they are being used repeatedly or if a function does to many things, introducing errors or bugs in the system more easily without the developer knowing it. Other metrics such as effort, difficulty, program’s level, time or number of bugs delivered are estimated metrics based on the number of operands and operators, also allowing a programmer to predict problems in his system and predict the difficulty and effort required to improve it.

Finally, the maintainability index is the most complete metric since it calculates the maintainability of a software system, in this case, of each file of a certain project, taking into account not only the number of lines of code but also the number of comments, the volume of the respective implementation and the cyclomatic complexity of the developed code. In this case, if all these values are critical the maintainability index will also present a critical value informing the programmer that he will have difficulty in modifying or adapting his system in the future, since it should have an excessive number of lines of code or commented lines or even functions that implement too many features or contain multiple possible paths, which can cause bugs or failures more easily.

All of these metrics can be calculated per file (.js or .ts files) or project, presenting total values, averages or estimates on the state of a software system.

6.2 Metrics Detection and Calculation

In this section, we describe the detection and calculation method used for each one of the metrics supported by the tool created.

Many of the metrics supported by this tool is calculated via AST or through the analysis of each file contents. Metrics such as the number of lines in a file, lines of code, blank lines, or commented lines is calculated by taking into account each line of a given file. As can be seen in the Algorithm 1 (p. 78), the number of blank lines is detected depending on whether a file’s line is empty or not. If this line’s content length is zero, the metric is incremented. As for the comments’ detection, these are done via regex. There are 4 distinct regex types that can detect each line of a comment:
The first regex presented detects single line comments, giving match with all lines of a file beginning with “//”. The second regex allows detecting the starting line of a multiline comment since it matches all the lines started by “/*” or “/**”. The third regex detects the contents of a multiline comment since it matches all the lines started by “*”. Finally, the fourth regex detects lines of code that end with “*/” or “/**” which represents the end of a multiline comment. If a line matches with any of these regex expressions, the number of comment lines is incremented.

All lines that are not empty or are not classified as comments are evaluated as lines of code. After detecting all these metrics, the total number of lines in a file is calculated through the length of that respective file.

The remaining metrics supported by this tool are detected and calculated by analyzing the AST that represents each file or through mathematical formulas. To create and to analyze a file’s AST, we used the VSCode’s typescript package that allows reading a file corresponding AST that contains nodes of type Node, each of them with a specific syntaxKind which represents the value of each respective node. For example, a function’s declaration has as its initial node, in its AST, a node with syntaxKind “FunctionDeclaration”.

Algorithm 1 Different lines type’s detection and calculation through a file’s content

1: numLines ← length(fileLines)  
2: numBlankLines ← 0  
3: numCodeLines ← 0  
4: numComments ← 0  
5: for each line ∈ fileLines do ▷ After reading each line of a certain file  
6: if length(line) = 0 then  
7: numBlankLines ← numBlankLines + 1  
8: else if line matches singleLineCommentRegex or line matches singleLineCommentBegin or line matches singleLineCommentMiddle or line matches singleLineCommentEnd then  
9: numComments ← numComments + 1  
10: else  
11: numCodeLines ← numCodeLines + 1  
12: end if  
13: end for  

78
The Algorithm 2 describes the method used to detect and to calculate metrics such as the number of loops, conditions, classes, functions, class’ properties and also the cyclomatic complexity. The number of loops in code is detected by “for”, “while” or “do” statements. Whenever one of these types of statements is found in the AST, the number of loops is increased. The number of conditions is incremented each time an AST node has the “if” or “switch” statements’ kind. Metrics such as the number of classes, functions, or classes’ properties are detected and calculated when a node representing “class”, “function” or “property” statements is found while going through the respective AST, increasing the metric that corresponds to the node type detected.

Cyclomatic complexity is also detected and calculated through AST analysis. However, because it is a more detailed analysis, it is done by another auxiliary function called “increaseCyclomatic” that accepts as a parameter the AST node and returns a boolean that represents if the value of the cyclomatic complexity of that file under analysis needs to be incremented or not. The function increaseCyclomatic(node) is presented by the Algorithm 3 (p. 80) pseudocode.

As Algorithm 3 (p. 80) shows, the function increaseCyclomatic(node) checks whether the node passed as parameter is of certain kinds:
Live Static Metrics Analysis

1. If the node represents a “catch clause”, the cyclomatic complexity should only be increased if its statements’ number is greater than zero;

2. If the node’s kind is a “catch”, a “condition” represented by “if” statements or a loop represented by “for”, “do” and “while” statements the cyclomatic complexity should be increased;

3. If the node represents a binary expression, the cyclomatic complexity should be increased if this expression has a “bar bar token” which in code is represented by “||” or a “ampersand ampersand token”. Otherwise, this function will return “false” and then the cyclomatic complexity won’t be incremented.

**Algorithm 3** increaseCyclomatic procedure’s definition

```plaintext
1: procedure INCREASE_CYCLOMATIC (node)
2:    switch node.kind do
3:        case CaseClause then
4:            return (length(node.statements) > 0)
5:        case CatchClause then
6:            case ConditionalExpression then
7:            case DoStatement then
8:            case ForStatement then
9:            case ForInStatement then
10:           case ForOfStatement then
11:           case IfStatement then
12:           case WhileStatement then
13:               return true
14:        case BinaryExpression then
15:            switch kind(node.operatorToken) do
16:                case BarBarToken then
17:                    case AmpersandAmpersandToken then
18:                        return true
19:                    default then
20:                        return false
21:                default then
22:                    return false
23:    end procedure
```

Finally, the latest values that are calculated directly through the AST analysis are the number of operands, operators and unique operands and operators, respectively. Their detection is done through the Algorithm 4 (p. 81).
Through this algorithm, it is verified that whenever a node that is an identifier or a literal expression is found, the number of operands is incremented and its text is stored in a data structure of type “Set” to avoid repeated data. If the analyzed node is a token or a keyword, the number of operators is incremented, saving the text of that node in another “Set” containing only the found operators. In the end, the length of each “Set” indicates the number of unique operators and operands present in a TypeScript or JavaScript file.

**Algorithm 4 Operators and operands calculation**

```plaintext
1: operatorsFound ← Set
2: operandsFound ← Set
3: operands ← 0
4: operators ← 0
5: if isIdentifier(node) then
6:   operands ← operands + 1
7:   operandsFound.add(node.escapedText)
8: else if isLiteralExpression(node) then
9:   operands ← operands + 1
10:  operandsFound.add(node.text)
11: else if isToken(node) or kind(node) ≥ FirstKeyword and kind(node) ≤ LastKeyword then
12:   operators ← operators + 1
13:  operatorsFound.add(node.text or node.kind)
14: end if
15: uniqueOperators ← operatorsFound.size
16: uniqueOperands ← operandsFound.size
```

From this last calculations (cf. Algorithm 4) it’s possible to calculate all Halstead metrics [Hal77]. The first Halstead metric that can be calculated is the length or more precisely the total value of operands and operators implemented in a file (cf. Equation 6.1).

\[
length = \text{operands} + \text{operators} \quad (6.1)
\]

Then, we can calculate the vocabulary metric that represents the total number of unique operands and unique operators in a file (cf. Equation 6.2).

\[
vocabulary = \text{uniqueOperators} + \text{uniqueOperands} \quad (6.2)
\]

The volume of an implementation (cf. Equation 6.3) doesn’t have a calculation formula as simple as the length or vocabulary, because it takes into account the latter two values calculated.

\[
volume = length \times \log(vocabulary) \quad (6.3)
\]
Live Static Metrics Analysis

All of the following metrics are calculated taking into account other Halstead metrics [Hal77], using more complex mathematical formulas.

The difficulty in understanding implemented code (cf. Equation 6.4) is calculated using the number of operators, operands and also the number of unique operands. The more operators and operands a developer use, more difficulty he has to understand his implementation, especially if he uses repeatedly the same operands, introducing many times some bugs into the developed software system [Hal77].

$$\text{difficulty} = \frac{\text{uniqueOperators}}{2} \times \frac{\text{operands}}{\text{uniqueOperands}}$$ (6.4)

The effort to understand an implementation is directly proportional to the volume of an algorithm and the difficulty in understanding it as can be verified in Equation 6.5 [Hal77].

$$\text{effort} = \text{volume} \times \text{difficulty}$$ (6.5)

The level of the implementations made in a file is inversely proportional to the difficulty in understanding an algorithm, as can be seen in the Equation 6.6 [Hal77].

$$\text{level} = \frac{1}{\text{difficulty}}$$ (6.6)

The effort metric is related to the coding time (in seconds) needed to understand and implement an algorithm. Halstead found [Hal77] that the relationship between these metrics could be verified by the Equation 6.7.

$$\text{time} = \frac{\text{effort}}{18}$$ (6.7)

The last metric is an estimation of the number of bugs that an implementation can present, which according to Halstead [Hal77] can be calculated in two different ways. The first one, represented by the Equation 6.8, takes into account the effort to understand an implementation raised to two-thirds, while the second formula (cf. Equation 6.9) takes into account the volume of the same implementation. However, in the developed tool, the calculation of this metric is made using Equation 6.8, since it is the most used and considered the most correct one to calculate the number of bugs delivered by a system.

$$\text{bugsDelivered} = \frac{\text{effort}^2}{3000}$$ (6.8)

$$\text{bugsDelivered} = \frac{\text{volume}}{3000}$$ (6.9)

The formula for calculating the maintainability of a software system takes into account other previously calculated metrics [Kuk, Mac, CALO94, SB14]. To calculate this value (cf. Equation 6.10, p. 83), we should take into account the volume of an implementation (V), the number of
Live Static Metrics Analysis

lines of code (LoC), the cyclomatic complexity (C) and the number of lines with comments (CL).

\[
\text{maintainability} = 171 - \frac{(5.2 \times \log(V)) - (0.23 \times C) - (16.2 \times \log(LoC)) + (50 \times \sin(\sqrt{2.4 \times CL}))}{171}
\]

(6.10)

6.3 Live Analysis

The liveness concept allows a user, in this case, a developer, to check the outcome of his programming action as he is developing his system, reducing or removing the edit-compile-run cycle. In this case, liveness is applied either to the analysis of software metrics, allowing to visualize their evolution as a developer changes his system, or to the refactoring or quick actions’ suggestion. The implementation of liveness in this context was possible through several features that the VSCode API contains.

The main live feature of this tool, as already mentioned, is related to visualizing the evolution of several metrics throughout the various programming actions, and for this, it was necessary to be aware of all the moments in which a developer is changing his system. VSCode lets us know when a file is changed through two events, one that checks if a file has been saved (\texttt{workspace.onWillSaveTextDocument}) and another that verifies if a file’s text has been modified (\texttt{workspace.onDidChangeTextDocument}). Although the second event is the one that seems to be the most correct to be subscribed, this event is triggered whenever a character is changed in a file, analyzing software metrics thousands of times in a short period. Thus, it was decided that the tool would subscribe the first event, \texttt{workspace.onWillSaveTextDocument}, to apply the liveness concept, because it is an event that is only triggered when a developer finishes changing a certain file.

Even though this event is only triggered when a developer stops programming, he can stop and immediately after he can change again that file in question, triggering 2 events of the same type, creating to metrics’ analysis almost at the same time. To prevent this from happening, a 10-second timeout was implemented that only allows the metrics to be analyzed 10 seconds after the event was triggered.

Another thing to keep in mind is when a developer modifies the active file, where he is programming because whenever this happens, the software metrics need to be changed to match the values for the new active file. This feature was also implemented through the VSCode API, subscribing the \texttt{window.onDidChangeActiveTextEditor} event. With this, whenever the active text editor changes, the tool gets its name and content through the \texttt{window.activeTextEditor} feature, allowing to create its AST and then to analyze its different software metrics.

All steps previously described are also used for the analysis and suggestion of refactorings and quick actions.
Live Static Metrics Analysis

```
subscriptions.push(vscode.workspace.onWillSaveTextDocument(function () {
    setTimeout(function () {
        readExtractMetrics(); // to read the file’s metrics
    }, 10000); // 10000 milliseconds or another configurable value
}});
```

Listing 6.2: workspace.onWillSaveTextDocument implementation example

6.4 Git Integration

The Git integration is done mostly using command-line instructions. This is only possible since VSCode allows to execute synchronous commands through a function called `execSync(commands)`. Thus, for this integration to be as simple as possible, not decreasing the extension’s performance, this process starts with the verification of the respective project’s Git repository URL. To do this, first, we need to go the folder where the project is stored locally (using the command “cd srcPath” where `srcPath` is the project current location) and then through a command like “`Git config -get remote.origin.url`” get the necessary URL to continue this integration.

Then we go to the temporary folder `tmp`, where a copy of the project will be stored, to remove any instance of that directory, that could already exist there. This step can be executed through the commands “cd tmp” and “rm -rf folder”, where `folder` is the name of the project’s folder. To later know the working directory where the project will be copied and where the software metrics will be analyzed, we can execute the “`pwd`”, which returns the path to the respective directory.

After that, the project is copied to the temporary folder via the “`cp -a srcPath initialDir`” command, where `srcPath` is the current project path and `initialDir` is the path for the `tmp` folder. After the project has been copied, the new folder will be accessed through “`cd tmp/folder`” and the command “`Git log -oneline -decorate> log.txt`” will be executed to create the `log.txt` file with all hashes that identify all commits made. Finally, the current working directory is re-wired through the “`pwd`” command, so that later it is possible to access the folder of the copied project, where the Git software metrics analysis will take place.

The process of copying the project folder from one side to another could be replaced by running a “`Git clone`” command creating an instance of the repository’s project, but at performance level this option would not be the best, since this task could take a long time to be finished, because it depends on the Internet connection, compromising the remaining behaviors of the tool.

So, after all these steps, each line of the `log.txt` file is read, using the path to this previously saved directory, to get the hash of each commit. With each one of this hashes the command “`Git checkout hash`” is executed, inside the correct directory (the command “`cd tmp/folder`” should be executed to access the correct folder). Finally, we need to remove the file `index.lock` after each “`Git checkout`” so that no problems occur between each Git checkout execution.

After each checkout, the metrics of that project’s are analyzed and saved using the Algorithms 1, 2, 3, 4, which are described in Section 6.2.
All previously described steps are described in Algorithm 5.

**Algorithm 5** Git integration.

1: `srcPath ← path(workspace) + “/src”`
2: `GitURL ← execSync(’
   cd ’+ srcPath + ’
   Git config –get remote.origin.url
’)`
3: `folder ← folder(GitURL)`
4: `initialDir ← execSync(’
   cd tmp
   rm -rf ‘+ folder + ‘
   pwd
’)`
5: `dir ← execSync(’
   cp -a ‘+ srcPath + ‘ ‘+ initialDir + ‘
   cd tmp/’+ folder + ‘
   Git log –oneline –decorate > log.txt
   pwd
’)`

▷ Reading each one of `log.txt` file’s lines (`log.txt` is located in folder `dir`)
6: for each line ∈ fileLines do
7: `hash ← hash(line)`
8: `execSync(’
   cd tmp/’+ folder + ‘
   Git checkout ‘+ hash + ‘
   rm -f ./Git/index.lock
’)`

▷ Metrics’ analysis of each Git commit
9: end for

### 6.5 Software Metrics Visualization

The software metrics analyzed and calculated by this tool are presented laterally through different views in the Visual Studio Code “Activity Bar”. For these to be presented to the user, he needs to click on the “Live Software Metrics” option, as shown in Figure 6.1 (p. 86).

Selecting this option, the user has access to 2 different views. The first one, named “Software Metrics”, enumerates all the metrics supported by this tool, presenting the name of each one of them and verifying their description when hovering them. The second view, named “Metrics Analysis”, displays the metrics that are under review and have been selected by the user.
As it can be seen in Figure 6.1, each option in the view “Software Metrics” contains an icon “+”, which allows the user to add that metric to the list of metrics under analysis on the second view.

By selecting the “+” icon, the user has access to a “Quick Pick Menu” where he can select which of the options he wants to use the selected metric. It has 3 options to choose from (Figure 6.2), where the first option allows analyzing that respective metric for the currently active text editor, the second option is to check that metric for the entire project and the third allows that metric to be always analyzed for that file in question. If the 3 options presented by this menu are selected, the metric selected is taken from the first view. When the active file changed, the metrics’ listing changes depending on the current context.

After the user completes the metric selection’s process, this metric is displayed in the second view, referring to the previously selected options. If a user is reviewing a particular file or the project, this option is displayed immediately after the metric’s name (cf. Figure 6.3, p. 87). Ahead of the name of each metric, there are 2 icons. The first one is the “-” icon, that allows deselecting the metric in question and the second one is a magnifying glass that allows visualizing, in a more detailed way, that metric. Then, underneath the name of each metric are presented 2 bar charts (cf. Figure 6.3, p. 87) that represent the evolution of the metrics during the development of the
Live Static Metrics Analysis

software system (A) and the evolution of the metrics over each Git commits (B). The bars of each chart are represented by ASCII characters so that they can be displayed.

Figure 6.3: Metric’s evolution overview.

By clicking on the “-” icon, if the metric is being analyzed for the currently active file and that file is simultaneously locked for constant analysis, the user has access to a second “Quick Pick Menu” (cf. Figure 6.4) where he can choose which option he wants to deselect.

Figure 6.4: Metric’s deselection quick pick menu.

If a user selects the magnifying glass icon, to view a metric in more detail, a webview opens automatically, presenting a line chart that compares the evolution of the metric throughout the programming actions and throughout the different Git commits. When he clicks on the magnifying glass icon and that metric has already been selected previously and is already being detailed in the line chart, that metric is taken from the webview. The content of the webview that contains the charts of the selected metrics changes according to the currently active text editor.

Figure 6.5: Metric’s evolution chart.
6.6 Summary

This chapter synthesized the approach used to implement the live analysis of software metrics, presenting not only the metrics supported by the developed tool (cf. Section 6.1, p. 75) but also its methods of detection and calculation (cf. Section 6.2, p. 77), through several algorithms. All metrics that have been selected to integrate this tool are explained in detail in this chapter, referring to their relevance to this research project.

Moreover, the integration method of the liveness concept, that allows giving feedback related to the different metrics calculated while programming, was also described in Section 6.3 (p. 83). Likewise, the methodology used to integrate these functionalities with Git was also mentioned in Section 6.4 (p. 84), so that the user could compare the evolution of his metrics with each version previously stored in the project’s Git repository.

Finally, we presented the visual form with which the evolution of the different calculated software metrics is presented to the user, as well as the relevance of the same. This visual functionality is described through algorithms and images that support the main explanation (cf. Section 6.5, p. 85).

The implementations described, in this chapter, also mention all the capabilities of the VSCode that allowed to implement all the presented algorithms and functionalities.

The next chapter, Chapter 7 (p. 89), will address the implementation of live refactorings and quick actions, describing the actions supported by this tool, as well as its methods of detection, suggestion, and execution.
Chapter 7

Live Refactorings and Quick Actions

Suggestion

7.1 Refactorings ......................................................... 89
7.2 Quick Actions ..................................................... 91
7.3 Suggestions Visualization ........................................ 94
7.4 Summary .......................................................... 95

This project has as main focus the analysis of software metrics. However, the tool created also allows the suggestion and execution of some refactoring methods or quick actions that improve the software system or help the programmer to develop his program.

This chapter describes the refactoring methods or quick actions supported by this plugin, that we adapted from the refactorix tool [Oetb, Oeta]. Furthermore, we explained how the liveness concept was conjugated with this subject and also how these methods are visually presented to the tool’s user.

7.1 Refactorings

The tool developed analyzes the “Split” and “Extract Variable” refactoring opportunities. As previously indicated these refactorings were adapted from the refactorix tool [Oetb, Oeta], but their source code was improved to include new features. These refactorings and the modifications made in each one of them are described in

Split Variable Declaration. Refactoring method that split a variable declaration from its initialization. Through refactorix [Oetb, Oeta], if a variable had not instantiated its type this would
Live Refactorings and Quick Actions Suggestion

have a default type named “type”. Now, the tool created can identify the type of the variable even when it is not defined.

**Before:**

```plaintext
1 const x = false;
2 const y: number = 1;
3 const name = new Name("Ana");
```

**After:**

```plaintext
1 let x: boolean;
2 x = false;
3
4 let y: number;
5 y = 1;
6
7 let name: Name;
8 name = new Name("Ana");
```

Listing 7.1: Tool’s Split Variable Declaration execution

**Extract Variable.** This refactoring simplifies a complex expression, by dividing it into two or more parts to better understand it. Through refactorix [Oetb, Oeta], the new constant variable created would be named as “var_name”, but now with this tool, the new variable defined as a name related to the initial variable that was extracted. Also, this tool also analyzes the type of the variable that will be extracted.

**Before:**

```plaintext
1 var x = false;
2 var y: number = 1;
```

**After:**

```plaintext
1 const var_x = false;
2 var x = var_x;
3
4 const var_y: number = 1;
5 var y: number = var_y;
```

Listing 7.2: Tool’s Extract Variable execution
The different refactoring opportunities are detected through the analysis of each file’s AST, as can be seen in the Algorithm 6. The different files’ ASTs are created and analyzed using the VSCode’s *typescript* package, as previously described in Section 6.2 (p. 77).

If the AST nodes represent a variable’s declaration and if that variable is in fact initialized (the node has to include the initialization signal, “=”), the refactoring that can be applied in this context is the “Split Variable Declaration”, to split the declaration and initialization of the variable represented by the respective node.

If the AST nodes are allusive to a variable’s statement and if that variable isn’t instantiated as constant, the refactoring that can be executed in this context is the “Extract Variable”, where the selected variable can be extracted to a new constant variable.

Algorithm 6 Refactoring opportunities detection.

```plaintext
1: if kind(node) = VariableDeclaration then
2:   if text(node) includes “=” then
3:     refactoring ← splitVariableDeclaration
4:   end if
5: else if kind(node) = VariableStatement then
6:   if text(node) not includes “const” then
7:     refactoring ← extractVariable
8: end if
9: end if
```

All refactorings that are detected, suggested and executed are not suggested again in the future, only if the developer rolls back his actions.

### 7.2 Quick Actions

The tool developed analyzes and suggests some quick actions that improve the software appearance and helps a developer implementing his system. In this way, actions such as the creation of getters and setters for class’ properties, the insertion of semicolons at the end of expressions that do not contain a semicolon, the modification of arrow functions and the access to class properties are suggested. As previously mentioned these actions were adapted from the refactorix tool [Oetb, Oeta], but some changes were made so that these actions could be performed equally in JavaScript and not just in TypeScript. All these quick actions listed above are described in more detail below.

**Property to Getter/Setter.** Suggests the implementation of “get” and “set” methods for class’ properties that don’t have this kind of methods implemented. An example of its execution in TypeScript is presented below.
Before:

```javascript
class Color {
  rgb: string;
}
```

After:

```javascript
class Color {
  private _rgb: string;
  get rgb(): string { return this._rgb; }
  set rgb(value: string) { this._rgb = value; }
}
```

Listing 7.3: Tool’s Getter and Setter implementation in TypeScript

This feature in the refactorix tool [Oetb, Oeta] could only be executed in a TypeScript software system, however, we change it so that a developer could use it in JavaScript. An example of its execution in JavaScript is presented below.

Before:

```javascript
class Color {
  constructor(rgb){
    this.rgb = rgb;
  }
}
```

After:

```javascript
class Color {
  constructor(rgb){
    this.rgb = rgb;
  }
  get rgb(): string { return this.rgb; }
  set rgb(value) { this.rgb = value; }
}
```

Listing 7.4: Tool’s Getter and Setter implementation in JavaScript

**Semicolon.** Checks if each expression is finished with a semicolon. This feature is in line with good programming practices and it’s why it has been included in the developed tool.
Live Refactorings and Quick Actions Suggestion

**Arrow function toggle.** This feature allows changing a single statement block to an arrow function’s expression and vice versa. Arrow function toggle was implemented as a way for the user to change quickly his programming style since the correction is automatic once the suggestion is accepted.

**Before:**

```
1   () => 0;
```

**After:**

```
1   () => { return 0; };
```

Listing 7.5: Tool’s arrow function toggle between single statement block and expression

**Property Access.** Toggles between “private”, “public” or “protected” access. This feature can only be suggested in TypeScript projects (as in the refactorix tool [Oetb, Oeta]) since JavaScript programs do not declare the access to class’ properties and it was implemented in the tool created as a way for the user to change quickly his programming style, since the correction is automatic once the suggestion is accepted.

Searching for quick actions’ opportunities is done by traversing the AST of a file, as it is done with the refactoring opportunities. Algorithm 7 (p. 94) shows how AST is traversed and analyzed to search for these respective opportunities.

If the node under analysis represents an expression or declaration and if it doesn’t include a semicolon in its text, the quick action to be suggested is to add the semicolon at the end of that expression or statement.

If the node represents the declaration of a class’ property and if the programming language used is TypeScript, the algorithm checks whether this property contains a get and set method. If there are no such methods associated with the property under analysis, the quick action that will be suggested is to create the required getter and setter. If this node doesn’t represent a property’s declaration but represents a class’ constructor, and if the programming language of the file is JavaScript, the algorithm checks whether each property declared inside that constructor contains a get and set method declared. If this doesn’t happen, the quick action that creates these methods will be suggested.

Finally, if there is a node that represents an arrow function, the algorithm checks if this function contains only a single statement or an expression. If it’s a single statement the quick action suggested is to convert this statement into an expression, if it’s an expression it’s suggested to convert that expression into a single statement.
Live Refactorings and Quick Actions Suggestion

Algorithm 7 Quick actions opportunities detection.

1: \textbf{if} \text{kind}(\text{node}) = \text{VariableStatement} \textbf{or} \text{kind}(\text{node}) = \text{ExpressionStatement} \textbf{then}  
2: \hspace{1em} \textbf{if} \text{text}(\text{node}) \text{ not includes } ";" \textbf{then}  
3: \hspace{2em} \text{action} \leftarrow \text{semicolons}  
4: \hspace{2em} \textbf{end if}  
5: \textbf{else if} \text{kind}(\text{node}) = \text{PropertyDeclaration} \textbf{then}  
6: \hspace{1em} \text{action} \leftarrow \text{toggle}  
7: \hspace{2em} \textbf{if} \text{language}(\text{document}) = "typescript" \textbf{and} \text{!hasGet()} \textbf{and} \text{!hasSet()} \textbf{then}  
8: \hspace{3em} \text{action} = \text{toGetterSetter}  
9: \hspace{2em} \textbf{end if}  
10: \textbf{else if} \text{kind}(\text{node}) = \text{ArrowFunction} \textbf{then}  
11: \hspace{1em} \textbf{if} \text{singleStatement} \textbf{then}  
12: \hspace{2em} \text{action} \leftarrow \text{toggleSingleStatementBlockExpression}  
13: \hspace{2em} \textbf{else}  
14: \hspace{3em} \text{action} \leftarrow \text{toggleBlockExpressionSingleStatement}  
15: \hspace{2em} \textbf{end if}  
16: \hspace{1em} \textbf{end if}  
17: \textbf{else if} \text{kind}(\text{node}) = \text{Constructor} \textbf{then}  
18: \hspace{1em} \textbf{if} \text{language}(\text{document}) = "javascript" \textbf{and} \text{!hasGet()} \textbf{and} \text{!hasSet()} \textbf{then}  
19: \hspace{2em} \text{action} \leftarrow \text{toGetterSetter}  
20: \hspace{2em} \textbf{end if}  
21: \textbf{end if}

All these quick actions that are detected, suggested and executed are not suggested again in the future, only if the developer rolls back his actions.

7.3 Suggestions Visualization

Since the analysis of refactoring and other quick actions’ opportunities is a complement to this research project, it’s necessary that the user first activates the analysis of these actions. To do this, he needs to click on the “Refactoring” option found in the Visual Studio Code “Status Bar”, as shown in Figure 7.1.

![Figure 7.1: Refactoring and quick actions suggestion’s activation.](image)

After activating the analysis of refactoring and quick actions’ opportunities, these restructurings are presented to the user through a linting provider that allows creating actions that are suggested and executed when the user selects them. These actions are presented by underlining the code and
listed through the menu presented by the “Bulb Linting Provider” of the VSCode, as shown in Figure 7.2.

![Figure 7.2: Refactoring or quick actions’ suggestion using the VSCode “Bulb Linting Provider”.](image)

In addition to presenting the suggestions through the “Bulb Linting Provider”, these are also listed in the “Problem Panel” which is usually located in the lower part of the Visual Studio Code’s text editor, as shown in Figure 7.3. In each of the enumerated descriptions, the user can still check the line and column in which that suggestion is being applied. By clicking on the description, its location in the code is highlighted and the “bulb” icon is displayed so that the user can execute that suggestion.

![Figure 7.3: “Problem Panel” with the description of each refactoring or quick action suggested.](image)

### 7.4 Summary

This chapter described the different refactorings and quick actions supported by the tool created, presenting examples of their performance and their importance to the user.

Furthermore, it was also described the approach, using different algorithms, for the detection and suggestion of refactoring and quick actions’ opportunities, to improve a software system, through its structure and appearance. All of this was described in Section 7.1 (p. 89) and Section 7.2 (p. 91).

Finally, how these suggestions are presented to the tool’s user was reported in Section 7.3 (p. 94), describing also the visual forms in which these actions are represented and seen, either through the Visual Studio Code’s “Bulb Linting Provider” or “Problem Panel”.

95
Live Refactorings and Quick Actions Suggestion
The objective of this dissertation was to develop a tool capable of reducing the effort in understanding and maintaining a software system. In order to validate the performance of the tool relative to its hypothesis and research questions, (cf. Section 4.3, p. 61) and (cf. Section 4.4, p. 62), respectively, we developed a controlled experiment that scrutinizes the respective tool’s execution in a software system created as a case study. Thus, in this chapter, we describe the empirical evaluation carried out, throughout its objectives, Section 8.1, guidelines (cf. Section 8.2, p. 98) and planning (cf. Section 8.3, p. 99), tasks performed, (cf. Section 8.4, p. 100) and, consequently, the respective results obtained and their extensive analysis, Section 8.5 (p. 102), and also the main validation’s threats (cf. Section 8.8, p. 126).

8.1 Objectives

With this controlled experiment, using the developed case study (cf. Section 5.6, p. 71) with the different defined tasks (cf. Section 8.4, p. 100), we wanted to evaluate how well our solution
Empirical Evaluation

worked when tested by other people, namely if it helped them improve the quality of their code, their development time and also the convergence of their solution to an optimal solution. With all this, once the participants were divided into two groups, one using the tested tool and another in which the tool was not used, it was expected that the experiment participants who used the tool would improve better the software used and its metrics, improving also the ability to understand and maintain it rather than those that didn’t use the tool.

Besides, we also wanted to evaluate the users’ ability to identify the different software metrics throughout the software’s code, verifying if they were able to improve the respective metrics, taking into account the evolution of those metrics being analyzed.

Regarding the tool execution, we also tried to verify if the respective participants could understand the purpose of the tool developed, as well as all the implemented functionalities, from the evolution of software metrics to the suggestion of refactorings and quick actions, in order to be able to improve the project under analysis, without having to resort to the tool’s tutorial multiple times to understand how to use it.

In general, we wanted to see the evolution of two different groups of participants throughout the different experiment’s tasks, with and without the tool developed, allowing us to evaluate their performance and also the tool’s performance, as well as its importance and relevance, verifying if the participants showed interest in using such a tool like the one implemented, during their daily programming routine.

Thus, we were hoping to have results that allowed us to compare the evolution of the different metrics, the final code quality and also the participant’s development time, with and without the tool created, in order to validate the hypothesis enumerated in Section 4.3 (p. 61) as a solution to the problems previously described in Section 4.1 (p. 59).

8.2 Guidelines

To carry out the validation process of the solution proposed in the Chapter 5 (p. 65), some guidelines were defined so that it was possible to achieve the objectives as well as to prove the main hypothesis of this dissertation (cf. Section 4.3, p. 61) throughout a controlled experiment. These guidelines are described below.

All participants should be from the computer engineering’s area. In this case, it was important that all participants be acquainted with software systems and with the consequent comprehension and analysis of code. Thus, all the participants should have experience in the field of computer engineering, namely in the area of software development.

Educational component. As this empirical study occurred mainly with students in a controlled environment, this experiment should contribute to their learning process, namely about the development and maintenance’s process of a software system with better structure and quality.
Motivation of the participant. The participant should be motivated to take part in this experiment by knowing that the tool created is unique and that it helps a developer while he is programming.

Providing a tool and project’s tutorial. In order for this experiment to be performed in the best way so that in the future the comparison and analysis of the obtained results was done in a reliable way, it was necessary that all the participants had access to a tutorial about the tool and project that they were using and analyzing, knowing all about their features, learning how to use it. Preferably, this tutorial and learning phase should be performed before the start of the controlled experiment.

Limit the experiment’s duration. All participants should be advised of the maximum duration of each task they will perform.

Tool relationship. To maximize confidence in the results obtained, participants shouldn’t have any experience with the tool being tested.

8.3 Experience Planning

This controlled experiment had the objective of getting the participant to experience the concept of liveness, while programming, observing the evolution of his software metrics and if whether or not there is a reduction of effort in understanding and maintaining a software system, increasing its quality, reducing the development time, allowing to converge faster to a good solution. For this, this controlled experiment had to be planned previously, delineating certain important parameters.

Device. For this controlled experience, the only device required was a computer with any operating system installed. In this case, it was used a MacBook with MacOS operating system installed.

Local. The experiment was executed indoors in a room of the Department of Informatics Engineering of the Faculty of Engineering of the University of Porto, where the necessary equipment for its realization was located.

Participants. All participants of this experience participated freely and spontaneously, fostered by the curiosity to use and understand the tool developed. These participants were divided into two distinct groups, one of which uses the developed tool and the other not, in order to validate its relevance and need. The sample of participants only includes people who are in direct contact with the area of computer engineering and programming, namely students, researchers, and teachers. These participants signed a consent (cf. Appendix A, p. 151) to participate in this experiment after becoming aware of the experience’s main context and also of the confidentiality of the data provided by them.

Duration. Before starting the controlled experiment, we tried to test it ourselves by checking both the results obtained and the time we need to complete each task, since it is thought
that limiting the duration of the experiment can make the participants manage better their own time, valuing their experience with the tool developed. Each one of the tasks has a predefined duration distributed in proportion to its difficulty and complexity, allowing a maximum total duration of 20 to 50 minutes for this experiment. However, even so, each of the four tasks was timed to compare the time-to-time with which it was truly necessary to perform the different tasks. This metric is not only important as a term of comparison between the time taken to accomplish each task but also as a form of comparison between participants, trying to validate the second research question (cf. Section 4.4, p. 62).

**Tool Exposure.** All participants showed interest in participating in this experiment, as they consider relevant the use of a tool of this kind to assist the software development, although they have never had previous contact with this specific tool. As such, all participants initially received an explanation and tutorial about the tool and how to use it, as the first task of this experiment. Through this task, called Task 0, each participant was exposed to all the features supported by the tool, thus obtaining feedback about how easy was executing it. Since this first task was an initialization and comprehension’s task, we give it a short but necessary time duration, so that the participants could assimilate all the tool’s functionalities.

**Questionnaire.** A questionnaire (cf. Appendix B, p. 153) was developed to answer each one of the tasks described in Section 8.4. This questionnaire begins with some questions that categorize the experience’s participants. There are a group of questions for each task performed. Finally, it was some questions related to the usefulness and usability of the tool, taking into account the System Usability Scale [Bro13].

**Difficulty.** The difficulty of each task was defined by this experiment’s authors. But, even so, each participant could evaluate it through the questionnaire developed (cf. Appendix B, p. 153) where he had questions where he could answer about the difficulty in understanding the purpose of the task and its execution. In addition, the experiment’s controller throughout the experience tries to perceive the difficulty with which the participants performed each task, seeing if they finished it with success or failure.

**Data Integrity.** All of the participants signed the participation consent, Appendix A (p. 151), in advance, where they were informed of the tool and project’s authorship. Equally, in order to maximize the criticism and to prevent them from trying to be generous in their responses, which would have a negative influence on the data collected, each one was encouraged to be critical and to offer his sincere opinions and suggestions, since all the answers given in the questionnaire were anonymous (cf. Appendix B, p. 153).

### 8.4 Testing Tasks

All participants were invited to perform four distinct tasks followed by a set of characteristic questions, to evaluate and validate each one of the different situations. These participants were
Empirical Evaluation

divided into two distinct groups, one of which used the developed tool and the other not. This
second group could only execute a script to check if the purpose of each task had been fulfilled.

**Task T0: Recognition and experience of tool and project’s functionalities.**

This task, that should last between 5 to 10 minutes, was subdivided into two distinct tasks
that allowed the participant to understand both the tool developed and the project used as a
case study.

**Tool/Script Recognition.** The participant should learn how to use the functionalities pro-
vided by the developed tool or script, depending on if he used it or not. After this
introductory phase, the objective was to experience the functionalities of the project
used as a case study.

**Project Recognition.** After a participant learns all the functionalities of the developed tool,
he should inspect the project that he will be using, in order to understand its structure
and all the features implemented on it. Also, he could also run the tool developed, to
analyze the software metrics that characterize the project at that initial phase.

**Task T1: Add an object to an array without duplicates, with as few metrics as possible**

The participant had the goal of implementing the `addEmployee()` function, in 5 minutes,
in which an object of the type `Employee` need to be added to an array that holds all the
employees of a supermarket. This array shouldn’t have repeated employees, as it happens
in a normal supermarket.

During this task, the participant had to respect a set of metrics defined by the authors, such
as the number of lines of code that should be lower than 70, the function’s cyclomatic
complexity that should be lower than 4, its number of loops and conditions that should be
about 2. He could also change the data structure where the supermarket’s employees were
stored if he thought that it benefited the implementation of this function and minimizes the
desired metrics.

**Task T2: Develop a function’s code with many actions involved**

This task allowed the participants to implement the `purchase()` function code, where a pur-
chase can be made in a supermarket. This function was complex because it affected several
other features and data structures implemented in the project created as a case study. To
implement this function all the participants has a maximum time of 20 minutes.

During this task, the participant had to respect a set of metrics defined by the authors. They
have to try to develop a function with less than 55 lines of code, with a cyclomatic com-
plexity lower than 20 and an effort approximately equal to 2600 with 0 bugs delivered and
a maintainability index higher than 60.

**Task T3: Refactoring a function**
Empirical Evaluation

In this task, with the duration of 10 minutes, the participant needed to improve the check-Discount() function’s code, by refactoring it, taking into account its current structure, his software engineering knowledge, and software metrics.

In this way, the goal of this task was to reduce the number of lines of code to a maximum of 130, the implementation and comprehension’s effort and volume to 20,000 and 750, respectively and its cyclomatic complexity to less than 30, increasing in this way its maintainability index.

The first two development tasks (T1 and T2) were tasks where the participant had to effectively create the code for two functions, from the root, being able to refactor it after he finished his development or if he couldn’t fulfill the proposed objectives. The last task was a simple refactoring task.

8.5 Tasks’ Evaluation

The controlled experiment was carried out by 16 participants, all students, finalists of the Integrated Master in Informatics and Computing Engineering, with competencies in Software Engineering.

Most of the analyzed results were obtained by timing the execution of each task, as well as analyzing the evolution of each metric over time. In addition, we also took into account the responses given by the participants to the questionnaire to which they were subjected (cf. Appendix B, p. 153), as well as the opinion of the experience controller. All the form’s questions were related to the participants’ characterization, the tasks performed and also about the evaluation of the tool created.

Regarding the main questions of the questionnaire (cf. Appendix B, p. 153), these were based on the Likert scale [Lik32, Alb97], assigning values between 1 and 5, which correspond to “I totally disagree”, “I disagree”, “I have no opinion”, “I agree” and “I totally agree” or “I never program”, “I rarely program”, “I program casually”, “I program multiple times” and “I program daily” or even “I never used”, “I hardly ever use”, “I rarely use”, “I use casually” and “I use frequently”. Also, there were some multiple-choice or single-selection questions that were analyzed through charts.

8.5.1 Participants’ Characterization

We analyze some characteristic aspects of the participants of the controlled experiment, such their gender, age, and academic qualifications.

As it can be seen in Figure 8.1 (p. 103), the distribution of participants by gender is very uneven since 87% of the participants were men and only 13% were women. Besides, we verified that all the participants had the same academic degree, that in this case, it was the bachelor’s degree since all of them were finalists of the Integrated Master in Informatics and Computing Engineering.
Likewise, the age group of all participants also didn’t vary, since all of them were between 18 and 25 years old, respectively.

Gender and age factors weren’t considered very significant for this experiment. However, the academic degree, that didn’t vary either, was an important factor to analyze since it allowed to understand if all the participants had or not the same technological knowledge. In addition to this characterization elements, other questions were analyzed that allowed to verify the knowledge of the participants regarding software engineering, programming (cf. Section 8.2, p. 103), TypeScript (cf. Figure 8.3, p. 104) and regarding the refactoring concept (cf. Figure 8.4, p. 104).

As previously noted, other factors were taken into account. One of these factors was the experience in programming (cf. Figure 8.2a), which was considered a preponderant and significant factor for this experiment. Regarding this factor, all participants confirmed to have experience in programming, among which 56.25% said that they program daily, 31.25% program often or multiple times and only 12.5% said that they program casually.

We were also questioned whether these 16 participants would normally have difficulty in understanding a software system when they analyzed it for the first time (cf. Figure 8.2b) and most of the participants demonstrated that they didn’t have difficulty comprehending programs since 81.25% disagreed with the statement presented (37.5% said that they have no difficulty and 43.75% said
that they have little difficulty). Although 81.25% is a high percentage, 8.75% of participants, even though most of them are constantly in contact with software systems (cf. Figure 8.2a, p. 103), have difficulty in understanding the software programs in which they work.

Participants then had to give us information about their TypeScript knowledge (cf. Figure 8.3). Most of the participants (63.5%) had never used it or knew little or nothing about this language except that it was similar to JavaScript. In this way, only 37.5% (25% uses TypeScript frequently and 12.5% uses it casually) indicated to have good knowledge in TypeScript.

Through the different tasks, the experiment’s controller recorded whether or not the participants understood TypeScript, and in fact, only 12.5% of the 16 participants actually knew this language and were able to use all of its features correctly.

It was also verified if the participants knew what a refactoring was (cf. Figure 8.3) and 87.5% of them were able to select the correct definition of a refactoring, demonstrating that they really knew in what a refactoring consists, which let them understand what the Task 3’s objectives really were since this task was only a refactoring exercise.

Finally, we tried to verify if the participants had already used software metrics analysis or refactoring suggestions tools (cf. Figure 8.5, p. 105), with 56% of them saying that they had never
used metric’s analysis tools or knew what they were (cf. Figure 8.5a). Regarding refactoring suggestion’s tools, 72% said they have already used tools like that, namely those that have previously been installed in the different IDEs (cf. Figure 8.5b).

![Figure 8.5: Verification if the participants already used software metrics or refactoring tools.](image)

### 8.5.2 Task 1

After completing Task 1, each participant answered a set of questions, depending on whether or not the tool was used. One of the first questions that they had to answer was whether they were able to identify the metrics analyzed in this task (cf. Figure 8.6a, p. 106) to which 75% of the participants, whether they had used the tool or not, demonstrated that they understood 100% all the metrics that were calculated. The remaining 4 participants were equally distributed by the remaining options, demonstrated that they understood all the metrics, but not with so much confidence as the 12 participants that were able to identify clearly all the necessary metrics.

Another question that they also had to answer and which is intertwined with the previous question was whether or not they had difficulty in understanding the purpose of this task and the reason behind calculating that software metrics (cf. Figure 8.6b, p. 106). With this, it was verified that 87.5% of the participants, whether or not using the tool, were able to fully understand the purpose of the task and the reason why we calculate metrics, in that task, such as number of lines of code, cyclomatic complexity, number of conditions or number of loops. Then the remaining participants had different answers since 1 participant who didn’t use the tool said that he understood the purpose of the task while another one who used the tool said that he wasn’t able to comprehend it and that he had difficulties in understanding the purpose of those selected metrics.
Empirical Evaluation

Figure 8.6: Checking if the participants were able to identify and understood the necessary metrics in Task 1.

Moreover, we verified that the time that each participant took to complete each task and also if he was able to reach the goals established for each metric. As can be seen in the Figure 8.7, more participants were able to finish the task in time, using the tool created, than those who did not use it.

Figure 8.7: Completion of Task 1.

The execution times of each participant, taking into account the use or non-use of the tool in the Task 1 are presented in Figure 8.8, where “A1”, “A2”, “A3”, “A4”, “A5”, “A6”, “A7” and “A8” are the values of time for each of the 8 participants who used the tool or not. Pink-painted cells represent cases where participants exceeded the legislated time this task. Thus, taking into account only the samples in which the participants were able to finish the task in time, the average execution time of this task, its first and third quartiles (Q1 and Q3, respectively) and respective deviation were calculated, as it can be seen in Table 8.1 (p. 107).

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>A5</th>
<th>A6</th>
<th>A7</th>
<th>A8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>With Tool</td>
<td>5min</td>
<td>5min 27s</td>
<td>4min 2s</td>
<td>2min 25s</td>
<td>3min 13s</td>
<td>1min 15s</td>
<td>5min 19s</td>
<td>2min 17s</td>
</tr>
<tr>
<td></td>
<td>Without Tool</td>
<td>6min 11s</td>
<td>4min 42s</td>
<td>3min 49s</td>
<td>5min 21s</td>
<td>3min 23s</td>
<td>5min 30s</td>
<td>2min 10s</td>
<td>2min 31s</td>
</tr>
</tbody>
</table>

Figure 8.8: Task 1 execution’s times.
Empirical Evaluation

Table 8.1: Task 1’s average time, respective quartiles and deviation values.

<table>
<thead>
<tr>
<th>Task 1</th>
<th>Q1</th>
<th>Average Time</th>
<th>Q3</th>
<th>Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>With Tool</td>
<td>3min 2s</td>
<td>2min 19s</td>
<td>3min 50s</td>
<td>1min 21s</td>
</tr>
<tr>
<td>Without Tool</td>
<td>3min 19s</td>
<td>2min 31s</td>
<td>3min 49s</td>
<td>1min 1s</td>
</tr>
</tbody>
</table>

Through the analysis of the Table 8.1, it was verified that on average the time values for the execution of Task 1 were better using the tool than without it, but the difference between the values wasn’t accentuated enough, also because of the deviations’ values.

As it can be seen in Figure 8.9a, all the participants who were able to finish the task in time were able to achieve the desired goals by reaching all the metric values requested in Task 1. The others also managed to reach the goals, but they exceeded their time and their results were not taken into account. It was also verified in Figure 8.9b that 87.5% of the participants found the task easy to execute, with or without the tool.

![Figure 8.9: Checking if the participants were able to achieve the Task 1’s goal and if it was easy to do.](image)

(a) Achieving the main goal.  
(b) Task’s easiness.

After all of these questions, there were a couple of specific questions that the participants had to answer, regarding whether or not they used the tool created

**With Tool**

Those who used the tool had to answer some usability questions that will be discussed later in Section 8.6 (p. 115). In addition, they were asked if they felt that the tool had helped them during that task or not, to which 87% of the 8 participants that used it answered positively, demonstrating that they verified the impact of the tool and the relevance of it relative to the evolution of the software metrics that were analyzed in this task.
Empirical Evaluation

Figure 8.10: Tool’s help performing Task 1.

Without Tool

In order to evaluate the opinion of the participants who didn’t use the tool, about the possibility of using it in the future and if it could help them, two different questions were asked. The first one was to verify if the participants thought that they would be able to execute the Task 1 more easily, using the tool (cf. Figure 8.11a) and the second one was to verify if they thought that they could carry out the task faster if they had used it (cf. Figure 8.11b).

Figure 8.11: Participants’ who didn’t use the tool evaluating if it would help them performing Task 1 easily and faster.

With these questions, it was verified that 50% of the participants who didn’t use the tool thought that when they could use it, they could perform the task more easily (cf. Figure 8.11a), and the remaining 50% don’t know if this would actually improve the easiness of the task. With these values, we can also verify that none of the participants thought that the tool wouldn’t be needed to execute Task 1, more easily.
Empirical Evaluation

With the second question (cf. Figure 8.11b, p. 108), it was found that 50% of participants who didn’t use the tool thought that it might help them do the task faster. It was also verified that in this case, 13% of the participants thought that the tool wouldn’t improve this task resolution’s time, while 37% think otherwise.

8.5.3 Task 2

After finishing Task 2, each participant answered another set of questions equal to the one in Task 1 (cf. Section 8.5.2, p. 105). The first question they had to answer was if they were able to identify the metrics analyzed in this task (cf. Figure 8.12a) to which 56.25% of the participants said that they understood 100% all the metrics calculated in there. The remaining 7 participants were mostly distributed by the second best answer, with 25% of them saying that it was easy to understand it.

The second question that was related to the first one checks if they had difficulty in comprehending the objective of this task (cf. Figure 8.12b), which 56.25% of the participants said that they were able to fully understand the purpose of the task and the reason why metrics, such as number of lines of code, cyclomatic complexity, number of bugs, effort or maintainability index, were calculated. Then the remaining participants had different answers. Another thing verified was that whoever used the tool understood better the purpose of the task than the others.

![Figure 8.12: Checking if the participants were able to identify and understood the necessary metrics in Task 2.](image)

(a) Identifying the metrics  
(b) Difficulty in understanding

Again, it was also calculated the execution time of Task 2 and if the participants were able to fulfill all the metrics requested in it. As it can be seen in the Figure 8.13 (p. 110), more participants were able to finish the task in time, using the tool created, than those who did not use it, as it happened in Task 1 (cf. Section 8.5.2, p. 105).

The Task 2’s execution times of each participant, taking into account the use or non-use of the respective tool are presented in Figure 8.14 (p. 110), using the same method used in the evaluation of Task 1 (cf. Section 8.5.2, p. 105). Then, taking into account only the samples in which the participants were able to finish the task in time, the average execution time of this task, its first and
Empirical Evaluation

Third quartiles (Q1 and Q3, respectively) and the respective deviation were calculated, as it can be seen in Table 8.2.

![Figure 8.13: Completion of Task 2.](image)

![Figure 8.14: Task 2 execution’s times.](image)

<table>
<thead>
<tr>
<th>Task 2</th>
<th>Q1</th>
<th>Average Time</th>
<th>Q3</th>
<th>Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>With Tool</strong></td>
<td>11min 34s</td>
<td>15min 20s</td>
<td>18min 56s</td>
<td>4min 7s</td>
</tr>
<tr>
<td><strong>Without Tool</strong></td>
<td>15min 45s</td>
<td>16min 38s</td>
<td>17min 44s</td>
<td>1min 46s</td>
</tr>
</tbody>
</table>

Through the analysis of the Table 8.2, it was verified that on average the time values for the execution of Task 2 were better using the tool than without it, but the difference between the values wasn’t very accentuated.

![Figure 8.15: Checking if the participants were able to achieve the Task 2’s goal and if it was easy to do.](image)

Table 8.2: Task 2’s average time, respective quartiles and deviation values.
Empirical Evaluation

Then, as it can be analyzed in Figure 8.15a (p. 110), 62.5% of the participants who were able to finish the task in time were also able to achieve the desired objectives by reaching all the metrics requested in Task 2. Two participants, one with the tool and the other without it, weren’t able to achieve all the task’s goals because they didn’t reach the desired effort value. Then, others managed to reach the goals, but they exceeded their time and their results were not taken into account. It was also verified in Figure 8.15b (p. 110) that 87.5% of the participants confirmed that they thought that the task was completely easy to accomplish.

After all of these questions, as it happened in Task 1 (cf. Section 8.5.2, p. 105), there were a set of questions that the participants had to answer, regarding whether or not they used the tool developed.

**With Tool**

Those who used the tool had to answer some usability questions that will be discussed later in Section 8.6 (p. 115). Moreover, we asked them if they felt the tool had helped them during this task’s exercise resolution or not, to which 87% of them answered positively, demonstrating that they verified the impact of the tool relative to this task software metrics' evolution.

![Figure 8.16: Tool’s help performing Task 2.](image)

**Without Tool**

To evaluate the opinion of the participants who didn’t use the tool, about the possibility of using it in the future and if it could help them, two different questions were asked. The first one was to verify if the participants thought that they would be able to execute the Task 2 more easily, using it (cf. Figure 8.17a, p. 112) and the second one was to verify if they thought that they could carry out the task faster if they had used the tool (cf. Figure 8.17b, p. 112).

With these questions, it was verified that 50% of the participants who didn’t use the tool thought that when they could use it, they could perform the task more easily (cf. Figure 8.17a, p. 112) and the remaining 50% don’t know if this would actually improve the easiness of the task. With these values, we can also verify that none of the participants thought that the tool wouldn’t be needed to execute Task 2 more easily.
Empirical Evaluation

With the second question (cf. Figure 8.17b), it was found that 50% of participants who didn’t use the tool thought that it might help them do the task faster. It was also verified that in this case, 13% of the participants thought that the tool wouldn’t improve this task resolution’s time, while 37% think otherwise.

![If I had used the tool, I would have fulfilled the task easily](image1)
![If I had used the tool, you would have fulfilled the task faster](image2)

Figure 8.17: Participants’ who didn’t use the tool evaluating if it would help them performing Task 2 easily and faster.

8.5.4 Task 3

After completing Task 3, as it happens in Task 1 (cf. Section 8.5.2, p. 105) and Task 2 (cf. Section 8.5.3, p. 109), each participant answered a set of questions, depending on if they used or not the tool developed. So like the other tasks, the first question they had to answer was if they were able to identify the different metrics related to that task (cf. Figure 8.18a, p. 113), to which 56.25% of the participants said that they understood 100% all the metrics calculated in there. The remaining 7 participants were mostly distributed by the second best answer, with 18.75% of them saying that it was easy to comprehend the respective metrics. We also verified through this question in this task that whoever used the tool identified the metrics more easily.

The second question that they had to answer was related to the first one. In this question, they had to evaluate the difficulty in understanding the objective of the task and why that set of metrics would be calculated (cf. Figure 8.18b, p. 113). Through this question, it was verified that 56.25% of the participants, whether or not they used the tool, were able to fully understand the objective of the task and the main reason why we analyze that set of software metrics composed by the number of lines of code, the cyclomatic complexity, number of bugs delivered, volume or the maintainability index. Then, the remaining participants had different answers, distributed by the different options possible. We also verified that whoever used the tool identified the objective of the task easily.
Empirical Evaluation

(a) Identifying the metrics
(b) Difficulty in understanding

Figure 8.18: Checking if the participants were able to identify and understood the necessary metrics in Task 3.

In this task, we also calculate the time that each participant took to complete this task and if he was able to reach the goals established. As it can be seen in Figure 8.19, more participants were able to finish the task in time, using the tool created, than those who did not use it, as it happened in Task 1 (cf. Section 8.5.2, p. 105) and Task 2 (cf. Section 8.5.3, p. 109).

Figure 8.19: Completion of Task 3.

The execution times of each participant, taking into account the use or non-use of the tool for Task 3 are presented in Figure 8.20. The colored cells represent the cases in which the participants couldn’t finish the task in time. Then, taking into account only the samples in which the participants were able to finish the exercise in time, the average execution time of this task, its first and third quartiles (Q1 and Q3, respectively) and respective deviation were calculated, as it can be seen in Table 8.3 (p. 114).

Figure 8.20: Task 3 execution’s times.
Empirical Evaluation

Table 8.3: Task 3’s average time, respective quartiles and deviation values.

<table>
<thead>
<tr>
<th>Task 1</th>
<th>Q1 Average Time</th>
<th>Q3 Average Time</th>
<th>Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>With Tool</strong></td>
<td>2min 5s</td>
<td>4min 21s</td>
<td>5min 9s</td>
</tr>
<tr>
<td><strong>Without Tool</strong></td>
<td>2min 43s</td>
<td>4min 48s</td>
<td>6min 3s</td>
</tr>
</tbody>
</table>

Through the analysis of the Table 8.3, it was verified that on average Task 3’s execution times were better using the tool than without it, but the difference between the values weren’t accentuated enough, because the standard deviation is too big to prove that the development time is better when using the tool.

As it can be seen in Figure 8.21a, 81.25% of the participants who were able to finish the task in time were able to achieve the desired objectives by reaching all the metric values requested in Task 3, but one participant that used the tool wasn’t able to fulfill the desired task’s volume. Then, the others managed to reach the goals, but they exceeded their time and their results were not taken into account. We also verified that 93.75% of the participants found the task easy to execute (choosing the highest options), with or without the tool (cf. Figure 8.21b).

Figure 8.21: Checking if the participants were able to achieve the Task 3’s goal and if it was easy to do.

Like it happens with Task 1 (cf. Section 8.5.2, p. 105) and Task 2 (cf. Section 8.5.3, p. 109), after all of these questions, we also did some questions about regarding the use or non-use of the tool developed.

**With Tool**

In Section 8.6 (p. 115) there are some usability questions and answers that the participants who used the tool had to answer. In addition to that, they had also to give us feedback about if they felt that the tool had helped them during this task or not.

Through this question in Task 3, 100% of them answered positively, demonstrating that all of them verified the impact of the tool about the software metrics’ evolution during the execution of this task.
Empirical Evaluation

Without Tool

To evaluate the opinion of the participants who didn’t use the tool in Task3, about the possibility of using it in the future and if it could help them, two different questions were asked. The first one was to verify if the participants thought that they would be able to execute the Task 3 more easily, using the tool (cf. Figure 8.22a) and the second one was to verify if they thought that they could accomplish the task’s goals faster if they had used it (cf. Figure 8.22b).

![Figure 8.22: Participants’ who didn’t use the tool evaluating if it would help them performing Task 3 easily and faster.](image)

Through these questions, it was verified that 50% of the kind of participants thought that maybe when they could use it, they could perform the task more easily (cf. Figure 8.22a) and then 13% of them thought that the tool wouldn’t improve the task’s easiness, while 37% think otherwise.

With the second question (cf. Figure 8.22b), it was found that 63% of these participants thought that the tool could help them execute the task faster than without it. It was also verified that in this case, 37% of the participants thought that the tool would, in fact, improve this task resolution’s time.

8.6 Usability Evaluation

At the end of the questionnaire (cf. Appendix B, p. 153), the participants who used the tool had to answer some questions related to more specific details of it as well as some usability questions, taking into account the system’s usability scale (SUS), which consists of 10 questions that aims to pointing out aspects that deserve attention regarding the developed tool’s usability [Bro13].

Figure 8.23 (p. 116) shows the percentages of each option relatively to each tool comprehension’s question and Figure 8.24 (p. 116) shows the percentages of answers given to the SUS questionnaire [Bro13]. The percentages are presented in a colored way, in order to reduce the comprehension effort of the respective tables. The percentages that negatively evaluate the tool’s
Empirical Evaluation

test

comprehension and usability were represented by reddish tones while the percentages representing positive values were represented by green tones, depending on their respective value.

Regarding the tool’s comprehension (cf. Figure 8.23), it was verified that all questions were scored with the highest scores, with a percentage equal or higher than 50%. However, questions related to the refactorings and quick actions’ suggestion system were the ones with the lowest percentage in the highest levels, since the participants did not interact much with this part of the tool during the controlled experiment. Besides, it was also verified that the participants understood how the tool works (scored 75%) and that it was easy to use (scored 87.5%), presenting well all the information (scored 62.5%) and all the calculated metrics (scored 62.5%). It was also checked that the participants understood the colors used in the tool (scored 75%), the buttons and their locations (scored 87.5%) and that, finally, they found the tool very intuitive (scored 75%). Notwithstanding this very positive feedback, each participant could leave some comments or opinions about the controlled experience or even about the tool that they used. These tool’s suggestions were mostly about the less well-scored questions and they are identified as possible future work, in Section 9.4 (p. 139).

Figure 8.23: Tool comprehension’s evaluation.

The SUS [Bro13] allows to give a score to a tool regarding its usability, taking into account the answers given to the different 10 questions that SUS questionnaire (cf. Appendix B, p. 153) has.

Figure 8.24: System’s Usability Scale.
Empirical Evaluation

This score varies between A and F, as it can be seen in Table 8.4.

Table 8.4: SUS grading scale.

<table>
<thead>
<tr>
<th>SUS Score</th>
<th>Grade</th>
<th>Adjective Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 80.3</td>
<td>A</td>
<td>Excellent</td>
</tr>
<tr>
<td>68 - 80.3</td>
<td>B</td>
<td>Good</td>
</tr>
<tr>
<td>68</td>
<td>C</td>
<td>Ok</td>
</tr>
<tr>
<td>51 - 68</td>
<td>D</td>
<td>Poor</td>
</tr>
<tr>
<td>&lt; 51</td>
<td>F</td>
<td>Awful</td>
</tr>
</tbody>
</table>

These values are calculated through several steps [Bro13]:

1. For odd questions (1, 3, 5, 7 and 9), subtract 1 from the score given by the participant;
2. For the even questions (2, 4, 6 and 8), the given answer must be subtracted from 5;
3. In the end, the final values of the odd and even questions must be added by participant and multiplied by 2.5 (eg. \((\text{oDDS+evens}) \times 2.5\)) and, then, we have to calculate the average of the results obtained.

Through this process, we obtained a score of 90.9375, which means that the tool is rated as Excellent, with grade A which is the highest possible score in this grading system. In this way, the participants show interest in the tested tool, since its usability value was high.

8.7 Research Question’s Validation

This section aims to present the validation of the research questions (cf. Section 4.4, p. 62) that underlie the hypothesis (cf. Section 4.3, p. 61) presented by this dissertation. In order to validate them, we decide to execute hypothesis tests to get some conclusions about the values obtained during the different tasks of this controlled experiment and consequently about the possibility of validating or not the different research questions and the respective main hypothesis.

All hypothesis tests performed for each one of the different research questions were performed using Independent-Samples t-tests [KK15] since we analyze independent variables for two distinct groups, which in this case consist of the participants who used or not the tool created.

Through this test, it is possible to reject or not a null hypothesis, through the analysis of a set of values. In our case, we used the SPSS software\(^1\) to perform this kind of tests, and the results’ output are divided into two different sections. The first one, named Group Statistics, gives us basic information regarding the sample that is being compared, including the sample’s size, its mean value, its standard deviation and also its standard error (cf. Figure 8.25, p. 118). Then,

---

\(^1\)IBM SPSS, [https://www.ibm.com/analytics/spss-statistics-software](https://www.ibm.com/analytics/spss-statistics-software), Last access on 26 June, 2019
Empirical Evaluation

it also has a second section, called **Independent Samples Test**, that shows us the most relevant values in the **Independent Samples t Test**, providing us with two different sets of information. The first one is about the **Levene’s Test for Equality of variances** (A) and the second is related to the **t-test values for Equality of Means** (B), as it can be seen in Figure 8.26. Through the **Levene’s Test** we have access to two different values:

- **F** that is the statistical test of Levene’s test;
- **Sig.** that is the p-value related to this statistic test.

In this case, if the **p-value** is too low (eg. p <0.001), we have to analyze the second row of each sample that in our case is the row related to the non-use of the tool by the different participants. Otherwise, we have to check the first one that is related to the values calculated when the participants used the tool developed. So, in the respective line to search, we have to check the **Sig. (2-tailed)** value that is the **p-value** related to the given statistic test (T) and the corresponding degrees of freedom (**DF**). The **Sig. (2-tailed)** value is displayed in the **t-test values for Equality of Means** group of information. In order to deny the null hypothesis, the value of this **p-value** must be lower than 0.05.

![Figure 8.25: Group Statistics’ results example.](image)

![Figure 8.26: Independent Samples Test’ results example.](image)

### 8.7.1 RQ1 - Code Quality

In order to validate the RQ1 research question (**cf.** Section 4.4, p. 62) concerning to the quality of the code of a software system, we used a **Independent Samples t Test** to verify the possibility of denying the respective null hypothesis related to this research question (**cf.** Table 8.5, p. 119).
Denying the respective null hypothesis, it is possible to validate the alternative hypothesis and therefore it is possible to validate the respective research question.

**RQ1** - When developers have constant exposure to software metrics, do they reach better solutions?

Table 8.5: Hypothesis Tests related to the research question RQ1.

<table>
<thead>
<tr>
<th>Null Hypothesis</th>
<th>Alternative Hypothesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H_0$: Developers didn’t reach better solutions when constantly exposed to their software metrics.</td>
<td>$H_1$: Developers reach better solutions when constantly exposed to their software metrics.</td>
</tr>
</tbody>
</table>

To verify if the quality of code improves when a developer is exposed to the constant evolution of software metrics, we measured the final values of two different metrics such as the number of lines of code and the maintainability of the system used as a case study, throughout the different tasks that the participants had to do.

Initially, we first tried to validate this research question using the number of lines of code. Table 8.6 presents the mean values of this metric in the different tasks, using or not the tool developed, as well as their standard deviations and standard errors. From this table, it was verified that on average, the number of final lines in all tasks is better, when we are using the tool, except for Task 3. However, the calculated standard deviations didn’t allow us to conclude with certainty that this may occur in all cases, not allowing to validate the respective research question using only this kind of information.

Table 8.6: T-Test’s group statistics about task’s lines of code.

<table>
<thead>
<tr>
<th>Type</th>
<th>Mean</th>
<th>Std. Deviation</th>
<th>Std. Error</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>With Tool</td>
<td>64.25</td>
<td>2.121</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>Without Tool</td>
<td>65.25</td>
<td>3.105</td>
<td>1.098</td>
<td></td>
</tr>
<tr>
<td>Task 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>With Tool</td>
<td>102.25</td>
<td>3.370</td>
<td>1.191</td>
<td></td>
</tr>
<tr>
<td>Without Tool</td>
<td>102.625</td>
<td>4.406</td>
<td>1.558</td>
<td></td>
</tr>
<tr>
<td>Task 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>With Tool</td>
<td>120.75</td>
<td>2.188</td>
<td>0.773</td>
<td></td>
</tr>
<tr>
<td>Without Tool</td>
<td>119.5</td>
<td>6.347</td>
<td>2.244</td>
<td></td>
</tr>
</tbody>
</table>

The independent-samples t-tests also calculates a more specific and complete table (cf. Table 8.9, p. 121), which allowed us to verify whether or not it was possible to deny the null hypothesis. In this case, like it was explain in Section 8.7 (p. 117), we have to analyze the first row of each task, since the p-value given by the variable Sig. in the Levene’s Test is always greater than 0.001. Here, in each respective line, we checked the p-value given by the Sig. (2-tailed) variable of the t-test for Equality of Means. As it can be seen in Table 8.7 (p. 120), all the respective
**Empirical Evaluation**

*p*-value’s are greater than 0.05, which allowed us to verify that it isn’t possible to deny the null hypothesis, not concluding anything about this research question, regarding the number of lines of code.

Table 8.7: T-Test’s about task’s lines of code.

<table>
<thead>
<tr>
<th>Task</th>
<th>Type</th>
<th>Levene’s test</th>
<th>t-test for Equality of Means</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>F</td>
<td>Sig.</td>
</tr>
<tr>
<td>Task 1</td>
<td>With Tool</td>
<td>1.683</td>
<td>0.216</td>
</tr>
<tr>
<td></td>
<td>Without Tool</td>
<td>-0.752</td>
<td>12.365</td>
</tr>
<tr>
<td>Task 2</td>
<td>With Tool</td>
<td>0.343</td>
<td>0.567</td>
</tr>
<tr>
<td></td>
<td>Without Tool</td>
<td>-0.191</td>
<td>13.102</td>
</tr>
<tr>
<td>Task 3</td>
<td>With Tool</td>
<td>1.121</td>
<td>0.308</td>
</tr>
<tr>
<td></td>
<td>Without Tool</td>
<td>0.527</td>
<td>8.640</td>
</tr>
</tbody>
</table>

We ran exactly the same test sequence for the maintainability index’s value. In this case, only this calculated value was used after Task 2 and Task 3, since in Task 1, this metric wasn’t one of the desired metrics of this task. The new results obtained for maintainability can be verified in Table 8.8 and Table 8.9 (p. 121).

Table 8.8: T-Test’s group statistics about task’s maintainability.

<table>
<thead>
<tr>
<th>Type</th>
<th>Mean</th>
<th>Std. Deviation</th>
<th>Std. Error Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1 + Task 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>With Tool</td>
<td>71.626</td>
<td>4.902</td>
<td>1.733</td>
</tr>
<tr>
<td>Without Tool</td>
<td>69.395</td>
<td>2.620</td>
<td>1.733</td>
</tr>
<tr>
<td>Task 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>With Tool</td>
<td>88.023</td>
<td>7.146</td>
<td>2.526</td>
</tr>
<tr>
<td>Without Tool</td>
<td>83.274</td>
<td>14.6024</td>
<td>5.163</td>
</tr>
</tbody>
</table>

As it can be seen in Table 8.8, the mean values, standard deviations and errors for the maintainability of each task were also calculated, and like it happens in the Table 8.14 (p. 125), the mean value of the final maintainability of Task 2 (including the values from Task 1) and Task 3 was higher when using the tool. However, as we can analyze in Table 8.14 (p. 125), all the cases had high standard deviations, so we couldn’t deny the null hypothesis, using this information.

Analyzing Table 8.9 (p. 121), it is verified that the p-value of each task in the Levene’s Test was not very small, which means that we have to analyze the 1st line for each task, which corresponds to the maintainability’s values calculated when the participants used the tool created. Then, analysing the p-value given by the Sig. (2-tailed), we check that these respective values were higher than 0.05 which indicates that we couldn’t deny the null hypothesis related to this research question when analyzing the maintainability indexes.
Empirical Evaluation

Table 8.9: T-Test’s about task’s maintainability.

<table>
<thead>
<tr>
<th>Task Type</th>
<th>Type</th>
<th>F</th>
<th>Sig.</th>
<th>T</th>
<th>DF</th>
<th>Sig. (2-tailed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1 + Task 2</td>
<td>With Tool</td>
<td>2.417</td>
<td>0.142</td>
<td>1.135</td>
<td>14</td>
<td>0.275</td>
</tr>
<tr>
<td>Task 1 + Task 2</td>
<td>Without Tool</td>
<td>1.135</td>
<td>0.698</td>
<td>0.281</td>
<td>10.698</td>
<td>0.281</td>
</tr>
<tr>
<td>Task 3</td>
<td>With Tool</td>
<td>2.862</td>
<td>0.113</td>
<td>0.826</td>
<td>14</td>
<td>0.422</td>
</tr>
<tr>
<td>Task 3</td>
<td>Without Tool</td>
<td>0.826</td>
<td>0.171</td>
<td>0.428</td>
<td>10.171</td>
<td>0.428</td>
</tr>
</tbody>
</table>

Since through the two Independent Samples t Tests carried out with the number of lines of code and the maintainability index, we couldn’t deny the null hypothesis and, consequently, not concluding anything about this research question and not being able to validate it.

The impossibility of validating this research question can be due to the fact that the project used as case study is of reduced size, with little complexity and with tasks of reduced difficulty degree, which means that the implementation carried out by each participant was similar, with a similar number of code lines and maintainability’s indexes. In addition, the sample of participants was too small, which may have hindered the validation of this research question. These problems are described succinctly in Section 8.8 (p. 126), where we address all the validation’s threats.

8.7.2 RQ2 - Solution’s Convergence

In order to validate the RQ2 research question (cf. Section 4.4, p. 62) concerning to the solution’s convergence, we normalized the values of the respective maintainability index of the different tasks so that it was possible to verify their evolution per participant but within the same set of iterations. In this case, we only use the value of maintainability because of it is the most important metric that this tool calculates since it combines not only the number of lines of code but also the volume of the implementation, its cyclomatic complexity, and its comments’ number.

RQ2 - When developers have constant exposure to software metrics, do they converge faster to a good solution?

Table 8.10: Hypothesis Tests related to the research question RQ2.

<table>
<thead>
<tr>
<th>Null Hypothesis</th>
<th>Alternative Hypothesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H_0$: Developers didn’t converge faster to a good solution when constantly exposed to their software metrics.</td>
<td>$H_1$: Developers converge faster to a good solution when constantly exposed to their software metrics.</td>
</tr>
</tbody>
</table>

The maintainability’s evolution, both for the participants who used the tool and for those who didn’t use it, were normalized to a time interval between 0 and 1. In this way, it was possible to
Empirical Evaluation

correctly compare the evolution of this metric per participant, regardless of the time it took them to complete each task. Figure 8.27a and Figure 8.28a demonstrate the evolution of this metric per participant, over time. The lines painted with blue tones representing the participants that used the tool created both in Task 1, Task 2 and Task 3, while the rest, represented in red tones, are the ones that didn’t use it.

Figure 8.27a and Figure 8.28a demonstrate the evolution of this metric per participant, over time. The lines painted with blue tones representing the participants that used the tool created both in Task 1, Task 2 and Task 3, while the rest, represented in red tones, are the ones that didn’t use it.

Figure 8.27: Maintainability convergence collected from Task 1 and Task 2.

Through the Figure 8.27a and Figure 8.28a, a greater abundance of blue tones lines was found at higher values in the y-axis than reddish tint lines, indicating that participants which used the tool reached a better maintainability value in that period of time, however, since we couldn’t do a very accurate analysis through these charts, because of the excessive number of overlapping lines, we calculated the mean value of each iteration for each case (using or not using the tool), resulting in the line charts of the Figure 8.27b and Figure 8.28b. Each line chart is delimited by a shaded area, with the same coloration as the respective line, determined by the first and third quartiles of the maintainability convergence over time.

Furthermore, through the analysis of the line charts and also of their shaded areas delimited by the first and third quartiles, it was verified that the maintainability’s convergence during Task 1 and Task 2, was better for the participants who used the tool. However, as can be seen in
Empirical Evaluation

Figure 8.27b (p. 122), there was an overlap of this convergence initially, which was reduced over time since this area of intersection practically ceases to exist as time goes by. During Task 3 (cf. Figure 8.28b, p. 122), we verified that the maintainability’s convergence is also better for the participants who used the developed tool, even though there was a greater intersection between the delimiting areas of each line chart. Over time, as it happened during Task 1 and Task 2, this intersection area decreases, thus verifying a better and greater solution’s convergence when using the implemented tool.

However, these data weren’t enough to reject the null hypothesis, since we couldn’t draw an absolute conclusion from the charts described in Figure 8.27 (p. 122) and Figure 8.28 (p. 122), because there was some overlap of the solution’s convergence with and without the tool, and therefore, we did a **Independent Samples t Test** using the maintainability’s average value per participant and taking into account the hypothesis test described in Table 8.10 (p. 121). The mean results, the standard deviations and standard errors for each task are presented in Table 8.11 and through its analysis it was verified that the mean values of maintainability with the tool were superior to those calculated without the tool, but nevertheless, there was a high standard deviation that didn’t allow us to conclude that the solution with tool was truly better than without it.

![Table 8.11: T-Test’s group statistics about task’s maintainability to validate RQ2.](image)

<table>
<thead>
<tr>
<th>Type</th>
<th>Mean</th>
<th>Std. Deviation</th>
<th>Std. Error Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Task 1 + Task 2</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>With Tool</em></td>
<td>80.578</td>
<td>3.902</td>
<td>1.380</td>
</tr>
<tr>
<td><em>Without Tool</em></td>
<td>77.449</td>
<td>3.522</td>
<td>1.245</td>
</tr>
<tr>
<td><strong>Task 3</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>With Tool</em></td>
<td>77.605</td>
<td>5.599</td>
<td>1.979</td>
</tr>
<tr>
<td><em>Without Tool</em></td>
<td>72.8761</td>
<td>10.742</td>
<td>3.798</td>
</tr>
</tbody>
</table>

Then, effectively with the values calculated through the t-test, it is verified that the **p-value** given by the variable **Sig.** of the **Levene’s Test** is not very reduced in both cases, indicating that the first line of each test case, related to the use of the tool, should be analyzed. Verifying the respective p-values given by the **t-test**, represented by **Sig. (2-tailed)**, we concluded that it wasn’t possible to reject the null hypothesis, even seeing that the mean values of the maintainability, using the tool, were higher than those without it, since the **p-value** wasn’t lower than 0.05.

This way, it wasn’t possible to validate this research question, because we could conclude nothing about it since we couldn’t reject the null hypothesis related to this research question. Like it happened in Section 8.7.1 (p. 118), this could be due to the fact that the case study used was too small and simple, with easy tasks to do. The reduced sample of participants can also be influential in the impossibility of validating this research question. These problems are addressed in more detailed in Section 8.8 (p. 126), where we describe all the validation’s threats.
Empirical Evaluation

Table 8.12: T-Test’s about task’s maintainability.

<table>
<thead>
<tr>
<th>Task Type</th>
<th>Levene’s test (F, Sig.)</th>
<th>t-test for equality of means (T, DF, Sig.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1 + Task 2</td>
<td>With Tool: 0.858, 0.370</td>
<td>1.684, 14, 0.114</td>
</tr>
<tr>
<td></td>
<td>Without Tool: 1.684, 13.855</td>
<td>0.115</td>
</tr>
<tr>
<td>Task 3</td>
<td>With Tool: 1.614, 0.225</td>
<td>1.104, 14, 0.288</td>
</tr>
<tr>
<td></td>
<td>Without Tool: 1.104, 10.542</td>
<td>0.294</td>
</tr>
</tbody>
</table>

8.7.3 RQ3 - Development Time

To validate the RQ3 research question (cf. Section 4.4, p. 62) that is related to the development’s time, we measured the time that each participant took to complete each task. This research question was evaluated through an independent-samples t-tests like it happens in Section 8.7.1 (p. 118) and Section 8.7.2 (p. 121), taking into account the hypothesis test described in Table 8.13.

RQ3 - When developers have constant exposure to software metrics, is the development time reduced?

Table 8.13: Hypothesis Tests related to the research question RQ3.

<table>
<thead>
<tr>
<th>Null Hypothesis</th>
<th>Alternative Hypothesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>( H_0 ): Developers didn’t reduce their development time when constantly exposed to their software metrics.</td>
<td>( H_1 ): Developers reduce their development time when constantly exposed to their software metrics.</td>
</tr>
</tbody>
</table>

Unfortunately, during the controlled experiment, some participants exceeded the maximum time given to complete each task, but nevertheless, they had the opportunity to finish it, so that their software metrics could be checked in other research question’s evaluation as it happens in Section 8.7.1 (p. 118) and Section 8.7.2 (p. 121).

The times measured by task, including the outliers and excluding them, were analyzed to verify both the discrepancy in the different values timed with and without using the tool created, or whether if the participants that were using the tool had better values. This analysis can be verified in Figure 8.29 (p. 125), where Figure 8.29a (p. 125) presents the variance of the execution times of each task, including the outliers and the Figure 8.29b (p. 125) describes the variation of the task execution’s times without the respective outliers.

Through this analysis, it can be verified that on average the execution times of each task were mostly better when the developed tool was used, and the minimum times of each task were also obtained mainly when the participant could use the tool created. However, this analysis wasn’t
Empirical Evaluation

enough to validate the respective research question and that’s why a hypothesis test (cf. Table 8.13, p. 124) of type independent-samples t-tests was carried out.

As it can be seen in Section 8.7.1 (p. 118) and Section 8.7.2 (p. 121), we analyzed the average time, its standard deviation and standard error for each of the tasks performed (cf. Table 8.14, p. 125). As mentioned previously, it was verified that the average time of execution of each task is smaller when the participants use the developed tool, however, the standard deviation is too high to be able to conclude something through these values, since they do not deny in advance the null hypothesis. We then analyzed the Levene’s test results and the t-test (cf. Table 8.15, p. 126), respectively, to try to verify whether or not this research question could be validated.

As it can be seen in (cf. Table 8.15, p. 126), the p-value presented by the variable Sig. of Levene’s test is not sufficiently reduced, so the line that should be analyzed by each task is the 1st line, which is relative to the cases in which the participants used the developed tool. From there, analyzing the p-value given by Sig. (2-tailed) of the t-test of the cases in which the tool was used, we can verify that it is not possible to reject the null hypothesis since the p-value is not lower than 0.05.

Table 8.14: T-Test’s group statistics about task’s execution’s time.

<table>
<thead>
<tr>
<th>Type</th>
<th>Mean</th>
<th>Std. Deviation</th>
<th>Std. Error Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>With Tool</td>
<td>3min 52s</td>
<td>1min 58s</td>
<td>56s</td>
</tr>
<tr>
<td>Without Tool</td>
<td>4min 2s</td>
<td>1min 47s</td>
<td>52s</td>
</tr>
<tr>
<td>Task 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>With Tool</td>
<td>16min 10s</td>
<td>4min 28s</td>
<td>1min 34s</td>
</tr>
<tr>
<td>Without Tool</td>
<td>19min 48s</td>
<td>6min 1s</td>
<td>2min 7s</td>
</tr>
<tr>
<td>Task 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>With Tool</td>
<td>4min 20s</td>
<td>3min 12s</td>
<td>1min 8s</td>
</tr>
<tr>
<td>Without Tool</td>
<td>4min 17s</td>
<td>3min 38s</td>
<td>1min 17s</td>
</tr>
</tbody>
</table>
### Table 8.15: T-Test’s about task’s execution’s time.

<table>
<thead>
<tr>
<th>Task</th>
<th>Type</th>
<th>Levene’s test</th>
<th>t-test for equality of means</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>F</td>
<td>Sig.</td>
</tr>
<tr>
<td>Task 1</td>
<td>With Tool</td>
<td>0.114</td>
<td>0.740</td>
</tr>
<tr>
<td></td>
<td>Without Tool</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 2</td>
<td>With Tool</td>
<td>0.013</td>
<td>0.911</td>
</tr>
<tr>
<td></td>
<td>Without Tool</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 3</td>
<td>With Tool</td>
<td>0.026</td>
<td>0.873</td>
</tr>
<tr>
<td></td>
<td>Without Tool</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As it can be seen in Section 8.7.1 (p. 118) and Section 8.7.2 (p. 121), it was not possible to reject the null hypothesis and therefore we couldn’t conclude anything about the respective research question. This may again be due to the fact that the project used as a case study was too simple and small, with easy tasks to be executed. Besides, as the number of participants was reduced, it was not possible to make a very extensive collection of values. All of these problems will be addressed in the next section (cf. Section 8.8, p. 126), where we describe some validations’ threats.

### 8.8 Validation Threats

This dissertation aims to show the impact of the tool created that checks certain software metrics, while programming, in order to improve the development time, the code’s quality and also the implemented solution. To do so, it was necessary to validate the hypothesis and research questions described in Section 4.3 (p. 61) and Section 4.4 (p. 62), respectively. For this, we carried out a controlled experiment, from which we couldn’t validate any research question and, consequently, the main hypothesis presented. Through the extensive analysis done we verified that the results presented some threats related to the validation of the tool developed. These threats are listed below.

**Sample’s size.** To validate the hypothesis that a tool such as the one developed has an impact on the form and quality of how software is developed, we did an experiment with two populations, one using the developed tool and the other not, solving a set of equal tasks, with the same difficulty’s degree. Through these two populations would be expected to obtain different data and then verify the differences between them. In this case, the sample of participants was reduced, with a total of sixteen participants, eight of whom used the tool and eight others who didn’t use it. This was due to the short period of time in which it was possible to perform the validation of this dissertation, which meant that the main hypothesis (cf. Section 4.3, p. 61) couldn’t be verified with accuracy, nor could we deny all the null hypotheses that were taken into account during this experiments(cf. Section 4.4, p. 62). In
this way, this is perhaps one of the most important threats and for that it must be taken into account in future work, since it’s relevant to make a more extensive validation, with a larger participants’ sample, to verify the validity of the data and, consequently, of the tool.

**Sample’s characteristics.** Another threat that occurred was the homogeneity of the characterization of the different experiment’s participants since as it can be seen in Section 8.5.1 (p. 102), all the participants were within the same age group and with the same academic qualifications. The developed tool has a target audience all the programmers, more or less experienced, and therefore, should have been validated by people already with different levels of programming experience, with different academic degrees and with different ages or different number of programming’s years, in order to get different results that can be taken into account in the validation of this dissertation. In addition, there should also have been “expert” programmers between the participants’ sample, so that they could definitely validate the impact of the tool on the quality of the code of a software system.

**Social pressure.** Since each participant has to perform a group of different tasks, with a different degree of difficulty, with a predefined duration and with objectives previously explained, they may feel pressured during the experience, making it impossible for them to do what they want correctly. This threat is quite common in quasi-experiments and can be reduced or demystified by running the test with multiple participants at the same time so that an outlier doesn’t feel so much pressure and thus the impact on the results could be lower.

**Practice in TypeScript and JavaScript.** As verified in Section 8.5.1 (p. 102), half of the participants in this experiment had never used TypeScript or had used it very rarely and the other half indicated that they have been in contact, more often, with this language in the past. At the beginning of the experiment, the controller explained at the beginning that this language is somehow similar to JavaScript and that they could use the Internet for any doubts they could have, to which all participants responded that they knew how to program in JavaScript. However, throughout the different tasks, it was observed that few participants actually showed knowledge in TypeScript and even in JavaScript, harming the execution times of each task and even the evolution of the different metrics calculated. In this way, the lack of knowledge about the Typescript’s capabilities, which was the main language of the case study used, and also about JavaScript, was a threat to the validation of this dissertation. This problem can be solved doing a previous tutorial on the basic functionalities of the two programming languages, only deepening in detail more specific features that would be used throughout the different tasks, in order to reduce the impact of this problem on the collected data.

**Case Study dimension and complexity.** As the size of the participants’ sample, the dimension and complexity of the project used as a case study are also a threat to the validation of this dissertation. The tool developed is specific to complex and large software systems, as we think that through it, programmers can improve their developed code as well as the
code they are implementing. Since the project used as a case study was simple, with little complexity, due to the short time available to develop it, the tool didn’t demonstrate its full potential through this system, and it wasn’t possible to reject the null hypotheses enumerated in Section 4.4 (p. 62), in order to validate the hypothesis that supports this dissertation (cf. Section 4.3, p. 61). In a project that is too simple, the execution time of each task, as well as the solution’s convergence and the quality of the code will not be very different whether or not using the tool, not differing the two populations. Therefore, we thought that in the future, the developed tool could be tested in larger and more complex software systems, even in business environments, in order to truly validate the main hypothesis.

Task’s difficulty. Since the case studies created were of simple dimension and of low complexity, all the tasks to be carried out by the participants were also easy to execute which made them arrive at the correct solution quickly, fulfilling all the metrics, with and without the tool you created. If there were more complicated or complex tasks involving several different programming elements, it might be possible to obtain a higher quality dataset and to validate the different research questions listed in Section 4.4 (p. 62).

Test environment. The last threat verified was the environment in which the experiments were performed since all the tasks were executed using the Visual Studio Code, because the tool was developed for this IDE, on a computer with the macOS operating system. All the participants were familiarized with this IDE, but not with all their features, showing some doubts in executing the commands requested in the different tasks. Besides, almost none of the participants knew how to use a MacBook as well as its operating system, requiring external help from the experiment’s controller to perform some actions during the different tasks. With all this, the execution time of each task was impaired, damaging the results obtained. This threat can be reduced or extinct through a prior tutorial about the main features of Visual Studio Code, using different computers, with different hardware and operating systems to accommodate the preferences of most participants.

8.9 Discussion

After an extensive analysis of all the results obtained and calculated through multiple tests, we were able to make a discussion around them also taking into account some notes and observations that the experiment’s controller taken from the different experiences with the different participants.

One of the first details that were verified was the homogeneity of the participants in terms of age and academic qualifications, which led us to the belief that their programming knowledge might not be very distinct. It was also found that all participants said that they could understand easily software systems when they first analyzed them and that they had an almost daily practice of programming. However, it was found that some of the participants actually failed to fully understand the design and its features, even though it is small and simple. Furthermore, the focus, in this case, was on a software system developed in TypeScript, and it was found that the majority
Empirical Evaluation

of participants even claiming to have knowledge in this programming language, justifying that it was similar to JavaScript, actually had little knowledge about TypeScript, which made it difficult to execute the difficult tasks and therefore made their development times worse in some cases.

In addition to these data, it was verified that the participants effectively knew what the concept of refactoring consisted of, being able to apply it to the used system. Many of the participants also demonstrated throughout the different tasks that they were in fact concerned with the state and the aspect of the respective software system, trying to implement a code that goes according to the good rules of programming, with great quality. This extreme concern in programming with quality may also have led to the execution’s times of each task being higher than what it should be, and through this, it wasn’t possible to directly correlate the development time of a task with the final quality of it.

More specifically, it was verified that there was no great difficulty on the part of the participants in understanding the purpose of each task or even the metrics in which they should focus, although there was some difficulty in programming in TypeScript, as already mentioned above. Even so, this difficulty didn’t prevent them from finishing the tasks with better or worse execution times, having in most cases fulfilled all the desired metrics. However, in the last two tasks, there was always a participant who couldn’t achieve the desired value of one of the metrics selected. In the first case, in Task 2 (cf. Section 8.5.3, p. 109), it was a participant without the tool who couldn’t reach the desired level of effort and in the second case, in Task 3 (cf. Section 8.5.4, p. 112), it was a participant who was using the tool, which failed to achieve the desired development volume. Despite this, it could be said that the results were quite positive in general, since the majority of the participants demonstrated motivation and commitment throughout the tasks and later, through the questionnaire (cf. Appendix B, p. 153), demonstrated their satisfaction regarding both the tasks and the tool and also the possibility of using it in the future, to help them while programming.

In spite of all this analysis previously described and discussed, the main focus of this empirical evaluation was to try to validate the hypothesis that supports this dissertation (cf. Section 4.3, p. 61), through the validation of its research questions (cf. Section 4.4, p. 62). For that, different hypothesis tests were created related to respective research questions, each with a null hypothesis and an alternative hypothesis, in which the null hypothesis contradicts what we wanted to prove and the alternative hypothesis is in agreement with the focus of the respective research question. All hypothesis tests were performed using Independent-Samples t-tests [KK15] since we tried to analyze independent variables for two distinct groups, one that used the tool created and others that didn’t use it (cf. Section 8.7, p. 117).

Thus, we performed three hypothesis tests, related to the three research questions presented in Section 4.4 (p. 62).

The first hypothesis test was related to the first research question (cf. Section 8.7.1, p. 118), regarding the final quality of the code that each participant had achieved in each task. So, the null hypothesis stated that continuous exposure to the evolution of software metrics couldn’t improve a software system’s quality. Through the final values of some metrics, namely the number of lines of code and the maintainability index, it was verified that on average, the participants who used
the tool obtained better results, however the value of the standard deviations for each task was too high, so we couldn’t conclude anything with total certainty only taking into account these values. Therefore, we performed the Independent-Samples t-tests [KK15] for each one of these metrics, for each of the different tasks, and it was verified that it wasn’t possible to reject the null hypothesis referring to that test, thus not being able to conclude anything about the first research question, not being able to validate it. The impossibility of rejection of this null hypothesis may have occurred due to several factors. One of them could have been the reduced number of participants in the experiments since in total there were only 16 participants and only 8 of them used the tool. In addition, another factor that may have influenced this validation may have been the lack of complexity and the small size of the project used as a case study, because of we a larger and wider system, the participant could have seen better the evolution of each metric. Another factor, which may also have influenced the results obtained might be related to the easiness of each task, making the participants achieve easily the expected results. Finally, it is also thought that the constant willingness of the participants to implement “good code” might have meant that the final results of the metrics analyzed weren’t very different since everyone was careful to program with quality, whether they used the tool or not.

The second hypothesis test was related to the second research question (cf. Section 8.7.2, p. 121), regarding the fastness of the convergence for a better solution. Thus, the null hypothesis taking into the account stated that the constant exposure to software metrics didn’t allow a fast convergence to a good solution. For this part of the validation, only the value of the maintainability index was used, since we thought that these metrics was one of the most important metrics since it is calculated using other factors and also other software metrics. Since we had only the values of this metric over the different iterations that the participant had made in the tested system, we decided to normalize the respective values collected for a time interval between 0 and 1, in order to make a better comparison between them. After this normalization, the obtained results were compared, as it can be seen in Figure 8.27 (p. 122) and Figure 8.28 (p. 122), it was verified that there was a greater convergence of the maintainability to higher values when the participant used the tool created. Through this analysis, we checked that using our approach developers were able to improve the desired metric more consistently. However, as it can be seen from the respective charts and their areas delimited by their respective quartiles, the convergence values weren’t too distinct, with some intersection areas, which means that nothing could be absolutely concluded through this simple analysis. To perform a better validation, we executed the Independent-Samples t-tests [KK15], using the mean values of maintainability’s convergence calculated for each participant, and through this test it was verified that it was impossible to reject the respective null hypothesis, so it wasn’t possible to conclude anything about the validity of the respective research question. This impossibility of rejecting the null hypothesis, as it happened in the first case, can be due to the fact that the number of participants was reduced, as well as to the fact that the project used as a case study was too simple, with tasks also simplified and with a reduced difficulty’s degree. In addition, the constant concern to implement code with quality and well structured might have also influenced the participants to converge to an optimal solution.
almost identically, with or without the tool.

Finally, the third hypothesis test was related to the third research question (cf. Section 8.7.3, p. 124), regarding the reduction of the development time. Thus, the null hypothesis taking into the account stated that the constant exposure to software metrics didn’t reduce the respective development time. For this part of the validation, the total time that each participant took to complete each task was analyzed with two different sets of results, one with the total times including the cases in which the participants exceeded the given duration for the task and another without these exceptional cases. In both of them, we could verify that in average the development time is lower when the participant used the tool, also verifying that the minimum development time belongs to the same group. It was also verified that there was a big outlier in Task 2, for a participant who didn’t use the tool. However, as it can be seen in Figure 8.29 (p. 125), there was a great variance of the respective times in each task, and then it wasn’t possible to conclude with certainty that the cases in which the tool is used were effectively better than those in which they didn’t use it.

As occurred in the remaining hypothesis tests, we used the **Independent-Samples t-tests** [KK15] for the recorded times. Through this hypothesis test, it was concluded that it was impossible to reject the respective null hypothesis, and therefore it wasn’t possible to conclude anything about the validity of the respective research question. So, we couldn’t say with certainty that using the tool reduces the development time. Probably, as in the other hypothesis tests, the impossibility of rejecting the null hypothesis might be because the sampling was very small and the case study’s project was too simple, with little complexity, as well as all the tasks that the participants performed. In addition, the fact that the participants were always concerned about improving their code might have caused the development time to be higher than it should have been. Finally, in this specific case, the impossibility of rejecting the null hypothesis could also be due to the fact that the participants have demonstrated some difficulty in programming in TypeScript, not understanding all its functionalities, having to use the Internet to research more about this programming language, thereby increasing the total time they took to complete each task. Thus, the development time might not have been influenced by using or not of the tool developed, but by the difficulties of the participants. In addition, we used a MacBook to do the experience also increased the task’s execution time regardless of whether or not the tool was used since the participants mostly didn’t know how to use a computer like that.

Thus, it could be concluded that it wasn’t possible to conclude anything about the research questions presented (cf. Section 4.4, p. 62), not validating them neither validating the main hypothesis of this dissertation (cf. Section 4.3, p. 61), which might be due to several factors, some of them already mentioned as possible validation’s threats in Section 8.8 (p. 126). However, in terms of usability, it was verified that the participants who used the tool were pleasantly satisfied with it, showing a willingness to use it again. The participants who didn’t use it also demonstrated a willingness to do so, concluding in this way that the tool developed although it hadn’t been validated empirically, not validating the hypothesis that supports this dissertation, is important and relevant from the point of view of the different participants.
8.10 Summary

This chapter summarizes the empirical experience carried out after developing the tool created as a proposed solution (cf. Section 5, p. 65) to the main problem stated in Chapter A (p. 151), describing both its objectives (cf. Section 8.1, p. 97), guidelines (cf. Section 8.2, p. 98), planning (cf. Section 8.3, p. 99) and tasks that were executed by the different participants of this experiment (cf. Section 8.4, p. 100).

After all the experiments with the participants, we made an empirical evaluation, in order to analyze the results obtained during the different tasks performed, as well as the opinions given by the respective participants. Besides, we performed some hypothesis tests that allowed us to conclude that we hadn’t data enough to reject any of the null hypotheses regarding the different research questions defined this dissertation (cf. Section 4.4, p. 62).

Then, after the analysis of the values and results obtained and after making small conclusions about them, we described the various validation’s threats related to this experiment (cf. Section 8.8, p. 126) and we discussed, in detail, all the results calculated and analyzed previously, making a connection between them and the validation’s threats that were identified (cf. Section 8.9, p. 128). Also, we also discussed the impossibility of validating the different research questions and described our assumptions about these results against the identified threats.

Moreover, it was verified that the participants showed enthusiasm in using the tool created or in being able to use it in the future, having been verified that they considered it relevant, important and with great usability.
This chapter is a reflection of all the work developed throughout this dissertation. In this way, all the conclusions and lessons learned over this process are presented in Section 9.1, followed by all the scientific contributions made during this research project (cf. Section 9.2, p. 135). Following are the major difficulties experienced in this project and how they were overcome (cf. Section 9.3, p. 137). At the end of this chapter, it is possible to analyze a description of the future work (cf. Section 9.4, p. 139) that can still be done in this project, to increase its scientific contributions, improving the system created, helping even more the different developers while programming.

9.1 Summary

Over time, software systems are increasingly complex, with large dimensions, not allowing a developer to know the state of his program correctly. This is where the tool developed comes in since with it, a developer has direct access to a vast set of software metrics as he programs, receiving immediate feedback about his program and the modifications that he executes, being also aware of the evolution of the same metrics throughout the different Git commits made, previously. Besides, this tool allows its user to accept a set of refactoring and quick actions’ suggestions that aim to improve the program either structurally or visually.

This whole project is part of the research in live software development, that aims to reduce the effort required to understand and to change a software system, which was the starting point for the
Conclusions and Future Work

creation of the described tool. After doing a literature review on the state-of-the-art of the main concepts of this dissertation (cf. Chapter 3, p. 17), checking the diverse scientific contributions that already exists, it was possible to indicate which current problems could be solved (cf. Section 4.1, p. 59). In this way, we formulated the following hypothesis (cf. Section 4.3, p. 61):

"When developers have constant exposure to software metrics, it influences the form and final results of their solutions."

Through this statement we also described several research questions (cf. Section 4.4, p. 62), in order to support the proposed solution (cf. Chapter 5, p. 65). This whole solution (cf. Chapter 5, p. 65) was initially thought and idealized, starting with its contextualization (cf. Section 5.1, p. 65), requirements (cf. Section 5.3, p. 66), and architecture (cf. Section 5.5, p. 69), describing its various features and, then, the possible case studies that could be used to evaluate it (cf. Section 5.6, p. 71).

Next, all the implementation’s phases of the proposed solution were presented through pseudocode and explanations regarding decisions that we had to make, to circumvent the various problems that appeared throughout the main development. This implementation’s decisions were described separately in Chapter 6 (p. 75) and Chapter 7 (p. 89) highlighting our contributions both in the development of the software metrics’ analysis mechanism and the refactorings and quick actions’ suggestion system.

Then, with the empirical evaluation carried out (cf. Chapter 8, p. 97), we collected a set of data, through which we found that it wouldn’t be possible to reject any of the null hypotheses related to each research question:

**RQ1** “When developers have constant exposure to software metrics, do they reach better solutions?” Through the collection of software metrics such as the number of lines of code and the maintainability index obtained at the end of each task performed, we were able to verify that, on average, the participants who used the tool created reached better results than the others. However, there was a high standard deviation in these values, which didn’t allow us to make an absolute conclusion only through these data. To obtain more concrete answers, we carried out some hypothesis tests with these values, verifying that it wasn’t possible to reject the null hypothesis regarding the quality of code obtained through the constant exposure to different software metrics, since we hadn’t enough data to do so. Therefore, we couldn’t conclude anything about this research question, and therefore it wasn’t possible to validate it yet (cf. Section 8.7.1, p. 118).

**RQ2** “When developers have constant exposure to software metrics, do they converge faster to a good solution?” Regarding this research question, the convergence of the maintainability index of each participant in each task was verified in a period of time normalized between 0 and 1, in order to compare the evolution of this metric and if the participants that were using the tool were able to converge more quickly to a better solution than the others. Through the data collected, it was verified that, in fact, in cases where the tool was used, there was a faster convergence for a better solution than the opposite cases. However, analyzing the
Conclusions and Future Work

areas delimited by the quartiles of the respective convergence of this metric with and without
the tool developed, it was verified that there were overlapping areas which could indicate
that in some cases the convergence could be better without the tool and therefore it wouldn’t
be possible to reject the null hypothesis referring to this research question. Thus, we execute
a hypothesis test to check this case, and we verified that in fact, it wasn’t possible to reject
the null hypothesis, since we hadn’t data enough to do so, and for that reason, it wasn’t
possible to validate this research question yet (cf. Section 8.7.2, p. 121).

RQ3 "When developers have constant exposure to software metrics, is the development time re-
duced?" To verify if it was possible to reduce the development time of a software system
when we are constantly exposed to the evolution of different software metrics, we verified
the different times that each participant took to finish each task of the controlled experiment.
Through the average values for each case, it was verified that the average execution time for
each task was lower when using the tool. However, there was a high standard deviation
in that case, which didn’t allow us to make absolute conclusions only through these values.
Thus, we performed a hypothesis test taking into account the times of each participant, using
or not the tool developed, where it was verified that, in fact, we hadn’t data enough to reject
the respective null hypothesis and, therefore, it was not possible to validate this research
question at that time (cf. Section 8.7.3, p. 124).

Thus, through the analysis of these results, we couldn’t validate all the described research
questions, not being possible to validate the main hypothesis of this dissertation (cf. Section 4.3,
p. 61).

In this way, these data allowed us to understand that the tool developed is a tool that can be
inserted in a more large and complex programming context, with a larger sample’s size, to try to
obtain better results, trying to validate the hypothesis that supports this dissertation (cf. Section 4.3,
p. 61). However, it was noticed that the tool was well accepted by its users, who demonstrated
easiness in using it, rating it as excellent in terms of usability.

In this sense, it is verified that the application of the concept of liveness benefits a programmer,
giving him live and complete feedback while he is programming. In this way, the contribution of
this research corresponds to the objectives raised.

To summarize, it is possible to conclude that the live software metrics tool is advantageous and
is a starting point for new research focuses on live software development and even on the concept
of refactoring and software metrics.

9.2 Main Contributions

Section 4.1 (p. 59) listed some problems that current tools had and which our proposed solution
was intended to correct and implement. In this way, it was possible to create a tool for live
analysis of software metrics, which integrated the project’s Git commits and that can even suggest
refactorings and quick actions. Thus, the main contributions of this dissertation are the following:
Conclusions and Future Work

**Systematic Literature Review.** At the beginning of this dissertation a systematic literature review was performed on the state-of-the-art of live refactoring, analyzing all the tools, studies and also solution proposals that currently exist. It was through this systematic review that much of the literature review that supported this research project was obtained (cf. Chapter 3, p. 17), also allowing us to analyze current problems (cf. Section 4.1, p. 59) related to this subject, helping us to formulate the hypothesis (cf. Section 4.3, p. 61) and therefore the research questions (cf. Section 4.4, p. 62) that serve as base for this dissection. This paper will be submitted to a future conference.

**Visual Studio Code’s Extension.** Taking into account all the problems found through the analysis of the state-of-the-art (cf. Chapter 3, p. 17) and consequently, through the formulation of a solution hypothesis (cf. Section 4.3, p. 61) for the problem found, we developed a live tool that analyzes different software metrics. This tool implemented several aspects and concepts:

**Application of the liveness concept.** This concept is applied both during the analysis of different software metrics and also during the analysis of refactoring and quick actions’ opportunities. Whenever the programmer changes his program, he can check the evolution of his metrics and also check new suggestions.

**Analysis of a larger set of software metrics.** Many of the quality metrics’ solutions in Section 3.4 (p. 39) analyzed, present tools that support a small number of software metrics, namely number of lines of code and cyclomatic complexity. However, with our implemented solution, we were able to create a set of nineteen possible metrics to be analyzed.

**Git Integration.** Integration with Git allowed a programmer to receive more complete feedback regarding his project, being able to know more about the metrics of each Git version or even about all project.

**Creation of refactoring and quick actions’ suggestions system.** The focus of this dissertation was to analyze software metrics in a live way so that a programmer could improve his software system. Since many of the improvements can be made through refactorings, it was decided to include the analysis and suggestion of two types of refactorings, which aims to help the developer, as an extra. Also, we developed a quick actions’ hint system that focuses on the good TypeScript or JavaScript’s programming rules and that allows customizing the programmer’s code in a faster way.

**Information visualization.** The refactorings and quick actions suggestion system was developed through the “Linting Provider” that the Visual Studio Code API allows to use and change, but the visualization of the information regarding the evolution of software metrics was made in a different way from the common forms that are verified in other tools, since we used several potentialities provided by the Visual Studio Code’s API, namely “WebViews” or “TreeViews” to display the metrics evolution’s charts.
Conclusions and Future Work

**Empirical Evaluation.** After completing the development of the tool described in the previous point, an empirical evaluation was made on its use and on the results that result from it (*cf.* Chapter 8, p. 97). With this results obtained we were able to verify several aspects regarding both the sampling of participants and the execution of each task, allowing us to perform different hypothesis tests for each of the different research questions listed in Section 4.4 (p. 62). Through these hypothesis tests, none of the null hypotheses could be rejected, but this allowed us to gather conclusions about these same results.

**Developed tool’s publication.** Throughout the development of the proposed solution (*cf.* Chapter 5, p. 65), that included the implementation of an extension to the Visual Studio Code IDE, and even after the tool was created, a small research was done regarding other tools already that already exist in the Visual Studio Code Marketplace, in order to compare the proposed solution by us in Chapter 5 (p. 65) with the existing solutions in this IDE. All these tools analyzed through this research also served as a complement to the state-of-the-art of this dissertation. This paper will be submitted to a future conference.

### 9.3 Main Difficulties

Throughout the development of this dissertation we encountered several difficulties related to the implementation of the proposed solution, that was described in Chapter 5 (p. 65), Chapter 6 (p. 75) and Chapter 7 (p. 89), and also to the validation of the same, using the empirical evaluation described in Chapter 8 (p. 97). All these adversities are listed below:

**Development’s Difficulties.** As mentioned, there were several difficulties in implementing the proposed solution (*cf.* Section 5, p. 65). All of these difficulties have been overcome, but once they have existed they must be recorded and described, explaining the methods used to solve them.

**Liveness Integration.** The major challenge of this dissertation was the implementation of the liveness’ concept in the analysis of software metrics. The goal was to calculate the different metrics as a programmer changed his system, but it was not intended that these values were always calculated when a programmer entered or modified a single character, for example. This difficulty was overcome by using the Visual Studio Code’s API triggers, which tell us when a programmer has finished programming or saved the changes he made to a particular file. In addition to these triggers, a timeout was created to prevent a programmer from saving his changes and seconds later, changing its project again, creating two analyzes of software metrics, unnecessarily, when there should only be one.
Conclusions and Future Work

Git Integration. The Git integration was another difficulty that we encountered during this research project. When we reach the step, where we need to integrate the Git functionalities with the tool created, we had some synchronization problems since it was necessary to first perform the different Git checkouts and then execute the analysis of the respective metrics for each version, while programming and also analysing the metrics of the software system’s that was being modified. This problem was solved successfully, using the functionalities of the VSCode API, explained in Section 6.4 (p. 84), being able to create a fluid and harmonized workflow, allowing the tool to analyze at the same time the metrics of each Git version and also the metrics of the system while it was being programmed.

Information Visualization. The last implementation’s problem found, which might have had more impact on the tool’s usability, was to find a visual form to present the evolution of the software metrics of the system under analysis to its programmer. Several shapes were tested, with different characteristics, from tables to lines in the Visual Studio Code “Output panel” to bar or line charts, and then it was verified that the best way would be to present the evolution of each metric through a bar chart that could be verified in more detail through a line chart. Also through this method, a developer could choose each metric he wanted to analyze so there was no overload of information.

Validation’s Difficulties. In addition to all the difficulties enumerated about the proposed solution’s implementation, there were still some difficulties that we encountered during the validation of the hypothesis (cf. Section 4.3, p. 61) and, consequently, of the research questions (cf. Section 4.4, p. 62) that support this dissertation.

Participant Sample’s Size. In order to do an empirical evaluation strictly, in the best possible way, with as many results as possible, we need a large sample size. In this case, the sample to be taken into account was given by the number of participants in this experiment. Initially, it was expected that the controlled experiment would be performed by 40 people, in which 20 would use the tool and another 20 not, however, we were only able to test it with 16 people, thus reducing the sample size drastically, knowing in advance that the results obtained and the consequent validation could be compromised by that.

Case Study. Just as a large number of participants were required for validating rigorously this dissertation, we also need to use a project of great size and complexity to make the evolution of the different software metrics more drastic and evident in the software system and, consequently, in its quality. However, there was great difficulty in arranging a project of this magnitude, in which we were able to create different tasks from it and that was easy to understand and modify both by us and by the participants in the time stipulated in the experiment. To address this problem, a simpler and less complex project was created (cf. Section 5.6, p. 71), since there was no time to create something
more elaborate. By using this smaller project, we knew that we could be damaging the results obtained and also the respective validation of the different research questions (cf. Section 4.4, p. 62) and hypothesis (cf. Section 4.3, p. 61).

**Hypothesis’ Validation.** The greatest difficulty and problem encountered in the validation of our solution, which has already been described throughout Chapter 8 (p. 97), was that it was not possible to prove this dissertation’s hypothesis (cf. Section 4.3, p. 61), not being able to deny the null hypotheses described in Section 4.4 (p. 62). This difficulty arises mainly from two other major problems that have already been described in Section 8.8 (p. 126) and also in this section, where the first one is related to not having been able to obtain enough data to carry out a more extensive and more rigorous result’s analysis. A larger sample of participants could have a greater impact on the results obtained and thus we could have been able to deny some of the null hypotheses enumerated, in order to prove the main hypothesis that supports this dissertation. The second problem is related to the lack of complexity of the case study developed to validate our solution, since it didn’t have enough impact on the software metrics, and in this way we couldn’t prove that the tool was effective in reducing the development time, improving the code’s quality or the solution’s convergence, since the results obtained by the two tested populations were too similar. If the tool was tested in a larger population, with a larger and more complex software system, it would be expected that the results obtained would be better and that, consequently, it would have been possible to prove the main hypothesis.

### 9.4 Future Work

Like all research projects, there are always improvements that can be made as future work and this dissertation is no exception.

Throughout the controlled experience, the participants recorded their opinions and suggestions on the form that accompanied the experience (cf. Appendix B, p. 153) and we took into consideration all their suggestions as valid and perhaps as possible improvements to be implemented in the future. Therefore, some of the features that may be added later to the tool developed are:

- Have a button that can automatically run all metrics anytime a programmer wants, without having to wait 10 seconds after modifying the software system. In this way, programmers could control whether or not they wanted to receive information about their program;
- Have the option of adding multiple metrics simultaneously, without having to click the icon “+” individually in each metric, which would make the selection process more agile and faster;
- Highlight the location of the icons that complement the metrics since some participants believe that their location was not very intuitive since they thought that they would also
Conclusions and Future Work

have icons next to the different charts visualized and not only next to the name of each metric under analysis or to be analyzed;

- Highlight through a color system the state of each metric. For example, if the cyclomatic complexity were in critical value, its evolution and even its name should be visualized in red as a form of alert to the programmer. This was not possible before, because currently the Visual Studio Code API does not allow to change the color of the text displayed through a "TreeView", that is the method in which all the metrics are visualized, but it’s one of the features that will be included in the next release of VSCode’s API, and then we can change it there. Another solution would be to change the metrics visualization system, without using "TreeViews", in order to be able to encompass this feature without waiting for the new VSCode release.

As future work, we also intend to improve the validation protocol of the proposed solution. In the future we would like to test our tool with a larger sample of participants, with different academic qualifications and different levels of programming knowledge, such as different students of computer engineering, in different academic years, from different universities, thus having a better and wider sampling of participants. Besides, we would like to test our solution in complex large-scale projects, such as the VSCode, React or AngularJS source code, to check if the tool would actually help improve your software metrics. We also think that this tool could be tested in a large-scale company over a long period of time to verify whether or not the main hypothesis is proven, thus validating this dissertation’s hypothesis(cf. Section 4.3, p. 61), and also verifying the performance and impact of the same in a business environment, in a context of team programming.

Finally, the author of this dissertation will continue this research project during a PhD, where we can continue, improve and investigate more about the subject of this dissertation, being able to improve all the previously described aspects, as well as to include new features to the solution proposed by this dissertation (cf. Section 5, p. 65), demystifying the main problem presented here (cf. Section 4, p. 59).
References


REFERENCES


[Dem] Serge Demeyer. FAMIX 2.1—the FAMOOS information exchange model.


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


Appendix A

Participation Consent Declaration

This appendix has the consent declaration that all the controlled experiment’s participants had to read and sign to give us permission to collect their data and the data resulting from the experiments carried out, that were analyzed in (cf. Section 8.5, p. 102) and (cf. Section 8.5, p. 102).
DECLARAÇÃO DE CONSENTIMENTO

(Baseada na declaração de Helsínquia)

No âmbito da realização da tese de Mestre no Mestrado Integrado em Engenharia Informática e Computação da Faculdade de Engenharia da Universidade do Porto, intitulada Supporting Refactoring through Live Static Metrics Visualization, realizada pela estudante Sara Filipa Couto Fernandes, orientada pelo Professor André Restivo e sob a coorientação do Professor Ademar Aguiar, eu abaixo assinado, __________________________________________, declaro que compreendi a explicação que me foi fornecida acerca do estudo que irei participar, nomeadamente o caráter voluntário dessa participação, tendo-me sido dada a oportunidade de fazer as perguntas que julguei necessárias.

Tomei conhecimento de que a informação ou explicação que me foi prestada versou os objetivos e os respetivos métodos a serem empregues, assim como que será assegurada a máxima confidencialidade dos meus dados.

Explicaram-me, ainda, que poderei abandonar o estudo em qualquer momento, sem que daí advinham quaisquer desvantagens.

Por isso, consinto participar no estudo e na recolha de imagens necessárias, respondendo a todas as questões propostas.

Porto, __ de ____________ de _____

___________________________________________________

(Participante ou seu representante)
Appendix B

Questionnaire Applied to the Controlled Experience

This appendix describes the questionnaire used in the controlled experience and through which different results were obtained as verified in (cf. Section 8.5, p. 102). This questionnaire involves questions regarding the characterization of the different participants and also about each the task performed and the created tool’s usability.
Supporting Refactoring through Live Static Metrics Visualization

*Obrigatório

Caracterização do Participante

1. **Género** *

   *Marcar tudo o que for aplicável.*

   - [ ] Feminino
   - [ ] Masculino
   - [ ] Outra: __________________________

2. **Idade** *

   *Marcar tudo o que for aplicável.*

   - [ ] < 18 anos
   - [ ] 18 - 25 anos
   - [ ] 25 - 40 anos
   - [ ] > 40 anos

3. **Habilidades Literárias (grau concluído)** *

   *Marcar apenas uma oval.*

   - [ ] < 12º ano
   - [ ] 12º ano
   - [ ] Licenciatura
   - [ ] Mestrado
   - [ ] Doutoramento
   - [ ] Outra: __________________________

Conhecimentos e Prática Tecnológica

4. **Prática em Engenharia de Software** *

   *Marcar apenas uma oval.*

   1 2 3 4 5

   - [ ] Nunca programo
   - [ ] Programo diariamente

https://docs.google.com/forms/d/1gZRpon_7dxezbkr6RLODo7-jmDBQMB8JG9Q8o00ek/printform
5. Sinto dificuldade na compreensão de sistemas de software (código/funcionamento de um programa) *

Marcar apenas uma oval.

1 2 3 4 5

Discordo ☐ ☐ ☐ ☐ ☐ Concorde totalmente

6. Uso com frequência ferramentas de auxílio à análise de sistemas de software (ex: ferramentas de debugging, de análise de software, etc) *

Marcar apenas uma oval.

1 2 3 4 5

Discordo ☐ ☐ ☐ ☐ ☐ Concorde totalmente

7. Já utilizei alguma ferramenta de análise de métricas de software? *

Marcar apenas uma oval.

☐ Sim
☐ Não

8. Já utilizei alguma ferramenta de refactoring? *

Marcar apenas uma oval.

☐ Sim
☐ Não

9. Refactoring consiste em: *

Marcar tudo o que for aplicável.

☐ Verificar as métricas de um sistema de software
☐ Mudar um sistema de software até que este passe em todos os testes unitários
☐ Reestruturar um sistema de software, sem modificar os comportamentos previamente implementados
☐ Corrigir bugs do sistema de software
☐ Todas as anteriores
☐ Nenhuma das anteriores

10. No decorrer deste caso de estudo utilizei a ferramenta de análise de métricas de software *

Marcar apenas uma oval.

☐ Sim Passe para a pergunta 11.
☐ Não Passe para a pergunta 37.
### Tarefa 0 - Reconhecimento do Sistema

11. **Considero que esta tarefa foi de fácil realização** *
   *Marcar apenas uma oval.*

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Discordo</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Concorde totalmente</td>
</tr>
</tbody>
</table>

12. **O meu conhecimento em Typescript é** *
   *Marcar apenas uma oval.*

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nunca usei</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Uso frequentemente</td>
</tr>
</tbody>
</table>

13. **Atividades** *
   *Marcar apenas uma oval por linha.*

<table>
<thead>
<tr>
<th></th>
<th>1 - Muito difícil</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5 - Muito fácil</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identificar as diferentes features do projeto</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Identificar menu de análise de métricas de software</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Identificar botão de ativação de refactorings</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Identificar diferentes métricas de software</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Identificar funcionalidades do menu de análise de métricas de software</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Identificar gráficos com a evolução de cada métrica</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Tarefa 1

14. **Sentí dificuldade em entender o objetivo da tarefa** *
   *Marcar apenas uma oval.*

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Discordo</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Concorde totalmente</td>
</tr>
</tbody>
</table>
15. **Consegui identificar as métricas necessárias** *  
* Marcar apenas uma oval.  

1 2 3 4 5  
Discordo Concarde totalmente

16. **Considero que esta tarefa foi de fácil realização** *  
* Marcar apenas uma oval.  

1 2 3 4 5  
Discordo Concarde totalmente

17. **Consegui terminar a tarefa a tempo** *  
* Marcar apenas uma oval.  

Sim Não

18. **As métricas em que consegui atingir o objetivo foram** *  
* Marcar tudo o que for aplicável.  

- Número de linhas de código  
- Complexidade ciclomática  
- Número de loops  
- Número de condições  
- Todas as anteriores  
- Não consegui terminar a tarefa a tempo

19. **A ferramenta ajudou-me a cumprir o objetivo** *  
* Marcar apenas uma oval.  

Sim Não

**Tarefa 2**

20. **Sentí dificuldade em entender o objetivo da tarefa** *  
* Marcar apenas uma oval.  

1 2 3 4 5  
Discordo Concarde totalmente
21. **Consegui identificar as métricas necessárias** *

*Marcar apenas uma oval.*

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discordo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concordo totalmente</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

22. **Considero que esta tarefa foi de fácil realização** *

*Marcar apenas uma oval.*

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discordo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concordo totalmente</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

23. **Consegui terminar a tarefa a tempo** *

*Marcar apenas uma oval.*

- [ ] Sim
- [ ] Não

24. **As métricas em que consegui atingir o objetivo foram** *

*Marcar tudo o que for aplicável.*

- [ ] Número de linhas de código
- [ ] Complexidade ciclomática
- [ ] Esforço
- [ ] Número de bugs
- [ ] Maintainability
- [ ] Todas as anteriores
- [ ] Nenhuma das anteriores

25. **A ferramenta ajudou-me a cumprir o objetivo** *

*Marcar apenas uma oval.*

- [ ] Sim
- [ ] Não

---

**Tarefa 3**
26. **Sentirdificuldade em entender o objetivo da tarefa** *

Marcar apenas uma oval.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discordo</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concordo totalmente</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

27. **Consegui identificar as métricas necessárias** *

Marcar apenas uma oval.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discordo</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concordo totalmente</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

28. **Considero que esta tarefa foi de fácil realização** *

Marcar apenas uma oval.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discordo</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concordo totalmente</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

29. **Consegui terminar a tarefa a tempo** *

Marcar apenas uma oval.

- Sim
- Não

30. **As métricas em que consegui atingir o objetivo foram** *

Marca tudo o que for aplicável.

- Número de linhas de código
- Esforço
- Complexidade ciclomática
- Maintainability
- Volume
- Todas as anteriores
- Nenhuma das anteriores

31. **A ferramenta ajudou-me a cumprir o objetivo** *

Marcar apenas uma oval.

- Sim
- Não

**Avaliação da ferramenta testada**
### Questionnaire Applied to the Controlled Experience

Supporting Refactoring through Live Static Metrics Visualization

#### 32. Compreensão do funcionamento da ferramenta

Marcar apenas uma oval por linha.

<table>
<thead>
<tr>
<th></th>
<th>1 - Discordo</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5 - Concordo totalmente</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fácil compreensão</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fácil utilização</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rapidez da execução</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fácil compreensão da informação apresentada</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Facilidade em entender as métricas apresentadas</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Facilidade em entender os &quot;refactorings&quot; apresentados</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Facilidade em entender as &quot;quick actions&quot; apresentadas</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fácil compreensão das cores</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Utilidade dos botões</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interface intuitiva</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 33. Utilizaria a ferramenta novamente *

Marcar apenas uma oval.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discordo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Concorde totalmente</td>
</tr>
</tbody>
</table>

#### 34. Considere que a ferramenta é relevante e vantajosa para a análise de sistema de softwares *

Marcar apenas uma oval.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discordo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Concorde totalmente</td>
</tr>
</tbody>
</table>
35. **Avaliação geral**

*Marcar apenas uma oval por linha.*

<table>
<thead>
<tr>
<th></th>
<th>1 - Discordo</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5 - Concorde totalmente</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gostaria de usar esta ferramenta com frequência</td>
<td>☐ ☐ ☐☐☐</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Achei o sistema demasiado complexo</td>
<td>☐ ☐ ☐☐☐</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Achei o sistema fácil de utilizar</td>
<td>☐ ☐ ☐☐☐</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acho que é necessário a ajuda de uma pessoa com conhecimentos técnicos para usar o sistema</td>
<td>☐ ☐ ☐☐☐</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acho que as várias funções do sistema estão bem integradas</td>
<td>☐ ☐ ☐☐☐</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acho que o sistema apresenta muita inconsistência</td>
<td>☐ ☐ ☐☐☐</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acho que as pessoas aprenderiam rapidamente a usar este sistema</td>
<td>☐ ☐ ☐☐☐</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Achei o sistema muito complicado de usar</td>
<td>☐ ☐ ☐☐☐</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Senti-me confiante a usar o sistema</td>
<td>☐ ☐ ☐☐☐</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precisava de aprender muitos conhecimentos antes de conseguir usar este sistema</td>
<td>☐ ☐ ☐☐☐</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

36. **Opiniões, sugestões e críticas**

________

________

________

________

Pare de preencher este formulário.

**Tarefa 0 - Reconhecimento do Sistema**

37. **Considero que esta tarefa foi de fácil realização**

*Marcar apenas uma oval.*

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discordo</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Concorde totalmente</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>
38. **O meu conhecimento em Typescript é**
*Marcar apenas uma oval.*

1 2 3 4 5

Nunca usei Uso frequentemente

39. **Identifiquei as features do projeto facilmente** *
*Marcar apenas uma oval.*

1 2 3 4 5

Discordo Concordo totalmente

**Tarefa 1**

40. **Sentí dificuldade em entender o objetivo da tarefa** *
*Marcar apenas uma oval.*

1 2 3 4 5

Discordo Concordo totalmente

41. **Consegui identificar as métricas necessárias** *
*Marcar apenas uma oval.*

1 2 3 4 5

Discordo Concordo totalmente

42. **Considero que esta tarefa foi de fácil realização** *
*Marcar apenas uma oval.*

1 2 3 4 5

Discordo Concordo totalmente

43. **Consegui terminar a tarefa a tempo** *
*Marcar apenas uma oval.*

Sim Não
44. As métricas em que consegui atingir o objetivo foram *
   Marcar tudo o que for aplicável.
   - Número de linhas de código
   - Complexidade ciclomática
   - Número de loops
   - Número de condições
   - Todas as anteriores
   - Nenhuma das anteriores

45. Se tivesse utilizado a ferramenta, teria cumprido a tarefa mais facilmente *
   Marcar apenas uma oval.
   - Sim
   - Não
   - Talvez

46. Se tivesse utilizado a ferramenta, teria cumprido a tarefa mais rapidamente *
   Marcar apenas uma oval.
   - Sim
   - Não
   - Talvez

Tarefa 2

47. Sentir dificuldade em entender o objetivo da tarefa *
   Marcar apenas uma oval.

   1 2 3 4 5
   Discordo ○ ○ ○ ○ ○ Concordo totalmente

48. Conseguir identificar as métricas necessárias *
   Marcar apenas uma oval.

   1 2 3 4 5
   Discordo ○ ○ ○ ○ ○ Concordo totalmente
49. **Considero que esta tarefa foi de fácil realização** *

*Marcar apenas uma oval.*

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Discordo ☐ ☐ ☐ ☐ ☐ Conordo totalmente ☐

50. **Consegui terminar a tarefa a tempo** *

*Marcar apenas uma oval.*

☐ Sim

☐ Não

51. **As métricas em que consegui atingir o objetivo foram** *

*Marcar tudo o que for aplicável.*

☐ Número de linhas de código

☐ Complexidade ciclomática

☐ Esforço

☐ Número de bugs

☐ Maintainability

☐ Todas as anteriores

☐ Nenhuma das anteriores

52. **Se tivesse utilizado a ferramenta, teria cumprido a tarefa mais rapidamente** *

*Marcar apenas uma oval.*

☐ Sim

☐ Não

☐ Talvez

53. **Se tivesse utilizado a ferramenta, teria cumprido a tarefa mais facilmente** *

*Marcar apenas uma oval.*

☐ Sim

☐ Não

☐ Talvez

---

**Tarefa 3**
54. **Sentí dificuldade em entender o objetivo da tarefa**

*Marcar apenas uma oval.*

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Discordo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Conordo totalmente</td>
</tr>
</tbody>
</table>

55. **Consegui identificar as métricas necessárias**

*Marcar apenas uma oval.*

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Discordo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Conordo totalmente</td>
</tr>
</tbody>
</table>

56. **Considere que esta tarefa foi de fácil realização**

*Marcar apenas uma oval.*

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Discordo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Conordo totalmente</td>
</tr>
</tbody>
</table>

57. **As métricas em que consegui atingir o objetivo foram**

*Marcar tudo o que for aplicável.*

- [ ] Número de linhas de código
- [ ] Esforço
- [ ] Complexidade ciclomática
- [ ] Maintainability
- [ ] Volume
- [ ] Todas as anteriores
- [ ] Nenhuma das anteriores

58. **Consegui terminar a tarefa a tempo**

*Marcar apenas uma oval.*

- [ ] Sim
- [ ] Não

59. **Se tivesse utilizado a ferramenta, teria cumprido a tarefa mais facilmente**

*Marcar apenas uma oval.*

- [ ] Sim
- [ ] Não
- [ ] Talvez
60. **Se tivesse utilizado a ferramenta, teria cumprido a tarefa mais rapidamente** *

*Marcar apenas uma oval.*

- [ ] Sim
- [x] Não
- [ ] Talvez
Questionnaire Applied to the Controlled Experience