



# Integration of an Automatic Fault Localization Tool in an IDE and its Evaluation

**JOÃO MANUEL MADUREIRA DE LEÃO**

Julho de 2022

# **Integration of an Automatic Fault Localization Tool in an IDE and its Evaluation**

**João Leão**

**A dissertation submitted in partial fulfillment of  
the requirements for the degree of Master of Science,  
Specialisation Area of Software Engineering**

**Supervisor: Professor Alberto Sampaio  
Co-Supervisor: Professor Isabel Sampaio**

Porto, July 1, 2022



# Dedictory

To my parents, my girlfriend Ester, my good friends Nuno and José whom without this journey wouldn't be possible. And finally, to my supervisors, whose feedback and help were crucial to the thesis development.



# Abstract

Debugging is one of the most demanding and error-prone tasks in software development. Trying to address bugs has become overall more expensive as the software complexity and size have increased. As a result, several researchers attempted to improve the developers' debugging experience and efficiency by automating as much of the process as possible. Existing auto-finding tools will assist developers in automatically detecting bugs, however, they are not yet widely available to software engineers. Making such tools available to developers can save debugging time and increase the productivity.

Subsequently, the main goal of this dissertation is to incorporate an automatic fault localization tool into an Integrated Development Environment (IDE). The selected IDE was Visual Studio Code, a source-code editor developed by Microsoft for Windows, Linux, and macOS. Visual Studio Code is one of the most used IDEs and is known for its flexible API, which allows nearly every aspect of it to be customized. Furthermore, the chosen automatic fault localization tool was FLACOCO, a recent fault localization tool for Java that supports up to the most recent versions.

Nonetheless, this document contains a full overview of several fault localization methodologies and tools, as well as an explanation of the complete planning and development process of the produced Visual Studio Code extension. After the development and deployment were completed, an evaluation was carried out. The extension was evaluated through a user study in which thirty Java professionals took part. The test had two parts: the first involved users using the extension to complete two debugging tasks in previously unknown projects, and the second had them filling out a satisfaction questionnaire for further analysis.

Finally, the results show that the extension was a success, with the system being rated positively in all areas. However, it may be revised in light of the questionnaire responses, with the suggestions received being considered for future work.

**Keywords:** Fault Localization, Automated Debugging, Debugging, Fault Localization Tools, Fault Localization Techniques



# Resumo

A depuração é uma das tarefas mais exigentes e propensas a erros no desenvolvimento de software. Tentar resolver esses erros tornou-se mais dispendioso com os incrementos de complexidade e tamanho do software. Deste modo, ao longo dos últimos anos, vários investigadores tentaram melhorar a experiência de depuração e a eficiência dos desenvolvedores automatizando o máximo possível do processo. Existem ferramentas de localização de defeitos que assistem os desenvolvedores na detecção automática de bugs, no entanto estas ainda não se encontram amplamente disponíveis para os programadores. Tornar essas ferramentas disponíveis para todos certamente iria resultar na redução do tempo de depuração e no aumento da produtividade.

Assim sendo, o principal objetivo desta dissertação é incorporar uma ferramenta de localização automática de defeitos num IDE. Em termos de IDE, o Visual Studio Code, um editor de código-fonte desenvolvido pela Microsoft para Windows, Linux e macOS, foi selecionado. Este IDE tem ganho bastante popularidade, sendo um dos IDEs mais utilizados mundialmente. Além disso, o Visual Studio Code é reconhecido pela sua API flexível, que permite que quase todos os seus aspectos sejam personalizados. Adicionalmente, o FLACOCO, uma ferramenta de localização de defeitos baseada em SFL que suporta até as versões mais recentes do Java, foi escolhida como ferramenta de localização automática de defeitos.

Além do mais, esta dissertação contém um estudo sobre as técnicas de localização automática de defeitos e as suas ferramentas, bem como uma explicação do planeamento e implementação da extensão criada para o Visual Studio Code. Após o término da implementação e a posterior implantação, foi efetuada a sua avaliação. Procedeu-se a um teste de utilização com a participação de treze utilizadores proficientes na linguagem Java. O teste foi composto por duas componentes: na primeira os utilizadores utilizaram a extensão para completar duas tarefas de depuração em projetos por eles desconhecidos e na segunda foi-lhes fornecido um questionário de satisfação para posterior análise.

Os resultados obtidos sugerem que a extensão foi um sucesso, sendo que o sistema foi positivamente avaliado em todos os aspetos. No entanto a mesma poderá ser aprimorada tendo em consideração o feedback obtido na secção de resposta livre do questionário, sendo que o mesmo foi bastante valioso e as sugestões apuradas vieram a ser consideradas para trabalho futuro.





# Acknowledgement

Before anything else, I must thank my family members, including those who have sadly passed away but whose values have been passed down to me. Their unwavering love and support enabled me to get to where I am today, personally, academically, and professionally.

Furthermore, I would like to thank my girlfriend and friends for always being there for me and assisting me in maintaining a balance between my dedication to this thesis and my personal life.

I also want to thank Alberto Sampaio, my thesis advisor, and Isabel Sampaio, my co-advisor. They have always been able to provide valuable feedback in a timely manner and have accompanied me throughout the entire process, ensuring that it is being guided in the right direction.

Finally, I'd want to thank everyone who took the questionnaire. They were all very nice and enthusiastically engaged, providing excellent feedback not just by answering the questions but also by going a step further and provide additional feedback.



# Contents

<b>List of Figures</b>	<b>xv</b>
<b>List of Tables</b>	<b>xvii</b>
<b>List of Source Code</b>	<b>xix</b>
<b>List of Acronyms</b>	<b>xxi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Context . . . . .	1
1.2 Problem . . . . .	1
1.3 Objectives . . . . .	2
1.4 Approach . . . . .	2
1.5 Document Structure . . . . .	3
<b>2 State of Art</b>	<b>5</b>
2.1 Debugging . . . . .	5
2.1.1 Definition and Process . . . . .	5
2.1.2 Debugging Tools . . . . .	6
2.2 Error, Fault and Failure . . . . .	7
2.2.1 Definition . . . . .	7
2.3 Fault Localization . . . . .	7
2.3.1 Traditional Methods . . . . .	7
Program Logging . . . . .	7
Assertions . . . . .	8
Breakpoints . . . . .	8
2.3.2 Advanced Methods . . . . .	8
Slice-Based Techniques . . . . .	8
Program Spectrum-Based Techniques . . . . .	10
Data Mining-Based Techniques . . . . .	11
Machine Learning-Based Techniques . . . . .	11
Mutation-Based Techniques . . . . .	12
Model-Based Techniques . . . . .	12
Program State-Based Techniques . . . . .	13
Statistics-Based Techniques . . . . .	14
2.4 Fault Localization Techniques Comparison . . . . .	15
2.5 Fault Localization Tools and Integrations . . . . .	16
2.5.1 Aletheia . . . . .	16
2.5.2 CharmFL . . . . .	17
2.5.3 FLACOCO . . . . .	18
2.5.4 FLAVS . . . . .	19

2.5.5	GZoltar . . . . .	20
2.5.6	MUSEUM . . . . .	21
2.5.7	Tarantula . . . . .	21
2.5.8	UnitFL . . . . .	22
2.6	Summary . . . . .	22
<b>3</b>	<b>Value Analysis</b>	<b>25</b>
3.1	Innovation Process . . . . .	25
3.2	New Concept Development . . . . .	26
3.2.1	Opportunity Identification . . . . .	26
3.2.2	Opportunity Analysis . . . . .	27
3.2.3	Idea Generation . . . . .	28
3.2.4	Idea Selection . . . . .	29
3.2.5	Concept Definition . . . . .	35
3.3	Value Analysis . . . . .	35
3.3.1	Customer Value . . . . .	35
3.3.2	Perceived Value . . . . .	35
3.3.3	Benefit vs Sacrifice . . . . .	36
3.4	Value Proposition . . . . .	36
3.5	Summary . . . . .	37
<b>4</b>	<b>Analysis and Design</b>	<b>39</b>
4.1	Selection Process . . . . .	39
4.2	Requirements . . . . .	39
4.2.1	Functional Requirements . . . . .	40
	UC1: Open entry project folder . . . . .	40
	UC2: Execute FLACOCO tool . . . . .	41
	UC3: Open suspicious classes . . . . .	41
	UC4: Highlight suspicious lines of code . . . . .	42
4.2.2	Non-Functional Requirements . . . . .	43
4.3	Design . . . . .	43
4.3.1	Architecture . . . . .	43
4.3.2	Deployment . . . . .	44
4.4	Summary . . . . .	45
<b>5</b>	<b>Development</b>	<b>47</b>
5.1	Visual Studio Code Extensions . . . . .	47
5.1.1	Creating an Extension . . . . .	47
5.1.2	Extension Structure . . . . .	47
	Extension Manifest . . . . .	48
	Entry File . . . . .	48
	UX Guidelines . . . . .	48
5.2	Use Case Development . . . . .	49
5.2.1	UC1 - Open entry project folder . . . . .	49
5.2.2	UC2 - Execute FLACOCO tool . . . . .	50
5.2.3	UC3 - Open suspicious classes . . . . .	51
5.2.4	UC4 - Highlight suspicious lines of code . . . . .	53
5.3	Other FLACOCO Extension Details . . . . .	54
5.3.1	Logo . . . . .	54

5.3.2	README . . . . .	55
5.3.3	Deploy . . . . .	55
5.4	Summary . . . . .	56
<b>6</b>	<b>Evaluation and Experimentation</b>	<b>57</b>
6.1	Goal and Research Question . . . . .	57
6.2	Hypothesis . . . . .	57
6.3	Study Planning . . . . .	57
6.3.1	Test Scenario . . . . .	57
6.3.2	Evaluation Indicators . . . . .	58
6.3.3	Participants . . . . .	58
6.3.4	Questionnaire . . . . .	59
6.3.5	Hypothesis Evaluation . . . . .	60
6.4	Preparation . . . . .	60
6.5	Results and Discussion . . . . .	61
6.6	Limitations . . . . .	63
6.7	Summary . . . . .	63
<b>7</b>	<b>Conclusion</b>	<b>65</b>
7.1	Achieved Goals . . . . .	65
7.2	Limitations . . . . .	66
7.3	Future Work . . . . .	66
7.4	Final Remarks . . . . .	66
	<b>Bibliography</b>	<b>69</b>
	<b>Appendix A: User Study</b>	<b>75</b>
A.1	Questions . . . . .	75
A.1.1	Introduction . . . . .	75
A.1.2	Demographic . . . . .	76
A.1.3	User Experience . . . . .	78
A.2	Results . . . . .	81
A.2.1	User Times . . . . .	81
A.2.2	Demographic . . . . .	82
A.2.3	User Experience . . . . .	84



# List of Figures

1.1	Design Science Research Methodology Steps . . . . .	2
2.1	Debugging Steps . . . . .	6
2.2	Faulty code example . . . . .	9
2.3	An example showing the differences among Static, Dynamic, and Execution Slicing . . . . .	9
2.4	Spectrum-based fault localization input example . . . . .	10
2.5	Aletheia main components . . . . .	16
2.6	CharmFL ranking list output . . . . .	17
2.7	FLACOCO Architecture . . . . .	18
2.8	FLAVS flowchart . . . . .	19
2.9	GZoltar information flow . . . . .	20
2.10	UnitFL menu . . . . .	22
3.1	The three steps of the Innovation Process . . . . .	25
3.2	The New Concept Development (NCD) Model . . . . .	26
3.3	Hierarchical Decision Tree . . . . .	30
3.4	Multiplication of priority vector by the non-normalized priority matrix . . . .	32
3.5	Multiplication of priority matrix by the criteria priority vector . . . . .	34
4.1	Use Case Diagram . . . . .	40
4.2	FLACOCO Component Diagram . . . . .	43
4.3	FLACOCO Deployment . . . . .	44
5.1	Visual Studio Code UX Guidelines . . . . .	49
5.2	Flacoco Extension View . . . . .	50
5.3	Flacoco Extension Run . . . . .	51
5.4	Flacoco Status Bar and Information Message . . . . .	51
5.5	Flacoco Faulty Classes Tree . . . . .	52
5.6	Flacoco Highlight Example . . . . .	53
5.7	Flacoco Hover Message Example . . . . .	54
5.8	Flacoco Visual Studio Extension Icon . . . . .	54
5.9	VSCoDe Personal Access Token . . . . .	55
6.1	Relation between the number of subjects and number of usability problems detected . . . . .	58
A.1	Questionnaire Questions Part 1 . . . . .	75
A.2	Questionnaire Questions Part 2 . . . . .	76
A.3	Questionnaire Questions Part 3 . . . . .	77
A.4	Questionnaire Questions Part 4 . . . . .	78
A.5	Questionnaire Questions Part 5 . . . . .	79



A.6	Questionnaire Questions Part 6 . . . . .	80
A.7	Questionnaire Questions Part 7 . . . . .	81
A.8	Demographic Question 1 Results . . . . .	82
A.9	Demographic Question 2 Results . . . . .	82
A.10	Demographic Question 3 Results . . . . .	83
A.11	Demographic Question 4 Results . . . . .	83
A.12	User Experience Question 1 Results . . . . .	84
A.13	User Experience Question 2 Results . . . . .	84
A.14	User Experience Question 3 Results . . . . .	85
A.15	User Experience Question 4 Results . . . . .	85
A.16	User Experience Question 5 Results . . . . .	86
A.17	User Experience Question 6 Results . . . . .	86
A.18	User Experience Question 7 Results . . . . .	87
A.19	User Experience Question 8 Results . . . . .	87

# List of Tables

2.1	Evaluation of Fault Localization Techniques . . . . .	15
2.2	Comparative Analysis of Fault Localization Tools . . . . .	23
3.1	Fundamental scale . . . . .	30
3.2	AHP table for criteria comparison . . . . .	31
3.3	AHP table for criteria comparison . . . . .	31
3.4	Normalized criteria matrix . . . . .	31
3.5	Relative Priority of each criterion . . . . .	32
3.6	Random Consistency Index . . . . .	33
3.7	Comparison Matrix of Accuracy between Alternatives . . . . .	33
3.8	Comparison Matrix of Reliability between Alternatives . . . . .	33
3.9	Comparison Matrix of Performance between Alternatives . . . . .	33
3.10	Normalized Matrix for Alternatives-Accuracy Comparison and Local Priority	34
3.11	Normalized Matrix for Alternatives-Reliability Comparison and Local Priority	34
3.12	Normalized Matrix for Alternatives-Performance Comparison and Local Priority	34
3.13	Criteria/Alternatives Classification Matrix and Composite Priority . . . . .	34
3.14	Business Model Canvas . . . . .	37
4.1	Non-Functional Requirements . . . . .	43
6.1	Likert Scale . . . . .	59
6.2	Questionnaire's User Experience Section . . . . .	60
7.1	Thesis' Objectives . . . . .	65
A.1	User Times . . . . .	81



# List of Source Code

5.1	Implementation of the method addToFaulty . . . . .	52
-----	--	----



# List of Acronyms

$N_C$	Total number of test cases that cover a code statement.
$N_F$	Total number of failed test cases.
$N_S$	Total number of successful test cases.
$N_U$	Total number of test cases that do not cover a code statement.
$N_{CF}$	Number of failed test cases that cover a code statement.
$N_{CS}$	Number of successful test cases that cover a code statement.
$N_{UF}$	Number of failed test cases that do not cover a code statement.
$N_{US}$	Number of successful test cases that do not cover a code statement.
AHP	Analytic Hierarchy Process.
BP	Back-Propagation.
DSRM	Design Science Research Methodology.
FEI	Front End of Innovation.
FFE	Fuzzy Front End.
IDE	Integrated Development Environment.
MBFL	Mutation Based Fault Localization.
NCD	New Concept Development.
NPD	New Product Development.
RBF	Radial Basis Function.
SFL	Spectrum-based Fault Localization.



# Chapter 1

## Introduction

This chapter introduces the context and the problem under study. It also describes the approach taken as well as the objectives for this dissertation work. Finally, it presents the document structure, where the content of each chapter is summarized.

### 1.1 Context

As a result of the exponential growth of the software projects' size and complexity, finding faults has become a more burdensome and time-consuming activity. Such a process can be error prone when done manually, hence, many researchers have tried to decrease the human effort by reducing the amount of code that must be analysed before the software fault(s) can be precisely located (F. P. d. Silva, H. A. d. Souza, and M. L. Chaim 2018).

That is where Automatic Fault Localization comes in, a software engineering technique to assist programmers during a debug session by suggesting locations that are more likely to contain a fault. Its goal is to point the programmer towards the right area of the program that will enable them to find the relevant fault quicker. Without having to manually review faults, it reduces the overall effort of software development making it more accurate and efficient.

### 1.2 Problem

Whenever a fault is detected, it is necessary to trigger the debugging process. It is known that debugging is a very time-consuming activity and when not carried out properly, consequences can be dreadful (Adragna 2008). Therefore, one of the goals of research in the field of software engineering has been to automate the debugging process. In addition, there is the process of automatic repairing, which includes a fault location task, which is a very difficult task in this process.

The results that have been obtained with the existing localization techniques and tools still leave a great room for improvement in the fault finding task and its correction. Improvements can be achieved not only in terms of existing techniques and tools, but also their availability for everyday software development. Integrated development environments include several tools that assist in locating the faults, but the localization task remained essentially manual (Parnin and Orso 2011).

Existing auto-finding tools will allow the developer to automatically find faults, but they are not yet easily acquirable by software engineers. Making such tools a developer's tool can result in reduced debugging time and increased productivity.



### 1.3 Objectives

The main objective of this dissertation is the inclusion of an automatic fault location tool in an integrated development environment. This can be subdivided into:

1. Identify and describe existing automatic fault localization techniques and tools.
2. Identify the possible tools, and/or techniques, that show good results and can be included in an Integrated Development Environment (IDE).
3. Integrate an existing tool into a chosen IDE. Integration may require adapting, or improving, this tool.

### 1.4 Approach

This dissertation will be developed applying the Design Science Research Methodology (DSRM). This is a rigorous process of designing artifacts to solve problems, assess what is designed or what is working, and communicate results obtained (Knuth 2016).

DSRM consists of 6 steps, which are presented in the Figure 1.1, namely: Identify problem and motivate, define objectives of a solution, design and development, demonstration, evaluation and communication.

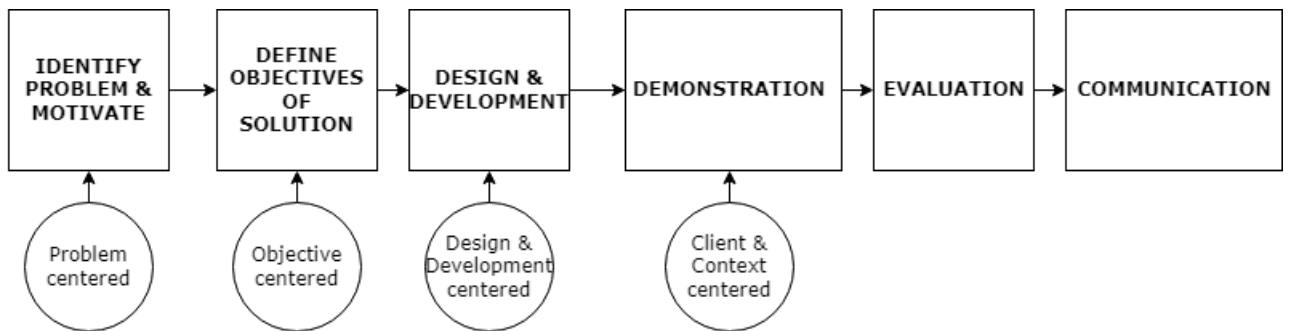


Figure 1.1: Design Science Research Methodology Steps (based on Lawrence, Tuunanen, and M. Myers 2010 ).

1. Identify problem and motivate: Having the problem defined, it begins a search for the best solution, thus it is necessary to understand what already exists in the literature of automatic fault localization tools and techniques, but it is also necessary to analyze what are the flaws that are still present so we can choose the best ones to integrate in an IDE.
2. Define objectives of a solution: Infer the goals of a solution from the problem definition and the knowledge of what is possible and feasible. The goals established for the development of this work were:
  - (a) Identify and describe existing automatic fault localization techniques and tools.
  - (b) Identify the possible tools, and/or techniques, that show good results and can be included in an IDE.
  - (c) Integrate an existing tool into a chosen IDE. Integration may require adapting, or improving, this tool.

3. Design and development: Taking into account the automatic fault tool(s) selected before, it is necessary to design an integration solution for the selected tool(s) and to select an IDE. After, the integration of an automatic fault localization tool will be carried out in the chosen IDE.
4. Demonstration: Application of the solution to the problem presented, through examples of applications with academic purposes.
5. Evaluation: The solution will be evaluated based on how useful it is to the developers and how satisfied they are with it.
6. Communication: The communication is done through the completion of a document, where it is presented all the details of the project (problem definition, state of art, value analysis, design and implementation, evaluation, discussion of results and conclusion) and the project presentation of this dissertation,

## 1.5 Document Structure

This document has the following structure:

1. Chapter 1 – **Introduction**: the context of this dissertation work is introduced as well as the problem at hand. Furthermore, the approach methodology is presented and the objectives of this dissertation are shown.
2. Chapter 2 – **State of Art**: the concepts used in this dissertation work are illustrated, automated debugging techniques as well as tools are analyzed and described to check the possibilities of including in an IDE.
3. Chapter 3 – **Value Analysis**: it is evaluated how this dissertation work can bring value to the customer (developer). This comprehends items such as value for the customer (developer) and perceived value. The New Product and Process Development is presented here in addition to the some value analysis definitions, the Analytic Hierarchy Process and the Business Model Canvas.
4. Chapter 4– **Analysis and Design**: The IDE and tools of choice, as well as the requirements and goals for making the most of the integration, are discussed.
5. Chapter 5 – **Development**: It is shown how the chosen tool was integrated into the IDE, going in detail how each use case was implemented.
6. Chapter 6 – **Evaluation and Experimentation**: the evaluation of the integration, as well as the outcomes, are demonstrated and discussed.
7. Chapter 7 – **Conclusion**: delineates the dissertation's conclusions, providing a concise review of the findings and proposing some future work.



## Chapter 2

# State of Art

This chapter presents some concepts as well as the techniques and tools under the context of automated debugging. It references the step 1 of the DSRM approach.

### 2.1 Debugging

When software developers seek to figure out why a program behaves the way it does, they must translate their questions about the behavior into a sequence of questions about the code, speculating on the reasons along the way. That is where debugging comes in action (Ko and B. Myers 2008).

Even though the struggle of trying to understand a program's behaviour has been known for decades, debugging is still one of the most challenging and time consuming aspects of software development. It often consumes more time than creating the piece of software itself (Beller et al. 2018). This section presents the definition of debugging as well as the concept of debugger.

#### 2.1.1 Definition and Process

Debugging is the process of detecting faults in a computer software or a system, and resolving them so that the program works correctly (Srivastva and Dhir 2017). These software defects are commonly known as "bugs", hence the term "debugging".

This considered tedious and complex task called debugging can be divided into 5 steps (Sayantini 2020) : identify the fault, find the fault location, analyze the fault, prove the analysis and cover lateral damage (see Figure 2.1). Firstly, the developer should identify the actual fault since a bad identification can lead into loads of wasted time, specially when the fault can be deceiving. Secondly, the developer needs to find the exact spot of the fault. After the correct exact spot of the fault was found, the third step is to analyse it in order to fix it. While we analyse, we must take care to not cause any collateral faults. Now for the fourth step, the developer should guarantee the problem doesn't happen again and write tests to cover this area of problem. Lastly, the developer should run all unit tests to make sure no collateral faults were triggered by this change.

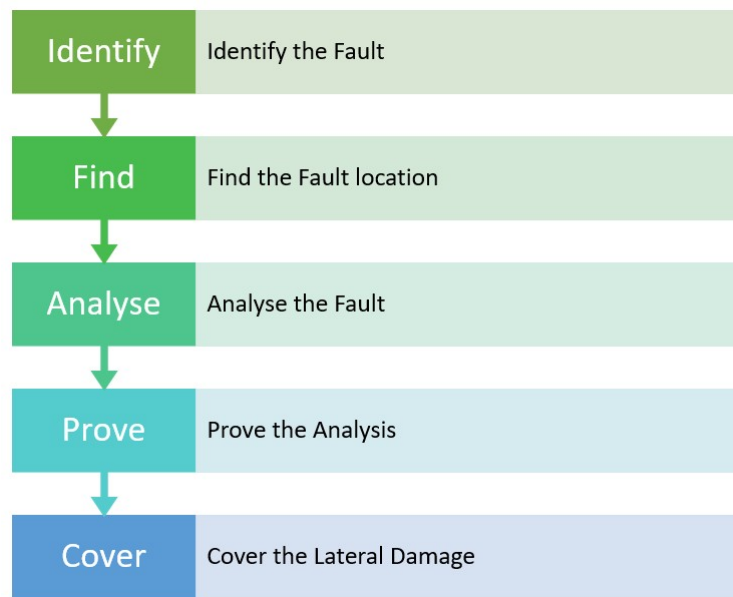


Figure 2.1: Debugging Steps (from Sayantini 2020)

### 2.1.2 Debugging Tools

During debugging, software engineers need to associate failures with its root defect. To complete this step efficiently, they often need to acquire a deep understanding and build a mental model of the software system at hand. This is where modern debugging tools come in: they aid software engineers in observing the system's dynamic behaviour (Beller et al. 2018).

Debugging tools or debugger is a program used to test and debug other programs. It helps to identify the faults of the code at the various stages of the software development process. Usually, debuggers offer a query processor, a symbol resolver, an expression interpreter, and a debug support interface. They also offer more sophisticated functions such as running a program step by step (single-stepping or program animation), stopping (breaking) (pausing the program to examine the current state) at some event or specified instruction by using what is called a breakpoint, and tracking the values of variables. Some debuggers also have the ability to modify program state while it is running, or even to continue execution at a different location in the program to bypass a crash or logical fault.

Nonetheless, there are different types of debuggers for different operating systems. For Unix and Linux GDB is used as the standard debugger, for Windows OS the Visual Studio is a powerful editor and debugger and lastly, for Mac OS LLDB is a high-level debugger. These are just the most common for each operating system, but there is a wide variety of choices in terms of debugging tools.

## 2.2 Error, Fault and Failure

### 2.2.1 Definition

The terms error, fault (defect), and failure are defined, respectively, as “erroneous state of the system”, “defect in a system or a representation of a system that if executed/activated could potentially result in an error”, and “an externally visible deviation from the systems specification” (ISO 2017).

The Standard IEEE (ISO 2017) also mentions that other synonym for a fault can be “bug” and states that “a failure may be caused by (and thus indicate the presence of) a fault” and “a fault may cause one or more failures”. Therefore, in this dissertation, it is used this terminology to any computer program. In this environment, faults are bugs in the code, and failures occur when the output diverges from what it was supposed to be.

Nonetheless, error detection is mandatory for fault localization. One should know something is wrong before locating the reasoning behind the fault. Failures are a basic form of error detection, but many errors remain hidden and never lead to a failure. Failure detection and array bounds checking are both examples of generic error detection mechanisms, that can be applied without detailed knowledge of a program (Abreu, Zoetewij, and A. J. v. Gemund 2007a).

## 2.3 Fault Localization

When a software program fails it can be caused by many faults that may reside in the program. To fix failures, faults should be located and corrected. That is where Fault Localization is needed. Fault localization is, essentially, a search over the space of program components (e.g. statements, variables, values, predicates) to find suspicious entities that might have participated in a program failure. It often involves inspection of numerous components and their interactions with the rest of system. Since there are many suspects in a program for a failure, automated fault localization techniques attempt to assist developers by refining and reducing the search space for faults.

According to Wong, Gao, et al. 2016, fault localization techniques can be divided into two categories: the more intuitive, traditional methods and the more complex, advanced ones. In the subsections that follows, these two categories will be described as well as its techniques.

### 2.3.1 Traditional Methods

Traditional fault localization techniques are very popular amongst programmers since they are the most intuitive and simple to perform. These include program logging, assertions and breakpoints.

#### Program Logging

Print statements are often used to create program logging in a chaotic way in order to monitor variable values and other information related to the current program state. If any faulty behaviour is detected, developers need to check the program logs or the printed run-time information to find the cause of such behaviours.

## Assertions

"The primary goal in writing assertions is to specify what a system is supposed to do rather than how it is to do it." (Rosenblum 1992). This means that developers can specify assertions in their code as a condition to terminate the execution if the program is acting erroneously, therefore it can also be used to detect any faulty behaviour at runtime.

## Breakpoints

A breakpoint is a particular point in the program where the code will stop executing, allowing the programmer to examine its current state. During this interruption, the programmer can check whether or not the program is functioning as it should, either by modifying the value of variables or just observing the progression of a bug.

### 2.3.2 Advanced Methods

#### Slice-Based Techniques

Program slicing has been proposed and applied to localize faults in the program. Essentially, slicing is a technique whose purpose is to extract all statements that affect some set of variables in the program. The set of all extracted statements is called a slice (Kusumoto et al. 2002). This abstracts the program into a reduced form by deleting irrelevant parts such that the resulting slice will still behave the same as the original program with respect to certain specifications.

Slicing techniques have been usually divided into **static slicing** and **dynamic slicing**. A program slice for a variable  $x$  at a statement  $n$  in program  $P$  ( $(x,n)$  is called slicing criterion). This means a set of program statements and/or predicate expressions in  $P$ , which possibly affect the value of  $x$  at  $n$ . The analysis result for all possible input data is called static slice (Weiser 1984), and that for a particular input with respect to a subset of selected variables, rather than for all possible computations is called dynamic slice (Korel and Laski 1990).

Static Slicing basically applies all possible runs without making any assumptions about a failing program run, therefore it has a crucial disadvantage. Since it contains all executable statements, it might generate a slice with statements that should not be included. That is where dynamic slicing is superior. Dynamic Slicing demands to be run with a specific value slice in order to identify the statements that do affect this particular value observed at a particular location. This surpasses Static Slicing, which has to analyse all the possible computations. Still it has its limitations, one known limitation is that it cannot capture when certain code should have been executed while it did not due to the error, this may cause the execution of certain critical statements in a program to be omitted and thus result in failures.

Furthermore, there is an alternative to static and dynamic slicing called execution slicing. Execution Slicing uses the tests to locate program bugs, basically an execution slice with respect to a given test case contains the set of statements executed by this test. The example below can further show the differences between static, dynamic and execution slice.

Code with a bug at $s_7$	
$s_1$	<code>input(a)</code>
$s_2$	<code>i = 1;</code>
$s_3$	<code>sum = 0;</code>
$s_4$	<code>product = 1;</code>
$s_5$	<code>if (i &lt; a){</code>
$s_6$	<code>    sum = sum + i;</code>
$s_7$	<code>    product = product × i;</code>
	<code>    //bug product = product × 2i</code>
$s_8$	<code>}else{</code>
$s_9$	<code>    sum = sum - i;</code>
$s_{10}$	<code>    product = product / i;</code>
$s_{11}$	<code>}</code>
$s_{12}$	<code>print (sum);</code>
$s_{13}$	<code>print (product);</code>

Figure 2.2: Faulty code example (Wong, Gao, et al. 2016)

Static slice for <i>product</i>		Dynamic slice for <i>product</i> with respect to a test case $a = 2$		Execution slice for <i>product</i> with respect to a test case $a = 2$	
$s_1$	<code>input(a)</code>	$s_1$	<code>input(a)</code>	$s_1$	<code>input(a)</code>
$s_2$	<code>i = 1;</code>	$s_2$	<code>i = 1;</code>	$s_2$	<code>i = 1;</code>
$s_3$		$s_3$		$s_3$	<code>sum = 0;</code>
$s_4$	<code>product = 1;</code>	$s_4$		$s_4$	<code>product = 1;</code>
$s_5$	<code>if (i &lt; a){</code>	$s_5$	<code>if (i &lt; a){</code>	$s_5$	<code>if (i &lt; a){</code>
$s_6$		$s_6$		$s_6$	<code>    sum = sum + i;</code>
$s_7$	<code>    product = product × i;</code>	$s_7$	<code>    product = product × i;</code>	$s_7$	<code>    product = product × i;</code>
$s_8$	<code>}else{</code>	$s_8$		$s_8$	
$s_9$		$s_9$		$s_9$	
$s_{10}$	<code>    product = product / i;</code>	$s_{10}$		$s_{10}$	
$s_{11}$		$s_{11}$		$s_{11}$	
$s_{12}$		$s_{12}$		$s_{12}$	<code>print (sum);</code>
$s_{13}$	<code>print (product);</code>	$s_{13}$	<code>print (product);</code>	$s_{13}$	<code>print (product);</code>

Figure 2.3: An example showing the differences among Static, Dynamic, and Execution Slicing (Wong, Gao, et al. 2016)

Using the code in figure 2.2 as the reference, assume it has the bug at  $s_7$ . Figure 2.3 demonstrates the results of the different techniques.

The static slice for the product contains all statements that could possibly affect the value of product. The dynamic slicing for product only contains the statements that do affect the value of product with respect to a given test case when  $a = 2$ . The execution slice with respect to a given test case contains all statements executed by this test.



## Program Spectrum-Based Techniques

Spectrum-based Fault Localization (SFL) techniques propose several approaches to find out exactly the faulty program entities. The majority of these techniques rank suspicious entities by using ranking metrics or statistical techniques. Nonetheless, artificial intelligence can also be applied to these techniques (H. Souza, M. Chaim, and Kon 2016). A program spectrum, also known as code coverage, is a collection of data that provides a detailed view on the behavior of software. This data is collected at run-time, and typically consist of a number of counters or flags for the different parts of a program. Spectrum-based fault localization uses information from this data executed by test cases to indicate entities more likely to be faulty (Abreu, Zoetewij, Golsteijn, et al. 2009). This program spectrum can be something like what we see in figure 2.4. This constitutes a binary matrix, whose columns correspond to  $N$  different parts of the program. The information that contains an error constitutes another column vector, the error vector. This vector represents a hypothetical part of the program that is responsible for all observed errors. So, fault localization consists in identifying the column vector that resembles the error vector the most (Abreu, Zoetewij, and A. J. v. Gemund 2007b).

$$\begin{array}{c}
 \begin{array}{c} M \text{ spectra} \end{array} \begin{array}{c} \begin{array}{c} N \text{ parts} \end{array} \\ \left[ \begin{array}{cccc} x_{11} & x_{12} & \dots & x_{1N} \\ x_{21} & x_{22} & \dots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M1} & x_{M2} & \dots & x_{MN} \end{array} \right] \end{array} \begin{array}{c} \text{errors} \\ \left[ \begin{array}{c} e_1 \\ e_2 \\ \vdots \\ e_M \end{array} \right] \end{array} \\
 \begin{array}{cccc} s(1) & s(2) & \dots & s(N) \end{array}
 \end{array}$$

Figure 2.4: Spectrum-based fault localization input example (Abreu, Zoetewij, and A. J. v. Gemund 2007b)

Having these data chunks, we need to find the resemblances between the columns in the matrix of program spectra. This can be achieved through similarity coefficients, which quantify our data on how much resemblance exists. There are many similarity coefficients, 2.3.2 depicts the ones that will be discussed in this document.

However, here are some key terms in order to fully comprehend the equations to come during this dissertation: Total number of test cases that cover a code statement ( $N_C$ ), Number of failed test cases that cover a code statement ( $N_{CF}$ ), Number of successful test cases that cover a code statement ( $N_{CS}$ ), Total number of failed test cases ( $N_F$ ), Total number of successful test cases ( $N_S$ ), Total number of test cases that do not cover a code statement ( $N_U$ ), Number of failed test cases that do not cover a code statement ( $N_{UF}$ ) and Number of successful test cases that do not cover a code statement ( $N_{US}$ ).

$$Dstar = \frac{(N_{CF})^*}{N_{UF} + N_{CS}}$$

$$Jaccard = \frac{N_{CF}}{N_{CF} + N_{UF} + N_{CS}}$$

$$Ochiai = \frac{N_{CF}}{\sqrt{N_F \times (N_{CF} + N_{CS})}}$$

$$Tarantula = \frac{\frac{N_{CF}}{NF}}{\frac{N_{CF}}{NF} + \frac{N_{CS}}{NS}}$$

$$WongII = N_{CS} - N_{CF}$$

### Data Mining-Based Techniques

Similar to machine learning, data mining builds a model finding anomalies, patterns and correlations from data. This can easily be used to fix fault localization problems since, usually, it is needed to find the pattern of the statement execution that leads to failure. Due to the huge amount of data that the complete execution trace has, reviewing it manually would be an extremely time-consuming task and not cost efficient. That is why some researchers tried to apply data mining in the fault localization field.

Firstly, in Cellier et al. 2011 it is combined two data mining techniques: Association Rules and Formal Concept Analysis (FCA). These techniques use as input a binary relation describing executions by subsets of program lines and verdicts. Then it tries to identify rules that correspond these executions to the program failures. Later it is computed the failure lattice for these rules. Quoting the definition given by the author, a failure lattice is "a partial ordering of the elements of the traces that are most likely to lead to failures". This is used to compute the ranking, which are needed afterwards to locate the faults.

Secondly, it was proposed another technique in Denmat, Ducassé, and Ridoux 2005. Here was shown that the method Tarantula (Jones, Harrold, and Stasko 2002a) can be re-interpreted as a data mining procedure. The data used in Tarantula are the executed statements and the verdict of a set of test cases. That data was transformed into an adequate form for data mining. Then from that data, association rules could be extracted. The value of these rules are evaluated based on two metrics, conf and lift, which are commonly used data mining metrics. These values can be used to determine whether or not a certain statement may contain a fault.

Lastly, a fault localization framework using data mining was proposed in Hirsch 2021. Here it is investigated a possible usage of a novel approach that inverts the hierarchy by identifying a starting point based on historical data and then apply other fault localization techniques. This historical textual data is formed by bug tickets, issue tickets, feature requests, pull requests, and commit messages. Then a machine learning algorithm will create a model with the responsibility of each code component.

### Machine Learning-Based Techniques

Machine learning is based on the idea that machines should be able to learn and adapt through experience and the use of data. Techniques that use machine learning have been successfully used in many areas of software engineering. In fault localization, these techniques have to learn the location of a fault based on input data. Examples of this data are the statement coverage and the success or failure from the execution result of each test case. After learning, they will adapt and try to deduce the suspicious program elements (Wong, Gao, et al. 2016).

In Ericwong and Yuqi 2011 for example, it is used an approach based on Back-Propagation (BP), a machine learning model to help programmers locate program faults. First it is

trained a BP neural network with the coverage data and the execution result (success or failure) gathered from the program, and then it is used the trained network to compute how suspicious is each executable statement, in terms of its likelihood of containing faults.

Another approach was taken in Wong, Debroy, et al. 2012 based on Radial Basis Function (RBF) networks, which are less susceptible to problems like paralysis and local minima which are often found in BP networks. Not only this, but also their ability to learn is proven better. This network is trained in a very similar way to the BP network, however, there is three innovative aspects in this work. First, it was introduced a method for representing test cases, statement coverage, and execution results within a modified RBF neural network formalism. Second, it was developed an unique algorithm to determine the number of hidden neurons and their receptive field centers at the same time. Third, it was used a weighted bit-comparison-based dissimilarity to estimate the distance between the statement coverage vectors of two test cases, instead of the usual traditional Euclidean distance which has proven inefficient in this context.

Lastly, we have an approach using the Deep Neural Network, another sub field of the machine learning family that is most commonly known as "Deep Learning". In deep learning, a computer model learns to perform directly from images, text, or sound. These models can achieve state-of-the-art accuracy, sometimes even over performing humans. In this case, the network was also trained very similar to the aforementioned but this one requires less percentage of examined statements to find out all faulty versions. Finally, even though this method proved is effectiveness and efficiency it still needs to be improve in order to handle multiple bug programs and big-scale software (W. Zheng, Hu, and J. Wang 2016).

### **Mutation-Based Techniques**

Instead of conventional program execution, mutation-based fault localization uses information from mutation analysis as inputs to its ranking metric or risk evaluation calculation (Jia and Harman 2011). As a result, it introduces some faults known as mutants into the program under investigation. Mutants are created using mutation operators, which are simple syntactic rules. Mutation testing is done by running the mutant programs through a set of test cases and observing the differences in behavior between the mutant and original program versions. As a result, the mutants might be classified as either killed or alive. Different approaches give different meanings to the mutants that are not killed and the ones that are (Zou et al. 2021).

MUSE (Moon et al. 2014) and Metallaxis-FL (Papadakis and Le Traon 2015) are two known Mutation Based Fault Localization (MBFL) tools. The differences between these two lie in the coefficient they use in order to calculate the suspicion level score and in the meaning of a killed mutant. MUSE uses its own coefficient score where Metallaxis-FL uses the Ochiai coefficient shown before (2.3.2). Also to count as killing a mutant in MUSE, a failed test case must be changed to successful, on the other hand to count as killing a mutant in Metallaxis, a failed test case merely needs to generate a different result (which may still be unsuccessful).

### **Model-Based Techniques**

Model-Based techniques work towards finding out all the possible causes of odd behaviour in an observed system based on what one knows about the system's expected behaviour

when its working correctly (Shchekotykhin, Schmitz, and Jannach 2016). Now, it will be shown some of the model-based techniques presented over the years.

This first model-based fault localization technique aims to help pin down the manual localization process into a small fraction of the whole system. Maruyama and Matsuoka 2008 consists of two parts: pre-fault model generation and model based anomaly detection. In the first place, it is collected function call traces from each process and it is originated an execution model that mirrors the function calling behaviours of the whole system. When there is a failure, the anomaly score is computed for each execution unit in the trial traces by comparing it against the attained model. This score quantifies how likely the execution unit is correlated with the fault. This way, the analyst can significantly reduce the time spent doing the manual localization by prioritizing the execution units with higher anomaly scores.

Nonetheless, model-based techniques can have different types based on their intention. It is the case of Mateis, Stumptner, and Wotawa 2000 which is a dependency model-based technique since its model derives from dependencies between statements in a program. Here was created a dependency model for Java programs in order to locate and partly repair bugs. This model handles all object-oriented features, loop constructs and global variables.

Moreover, there is a sequential model-based technique in Shchekotykhin, Schmitz, and Jannach 2016 aiming to reduce the time of the sequential diagnosis sessions.

The algorithm uses partial diagnoses, which are computed using a subset of the minimal conflicts. Conflicts are a set of components that don't work correctly as a whole at the same time given the observations and measurements. Then it is determined the best possible partitioning of the partial diagnoses which form a smaller search space than in the original problem setting. Lastly, it is guaranteed that the true problem cause, called preferred diagnosis, will be found.

Lastly, in Abreu, Zoetewij, and A. J. C. v. Gemund 2008 is presented an observation-based model technique. This approach uses dynamic information, namely abstraction of program traces, to generate a sub-model of the program under analysis. This model is used to compute the set of valid diagnoses, ranking them afterwards according to the likelihood of them being guilty of the failures. Despite most approaches in fault localization only present a single explanation, this approach also contains multiple fault explanations in the ranking diagnosis.

### **Program State-Based Techniques**

A program stores values in variables or constants. A program state consists of these variables or constants and their value at a given time during the execution of the program. As the program executes its state will change, either the variable may differ or even the contents stored in memory. Analyzing the variables or constants in a precise state is a good approach to locate bugs in the program.

In Zimmermann and Zeller 2001 is presented a technique called delta debugging. Using memory graphs, that deliver an overview of all data structures of the program, to capture and explore the program states. This way, it is possible to contrast a graph of a successful and a failed test. Also to test the level of suspicion, we can replace the values of a failed test in a corresponding value of a successful test at the same point. If it replicates, it is considered suspicious.

Another program state-based fault localization technique is shown in Zhang, N. Gupta, and R. Gupta 2006. Its approach strove to repeat executions of the program on the failing input and then switch the conditional branch results during these retries till it was discovered a predicate switching that caused the program to build the correct output. To achieve this end result, this approach uses a search strategy made of three important steps:

- Only one predicate switch at a time - the program behaviour is not explored by switching several predicates at the same time since its possibilities are immense;
- Last Executed First Switched (LEFS) Ordering - after having only one predicate switch at a time, it is needed to choose which predicate to look for first. For this LEFS is used, it is explored possible predicates in the reverse order of the predicate executions, the outcome found last will be the first;
- Prioritization-based (PRIOR) Ordering - Adding to the LEFS strategy it used the PRIOR strategy which consists of two main steps. In the first step, an algorithm is used to divide all branch predicate instances into those that are expected to be influenced by the faulty code and those that are not expected to be influenced by faulty code. The first group is explored before the second. In the second step, it is ordered the branch predicate within the first group according to their dependence distance from the wrong output value. The order result of these branch predicates is then used in our search.

This study shown us that can present valuable clues to the cause of the failure and, therefore, assist in fault localization field.

### Statistics-Based Techniques

Statistical debugging is a dynamic analysis to identify the underlying causes of software failures. Several algorithms were presented over the years, therefore, here will be shown just an overview of some of them.

In Liblit et al. 2005, it is possible to verify an algorithm that isolate bugs in programs containing multiple bugs that were not diagnosed. Their approach consists of instrumented predicates tested at particular program points. The idea behind it is to simulate the manner in which programmer usually find and fix bugs, which is by first identifying the most important bug, then fix it and repeat. To get into the algorithm, we should understand some nuances:

- Feedback report  $R$  - indicates whether the run of the program succeeded or failed, if a predicate is observed to be true at least once during run  $R$  then  $R(P) = 1$ , otherwise  $R(P) = 0$ .
- Bug profile  $B$  - a set of failing runs (feedback reports) that share the same cause of failure.
- Predicate  $P$  - simply a bug predictor, whenever  $R(P) = 1$  is likely that it belongs to the bug profile.

Finally, the algorithm consists on first computing the probability that  $P$  being true implies failure,  $\text{Failure}(P)$ , and the probability that the execution of  $P$  implies failure,  $\text{Context}(P)$  for each predicate. Predicates that have a  $\text{Failure}(P)$  minus  $\text{Context}(P)$  less or equal to 0 are discarded. The remaining predicates are prioritized based on their scores, which are an indicator of the connection between the predicate and the bug profile.

Another interesting statistical approach was taken by Liu et al. 2006 to localize software faults without prior knowledge of program semantics. It was called the SOBER technique and it was used to rank predicates that were suspicious. In this technique, a predicate  $P$  can be evaluated as true more than once in the execution of one test case. Compute  $\pi(P) = \frac{n(t)}{n(t)+n(f)}$ ; the probability that  $P$  is evaluated as true in each execution of a test case, where  $n(t)$  is the number of times  $P$  is evaluated as true and  $n(f)$  is the number of times  $P$  is evaluated as false. If the distribution of  $\pi(P)$  in failed executions is significantly different from that in successful executions, then  $P$  is related to a fault.

On other hand, there are other approaches just like the one in You, Qin, and Z. Zheng 2012, where it is exploited the statistical information of two sequentially connected predicates in the execution. It is built an execution graph for each execution of a test case, where the predicates are the vertices and the transition of two sequential predicates are the edges. Lastly, for each edge a suspicion value is calculated to quantify its likelihood of fault. This has proven more effective than older approaches.

## 2.4 Fault Localization Techniques Comparison

Several fault localization techniques were shown, presented some papers related to them as well as what these techniques consist of. However, there was no comparison between these families of techniques.

Here is presented an evaluation of some of them based on Zou et al. 2021 paper. This study was based on 357 real-world faults from the Defects4J dataset. Most of the tools from other techniques are not open source and this study compares some of the techniques that were presented in this thesis like SFL (2.3.2), slicing (2.3.2), MBFL (2.3.2) and data mining (2.3.2).

To evaluate these techniques they used the EXAM score. EXAM is the percentage of elements that have to be examined until finding a faulty element, averaged across all 357 faults. It is important to note that smaller EXAM scores are better. In the table 2.1, it is possible to see the results of this metric applied to the tools of the respective fault localization techniques.

Table 2.1: Evaluation of Fault Localization Techniques based on Zou et al. 2021

Family	Technique	EXAM
SFL	Ochiai	0.033
	Dstar	0.033
MBFL	Metallaxis	0.118
	MUSE	0.304
Slicing	union	0.207
	intersection	0.222
Data mining based	BugLocator	0.212

SFL procedures produce better outcomes based on table 2.1 alone; however, the decision on which technique the tool that will be integrated employs will be made in chapter 3 using a decision-making method considering more variables.

## 2.5 Fault Localization Tools and Integrations

Fault localization can be one of the most challenging tasks during software development. Having the right tools in this task can be the decisive factor on many studies. Therefore, many tools have been developed in order to reduce the amount of effort and time software developers have to spend on fault localization (Chandrasekaran et al. 2016). In this section, it is listed some of the tools researched, some of them already integrated with an IDE, along with a brief description.

### 2.5.1 Aletheia

Firstly, there is Aletheia (Golagha et al. 2018) which is a fault diagnosis toolchain aiming to help everyone reducing their failure analysis time. It is designed for C++ projects and its also compatible with the Google testing framework. Also, it was released as an open source tool on Github, being also integrated in Visual Studio as an extension.

Aletheia has three main components: data generation, failure clustering and fault localization as it is presented in figure 2.5.

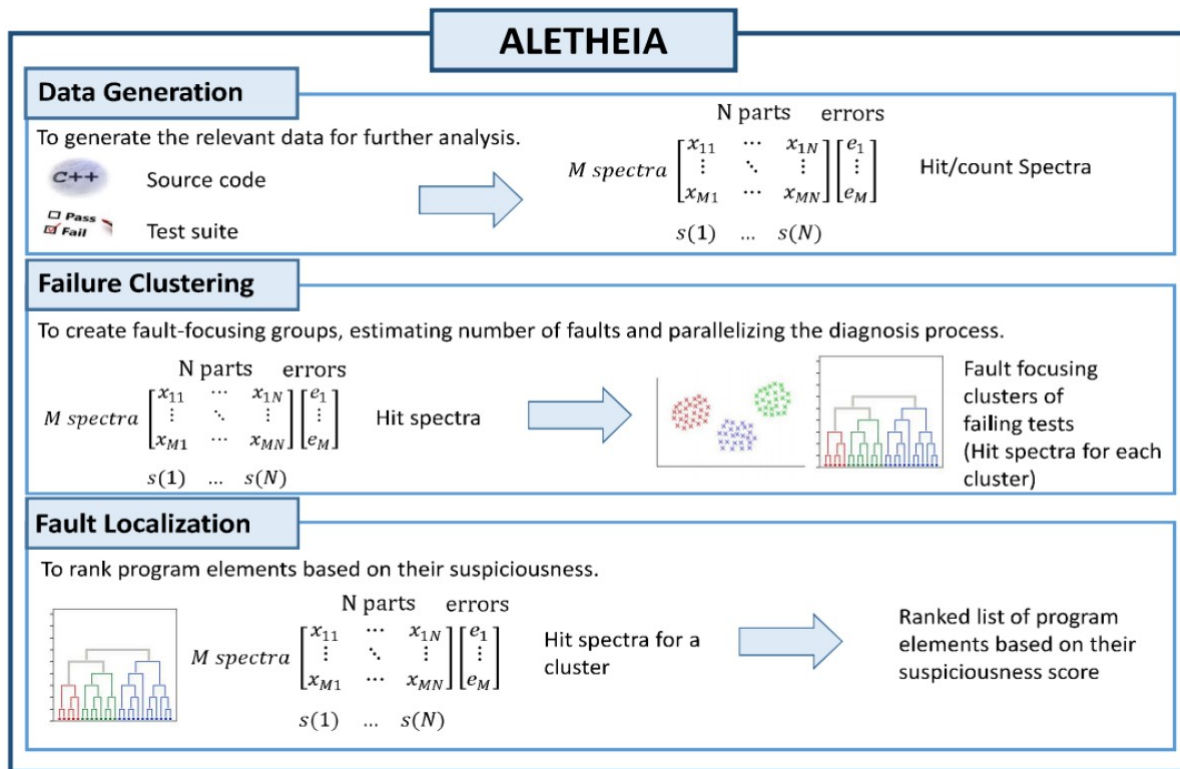


Figure 2.5: Aletheia main components (Golagha et al. 2018)

The first component, named data generation, intends to generate and prepare data for further analysis using the source-code and its test suite. Then, it runs the test, collecting coverage information and aggregating it into the hit/count spectra.

The second component is failure clustering. Here, it receives a hit spectrum (provided by the data generation component or by other tools such as GZoltar (see 2.5.5)). Firstly, it generates a hierarchical tree of failing tests. Then, the tree is divided into some clusters

using spectrum-based fault localization. In the end, it selects as many representatives for each cluster as the user desires.

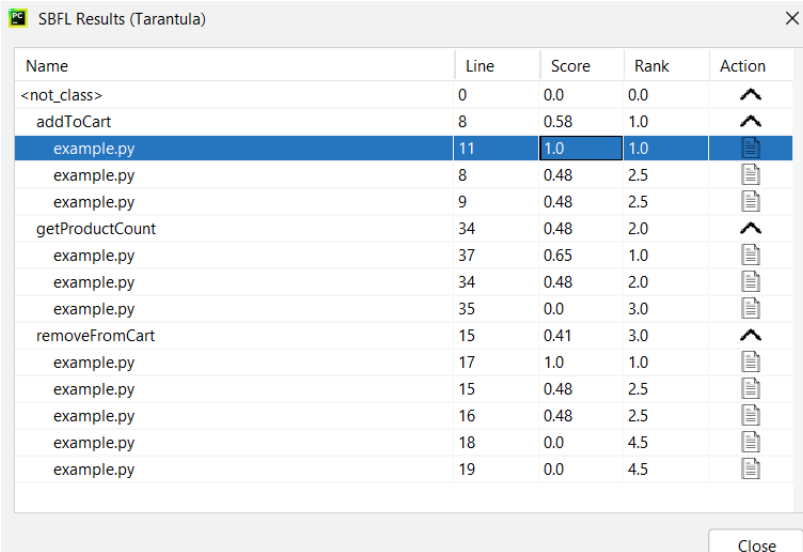
The last and third component, fault localization, uses the information summarized in a hit spectrum (of all tests or one cluster), returns a list of program elements with their ranks based on their likelihood to be faulty. Atheleia implements Tarantula, Ochiai, Jaccard and Dstar metrics (see in 2.3.2).

### 2.5.2 CharmFL

CharmFL (Sarhan et al. 2021) is an Open-source fault localization tool that has IDE integration with PyCharm, a popular Python development platform. This tool uses Spectrum-based fault localization (see in 2.3.2) to assist Python developers by examining their programs and generating useful data at run-time in order to create a ranked list of potentially faulty program elements.

It is possible to divide this tool into two parts, the GUI and the frameworks architecture. In the GUI, program elements are highlighted with different shades of red color based on the suspicion level scores. The darker the color is, the more suspicious the element is. Also a great advantage of this tool is that the user can choose multiple metrics like Tarantula and Ochiai , Dstar and Wong II (see in 2.3.2).

In the end, the results figure (2.6) shows the program elements hierarchically, next to them there are their positions in the source code, their ranks, and their scores.



Name	Line	Score	Rank	Action
<not_class>	0	0.0	0.0	⬆
addToCart	8	0.58	1.0	⬆
example.py	11	1.0	1.0	📄
example.py	8	0.48	2.5	📄
example.py	9	0.48	2.5	📄
getProductCount	34	0.48	2.0	⬆
example.py	37	0.65	1.0	📄
example.py	34	0.48	2.0	📄
example.py	35	0.0	3.0	📄
removeFromCart	15	0.41	3.0	⬆
example.py	17	1.0	1.0	📄
example.py	15	0.48	2.5	📄
example.py	16	0.48	2.5	📄
example.py	18	0.0	4.5	📄
example.py	19	0.0	4.5	📄

Figure 2.6: CharmFL ranking list output (Sarhan et al. 2021)

About the framework's architecture, it is important to note that it can be used as a stand-alone tool or integrated in other IDEs too as a plug-in. Also, in order to collect the program's spectra it uses the popular coverage measuring tool for Python, called "coverage.py". The framework transforms the statement level of the coverage measuring tool to method and class levels as shown in 2.6. Later, the classes are sorted based on their suspicion level scores, then the functions, and finally the statements. Finally, to make the coverage matrix, it is used the ".coveragerc" file where the user can configure the measurement.



### 2.5.3 FLACOCO

FLACOCO (A. Silva et al. 2021) is a recent fault localization tool for Java, it uses SFL (2.3.2) and can be used abroad a wide variety of Java versions. This is entirely possible due to this tool being built on top of JaCoCo, a top tier code coverage library with almost a decade of existence. Therefore by update the JaCoCo version, the tool is able to support new Java versions which is a major advantage comparing to other existing fault localization tools.

In figure 2.7 is present the architecture of FLACOCO which can be divided into seven different components: User Interface, Test Detection, Instrumentation and Test Execution, Coverage Collection, Suspiciousness Computation, AST Bridge and File Exporter. Below will be given a brief explanation of each component in this order.

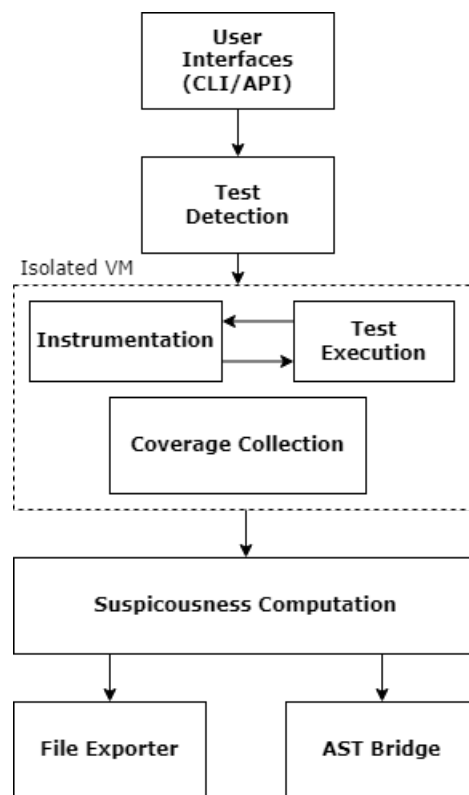


Figure 2.7: FLACOCO Architecture based on (A. Silva et al. 2021)

The first component, User Interface, is the entry point of the tool and it can be used in the command line or as a Java API. Secondly in Test Detection, it is detected the tests to be executed and analyzed with supported test framework (JUnit 3,4 and 5). Thirdly, all of the tests before mentioned that were chosen are executed and instrumented by JaCoCo. Next, it is collected the information related to the test result, exceptions thrown and the coverage. After this test suite is executed, the results will be used to compute the suspicion level score using the Ochiai coefficient (2.3.2). In addition, FLACOCO can provide an output as an AST (abstract syntax tree) using the AST Bridge or as JSON or CSV using the File Exporter. Lastly, it is important to note that FLACOCO does not have an IDE integration present in the paper (A. Silva et al. 2021).

### 2.5.4 FLAVS

FLAVS (N. Wang et al. 2015), which stands for Fault Localization Add-in for Visual Studio, has an IDE integration with Visual Studio coded in C#. It can be used to develop projects in Visual C, Visual Basic, C#, and other languages.

Figure 2.8 represents the FLAVS flowchart with its main functions, below will be given a brief description of each step present in the flowchart.

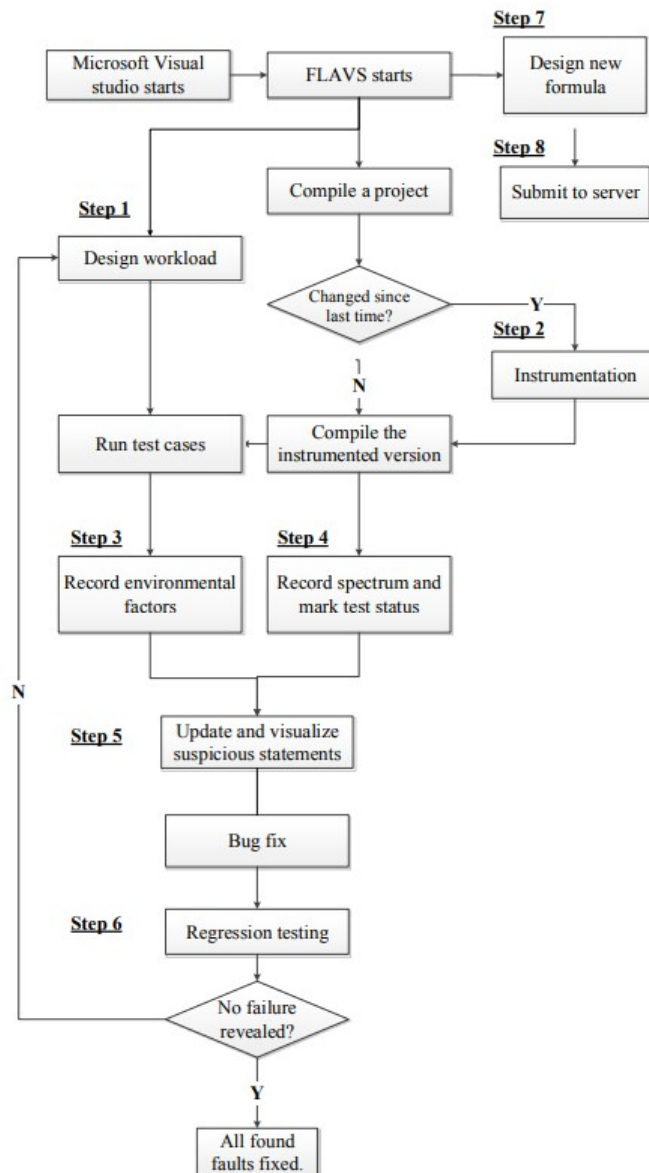


Figure 2.8: FLAVS flowchart (N. Wang et al. 2015)

Starting with the step one, here is defined the amount of test cases a test has to do, the set of test cases executed, the execution time of each test case and the intervals between each test cases (this can be defined by the user). In step two is provided an automatic instrumentation mechanism, which is triggered before the program compilation. Later in step three, environmental factors like memory consuming, CPU usage and thread numbers

are kept in check. Step four saves all the coverage information automatically and the developer can be asked to mark the test status, i.e., successful or failed (it also can be done automatically). After in step five the suspicion level of each statement is calculated to later be shown and highlighted. Finally in step six, it is provided the option to rerun all previous test cases. Step seven and eight are optional and basically they let the users create their own new technique and upload it to the servers respectively.

### 2.5.5 GZoltar

GZoltar (Campos et al. 2012a) is a SFL tool that provides an infrastructure to automatically instrument the source code and produce run time data. Additionally, this data is examined in order to minimize the test suite and return a ranked list of the suspicious elements. GZoltar generates a HTML visualization and a warning marking these suspicious elements in the code with a high chance of being faulty. This tool implements SFL and can employ several similarity coefficients such as Ochiai, Dstar, Jaccard (see their formulas in 2.3.2). Figure 2.9 presents the information flow of GZoltar.

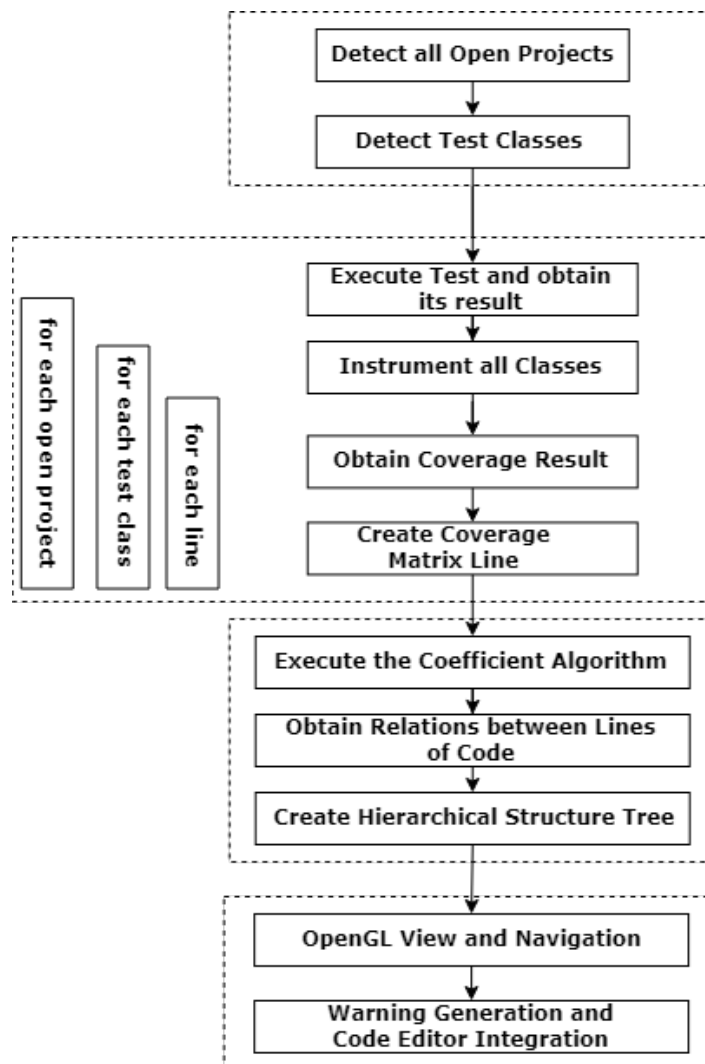


Figure 2.9: GZoltar information flow based on (Campos et al. 2012b)

At this moment, GZoltar is available as a command line interface, ant task, maven plug-in and Visual Studio Code IDE integration. Also, it is important to denote a paper that describes this integration of the GZoltar tool into the IDE Visual Studio Code.

In Brito 2020, the author argued that the GZoltar tool should transition from an Eclipse plug-in to a Visual Studio Code plug-in because Eclipse's popularity has been falling ever since then. This paper showed state-of-art techniques, the GZoltar tool as the major focus, demonstrated the Visual Studio Code principles that were essential to know, whether they were visual concepts or API related, and explained how the integration was designed.

Nonetheless, the created extension was published in the Visual Studio Marketplace being available to a wide variety of users and conducted an user study to prove the effectiveness of the plug-in which turned out to be successful.

### 2.5.6 MUSEUM

MUSEUM (Hong et al. 2015) is a MBFL technique (2.3.2) for real-world multilingual programs. A multilingual program is made up of numerous pieces of code written in various languages that communicate with one another via language interfaces. Also, MUSEUM is language-independent since it generates syntactic mutations and statistical reasoning based on the outcomes of testing on a target program and its mutants.

MUSEUM requires no extra build/runtime environments, merely a mutation tool and a coverage assessment tool for the target programming languages. This is a significant benefit over alternative debugging techniques that necessitate the use of specialized infrastructure such as virtual machines or compilers.

In terms of its fault localization process, MUSEUM divides its process into four steps:

- MUSEUM accepts the target source code and the target program's test cases as input;
- MUSEUM creates mutants, each of which is obtained by modifying a single statement of the target code;
- MUSEUM examines the mutants' testing data to identify faulty; assertions;
- MUSEUM uses its own suspiciousness metric to evaluate the test results and outputs a suspiciousness score ranking.

Finally, MUSEUM was implemented targeting Java and C primarily (author said it would support other languages later but not possible to confirm) and it is not integrated with any IDE.

### 2.5.7 Tarantula

Tarantula (Jones, Harrold, and Stasko 2002b) tool is a program visualization system to aid fault localization. This tool is written in Java and the SFL metric named Tarantula (check 2.3.2). Its input is software's source code and the results of executing a test suite in order to provide a visual mapping of the participation of each program statement in the testing. Also, the program statements are colored using a from red to yellow to green: the greater the percentage of failed test cases that execute a statement, the brighter and more red the statement be. Finally, there is no IDE integration with this tool.

### 2.5.8 UnitFL

UnitFL (Chen and N. Wang 2016) is an add-in of Microsoft Visual Studio written in C#. It supports projects written in C#, C and Visual Basic. UnitFL has the options to run in test and fault localization mode (see in figure 2.10). Both modes include unit test design, visualized guidance and regression test modules.

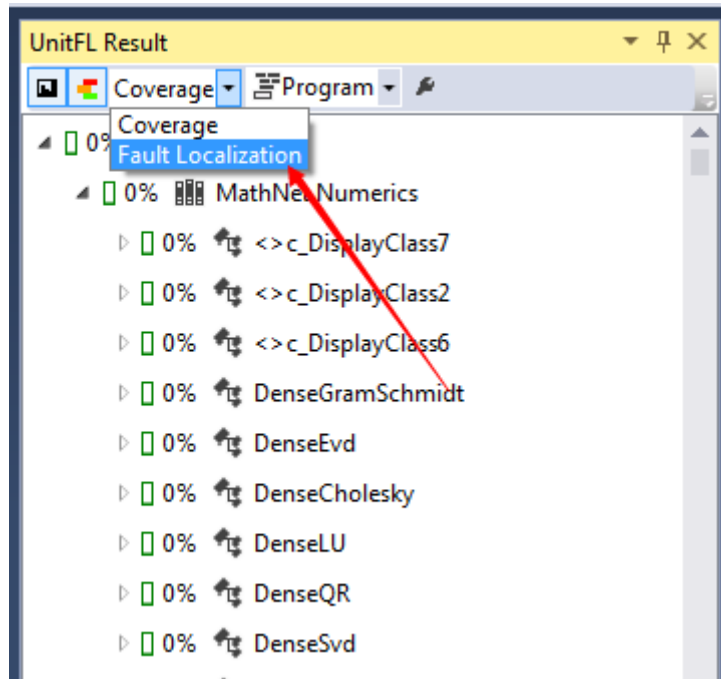


Figure 2.10: UnitFL menu *UnitFL* n.d.

UnitFL uses program slicing (2.3.2) and dynamic program instrumentation approaches in order to reduce the program execution overhead. Besides, it also puts up different floors of granularities for fault localization analysis to provide different aspects of execution during the program analysis. Lastly, it evaluates each unit test performance to discover underlying bugs and shows with different colors based on their suspicion level ranging from green to red.

## 2.6 Summary

Over this chapter, multiple fault localization tools that leverage several different techniques were demonstrated, and some of them were even integrated with an IDE. However, it is necessary to investigate which tools are accessible for integration into an IDE, whether they already have an integration, and whose family of automatic fault localization techniques they belong to.

Table 2.2 summarizes the thesis' findings, including details on the technique employed, whether or not it includes IDE integration and the year in which the associated paper was published.

Table 2.2: Comparative Analysis of Fault Localization Tools

<b>Tool</b>	<b>Technique</b>	<b>IDE Integration</b>	<b>Language Supported</b>	<b>Open Source</b>	<b>Year</b>
<b>Atheleia</b>	SFL	Visual Studio	C++	Yes	2018
<b>CharmFL</b>	SFL	PyCharm	Python	Yes	2021
<b>FLACOCO</b>	SFL	None	Java	Yes	2021
<b>FLAVS</b>	SFL	Visual Studio	C/Visual Basic/C#	No	2015
<b>GZoltar</b>	SFL	Visual Studio Code	Java	Yes	2012
<b>MUSEUM</b>	MBFL	None	Multi language	No	2015
<b>Tarantula</b>	SFL	None	Java	No	2002
<b>UnitFL</b>	Slicing	Visual Studio	C#/C/Visual Basic	No	2016



## Chapter 3

# Value Analysis

This chapter addresses how can this dissertation bring value. Nuances about the innovation process and the business process will be presented, as well as an identification and analysis of opportunities. Lastly, there will be a value proposition and the use of Analytic Hierarchy Process (AHP) for multi criteria decision to aid choosing the best fault localization technique to focus.

### 3.1 Innovation Process

Accordingly to P. Koen, Bertels, and Kleinschmidt 2014, the innovation process can be split into three different steps (see 3.1): Fuzzy Front End (FFE) also known as Front End of Innovation (FEI), New Product Development (NPD) and Commercialization.

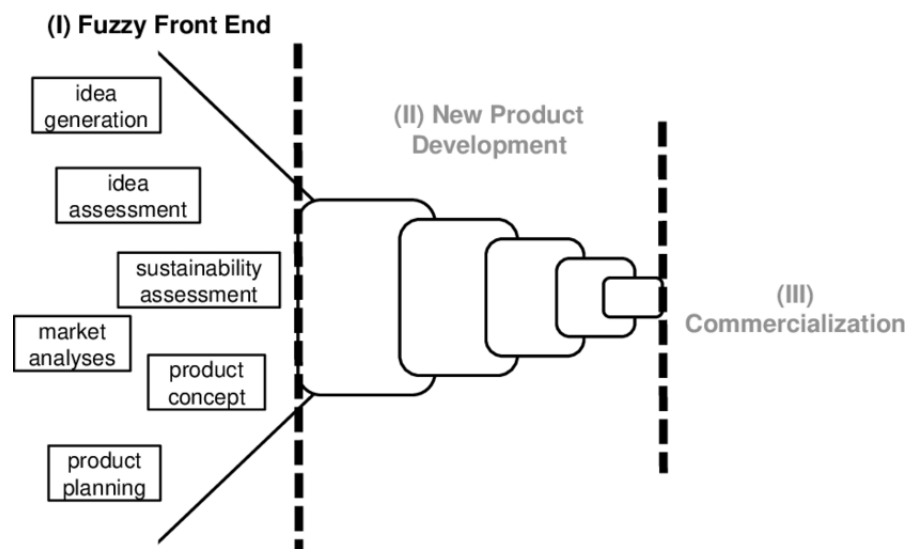


Figure 3.1: The three steps of the Innovation Process (Dimitrijevic 2014)

The first step, the **FFE**, represents a chaotic and experimental state where there are a lot of uncertainties, mostly financial ones. The second step, the **NPD**, has concrete and goal-oriented plans with formalized dates and teams focused on process and/or product development. The last and third step, the **Commercialization**, is the final result originated from the last two steps and the main goal of the innovative process (P. A. Koen et al. 2002).



## 3.2 New Concept Development

New Concept Development (NCD) model is a framework for Front End of Innovation that provides a common knowledge for the field. Additionally, it is divided into three fundamental parts:

- **The engine** - at the center of the model there is the engine, which provides power to the innovation process. It consists of two separate segments: organizational attributes/teams and collaboration.
- **The wheel** - in the inner part of the model it is present the wheel. It contains the five fundamental elements of the front end of innovation: opportunity identification, opportunity analysis, idea generation, idea selection and concept definition.
- **The rim** - that last element, the rim, comprises the environmental factors that influence the engine and form the five activity elements.

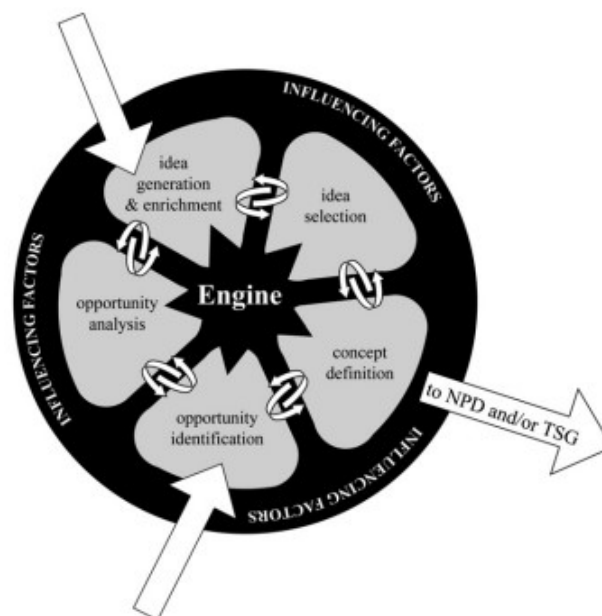


Figure 3.2: The New Concept Development (NCD) Model(P. Koen, Bertels, and Kleinschmidt 2014)

The circular shape of the model in Figure 3.2 is intended to depict how thoughts move between the five elements. The arrows pointing into the model show potential project starting points, which can be Opportunity Identification or Idea Generation and Enrichment. The projects that exit through the NPD or Technology Stage Gate processes are represented by the arrow pointing out of the model.

### 3.2.1 Opportunity Identification

Opportunity identification is when the organization identifies the different available opportunities to eventually allocate its resources. This can be done in different ways, such as making existing processes better, more efficient or effective. The opportunity can also be

an entirely new direction for the business, a new service offering or a new manufacturing process (P. A. Koen et al. 2002).

In the context of this dissertation, the opportunity identification lies in the fault localization field. Since the everyday programmer faces faults, finding the underlying cause can be a hard task to perform with the ever increasing scale of today's software and its complexity. Adding to this fact, the fault localization tools are mostly deprecated, with no support whatsoever or hidden from the average person. That is where is the opportunity, to provide the integration of a fault localization tool with a popular IDE to make sure its availability to everyone.

### 3.2.2 Opportunity Analysis

After having the opportunity identified, it is mandatory to understand if the opportunity is worth pursuing. In order to know this, information on reducing uncertainties about how attractive is the opportunity, the size of the future development and the risk tolerance should be scouted.

Therefore, a SWOT analysis is an adequate way to evaluate and analyse the opportunity. It is usually associated with the strategic planning, which consists in a sequential set of analyses and choices that will eventually lead to the strategy with better odds of succeeding. The SWOT analysis process has four components: **Strengths, Weaknesses, Opportunities** and **Threats** (Gürel 2017). Strengths describes what the opportunity excels at, weaknesses are the internal issues that stop the opportunity from its prime level, opportunities are the favourable external factors that add value to the opportunity and lastly, threats are external factors that can harm our opportunity.

<b>Strengths</b> <ul style="list-style-type: none"> <li>• Open Source</li> <li>• User-friendly</li> </ul>	<b>Weaknesses</b> <ul style="list-style-type: none"> <li>• Development team reduced to one member.</li> <li>• Non existent Certificate</li> </ul>
<b>Opportunities</b> <ul style="list-style-type: none"> <li>• Software development is growing larger everyday</li> <li>• Lack of up-to-date integrated fault localization tools</li> </ul>	<b>Threats</b> <ul style="list-style-type: none"> <li>• Big companies have their own tools.</li> </ul>

#### Strengths

- Open Source: It will be available for free in the marketplace and in an open source community, closing the gap with the customer and being open to new changes.
- User-friendly: Developers will not need to use external tools or find themselves surrounded by deprecated ones. The plugin will have a friendly interface for the code editor.

#### Weaknesses

- Development team reduced to one member: Being the author of this thesis the only member on this project, it faces a disadvantage in terms of time and in comparison with other multi-element team projects.

- Non existent Certificate: Since it is an academic project, it does not have a certificate of approval or trust like big companies do. This might scare away developers that want options validated by big entities.

### Opportunities

- Software development is growing larger everyday: It is everywhere. Since software is needed everywhere and it is only getting larger and more complex, developers need fault localization tools in order to be more efficient. As a result, the fault localization field also grows bigger and it is a necessity.
- The lack of up-to-date integrated fault localization tools show up as an opportunity since it becomes a need for the developers who want to have a better experience debugging and need to be more efficient.

### Threats

- Big companies have their own tools: Organizations will rather use their own developed tools than to use an open source one with no guarantee of success.

### 3.2.3 Idea Generation

Idea Generation is the evolutionary process that includes the birth, development and maturation of the opportunity into a definitive idea (P. A. Koen et al. 2002). More often than not, the idea is not straightforward. During the process of generating the idea there will be several step backs where it will be needed to reshape, combine, modify or even upgrade what was previously thought of. Additionally, brainstorming sessions where multiple perspectives can be gathered are essential since a new concept can arise outside the confines of the current one. In this dissertation, the idea was originally thought of by my supervisor ProfDoc Alberto Sampaio and after some brainstorming sessions it was formalized to be an integration of a software localization tool into an IDE. Nevertheless, this idea is still not complete since there were still some questions left to answer in order to complete the idea:

- **Q1**: What is the fault localization technique that the tool has to have?
- **Q2**: What tool will be used?
- **Q3**: What IDE will be used?

As a result, those three questions all depend on **Q1** "What is the fault localization technique that the tool has to have?". Having stated that, **Q1** should be the primary focus. After considerable investigation (2.4), four alternatives were found in order to answer **Q1**:

- **A1**: Spectrum Based Fault Localization Technique
- **A2**: Mutation Based Fault Localization Technique
- **A3**: Program Slicing Fault Localization Technique
- **A4**: Data Mining Based Fault Localization Technique

### 3.2.4 Idea Selection

Idea selection is a critical but tricky process in order to attain the most value out of the opportunity. This process that can be as simple as choosing one idea out of a set of many self-generated ideas, can be the decisive factor of success or failure. However, this idea is allowed to grow and advance but the basis must be there (P. A. Koen et al. 2002).

Furthermore, there is no system that ensures the correct decision is made. Having said that, there are various strategies that can help with the idea selection process, ensuring that it has a better chance of succeeding. This is the case with AHP, the approach employed for that goal in this dissertation.

#### **Analytical Hierarchical Process**

The AHP is a method to aid in the process of deciding complex decisions, using math and psychology. Accordingly to Nicola 2018, it is divided into seven steps:

- Structure the decision hierarchy from the top with the goal of the decision, its criteria and alternatives.
- Compare the alternatives and the criteria.
- Determine the relative priority for each criterion.
- Examine the coherence of relative priorities.
- Construction of a parity comparison matrix for each criterion, taking into account all of the alternatives
- Obtain the composite priority for the alternatives.
- Alternative's choice

For the first step, the problem was already exposed through a question in section 3.2.3, which asked "What is the fault localization technique that the tool has to have?". To answer this question it will be used the following criteria:

- Accuracy: the proportion of correct localized faults in the total number of faults.
- Reliability: the consistency of the solution.
- Performance: the percentage of code statements that need to be examined until the first faulty statement is reached.

For the alternatives, we will use the ones tested in 2.4, since it is the only comparison data it could be found. And, these are:

- Spectrum Based
- Mutation Based
- Program Slicing
- Data Mining Based

Now it is possible to generate the AHP decision hierarchy tree:

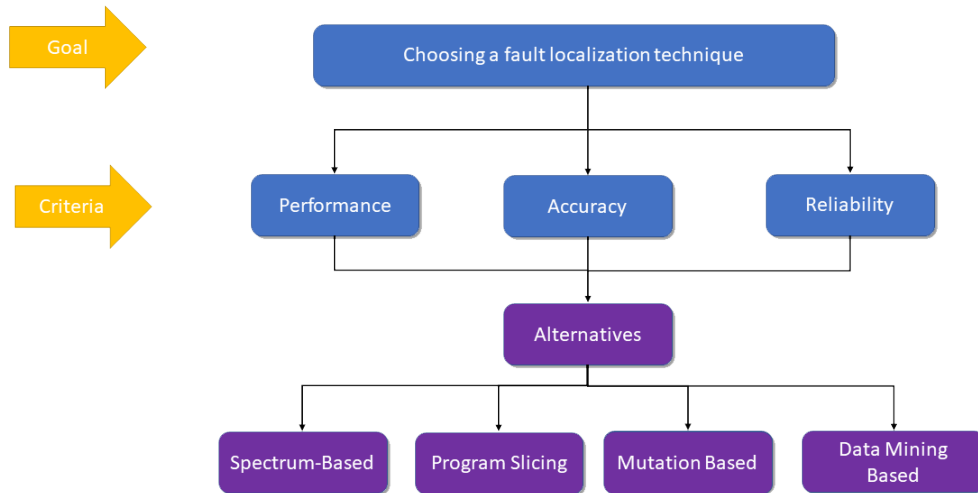


Figure 3.3: Hierarchical Decision Tree

### Criteria comparison

The second phase entails using a comparison matrix to establish priorities among the elements at each level of the hierarchy. In order to correctly compare our alternatives with the defined criteria, it is needed an importance scale. This scale is presented in with the rating explanations.

Table 3.1: Fundamental scale from Saaty 2008

Intensity of Importance	Definition	Explanation
1	Equal Importance	Two activities contribute equally to the objective
2	Weak or slight	
3	Moderate importance	Experience and judgement slightly favour one activity over another
4	Moderate plus	
5	Strong importance	Experience and judgement strongly favour one activity over another
6	Strong plus	
7	Very strong or demonstrated importance	An activity is favoured very strongly over another; its dominance demonstrated in practice
8	Very, very strong	
9	Extreme importance	The evidence favouring one activity over another is of the highest possible order of affirmation
Reciprocals of above	If activity i has one of the above non-zero numbers assigned to it when compared with activity j, then j has the reciprocal value when compared with i	A reasonable assumption
1.1-1.9	If activities are very close	May be difficult to assign the best value but when compared with other contrasting activities the size of small numbers would not be too noticeable yet they can still indicate the relative importance of the activities

Finally, a weight is provided to each of the criteria described above using the AHP fundamental scale (table 3.1). In addition, paired procedures are used to compare each criterion (table 3.2).

Table 3.2: AHP table for criteria comparison

	<b>Accuracy</b>	<b>Reliability</b>	<b>Performance</b>
<b>Accuracy</b>	1	2	5
<b>Reliability</b>	1/2	1	3
<b>Performance</b>	1/5	1/3	1

After analyzing the table 3.2, it is possible to infer that:

- Accuracy is very slightly more crucial than reliability.
- Performance isn't nearly as critical as accuracy.
- Reliability is more important than performance.

The author reasoning for this is that accuracy is the most determining factor in a algorithm specially in fault localization since the whole point is to discover the most faults possible, paired with it is reliability because in order to be a good technique it needs to be consistent. Lastly, performance is a useful metric to have since it can decide between two really good options but is not as mandatory as the last two.

### Criterion Relative Priority

Now, in the third step, there is two important steps to do. Firstly, in order to match all criteria to the same unit, the values of the comparison matrix will be normalized. To accomplish this, we must compute the sum of each column:

Table 3.3: AHP table for criteria comparison

	<b>Accuracy</b>	<b>Reliability</b>	<b>Performance</b>
<b>Accuracy</b>	1	2	5
<b>Reliability</b>	1/2	1	3
<b>Performance</b>	1/5	1/3	1
<b>Sum</b>	17/10	10/3	9

And then, normalize by dividing each matrix value by the sum of its associated column:

Table 3.4: Normalized criteria matrix

	<b>Accuracy</b>	<b>Reliability</b>	<b>Performance</b>
<b>Accuracy</b>	10/17	3/5	5/9
<b>Reliability</b>	5/17	3/10	1/3
<b>Performance</b>	2/17	1/10	1/9

Lastly, the priority vector must then be obtained in order to establish the relevance order of each criterion. To do so, the normalized matrix of the previous point is used in order to find the arithmetic mean of each row's values.

Table 3.5: Relative Priority of each criterion

	Accuracy	Reliability	Performance	Relative Priority
Accuracy	10/17	3/5	5/9	0.5813
Reliability	5/17	3/10	1/3	0.3092
Performance	2/17	1/10	1/9	0.1096

### Relative Priorities Coherence

Now, in order to quantify how consistent the judgments were for large samples of entirely random judgments, the fourth step is to calculate the Consistency Ratio (CR). If the RC is more than 0.1, the judgements are untrustworthy since they are too near of randomness.

To begin, obtain the value of  $\lambda_{max}$ , which indicates the biggest eigenvalue of a matrix A, using the equation below:

$$Ax = \lambda_{max}x$$

Where x denotes the priority vector of each criterion and Ax denotes the multiplication of this vector by the non-normalized priority matrix. The matrix 3.4 below displays the outcomes of this multiplication.

$$\begin{bmatrix} 1 & 2 & 5 \\ 1/2 & 1 & 3 \\ 1/5 & 1/3 & 1 \end{bmatrix} \times \begin{bmatrix} 0.5813 \\ 0.3092 \\ 0.1096 \end{bmatrix} = \begin{bmatrix} 1.7477 \\ 0.9287 \\ 0.3289 \end{bmatrix}$$

Figure 3.4: Multiplication of priority vector by the non-normalized priority matrix

Then, it is possible to calculate the matrix eigenvalue:

$$\lambda_{max} = \frac{\frac{1.7477}{0.5813} + \frac{0.9287}{0.3092} + \frac{0.3289}{0.1096}}{3} \approx 3.004$$

Finally, the division between the Consistency Index (CI) and the Random Index (RI) is required to compute the Consistency Ratio (CR). To calculate the CI, this is the formula:

$$CI = \frac{\lambda_{max} - n}{n - 1}$$

Where n is the number of criteria it was used. Because the value of  $\lambda_{max}$  has already been determined, the IC can be calculated quickly:

$$CI = \frac{3.004 - 3}{3 - 1} \approx 0,002$$

Then, the Random Index (RI) can be obtained in table 3.6, where the value depends on the number of criteria used:

Table 3.6: Random Consistency Index (Nicola 2018)

1	2	<b>3</b>	4	5	6	7	8	9	10	11	12	13	14	15
0.00	0.00	<b>0.58</b>	0.90	1.12	1.24	1.32	1.41	1.45	1.49	1.51	1.48	1.56	1.57	1.59

Since it was used three criteria, the RI is 0.58 and lastly, the value of CR is obtained:

$$CR = \frac{0,002}{0.58} \approx 0,0034$$

Then, it is possible to infer that the criteria is trustworthy since the obtained CR value was approximately 0.0034, which is less than 0.1.

### Alternative's Parity Comparison Matrix For Each Criterion

The fifth step is to observe the relative relevance of each of the alternative in the problem at hand. To accomplish this, the creation of the comparison matrices (this time, between each criterion and alternative), the respective normalization matrix, and the calculation of the priority vector are all repeated. Table 3.7,3.8 and 3.9 show the preliminary comparison using the fundamental scale (Table 3.1).

Table 3.7: Comparison Matrix of Accuracy between Alternatives

<b>Accuracy</b>				
	<b>Spectrum Based</b>	<b>Mutation Based</b>	<b>Program Slicing</b>	<b>Data Mining Based</b>
<b>Spectrum Based</b>	1	5	7	7
<b>Mutation Based</b>	1/5	1	3	6
<b>Program Slicing</b>	1/7	1/3	1	5
<b>Data Mining Based</b>	1/7	1/6	1/5	1

Table 3.8: Comparison Matrix of Reliability between Alternatives

<b>Reliability</b>				
	<b>Spectrum Based</b>	<b>Mutation Based</b>	<b>Program Slicing</b>	<b>Data Mining Based</b>
<b>Spectrum Based</b>	1	3	3	7
<b>Mutation Based</b>	1/3	1	2	5
<b>Program Slicing</b>	1/3	1/2	1	5
<b>Data Mining Based</b>	1/7	1/5	1/5	1

Table 3.9: Comparison Matrix of Performance between Alternatives

<b>Performance</b>				
	<b>Spectrum Based</b>	<b>Mutation Based</b>	<b>Program Slicing</b>	<b>Data Mining Based</b>
<b>Spectrum Based</b>	1	5	7	5
<b>Mutation Based</b>	1/5	1	5	2
<b>Program Slicing</b>	1/7	1/5	1	1/5
<b>Data Mining Based</b>	1/5	1/2	5	1

These matrices are then normalized, and the local priority vector for each one of them is determined. Table 3.7, 3.8 and 3.9 show the result as well as their local priority vector.



Table 3.10: Normalized Matrix for Alternatives-Accuracy Comparison and Local Priority

		Accuracy			
	Spectrum Based	Mutation Based	Program Slicing	Data Mining Based	Local Priority
Spectrum Based	0.6731	0.7692	0.625	0.3684	0.6089
Mutation Based	0.1346	0.1538	0.2678	0.3158	0.218
Program Slicing	0.0961	0.0513	0.0892	0.2631	0.1249
Data Mining Based	0.0961	0.0256	0.0178	0.0526	0.0480

Table 3.11: Normalized Matrix for Alternatives-Reliability Comparison and Local Priority

		Reliability			
	Spectrum Based	Mutation Based	Program Slicing	Data Mining Based	Local Priority
Spectrum Based	0.5526	0.6383	0.4839	0.3889	0.5159
Mutation Based	0.1842	0.2128	0.3226	0.2778	0.2493
Program Slicing	0.1842	0.1064	0.1613	0.2778	0.1824
Data Mining Based	0.0789	0.0425	0.0322	0.0556	0.0523

Table 3.12: Normalized Matrix for Alternatives-Performance Comparison and Local Priority

		Performance			
	Spectrum Based	Mutation Based	Program Slicing	Data Mining Based	Local Priority
Spectrum Based	0.6481	0.7463	0.3889	0.6097	0.5982
Mutation Based	0.1296	0.1492	0.2778	0.2439	0.2001
Program Slicing	0.0926	0.0298	0.0555	0.0243	0.0505
Data Mining Based	0.1296	0.0746	0.2778	0.1212	0.1508

**Alternative's Composite Priority and the Choice** Table 3.7, 3.8 and 3.9 results are then combined with table 3.5 relative priority vector to create a new matrix:

$$\begin{bmatrix} 0.6089 & 0.5159 & 0.5982 \\ 0.218 & 0.2493 & 0.2001 \\ 0.1249 & 0.1824 & 0.0505 \\ 0.0480 & 0.0523 & 0.1508 \end{bmatrix} \times \begin{bmatrix} 0.5813 \\ 0.3092 \\ 0.1096 \end{bmatrix} = \begin{bmatrix} 0.5790 \\ 0.2257 \\ 0.1345 \\ 0.0606 \end{bmatrix}$$

Figure 3.5: Multiplication of priority matrix by the criteria priority vector

Then, the composite priority vector may be calculated using these values by multiplying the result of each alternative with the priority vector. Based on the criteria, this vector will indicate the relevance of the alternative. Table 3.13 summarizes the findings:

Table 3.13: Criteria/Alternatives Classification Matrix and Composite Priority

	Accuracy	Reliability	Performance	Composite Priority
Spectrum Based	0.6089	0.5159	0.5982	0.5790
Mutation Based	0.218	0.2493	0.2001	0.2257
Program Slicing	0.1249	0.1824	0.0505	0.1345
Data Mining Based	0.0480	0.0523	0.1508	0.0606

To sum up, the AHP method and the related findings of the global priority vector from Table 3.13 indicate that **Spectrum Based are the preferred techniques**, followed by Mutation Based, Program Slicing and Data Mining Based.

### 3.2.5 Concept Definition

Concept definition is the final element of the NCD, providing the only exit to this framework (P. A. Koen et al. 2002). This is essentially the creation of the dissertation, therefore the primary objectives and concepts should be synthesized here.

That being the case, the main objectives of this project are describing fault localization techniques and its state of art (done previously in 2), choose a fault localization technique to integrate a tool from to an IDE (technique chosen in 3.2.4) and integrate the tool into an IDE. The tool and IDE will be presented in the Design chapter.

## 3.3 Value Analysis

In order to improve product creation or service delivery a lot of organizations use Value Analysis. This is a process aiming to identify and get rid of product and service features that don't add any value to the customer or the product but do imply costs to manufacture or provide the service (Rich, Holweg, and Wirtshaftsing 2000). Therefore, in this section concepts like Customer Value, Perceived Value, Benefits and Sacrifices will be explained in order to further understand Value Analysis and its connection with the customer.

### 3.3.1 Customer Value

According to Mahajan 2020, "customer value is the perception of what a product or service is worth to a customer versus the possible alternatives". This can be converted into the following equation:

$$CustomerValue = \frac{Benefits}{Costs}$$

The costs can be something related with the price or even other types of costs like time or effort. On the other hand, the benefits are the advantages of the said product or service.

In the context of this dissertation, which is a integration of a fault localization tool into an IDE, something worth to the customer is having a reliable, effective and easy to use way to use these tools on their favourite code editor.

### 3.3.2 Perceived Value

Perceived value can be described as the value of a product or service for the customer measured in their own perception of the said product/service's attractiveness (*What is Perceived Value?* 2020). It is known that a customer will be willing to pay more if they think the product is valuable to them, thus factors like the brand's name or reputation can affect the perceived value as well.

In this context, the perceived value of the customer should be up to the mark. Since the user will be able to use the same reliable tools available in their IDE for free (open-source) and without much complications.

### 3.3.3 Benefit vs Sacrifice

The benefit and sacrifice heavily depend on the type of value that the product or service provide to the customer. In Smith et al. 2007, it is presented four major types : functional/instrumental value, experiential/hedonic value (related with providing appropriate feelings and emotions), symbolic/expressive value (related with attaching the customer to the product or giving it some psychological meaning), and cost/sacrifice value (related with minimizing the costs of purchasing a product/service).

The integration done in this thesis falls under the functional/instrumental value, where it really matters how efficient, practical and useful the product/service is. The list below identifies the benefits and sacrifices for this dissertation:

#### Benefits

- **Practical** - Simple, practical, user-friendly interface.
- **Dependency Reduction** - Less reliant on a debugging tool.
- **Useful** - Excellent addition to the IDE's testing capabilities.
- **Free** - Available for free on the marketplace and to the open source community.

#### Sacrifices

- **Time/Effort** - Time and effort employed by the open source community.

## 3.4 Value Proposition

A Value Proposition is used to summarize the reasons why a customer should buy/use a product/service. If it is enthralling enough, it should persuade any customer that is pensive about the matter.

The Value Proposition Canvas was originally created by Dr Alexander Osterwalder as a framework to ensure that there is a suitable connection between the product and the market (Osterwalder et al. 2014). A similar canvas (see 3.14), Business Model Canvas, is used to demonstrate the value of the current dissertation.

Table 3.14: Business Model Canvas

<u>Key Partners</u> <ul style="list-style-type: none"><li>• Open Source community;</li><li>• Digital Libraries;</li><li>• IDE developers.</li></ul>	<u>Key Activities</u> <ul style="list-style-type: none"><li>• Identifying the best fault localization techniques and tools;</li><li>• Adapt and implement the technique/tool into an IDE.</li></ul>	<u>Value Propositions</u> <ul style="list-style-type: none"><li>• <b>Implementation of a fault localization tool/technique into a compatible IDE;</b></li><li>• Open Source solution;</li><li>• User Friendly;</li><li>• Reducing dependency on external tools.</li></ul>	<u>Customer Relationships</u> <ul style="list-style-type: none"><li>• <b>Online documentation:</b> All of the work will be documented adequately in online tools and in this dissertation;</li><li>• <b>Release Notes.</b></li></ul>	<u>Customer Segments</u> <ul style="list-style-type: none"><li>• <b>Independent Developers;</b></li><li>• <b>Students.</b></li></ul>
	<u>Key Resources</u> <ul style="list-style-type: none"><li>• <b>Available Fault Localization Tools;</b></li><li>• <b>Compatible IDE.</b></li></ul>		<u>Channels</u> <ul style="list-style-type: none"><li>• <b>Plugins Marketplace;</b></li><li>• <b>Email for technical support.</b></li></ul>	
<u>Cost Structure</u> <p>No monetary costs included.</p>			<u>Revenue Streams</u> <p>No monetary profit included.</p>	

## 3.5 Summary

Several value analysis topics were introduced and addressed throughout this chapter. The approach began with the innovation process, which was divided into three parts, one of which, the NCD, was primarily focused on. Each of the five key parts of the NCD (opportunity identification, opportunity analysis, idea production, idea selection, and concept definition) was examined and applied to this dissertation.

Nonetheless, among these five key parts, it is crucial to highlight the idea selection, which played a significant role in the thesis' enrollment. Using the AHP method to aid in the selection, **Spectrum Based Fault Localization** techniques were determined to be the winner with the highest composite priority.

Finally, ideas like as customer value, perceived value, benefit, and sacrifice were demonstrated in order to better understand the client and what value this dissertation may provide to him. In addition, a value proposition was developed to strengthen this relationship with the consumer and demonstrate why they should utilize the end product.



## Chapter 4

# Analysis and Design

This chapter outlines what was done prior to the start of implementation. The tool and IDE selection, the requirements, and some diagrams to help grasp the overall structure of the project.

### 4.1 Selection Process

In chapter 3, it was determined that the tool would have to use SFL approaches, as well as to be open source in order to be integrated. However, according to table 2.2 where it was included the tools discovered as well as other pertinent information, there were four SFL technique based tools that fulfilled this criteria:

- **Atheleia**
- **CharmFL**
- **FLACOCO**
- **GZoltar**

Choices had to be made, and there was a preference for Java-supported tools which narrowed the options down to **FLACOCO** (2.5.3) and **GZoltar** (2.5.5). However, due to **FLACOCO** not having any IDE integration, it was decided to integrate this tool.

The choice for IDE integration was between **IntelliJ** and **Visual Studio Code**. When looking at an index of the most popular IDE's (Carbounelle 2022), they are the only ones supporting Java and with a positive trend rise, indicating that they are both gaining traction over the past year. Yet, given to Microsoft's extensive and well-documented Extension API<sup>1</sup>, it was decided to incorporate **FLACOCO** into the IDE **Visual Studio Code**.

Nonetheless, if this choice failed, a **GZoltar** integration with the IntelliJ IDE would be the alternative.

### 4.2 Requirements

Understanding the project's functional and non-functional requirements is critical for establishing the use cases for the solution's implementation.

---

<sup>1</sup><https://code.visualstudio.com/api>

### 4.2.1 Functional Requirements

Functional requirements specify exactly what the system must perform and describe the features that the system must have. Hence, in this section, the requirements identified during the project led to the development of use cases:

- Read the project folder the developer wants to analyze;
- Execute the FLACOCO tool analysis in the project;
- Highlight the suspicious lines of code in the developer's project.

It was thus able to identify some use cases (UC) related to the integration of the tool FLACOCO in the IDE Visual Studio Code. This is illustrated in the use case diagram 4.1:

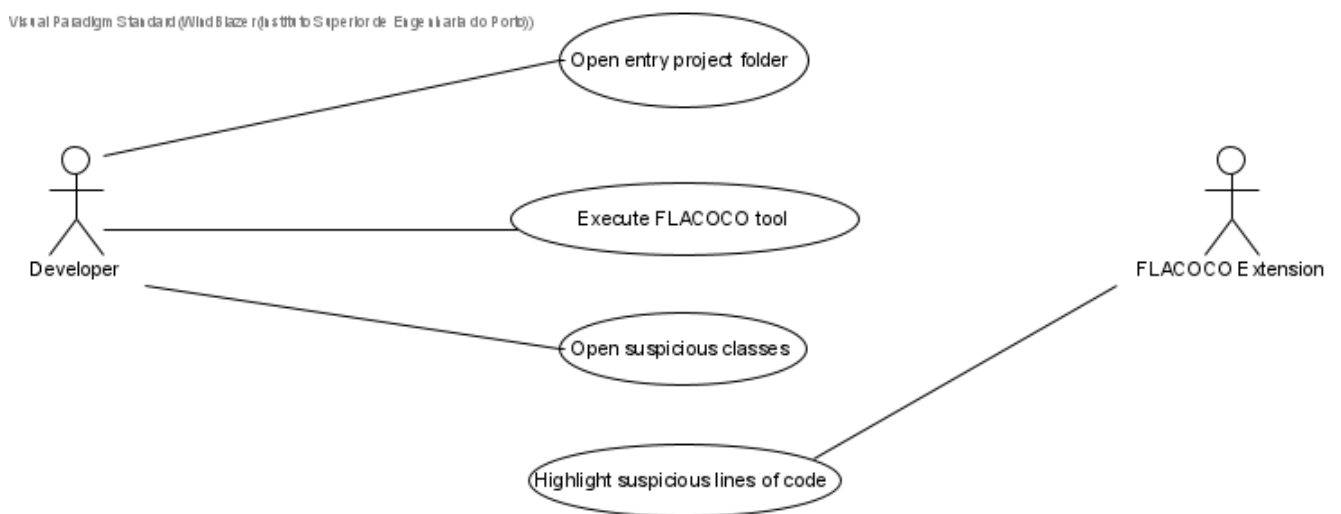


Figure 4.1: Use Case Diagram

#### UC1: Open entry project folder

##### Goal:

Display the opened folder to be analyzed in the Visual Studio Code FLACOCO Extension UI.

##### Actor:

Developer

##### Preconditions:

The project folder is in Java language.

##### Trigger:

The extension is triggered if the project folder is in the Java language.

##### Success Scenario

1. The developer opens the Visual Studio Code FLACOCO Extension UI.
2. The project name is shown successfully in the Visual Studio Code FLACOCO Extension UI under the opened folders.

**Fail Scenario**

1. The developer opens the Visual Studio Code FLACOCO Extension UI.
2. The project name is not shown in the Visual Studio Code FLACOCO Extension UI under the opened folders.
3. Visual Studio Code presents an error message.

**UC2: Execute FLACOCO tool****Goal:**

Run the FLACOCO tool to examine the specified project.

**Actor:**

Developer

**Preconditions:**

The UC1 (Read entry project folder) has to be completed successfully.

**Success Condition**

1. The developer presses the Run button in the Visual Studio Code FLACOCO Extension UI.
2. Visual Studio Code presents an information message indicating that the FLACOCO Extension is running.
3. Visual Studio Code presents an information message indicating that the FLACOCO Extension ran successfully.

**Fail Condition**

1. The developer presses the Run button in the Visual Studio Code FLACOCO Extension UI.
2. Visual Studio Code presents an error message.

**UC3: Open suspicious classes****Goal:**

Display the names of suspicious classes in the Visual Studio Code FLACOCO Extension UI so that the developer can easily be redirected by pressing them.

**Actor:**

Developer

**Preconditions:**

The UC2 (Execute FLACOCO tool) has to be completed successfully.

**Success Scenario**

1. The Visual Studio Code FLACOCO extension extracts the data from the Excel file created having the suspicious classes from running the FLACOCO tool.



2. The Visual Studio Code FLACOCO Extension uses the extracted data to display a list of the suspicious classes names in its UI.
3. The developer presses the desired class to check.
4. Visual Studio Code redirects the developer to the folder of the corresponding class.

### **Fail Scenarios**

#### **First**

1. The Visual Studio Code FLACOCO extension extracts the data from the Excel file created having the suspicious classes from running the FLACOCO tool.
2. There is no data.
3. Visual Studio Code presents an error message.

#### **Second**

1. The Visual Studio Code FLACOCO extension extracts the data from the Excel file created having the suspicious classes from running the FLACOCO tool.
2. The Visual Studio Code FLACOCO Extension uses the extracted data to display a list of the suspicious classes names in its UI.
3. The developer presses the desired class to check.
4. Visual Studio Code doesn't redirect the developer to the folder of the corresponding class.
5. Visual Studio Code presents an error message.

### **UC4: Highlight suspicious lines of code**

#### **Goal:**

Highlight suspicious lines of code in the project's source code based on their level of suspicion.

#### **Actor:**

System (FLACOCO Extension)

#### **Preconditions:**

The UC2 (Execute FLACOCO tool) has to be completed successfully.

#### **Success Condition**

1. The Visual Studio Code FLACOCO extension extracts the data from the Excel file created having the lines of code and its suspicion level from running the FLACOCO tool.
2. Visual Studio Code FLACOCO extension uses the received data to highlight the lines of code in the project source code files.

#### **Fail Condition**

1. The Visual Studio Code FLACOCO extension extracts the data from the Excel file having the lines of code and its suspicion level from running the FLACOCO tool.

2. There is no data.
3. Visual Studio Code presents an error message.

### 4.2.2 Non-Functional Requirements

Non-functional requirements are those that define how the system should operate. They can be thought of as system properties.

As a result, Table 4.1's objective is to describe each non-functional requirement employed in the project's thesis.

Table 4.1: Non-Functional Requirements

	Requirement
<b>Portability</b>	Specifies how a system can be launched in one environment or another.
<b>Usability</b>	Features linked to the aesthetics and consistency of the user interface are evaluated.

1. **Portability:** The application should be accessible for all operating systems supported by Visual Studio Code, this includes Windows, Linux, and MacOS;
2. **Usability:** The application is designed to be flexible, user-friendly, and information-rich.

## 4.3 Design

### 4.3.1 Architecture

In this section, the architecture of the generated solution will be revealed in order to describe the software. A general understanding of how software will work is the starting point for any successful endeavor. Thus, figure 4.2 was designed to achieve that purpose, while adhering to the recommended practices for a Visual Studio Code extension.

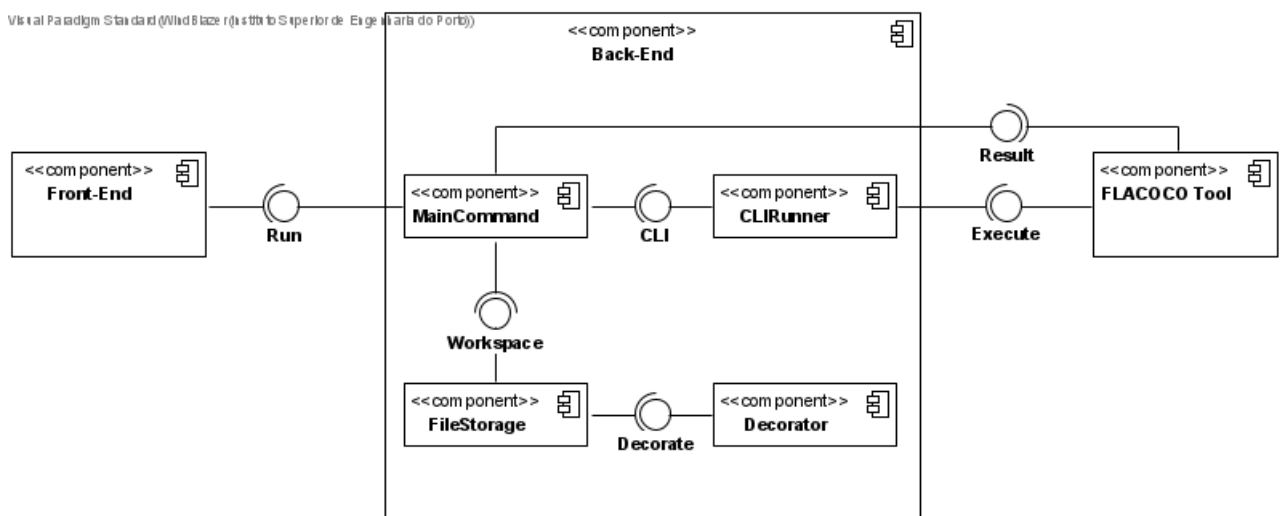


Figure 4.2: FLACOCO Component Diagram

Figure 4.2 depicts the three major components of the FLACOCO Visual Studio Extension.

Firstly, the **Front End** or **View** (following the Visual Studio Code UX guidelines<sup>2</sup>). This component serves as the user interface, presenting the user with all of the information processed in the back end as well as delivering all of the orders requested by the user to the back end.

Secondly, there is the **Back End**. In the Back End there are several components, therefore it will be given a brief sum of their purpose. The component `MainCommand` accepts the user requests and acts accordingly to its needs, if it is needed to interact with `FLACOCO` in order to get the analysis result it will request the `CLIRunner` component, otherwise if it is file related it will request the `FileStorage`. In order to highlight the suspicious lines of code in the project's source code it is used the `Decorator` component, which is linked to a file.

Finally, the **FLACOCO tool** does the analysis as directed by the `CLIRunner` and reports the results back to the `MainCommand`.

### 4.3.2 Deployment

Deployment is the process of making software available to the target customers in a specified environment (Williams 2021). A deployment diagram was constructed to aid the process, this diagram can therefore be found in Figure 4.3 .

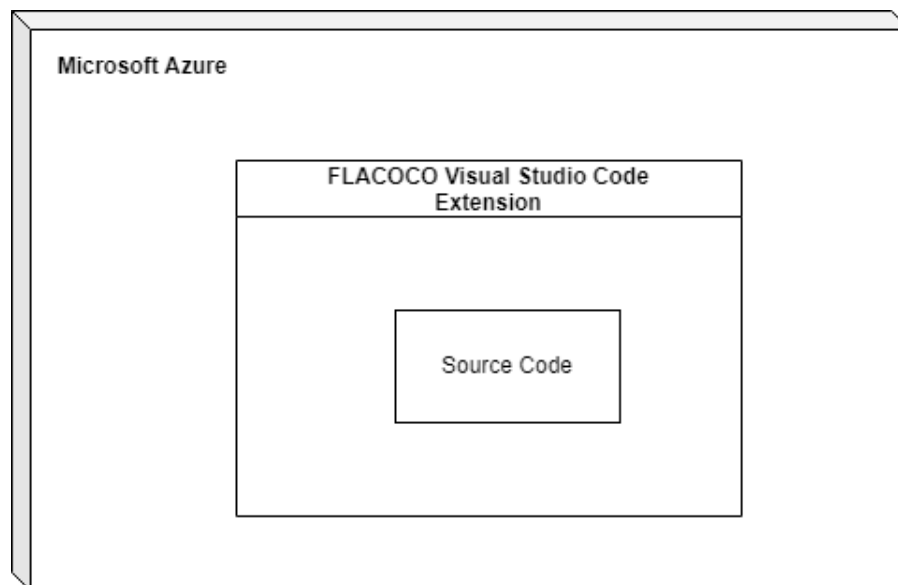


Figure 4.3: FLACOCO Deployment

As can be seen in the Figure 4.3, it was adopted the method of deploying the software in Microsoft Azure. Since Microsoft owns Visual Studio Code as well as **Azure**, a well-known cloud deployment provider, it offers excellent support and compatibility for deploying your Visual Studio Code extensions, therefore it only made sense to do so.

Chapter 5 will go through how this deployment was carried out in further detail.

<sup>2</sup><https://code.visualstudio.com/api/ux-guidelines/views>

## 4.4 Summary

There were a few noteworthy points in this chapter. To begin, FLACOCO was chosen as the automatic fault localization tool and Visual Studio Code as the IDE to integrate it on.

Moreover, four use cases were created to meet the functional requirements: open entry project folder, execute FLACOCO tool, open suspicious classes and highlight suspicious lines of code. Additionally, two non-functional requirements were imposed: the extension must be usable in all VSCode-supported operating systems and be user-friendly.

Finally, a component and deployment diagram were used to describe the extension's architecture.



## Chapter 5

# Development

This chapter will discuss the software development process, therefore it will first detail how Visual Studio Code extensions function, followed by an explanation of how each use case mentioned in Chapter 4 was implemented. Finally, any other pertinent information about the development will be shared.

### 5.1 Visual Studio Code Extensions

Visual Studio Code is a source code editor developed by Microsoft for Windows, Linux, and macOS. It is noted for being a lightweight and fast IDE, but it lacks some of the built-in features that other IDEs offer. Nonetheless, Visual Studio Code excels in having a large number of extensions in its marketplace and a highly extendable API, with practically every aspect of VS Code being extremely customizable.

#### 5.1.1 Creating an Extension

Some prerequisites must be completed before creating the extension. The device must have Node.js, Yeoman, and the VS Code Extension Generator installed.

Yeoman is a web scaffolding tool for web applications, and as such it can provide a generator ecosystem. Being available in the package manager Node, through a simple Node command it can be created a VS Code Extension Generator walking the user through the steps required to create an extension:

```
npm install -g yo generator-code
```

After executing the generator, it will be prompted with some customization questions, the most important of which is whether the extension is written in TypeScript or JavaScript and whether the package management used is Yarn or Node.js. FLACOCO Visual Studio Code Extension was written in Typescript and used the node package manager.

Following that, a base folder structure with a simple extension implementation will be built.

#### 5.1.2 Extension Structure

There are some files created by the generator, however, the primary files that are required to comprehend the extension are package.json (**Extension Manifest**) and extension.ts (**Entry File**).

## Extension Manifest

The Extension Manifest for each VS Code extension must be package.json. The package.json file has a mix of Node.js fields like scripts and devDependencies as well as VS Code fields (Microsoft 2022). Some of the most significant VS Code fields are as follows:

- **Name** and **Publisher**: To identify an extension, VS Code uses <publisher>.<name> as an unique ID;
- **Main**: Extension's entry point;
- **Activation Events**: set of JSON statements that, if they occur, activate your extension. The presence of a file in the folder of a specific programming language, for example, can be activated by the onLanguage Activation Event (e.g., "onLanguage:java").
- **Contribution Points**: a set of JSON declarations that extend the functionality of Visual Studio Code. For example, contributing the UI with a command, define when the command should be enabled or even the condition under which specific items should show during that command.

## Entry File

Two functions are exported by the extension entry file: **activate** and **deactivate**.

When your registered Activation Event occurs, activate is executed. The **activate** function will include the core logic flow and code for your extension; here you will register your commands and control any section of the extension using the VS Code API.

**Deactivate** function allows you to clean up before your extension is deactivated. Many extensions may not require specific cleanup, and the deactivate function may be omitted. This is the approach to use if an extension wants to conduct an operation while VS Code is closing down or the extension is disabled or removed (Microsoft 2022).

## UX Guidelines

Figure 5.1 was created in order to completely comprehend some of the expressions that will be utilized throughout the development explanation and what they represent in the Visual Studio UI.

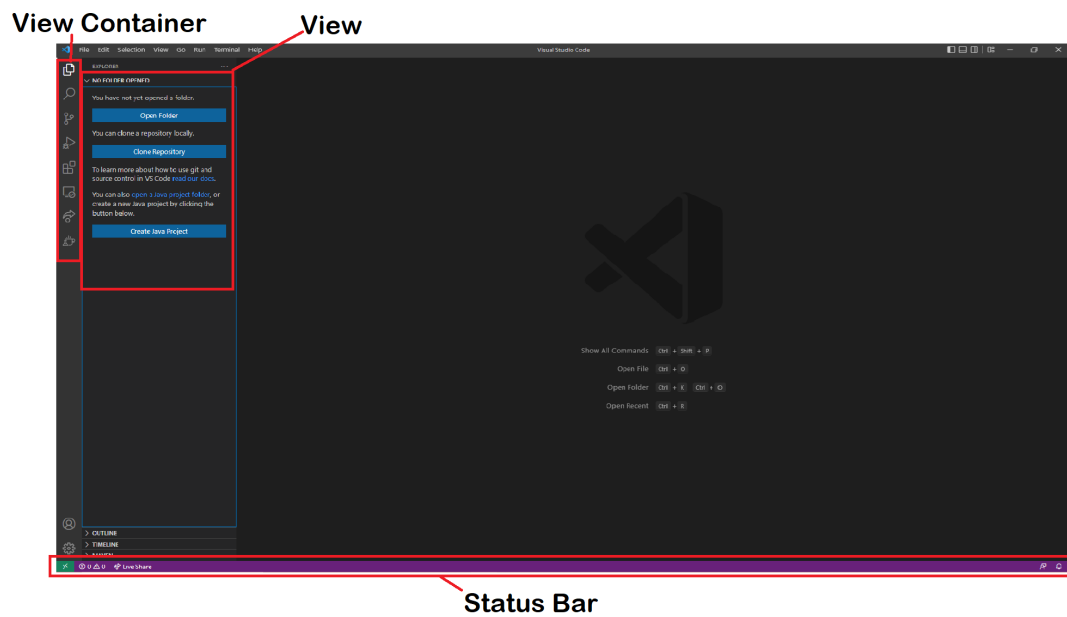


Figure 5.1: Visual Studio Code UX Guidelines (Microsoft 2022)

- **View Container:** here it will be present the icon of the extension developed;
- **View:** content containers that can show in the sidebar or panel. Views can contain tree views or custom views, as well as view actions.
- **Status Bar:** located at the bottom of the VS Code workbench and displays workspace-related information and actions.

## 5.2 Use Case Development

The four use cases will be presented in this section in order to demonstrate all the features and sub-functionalities of the software. The following approach was used to document the respective use cases: first, a description of the UC will be provided, indicating what it does and who executes it, then figures referencing the result will be made presented, and finally, it will be explained how some relevant parts of its coding were implemented, with the possibility of inserting relevant code snippets.

### 5.2.1 UC1 - Open entry project folder

This use case occurs when the developer opens a Java project folder to be analyzed and the folder name is displayed in the View menu when using the extension (5.2).



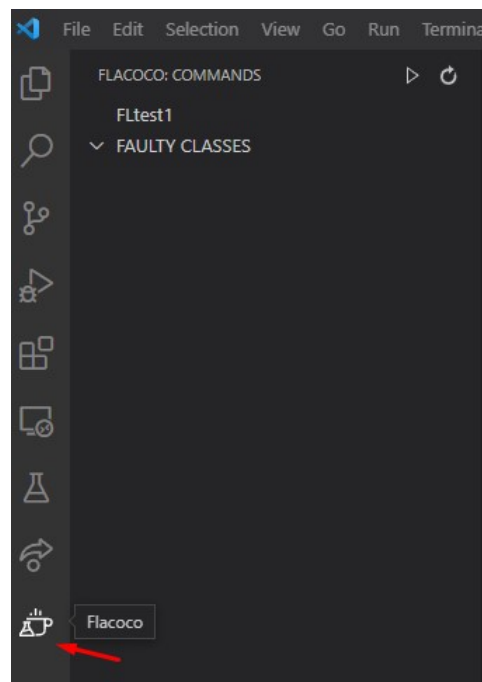


Figure 5.2: Flacoco Extension View

Furthermore, some triggers were implemented for this use case. By creating the "on-Language:java" Activation Event in the package.json (5.1.2), it was possible to make the extension icon visible in the View Container if the developer opened a project folder that uses the Java language.

Visual Studio Code requires data to display in the newly opened view after opening the extension. For this, the Tree View API was used, which allows the content to be shown as a tree. First, a TreeDataProvider was created in order to provide data to the view, which must extend from the TreeItem class. These are our elements in the view; they include attributes such as labels and a collapsible state that indicates whether or not they have children (meaning if they can be opened or not). Figure 5.2 shows two TreeItems that were implemented: the first with the label as the folder name and no children; the second with "FAULTY CLASSES" as the label and the possibility to have children.

Nonetheless, on the Contribution Points field of the Extension Manifest, the "FLACOCO:COMMANDS" label, as well as the refresh and run buttons, are defined. The label, as well as the image linked to specific commands, can be defined there (like run and refresh).

Finally, it is important to note that a ".flacoco" folder is created in the root of the user's project, where the tool's jar is also moved, and an empty CSV is created there to receive future data from the tool analysis.

### 5.2.2 UC2 - Execute FLACOCO tool

This use case occurs when the developer launches the extension analysis by clicking the Run icon in the View Toolbar or by hovering over the folder name and pressing the Run icon, as shown in Figure (5.3).

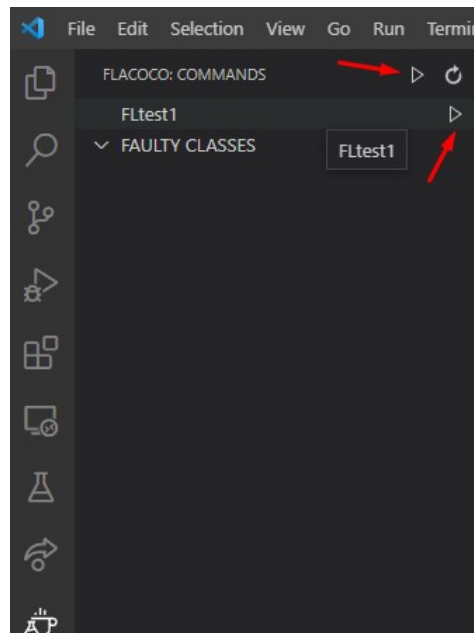


Figure 5.3: Flacoco Extension Run

In terms of code, the command Run was linked to a function that will define its logic. In that function, it calls a class that constructs the FLACOCO tool command from formatted strings and then executes it to get the tool's output into the previously prepared empty CSV.

Furthermore, after the run has begun, it is critical to keep the user informed of what the extension is doing. As a result, the Extension API was utilized to generate a Status Bar and an Information Message popup, as seen in Figure 5.4. This was accomplished by utilizing the `vscode.window.createStatusBarItem` (where the text and position can be specified) and `vscode.window.showInformationMessage` functions (where it can be defined the text present in the message). This way, even if long background work is being performed, users are aware that the extension is still operational, despite the inability to display results immediately.

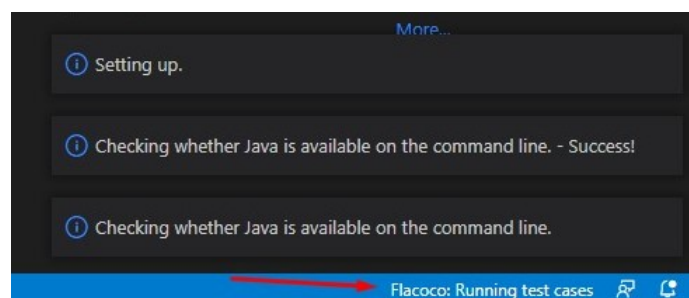


Figure 5.4: Flacoco Status Bar and Information Message

### 5.2.3 UC3 - Open suspicious classes

This use case happens when the extension displays the names of the defective classes so that the user may easily be redirected to the class source file by simply clicking them, as presented in Figure 5.5 .

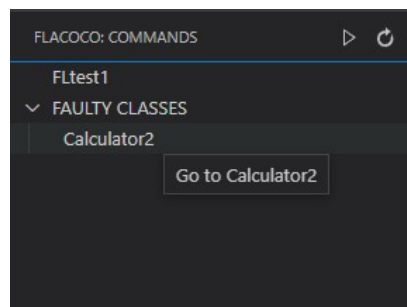


Figure 5.5: Flacoco Faulty Classes Tree

Coding-wise, it is necessary to first read the tool's output, which was saved as a CSV (5.2.2), but in order to parse that data, it is necessary to understand how the data was organized. That being said, the data in the CSV is ordered as follows: first, the Java class name, then the number of the suspicious code line, and last, the suspicious level, which can range between 0 and 1.

Only the first piece of data, the class name, is required for this use case. Using a standard Typescript split, the first piece is sent to a function that searches for the full path of the desired class and creates an object with the class name and full path. If this object is not repeated, it will be added to an array containing all of the faulty classes.

```

1  private static async addToFaulty(folder: string) {
2      let faultyClass: TreeFolder;
3      let label: string;
4      let glob: string;
5
6      let folderToFind = folder.replace(/\.\/g, '/') + '.java';
7      label = folder.split('.').pop()!;
8
9      glob = '**/*${folderToFind}';
10
11     await vscode.workspace
12         .findFiles(glob, '/node_modules/', 1)
13         .then((uris: vscode.Uri[]) => {
14             uris.forEach((uri: vscode.Uri) => {
15                 faultyClass = new TreeFolder(label, uri.fsPath);
16                 if (
17                     !Decorator.faultyClasses.some((f) => f.path ===
18                     faultyClass.path)
19                 ) {
20                     Decorator.faultyClasses.push(faultyClass);
21                 }
22             });
23         });
24 }

```

Listing 5.1: Implementation of the method addToFaulty

Looking at this code, the main Extension API function that should be focused on is the `vscode.workspace.findFiles` function. Three arguments were used: include pattern, exclude

pattern, and number of maximum results. In the first, there is a pattern to search the full path of a file ending in the class name plus the ".java" since it is a java file, in the second, there is no need to search in the node modules folder, which would consume a lot of the extension response time, and finally, only one match should be possible.

Lastly, the array containing all of the faulty classes discovered during the analysis will be utilized to populate the children of the TreeItem labeled "FAULTY CLASSES" described in 5.2.1. This will be linked with the `vscode.open` command from the Extension API, with the entire path of the faulty class as an argument, so that the user can click the class name and be redirected to the source file.

#### 5.2.4 UC4 - Highlight suspicious lines of code

This use case is triggered when the user switches to a different text editor, either by being redirected on pressing the class name in UC3 (5.2.3) or by manually going through each class file. Then, in each file, the suspicious lines of code are highlighted as shown in Figure 5.6.

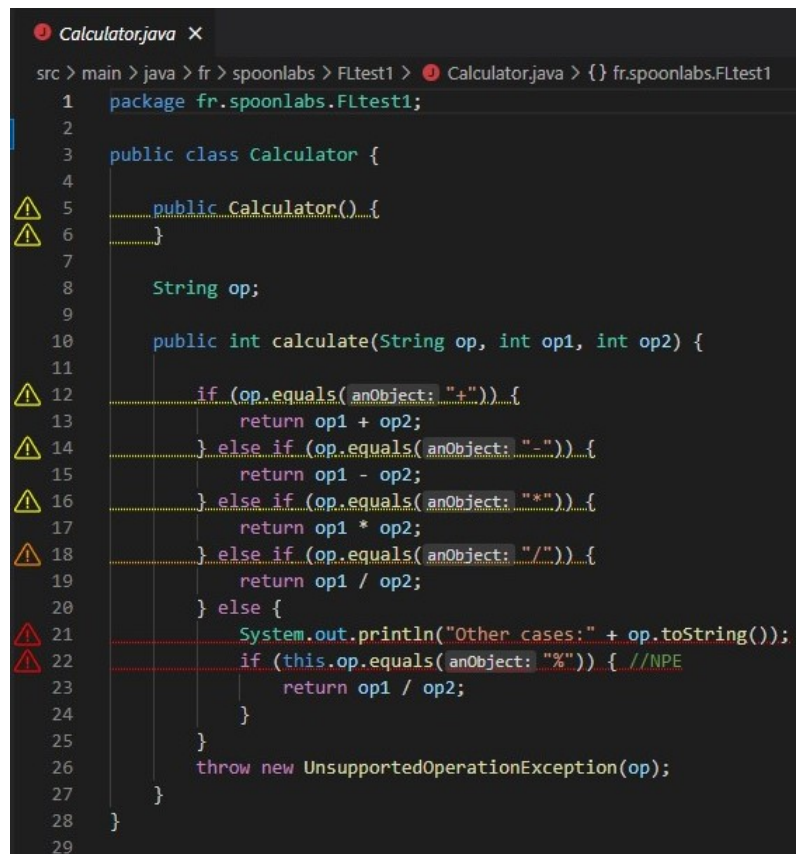


Figure 5.6: Flacoco Highlight Example

In terms of coding implementation, using the tool's output, which included the class file name, number of suspicious code lines, and suspicion level, it was possible to associate an image of a specific color with each line and its level. **Green** denoted a **low** level of suspicion, **yellow** a **medium** level, **orange** a **high** level, and **red** a **very high** level. An extra step was taken here by adding a `hoverMessage` property so that when the user hovers over a specific

line, a text indicating its level appears (Figure 5.7). This measure can assist colorblind people in using the extension.

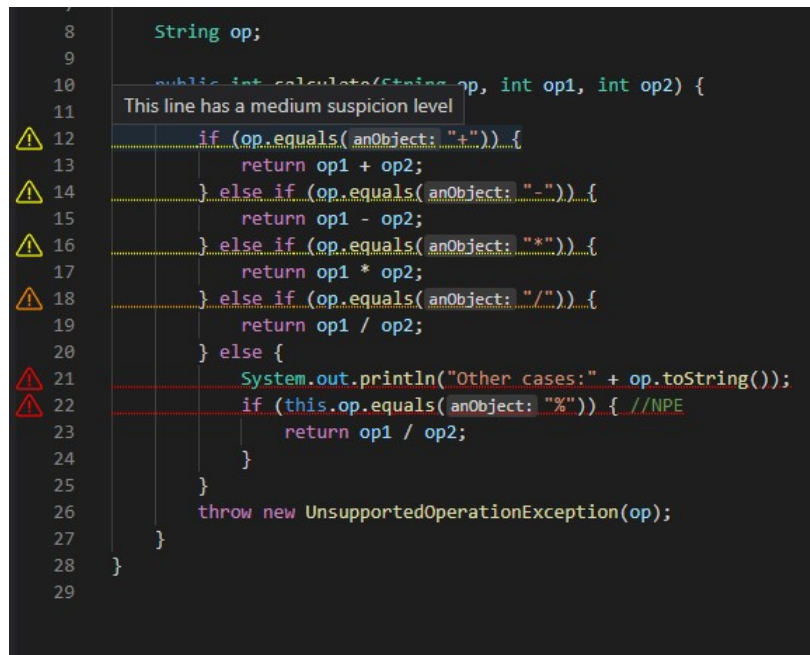


Figure 5.7: Flacoco Hover Example

The image and type of highlight were assigned to the code lines using the Extension API method *createTextEditorDecorationType*. This decoration would be triggered whenever the user switched to that text editor via the *onDidChangeActiveTextEditor* event, but would be removed if the user saved the document via the Extension API's *onWillSaveTextDocument* event.

## 5.3 Other FLACOCO Extension Details

### 5.3.1 Logo

Visual Studio Code presents an icon of each active extension in its View Container, therefore, an icon was needed.

The logo needed to reflect the tool's identity, it is a tool to help the debugging task using the tests in Java projects. Thus, the idea was to combine the Java logo with a testing logo, as illustrated in Figure 5.8.



Figure 5.8: Flacoco Visual Studio Extension Icon

### 5.3.2 README

Along with the extension's implementation, a README file was created to serve as a guide for any new users of the extension. This README can be found here<sup>1</sup>. Its contents included a brief overview of the tool and its goals, as well as the prerequisites for using it, instructions on how to set it up and use it. The guide attempted to include a lot of pictures to make it easy to follow, as well as release notes to keep the user up to date on its contents.

### 5.3.3 Deploy

After finishing the extension, the next step was to submit it to the VS Code Extension Marketplace so that others could find, download, and use it. To do so, first install the library vsce, which stands for "Visual Studio Code Extensions" and is a command-line tool for packaging, publishing, and managing VS Code extensions. This can be accomplished with the following command:

```
npm install -g vsce
```

After completing the first step, it was necessary to establish an organization. This was possible by filling out some basic information on the Azure Dev Ops page<sup>2</sup>. The Personal Access Token must then be obtained. Figure 5.9 shows how to do this on the top right of the page.

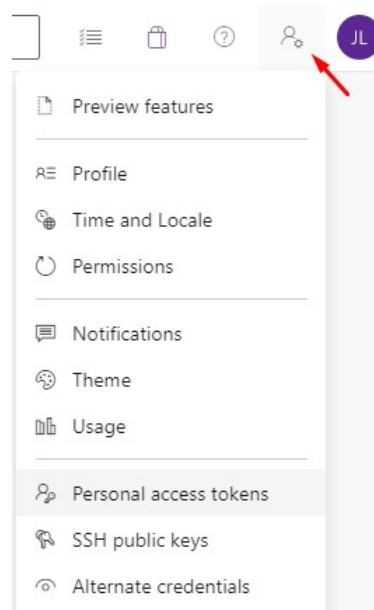


Figure 5.9: VSCode Personal Access Token

It is also necessary to create a publisher. It is possible to create one by filling out the name and identifier for the created extension on the marketplace publisher management page<sup>3</sup>. After completing all of the preceding steps, go to the root directory and execute the following commands:

<sup>1</sup><https://marketplace.visualstudio.com/items?itemName=JoaoLeao.flacoco>

<sup>2</sup><https://dev.azure.com>

<sup>3</sup><https://marketplace.visualstudio.com/manage/publishers>

```
vsce login <publisher identifier>
```

The Personal Access Token obtained previously will be required after using this command.

```
vsce publish
```

Finally, once all preceding procedures have been performed successfully, the extension will go live upon approval from the Visual Studio Marketplace.

## 5.4 Summary

This chapter went into greater detail on how the extension was created.

It began by introducing the basic ideas of Visual Studio Code Extensions, such as its UX Guidelines and anatomy. Then, for each use case, it was provided a detailed description of its purpose and how it was implemented, along with an image to illustrate it.

Finally, the logo, the extension guide provided in the README and how it was deployed into the Visual Studio Marketplace were demonstrated.

## Chapter 6

# Evaluation and Experimentation

The solution evaluation and results discussion are presented in this chapter. It begins with the presentation of the research question, the hypothesis outlining, as well as the metrics by which the findings will be measured and how the user study was performed. This refers to step 5 from the DSRM approach used in this dissertation.

### 6.1 Goal and Research Question

As previously stated in this thesis, the final goal is to integrate a fault localization tool into an IDE. Therefore, the following research question was developed:

**RQ1:** Is FLACOCO Visual Studio Code Extension a valuable tool for developers?

Thereafter, the Visual Studio Code Extension FLACOCO will be referred just as "extension".

### 6.2 Hypothesis

Two hypotheses were derived from the Research Question. They have been defined in order to assess the success of the integration. The Null hypothesis  $H0$  and the alternative hypothesis  $H1$ . The Null hypothesis of a test always predicts failure, while the alternative hypothesis predicts success:

- $H0$ : The extension was not deemed valuable by the end user.
- $H1$ : The extension was deemed valuable by the end user.

### 6.3 Study Planning

In order to validate the hypothesis, an user study was performed. This user study has two parts that complement each other: **Test Scenario** and **Questionnaire Application**. The success of it will decide on whether the Null hypothesis is refuted or not.

#### 6.3.1 Test Scenario

The goal of this test scenario is to instruct the users in how to perform tasks using the extension on a daily basis, whether academically or professionally. In the Test Scenario the participants will have to perform two debugging tasks using the extension's current version



in the IDE Visual Studio Code. The time that each user takes to finish the individual tasks will be recorded.

### 6.3.2 Evaluation Indicators

The concept of value for the end user, on the other hand, is tremendously subjective (Tanner 2022). Subsequently, the extension should be evaluated by its usability and usefulness, both of which should be major reasons of why someone would consider something valuable (L. F. d. Silva and Oliveira 2020). With that in mind, the hypothesis will be evaluated using the following indicators:

- **Real-life Usefulness:** It is necessary to assess if the integration mitigates real-life issues for the user;
- **User's Usability:** User usability for the project in question is a crucial issue to consider, as it is a support tool to improve the developer's quality of life, and their feedback will condition the project's development.

### 6.3.3 Participants

Initially, it was necessary to choose participants for the Test Scenario. The use of the tool requires prior Java knowledge. As a result, only developers with previous experience with the Java language (being it professionally or academic) could be invited.

Nonetheless, the study should have a minimum number of participants in order to proceed. According to a research made by Virzi 1992, there is a relation between the number of participants in an evaluation session and the percentage of usability problems detected.

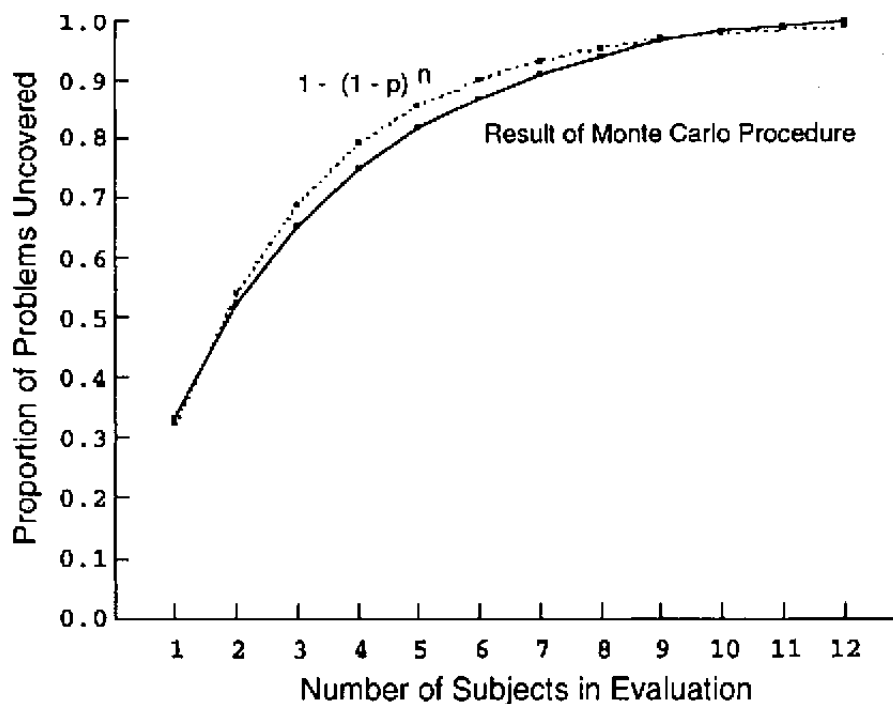


Figure 6.1: Relation between the number of subjects and number of usability problems detected

According to Figure 6.1, Virzi found that with just three participants the percentage of detected problems would be around 65%; with nine, 95% of the problems are detected. As a result, the minimum number of participants for the study should be nine, since its percentage of problems detected is close to 100%. Naturally, as more people test the program, the likelihood of a new error being discovered decreases.

For convenience, the participants would be invited from developer's Discord<sup>1</sup> communities as well as students currently attending the ISEP's Masters Degree in Software Engineer.

### 6.3.4 Questionnaire

A questionnaire was developed to be presented to the Java developers that will participate in the study. The questionnaire was designed recurring to the Google Forms platform. The responses supplied by these developers are critical in determining whether or not the extension achieved meaningful and valuable consequences. Furthermore, it is particularly useful to establish whether the questionnaire touches on the relevant matters required to assess the validation of the integration.

The questionnaire contained two groups of questions:

- **Demographic:** Section with questions designed to elicit information about the respondent's history and experience. This is critical for forming connections with the responses provided.
- **User Experience:** Section including questions meant to elicit information on the participant's experience with the extension. It is designed to retrieve information about the two indicators specified in 6.3.2. Lastly, it offers a choice to give any additional feedback.

Moreover, this questionnaire will use a symmetric Likert Scale (presented in Table 6.1) in the User Experience section. This means that the position of neutrality (neutral/don't know) is located exactly halfway between the two extremes of strongly disagree (SD) and strongly agree (SA), and it allows a participant to choose any response in either direction in a balanced and symmetric manner (Joshi et al. 2015).

Table 6.1: Likert Scale

Description
Strongly Disagree
Disagree
Neutral
Agree
Strongly Agree

Lastly, Table 6.2 presents the questions in the User Experience section of the questionnaire which are the ones applying the Likert scale. It is also shown which indicator they pretend to evaluate. Nonetheless, the full questionnaire is included in the Appendix A.

<sup>1</sup><https://discord.com/>

Table 6.2: Questionnaire's User Experience Section

	Question	Indicator
1	The readme file clearly and objectively describes how to utilize the extension.	Usability
2	It was easy to start using the extension.	Usability
3	The extension interface presented information in a structured manner.	Usability
4	The information in the extension interface is presented in a clear manner.	Usability
5	The extension responds quickly.	Usability
6	The extension played an important role in locating the bug.	Usefulness
7	The extension is a positive addition to a developers' daily life	Usefulness
8	I would recommend the extension to other developers.	Usefulness

### 6.3.5 Hypothesis Evaluation

In Likert scales, the response categories have a rank order, but the intervals between values cannot be assumed to be equal. Thus, for ordinal data, the mean is incorrect and a statistic like **mode** would be advised (Jamieson 2005). Therefore, using the mode and the values in the Likert Scale, it is reasonable to conclude that the extension outcome is positive if the calculated mode is greater than Neutral, which would mean that it would be Agree or Strongly Agree. Based on such, the following statements can be made:

- **Usability** indicator is valid if the mode for these questions is **Agree** or **Strongly Agree** ( Mo > Neutral ).
- **Usefulness** indicator is valid if the mode for these questions is **Agree** or **Strongly Agree** ( Mo > Neutral ).

If both indicators are valid, it can be stated that the extension is deemed valuable for the developers.

## 6.4 Preparation

Following the planning, the preparation for the user study began. As a result, an extension and session guide had to be created, as well as faulty projects prepared for the participants to test the extension on.

The extension guide had previously been created while implementing the extension in form of a README file. This file was slightly adjusted in order to clarify some nuances that might have been less obvious.

Then, one or more faulty projects should be provided to the participants in order for them to test the extension. After some research, it was deemed necessary to have two projects: one being a sizable open source project and the other an academic project. Correspondingly, the Joda-Time<sup>2</sup> open source project, which is a replacement to the standard date and time

<sup>2</sup><https://www.joda.org/joda-time/>

library for Java and an academic project developed during the Bachelor's degree named Graph<sup>3</sup> were chosen. Afterwards, two bugs were introduced, one in each project, and it was ensured that the project could compile but that it would lead to unexpected behaviour during execution causing some test cases to fail. In the Joda-Time project, the line 435 of the *Hours* class was changed to an equal operation to 0 instead of 1 and in the Graph project, line 65 of the *RoutImpl* class was changed the equal operator "==" to a not equal "!=".

Subsequently, the session guide was created. This included a personal introduction, a presentation of the topic and the study goals, and then an explanation of how the procedure would work. In this procedure, it would be sent an invitation link to a video conference in Discord for the developers from the Discord communities and a Zoom link for the students attending the ISEP's Masters Degree in Software Engineer. Then, it would be ensured that the participants had Visual Studio Code, Java and Maven (due to the projects) installed in their machines. Following that, a link to the extension's installation and usage guide, as well as a link to the faulty projects they would use, would be provided. Nonetheless, the task at hand would be explained to them. This task consisted of localizing the two bugs, and in order to complete it, the developer had to correctly pinpoint the bug. When the activity was completed, the developer was required to send the number of the faulty line as well as its fix via email. The time taken by each developer to accomplish the task was likewise tracked this way. Finally, they would be given a 25 minute limit for each task. This was a rough estimate of the maximum time in order to not worry the participants about its content, the estimate was also based on the evaluations performed on GZoltar (Brito 2020 and Campos et al. 2012a).

Before beginning the test, any concerns that participants had would be addressed. However, no instructions on how to use the extension would be provided because the quality of the README file as a guide required to be tested. Whenever everyone was ready, they would start at the same time and in the same project. After the 25 minute limit, the other project test would begin with everyone at the same time again.

## 6.5 Results and Discussion

The study was able to enlist the participation of thirteen people, this included eleven working software engineers and two students currently doing their masters degree. The majority of the working software engineers were new to the field, with 1 to 2 years of experience (46,2%); however, some experienced developers participated as well, with experience of more than 5 years (15,4%). The large majority of participants (76,9%) were already familiar with Visual Studio Code, and Automatic Fault Localization tools were majorly unfamiliar amongst the participants, with only 53,8 percent having never worked with any of them. Finally, nine of these participants used Windows, one Linux, and three MacOS as their operating system. All of this information is shown in the graphs in the Appendix A.1.2.

All of the participants were able to identify and correct the problem in the 25 minutes time frame given. The average time to complete the task for the Joda-Time project was around 11 minutes (662 seconds), with the lowest time being 178 seconds and the highest time being 901 seconds. On the other hand, for the Graph project, the average time to complete the task was around 7 minutes (420 seconds), with the lowest time being 114

<sup>3</sup><https://github.com/JoaoLeao7/graph>

seconds and the highest time being 897 seconds. All the time data gathered is available in the Appendix A.1.

As previously mentioned, after completing the task the users were invited to answer a survey related to their experience using the extension. The questions addressed the usability and interface of the extension in a Likert Scale (6.3.4).

Firstly, regarding the quality of the README created as a guide, all of the users agreed that it achieved its purpose by clearly and objectively describing how to use the extension, having a mode of Agree (frequency of 7 users) to this statement.

The majority of the users also agreed on the extension being easy to pick on, having a mode of Strongly Agree (frequency of 7 users). It is impressive that only one user was Neutral to this question since three users claimed that they had no previous experience with the IDE Visual Studio Code.

Everyone agreed that the interface was well structured, presenting the information accordingly. However, when asking if the interface presented information in a clear manner; two users preferred to be neutral about it. It is possible that some users were overwhelmed by the highlight system presented in the extension or that they would prefer some extra feature to aid it. Nevertheless, the mode for both of these questions were Agree (frequency of 7 users on both).

Following up on the extension performance, whether it had a quick response or not, even though the majority still agreed on the extension having a quick and smooth performance, there was a huge minority that were Neutral about it. Therefore, in this question, the data set gathered is bimodal. This signifies that no single data value occurs with the greatest frequency. Instead, there are two data values that have the same frequency (Taylor 2020). These two data values were Agree and Neutral with a frequency of 5 users. In this case, the perception of how quickly something responds can be subjective, and the results reflect this. Even though most users thought the response time was adequate for the size of the test projects, some would prefer it to be faster. Nonetheless, this is a factor to consider; it will be attempted in the future to improve the extension's performance.

Moving on, most of the users (frequency of 12) believed the extension was critical in locating the bug, with a bimodal data set consisting of Agree and Strongly Agree with 6 users each. Furthermore, the majority agreed that the extension was a useful addition to their toolkit and that they would recommend it to other developers, both with a mode of Agree (frequency of 8 users on both).

The survey also included an open question for users to indicate any issues or suggestions they had during the session, which resulted in valuable feedback being gathered here. Some suggested that the code highlight should be shown in the VSCode Minimap as well, in order to provide a better overview of the source file that the user is inspecting. It was also suggested to not remove the highlights automatically after saving the file, but rather left up to the user with a toggle button option. Another useful suggestion was to be able to set up a keybinding to advance to the next severity level (first red highlights, then orange, and so on). Lastly, some users expressed a desire for this type of extension in the IntelliJ IDE.

Overall it was a very positive experience, all of the results to the questionnaire and the questionnaire itself is available in the Appendix A. Furthermore, in order to determine whether the indicators explained in 6.3.5 were valid, the mode for the usability and usefulness questions must be checked.

- **Usefulness:** The mode for this category was **Agree** with an absolute frequency of 22.
- **Usability:** The mode for this category was **Agree** with an absolute frequency of 31.

With these results in mind, it is possible to confirm that the Null Hypothesis (The end user does not value the extension, which is the integration of a fault localization tool into an IDE.") was refuted since both categories had **modes superior to Neutral**.

Fianlly, it is possible to state that the **FLACOCO Visual Studio Extension is a valuable tool for developers**.

## 6.6 Limitations

Despite the fact that the outcome was overwhelmingly positive, there were some setbacks in gathering User Feedback.

Some issues were unavoidable due to the online nature of the video conferences. Because users were required to test the extension on their computers using the projects provided to them, some encountered compilation issues that were resolved but caused delays in the video conferences.

Nonetheless, two users reported an error with the extension that caused it to stop working while analyzing the code; however, this was fixed without knowing the underlying cause.

Lastly, given the nature of the data and the limited number of participants, the result samples could not be generalized. However, it was possible to infer that the extension was a success amongst the participants who performed the test and answered the questionnaire.

## 6.7 Summary

The steps for evaluating the extension were presented in this chapter.

The research question was presented first. This was the primary premise of the hypothesis for which evaluation indicators were developed. A user study was then planned to validate these indicators. This included creating an extension guide, a plan for each video conference session, gathering participants, preparing test projects, and creating a questionnaire.

The user study was a success, despite some setbacks. All users were able to identify the bugs with the extension and were pleased with how the tool worked. This was reflected in the questionnaire results, which allowed the stated Null Hypothesis to be refuted, indicating that the FLACOCO extension was proven to be useful to developers.



## Chapter 7

# Conclusion

This chapter finishes the document by reviewing the work completed and providing some final thoughts. First, the initially established objectives are examined to determine whether they have been fully addressed. Followed by a description of the obstacles encountered and how they were overcome. Future work is also mentioned, as are some final considerations regarding the work done.

### 7.1 Achieved Goals

This section discusses the set goals for this thesis and whether they were completed or not. Table 7.1 displays their descriptions as well as whether or not they were completed.

Table 7.1: Thesis' Objectives

	Objective	Completed
<b>1</b>	Identify and describe existing automatic fault location techniques and tools.	<b>Yes</b>
<b>2</b>	Identify the possible tools, and/or techniques, that show good results and can be included in an Integrated Development Environment (IDE).	<b>Yes</b>
<b>3</b>	Integrate an existing tool into a chosen IDE. Integration may require adapting, or improving, this tool.	<b>Yes</b>

Thus, in addition to accomplishing all of the objectives put forth previously, this thesis offered several contributions to the automatic fault localization field:

1. For starters, it presented an in-depth examination of current automatic fault localization tools and techniques. It went through their concepts with examples and even a comparison.
2. Second, this thesis contributed with the design and implementation of the integration of the FLACOCO tool into Visual Studio Code, complementing the study of A. Silva et al. 2021 where the tool is described and is evaluated its effectiveness.
3. Third, it also contributed with an evaluation of the extension through an user study. The study was divided into two parts: one for participants to gain firsthand experience with the extension and the second for them to rate their experience. The results to the questionnaire were positive, deeming the extension useful to the developers.



## 7.2 Limitations

While developing this document, several challenges were faced. It is reasonable to state that the following constraints influenced the research and implementation of the thesis:

- **Information Limitations:** Due to the niche nature of Automatic Fault Localization, gathering information about the various existing techniques, comparing families, and tools was a difficult task. Furthermore, finding recently updated open-source automatic fault localization tools or papers about them is difficult.
- **User Study Limitations:** Gathering participants for the user study was not an easy task; they did not need to be experienced with Visual Studio Code, but they did need to be Java developers and be willing to do what could be a 1 hour session to complete the tasks and answer the questionnaire. Nonetheless, because everything had to be done online, attempting to work around everyone's schedule resulted in the creation of several video conference sessions along the way.

## 7.3 Future Work

Even though this thesis successfully achieved all the initially defined objectives, there are always improvements that can be made to enrich the outcomes produced. The responses to the questionnaires provided some insight into what aspects of the extension could be improved or added in the future. They are listed as follows:

- **Extension Performance:** Some users voiced concern about the extension response time as the project size grew exponentially larger. This should be reviewed to determine what is achievable in terms of performance.
- **VSCode Minimap Highlight:** The extension should not only highlight the source code file, but it should also be visible in the VSCode Minimap so that users may get a better overview of that particular file.
- **User Customization Improvements:** Some user options customization, such as whether or not the highlight should fade after saving, should be added. As well as keybinding configuration choices, such as skipping from red to orange warnings.

## 7.4 Final Remarks

The task planning was completed, as were all of the primary functionalities; as a result, the author believes that it was a successful project.

Since the author was unfamiliar with the topic, the project was considered a challenge from the start. However, the overall development of this thesis was very positive, owing to the valuable knowledge acquired in automatic fault localization, such as its techniques and tools, which was a very interesting topic to learn about and participate in. It is hoped that this project will contribute to the topic of automatic fault localization among programmers, which is extremely underappreciated for its value.

Finally, despite the demanding work required to complete this document and the limited experience in the field, it is very satisfying to be able to achieve the initial goals and to leave the comfort zone, by expanding knowledge on various areas of software development as well

as learning new technologies that will undoubtedly be valuable for the future career path or personal fulfillment.



# Bibliography

- Abreu, Rui, Peter Zoetewij, and Arjan J. C. van Gemund (2008). "An Observation-Based Model for Fault Localization". In: *Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*. WODA '08. Seattle, Washington: Association for Computing Machinery, pp. 64–70. isbn: 9781605580548. doi: 10.1145/1401827.1401841. url: <https://doi.org/10.1145/1401827.1401841>.
- (2007a). "On the Accuracy of Spectrum-based Fault Localization". In: *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pp. 89–98. doi: 10.1109/TAIC.PART.2007.13.
  - (2007b). "On the Accuracy of Spectrum-based Fault Localization". In: *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pp. 89–98. doi: 10.1109/TAIC.PART.2007.13.
- Abreu, Rui, Peter Zoetewij, Rob Golsteijn, et al. (Nov. 2009). "A practical evaluation of spectrum-based fault localization". In: *Journal of Systems and Software* 82, pp. 1780–1792. doi: 10.1016/j.jss.2009.06.035.
- Adragna, Pierre-Antoine (2008). "Software debugging techniques". In.
- Beller, Moritz et al. (2018). "On the Dichotomy of Debugging Behavior Among Programmers". In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 572–583. doi: 10.1145/3180155.3180175.
- Brito, Steven Carlos Lopes (2020). "An Automated Debugging plug-in for Visual Studio Code". In.
- Campos, José et al. (2012a). "GZoltar: an eclipse plug-in for testing and debugging". In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 378–381. doi: 10.1145/2351676.2351752.
- (2012b). "GZoltar: an eclipse plug-in for testing and debugging". In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 378–381. doi: 10.1145/2351676.2351752.
- Carbonnelle, Pierre (2022). *TOP IDE index*. <https://pypl.github.io/IDE.html>. Accessed: 2022-06-24.
- Cellier, Peggy et al. (July 2011). "Multiple Fault Localization with Data Mining". In: pp. 238–243.
- Chandrasekaran, Jaganmohan et al. (2016). "Evaluating the Effectiveness of BEN in Localizing Different Types of Software Fault". In: *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 26–34. doi: 10.1109/ICSTW.2016.44.
- Chen, Cheng and Nan Wang (2016). "UnitFL: A fault localization tool integrated with unit test". In: *2016 5th International Conference on Computer Science and Network Technology (ICCSNT)*, pp. 136–142. doi: 10.1109/ICCSNT.2016.8070135.
- Denmat, Tristan, Mireille Ducassé, and Olivier Ridoux (2005). "Data Mining and Cross-Checking of Execution Traces: A Re-Interpretation of Jones, Harrold and Stasko Test

- Information". In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ASE '05. Long Beach, CA, USA: Association for Computing Machinery, pp. 396–399. isbn: 1581139934. doi: 10.1145/1101908.1101979. url: <https://doi.org/10.1145/1101908.1101979>.
- Dimitrijevic, Katarina (Nov. 2014). "Transgressing Plastic Waste: Designedisposal Strategic Scenarios". In.
- Ericwong, W. and Yuqi (Nov. 2011). "BP NEURAL NETWORK-BASED EFFECTIVE FAULT LOCALIZATION". In: *International Journal of Software Engineering and Knowledge Engineering* 19. doi: 10.1142/S021819400900426X.
- Golagha, Mojdeh et al. (2018). "Aletheia: A Failure Diagnosis Toolchain". In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pp. 13–16.
- Gürel, Emet (Aug. 2017). "SWOT ANALYSIS: A THEORETICAL REVIEW". In: *Journal of International Social Research* 10, pp. 994–1006. doi: 10.17719/jisr.2017.1832.
- Hirsch, Thomas (Mar. 2021). "A Fault Localization and Debugging Support Framework driven by Bug Tracking Data". In.
- Hong, Shin et al. (2015). "Mutation-Based Fault Localization for Real-World Multilingual Programs (T)". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 464–475. doi: 10.1109/ASE.2015.14.
- ISO (2017). "ISO/IEC/IEEE 24765:2017(en) Systems and software engineering — Vocabulary". In: *ISO*.
- Jamieson, Susan (Jan. 2005). "Likert Scales: How to (ab) Use Them". In: *Medical education* 38, pp. 1217–8. doi: 10.1111/j.1365-2929.2004.02012.x.
- Jia, Yue and Mark Harman (2011). "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Transactions on Software Engineering* 37.5, pp. 649–678. doi: 10.1109/TSE.2010.62.
- Jones, James A., Mary Jean Harrold, and John Stasko (2002a). "Visualization of Test Information to Assist Fault Localization". In: *Proceedings of the 24th International Conference on Software Engineering*. ICSE '02. Orlando, Florida: Association for Computing Machinery, pp. 467–477. isbn: 158113472X. doi: 10.1145/581339.581397. url: <https://doi.org/10.1145/581339.581397>.
- (2002b). "Visualization of Test Information to Assist Fault Localization". In: *Proceedings of the 24th International Conference on Software Engineering*. ICSE '02. Orlando, Florida: Association for Computing Machinery, pp. 467–477. isbn: 158113472X. doi: 10.1145/581339.581397. url: <https://doi.org/10.1145/581339.581397>.
- Joshi, Ankur et al. (Jan. 2015). "Likert Scale: Explored and Explained". In: *British Journal of Applied Science & Technology* 7, pp. 396–403. doi: 10.9734/BJAST/2015/14975.
- Knuth, Donald E. (2016). "Design Science Research Methodology Enquanto Estratégia Metodológica para a Pesquisa Tecnológica". In: *Revista Espacios* 38.6, p. 25.
- Ko, Andrew and Brad Myers (2008). "Debugging reinvented". In: *2008 ACM/IEEE 30th International Conference on Software Engineering*, pp. 301–310. doi: 10.1145/1368088.1368130.
- Koen, Peter, Heidi Bertels, and Elko Kleinschmidt (May 2014). "Managing the Front End of Innovation Part-I Results From a Three-Year Study". In: *Research-Technology Management* 57. doi: 10.5437/08956308X5703199.
- Koen, Peter A. et al. (2002). "1 Fuzzy Front End : Effective Methods , Tools , and Techniques". In.
- Korel, Bogdan and Janusz Laski (1990). "Dynamic slicing of computer programs". In: *Journal of Systems and Software* 13.3, pp. 187–195. issn: 0164-1212. doi: <https://doi.org/>

- 10.1016/0164-1212(90)90094-3. url: <https://www.sciencedirect.com/science/article/pii/0164121290900943>.
- Kusumoto, Shinji et al. (2002). "Experimental Evaluation of Program Slicing for Fault Localization". In: *Empirical Softw. Engg.* 7.1, pp. 49–76. issn: 1382-3256. doi: 10.1023/A:1014823126938. url: <https://doi.org/10.1023/A:1014823126938>.
- Lawrence, Carl, Tuure Tuunanen, and Michael Myers (Mar. 2010). "Extending Design Science Research Methodology for a Multicultural World". In: vol. 318, pp. 108–121. isbn: 978-3-642-12112-8. doi: 10.1007/978-3-642-12113-5\_7.
- Liblit, Ben et al. (2005). "Scalable Statistical Bug Isolation". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: Association for Computing Machinery, pp. 15–26. isbn: 1595930566. doi: 10.1145/1065010.1065014. url: <https://doi.org/10.1145/1065010.1065014>.
- Liu, Chao et al. (2006). "Statistical Debugging: A Hypothesis Testing-Based Approach". In: *IEEE Transactions on Software Engineering* 32.10, pp. 831–848. doi: 10.1109/TSE.2006.105.
- Mahajan, Gautam (2020). "What Is Customer Value and How Can You Create It?" In: *Journal of Creating Value* 6.1, pp. 119–121. doi: 10.1177/2394964320903557. eprint: <https://doi.org/10.1177/2394964320903557>. url: <https://doi.org/10.1177/2394964320903557>.
- Maruyama, Naoya and Satoshi Matsuoka (2008). "Model-based fault localization in large-scale computing systems". In: *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–12. doi: 10.1109/IPDPS.2008.4536310.
- Mateis, Cristinel, Markus Stumptner, and Franz Wotawa (2000). "Modeling Java Programs for Diagnosis". In: *Proceedings of the 14th European Conference on Artificial Intelligence*. ECAI'00. Berlin, Germany: IOS Press, pp. 171–175.
- Microsoft (2022). *Extension API*. <https://code.visualstudio.com/api>. Accessed: 2022-05-31.
- Moon, Seokhyeon et al. (2014). "Ask the Mutants: Mutating Faulty Programs for Fault Localization". In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pp. 153–162. doi: 10.1109/ICST.2014.28.
- Nicola, Susana (2018). *Análise de Valor*. [https://moodle.isep.ipp.pt/pluginfile.php/187507/mod\\_resource/content/1/An%C3%A1lise\\_Valor\\_Aula\\_4\\_21NOV\\_2018\\_1hora\\_AHP.pdf](https://moodle.isep.ipp.pt/pluginfile.php/187507/mod_resource/content/1/An%C3%A1lise_Valor_Aula_4_21NOV_2018_1hora_AHP.pdf). Accessed: 2022-02-17.
- Osterwalder, Alexander et al. (2014). *Value proposition design*. Wiley.
- Papadakis, Mike and Yves Le Traon (Aug. 2015). "Metallaxis-FL: Mutation-Based Fault Localization". In: *Softw. Test. Verif. Reliab.* 25.5–7, pp. 605–628. issn: 0960-0833. doi: 10.1002/stvr.1509. url: <https://doi.org/10.1002/stvr.1509>.
- Parnin, Chris and Alessandro Orso (2011). "Are Automated Debugging Techniques Actually Helping Programmers?" In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA '11. Toronto, Ontario, Canada: Association for Computing Machinery, pp. 199–209. isbn: 9781450305624. doi: 10.1145/2001420.2001445. url: <https://doi.org/10.1145/2001420.2001445>.
- Rich, Nick, Matthias Holweg, and Wirtschaftsring (2000). "INNOREGIO: dissemination of innovation and knowledge management techniques". In.
- Rosenblum, David S. (1992). "Towards a Method of Programming with Assertions". In: *Proceedings of the 14th International Conference on Software Engineering*. ICSE '92. Melbourne, Australia: Association for Computing Machinery, pp. 92–104. isbn: 0897915046. doi: 10.1145/143062.143098. url: <https://doi.org/10.1145/143062.143098>.

- Saaty, Thomas (Jan. 2008). "Decision making with the Analytic Hierarchy Process". In: *Int. J. Services Sciences Int. J. Services Sciences* 1, pp. 83–98. doi: 10.1504/IJSSCI.2008.017590.
- Sarhan, Qusay et al. (Aug. 2021). "CharmFL: A Fault Localization Tool for Python". In: Sayantini (2020). *What is Debugging and Why is it important?* Last accessed 10 January 2022. url: <https://www.edureka.co/blog/what-is-debugging/>.
- Shchekotykhin, Kostyantyn, Thomas Schmitz, and Dietmar Jannach (2016). "Efficient Sequential Model-Based Fault-Localization with Partial Diagnoses". In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence. IJCAI'16*. New York, New York, USA: AAAI Press, pp. 1251–1257. isbn: 9781577357704.
- Silva, André et al. (2021). *FLACOCO: Fault Localization for Java based on Industry-grade Coverage*. Tech. rep. 2111.12513. arXiv. url: <http://arxiv.org/pdf/2111.12513>.
- Silva, Fabio Pereira da, Higor Amario de Souza, and Marcos Lordello Chaim (2018). "Usability Evaluation of Software Debugging Tools". In: *Proceedings of the XIV Brazilian Symposium on Information Systems. SBSI'18*. Caxias do Sul, Brazil: Association for Computing Machinery. isbn: 9781450365598. doi: 10.1145/3229345.3229410. url: <https://doi.org/10.1145/3229345.3229410>.
- Silva, Leandro Flores da and Edson Oliveira (2020). "Evaluating Usefulness, Ease of Use and Usability of an UML-Based Software Product Line Tool". In: *Proceedings of the 34th Brazilian Symposium on Software Engineering. SBES '20*. Natal, Brazil: Association for Computing Machinery, pp. 798–807. isbn: 9781450387538. doi: 10.1145/3422392.3422402. url: <https://doi.org/10.1145/3422392.3422402>.
- Smith et al. (Dec. 2007). "Customer Value Creation: A Practical Framework". In: *Journal of Marketing Theory and Practise* 15, pp. 7–23. doi: 10.2753/MTP1069-6679150101.
- Souza, Higor, Marcos Chaim, and Fabio Kon (July 2016). "Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges". In: Srivastva, Shreya and Saru Dhir (2017). "Debugging approaches on various software processing levels". In: *2017 International conference of Electronics, Communication and Aerospace Technology (ICECA)*. Vol. 2, pp. 302–306. doi: 10.1109/ICECA.2017.8212821.
- Tanner, Jason (2022). *Defining Customer Value*. <https://appliedframeworks.com/defining-customer-value/>. Accessed: 2022-06-5.
- Taylor, Courtney (2020). *Definition of Bimodal in Statistics*. <https://www.thoughtco.com/definition-of-bimodal-in-statistics-3126325>. Accessed: 2022-06-24.
- UnitFL (n.d.). <https://marketplace.visualstudio.com/items?itemName=Wangnangg.UnitFL>. Accessed: 2022-02-06.
- Virzi, Robert A. (1992). "Refining the Test Phase of Usability Evaluation: How Many Subjects Is Enough?" In: *Human Factors* 34.4, pp. 457–468. doi: 10.1177/001872089203400407. eprint: <https://doi.org/10.1177/001872089203400407>. url: <https://doi.org/10.1177/001872089203400407>.
- Wang, Nan et al. (2015). "FLAVS: A Fault Localization Add-in for Visual Studio". In: *Proceedings of the First International Workshop on Complex Faults and Failures in Large Software Systems. COUFLESS '15*. Florence, Italy: IEEE Press, pp. 1–6.
- Weiser, Mark (1984). "Program Slicing". In: *IEEE Transactions on Software Engineering* SE-10.4, pp. 352–357. doi: 10.1109/TSE.1984.5010248.
- What is Perceived Value?* (2020). <https://www.investopedia.com/terms/p/perceived-value.asp>. Accessed: 2022-02-12.
- Williams, Elise (2021). *What is Deployment in Software*. <https://pdf.wondershare.com/business/what-is-software-deployment.html>. Accessed: 2022-05-31.

- Wong, W. Eric, Vidroha Debroy, et al. (2012). "Effective Software Fault Localization Using an RBF Neural Network". In: *IEEE Transactions on Reliability* 61.1, pp. 149–169. doi: 10.1109/TR.2011.2172031.
- Wong, W. Eric, Ruizhi Gao, et al. (2016). "A Survey on Software Fault Localization". In: *IEEE Transactions on Software Engineering* 42.8, pp. 707–740. doi: 10.1109/TSE.2016.2521368.
- You, Zunwen, Zengchang Qin, and Zheng Zheng (2012). "Statistical fault localization using execution sequence". In: *2012 International Conference on Machine Learning and Cybernetics*. Vol. 3, pp. 899–905. doi: 10.1109/ICMLC.2012.6359473.
- Zhang, Xiangyu, Neelam Gupta, and Rajiv Gupta (2006). "Locating Faults through Automated Predicate Switching". In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. Shanghai, China: Association for Computing Machinery, pp. 272–281. isbn: 1595933751. doi: 10.1145/1134285.1134324. url: <https://doi.org/10.1145/1134285.1134324>.
- Zheng, Wei, Desheng Hu, and Jing Wang (Jan. 2016). "Fault Localization Analysis Based on Deep Neural Network". In: *Mathematical Problems in Engineering* 2016, pp. 1–11. doi: 10.1155/2016/1820454.
- Zimmermann, Thomas and Andreas Zeller (2001). "Visualizing Memory Graphs". In: *Revised Lectures on Software Visualization, International Seminar*. Berlin, Heidelberg: Springer-Verlag, pp. 191–204. isbn: 3540433236.
- Zou, Daming et al. (2021). "An Empirical Study of Fault Localization Families and Their Combinations". In: *IEEE Transactions on Software Engineering* 47.2, pp. 332–347. doi: 10.1109/TSE.2019.2892102.





# Appendix A: User Study

## A.1 Questions

### A.1.1 Introduction

#### Satisfaction survey regarding the Visual Studio Code flacoco extension

This questionnaire seeks to ascertain the level of satisfaction with the flacoco extension for the Visual Studio Code IDE.

It was developed as part of the "Automatic Fault Localization tool IDE integration" master's thesis in Informatics Engineering, Specialization in Software Engineering, from ISEP. Participation in the study is entirely optional, and you may opt out at any moment. All responses are anonymous and will only be used in the academic context revealed.

THE COMPLETION AND DELIVERY OF THE QUESTIONNAIRE PRESUMES CONSENT TO TAKE PART IN THE STUDY.

Estimated time to complete: 2 minutes.

Figure A.1: Questionnaire Questions Part 1

### A.1.2 Demographic

#### Demographic

This section intends to better understand the distribution profile of the participants.

Select the highest level of education you have completed so far. \*

☐ High School Graduate

☐ Bachelor's Degree

☐ Master's Degree

☐ Doctorate Degree

☐ Other

How many years of professional experience do you have in the computer programming field? \*

☐ Less than 1 year

☐ Between 1 and 2 years

☐ Between 2 and 5 years

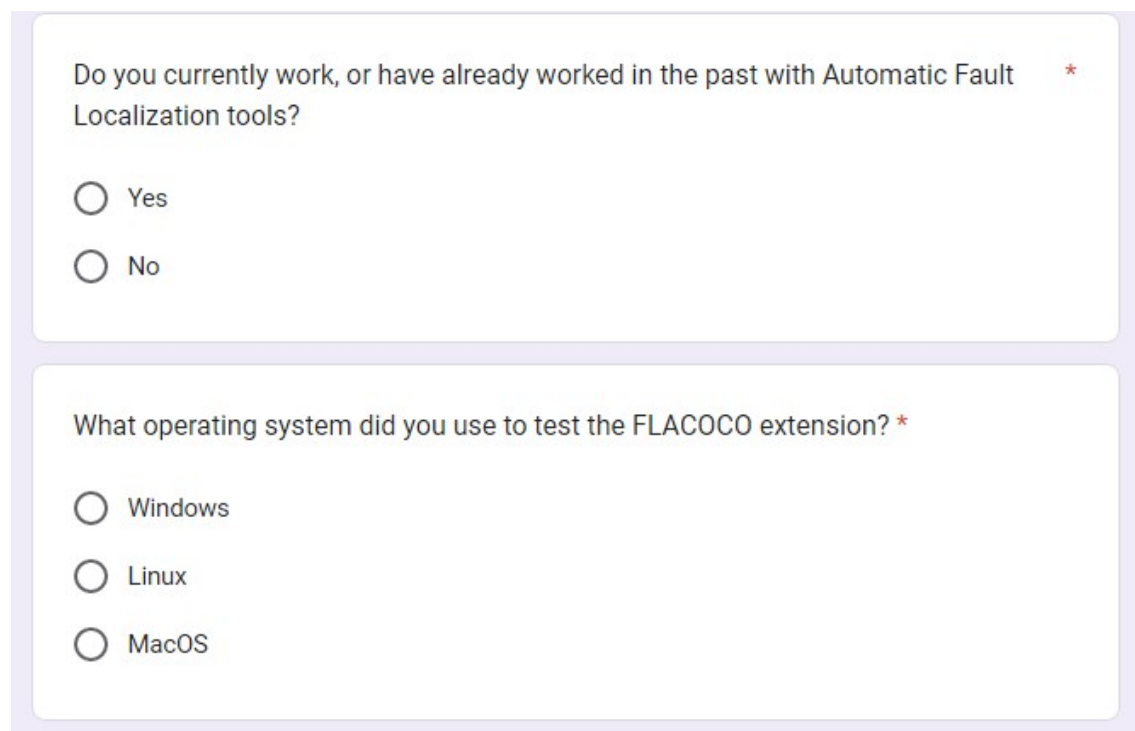
☐ More than 5 years

Do you currently use, or have already used in the past Visual Studio Code? \*

☐ Yes

☐ No

Figure A.2: Questionnaire Questions Part 2



Do you currently work, or have already worked in the past with Automatic Fault Localization tools? \*

☐ Yes

☐ No

What operating system did you use to test the FLACOCO extension? \*

☐ Windows

☐ Linux

☐ MacOS

Figure A.3: Questionnaire Questions Part 3

### A.1.3 User Experience

#### User Experience

The purpose of this section is to learn about the participant's experience with the extension. Choose the best choice for you.

The readme file clearly and objectively describes how to utilize the extension. \*

☐ Strongly Disagree

☐ Disagree

☐ Neutral

☐ Agree

☐ Strongly Agree

It was easy to start using the extension \*

☐ Strongly Disagree

☐ Disagree

☐ Neutral

☐ Agree

☐ Strongly Agree

Figure A.4: Questionnaire Questions Part 4

The extension interface presented information in a structured manner. \*

☐ Strongly Disagree

☐ Disagree

☐ Neutral

☐ Agree

☐ Strongly Agree

The information in the extension interface is presented in a clear manner. \*

☐ Strongly Disagree

☐ Disagree

☐ Neutral

☐ Agree

☐ Strongly Agree

The extension responds quickly. \*

☐ Strongly Disagree

☐ Disagree

☐ Neutral

☐ Agree

☐ Strongly Agree

Figure A.5: Questionnaire Questions Part 5

The extension played an important role in locating the bug.

- ☐ Strongly Disagree
- ☐ Disagree
- ☐ Neutral
- ☐ Agree
- ☐ Strongly Agree

The extension is a positive addition to a developers' daily life. \*

- ☐ Strongly Disagree
- ☐ Disagree
- ☐ Neutral
- ☐ Agree
- ☐ Strongly Agree

Figure A.6: Questionnaire Questions Part 6

I would recommend the extension to other developers. \*

☐ Strongly Disagree

☐ Disagree

☐ Neutral

☐ Agree

☐ Strongly Agree

You can write here if you believe there are any difficulties that were not addressed in the previous question. You can also make suggestions for improvement.

A sua resposta

Figure A.7: Questionnaire Questions Part 7

## A.2 Results

### A.2.1 User Times

Table A.1: User Times

Participant	Joda-Time Task (in seconds)	Graph Task (in seconds)
<b>1</b>	713	897
<b>2</b>	605	181
<b>3</b>	665	235
<b>4</b>	736	231
<b>5</b>	901	592
<b>6</b>	613	414
<b>7</b>	585	303
<b>8</b>	844	364
<b>9</b>	178	114
<b>10</b>	664	481
<b>11</b>	737	604
<b>12</b>	693	535
<b>13</b>	676	511
Average	662	420



### A.2.2 Demographic

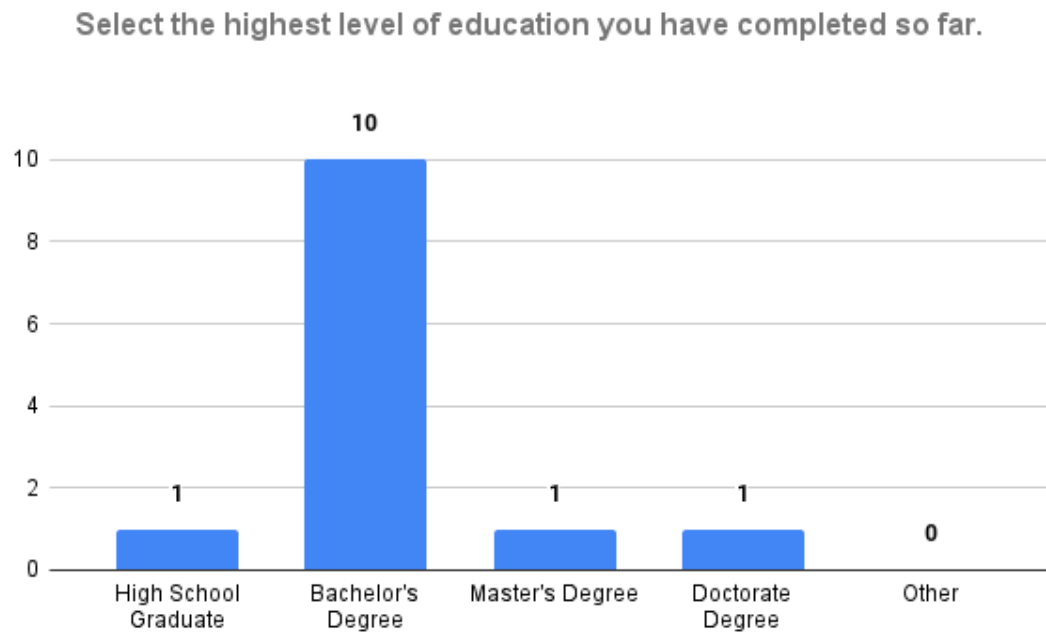


Figure A.8: Demographic Question 1 Results

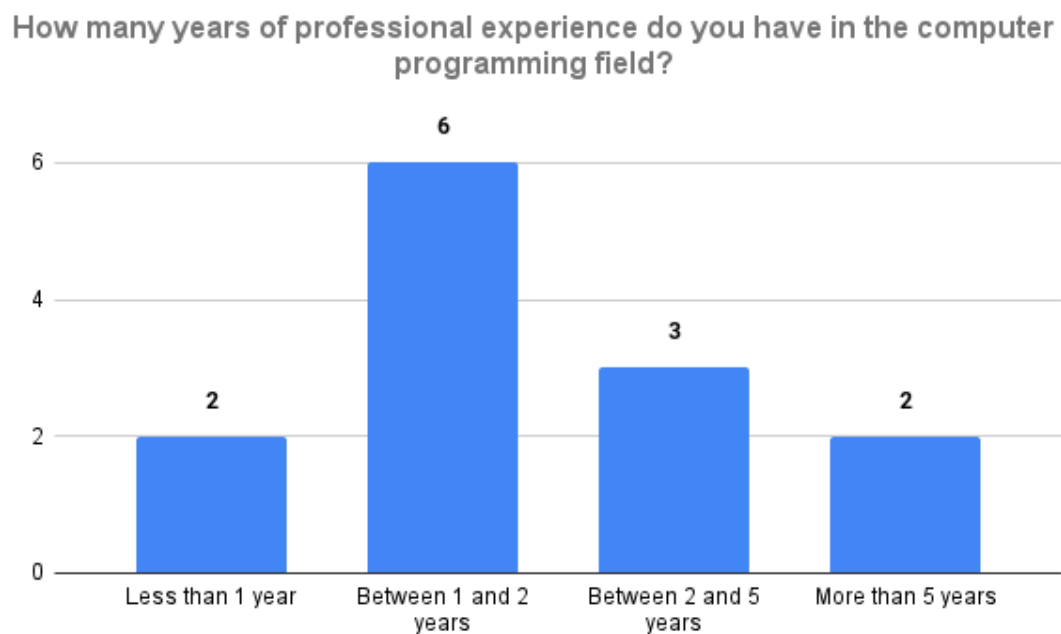


Figure A.9: Demographic Question 2 Results

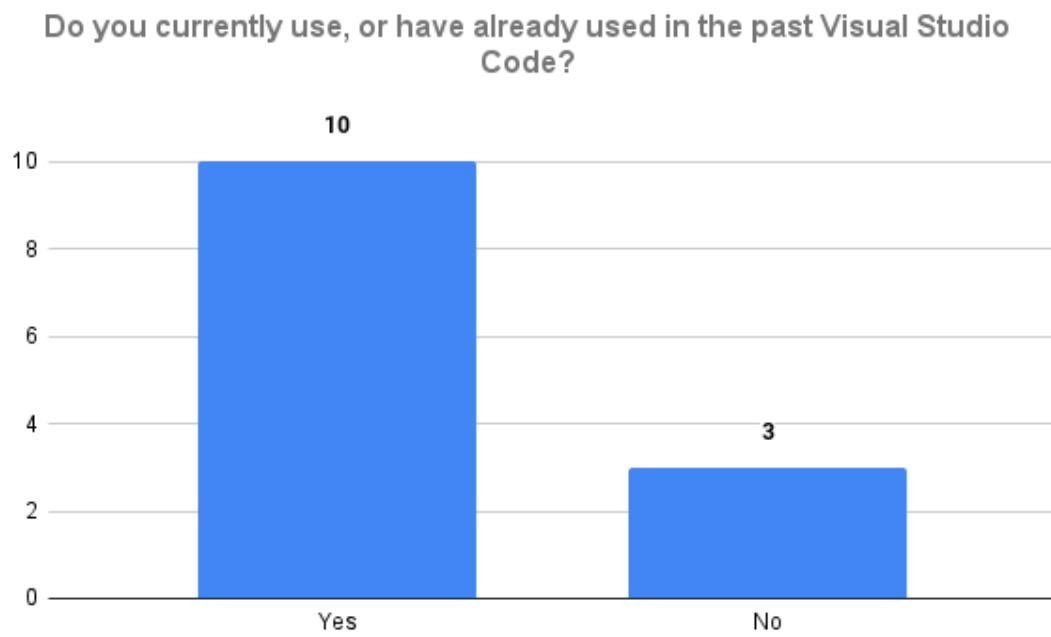


Figure A.10: Demographic Question 3 Results

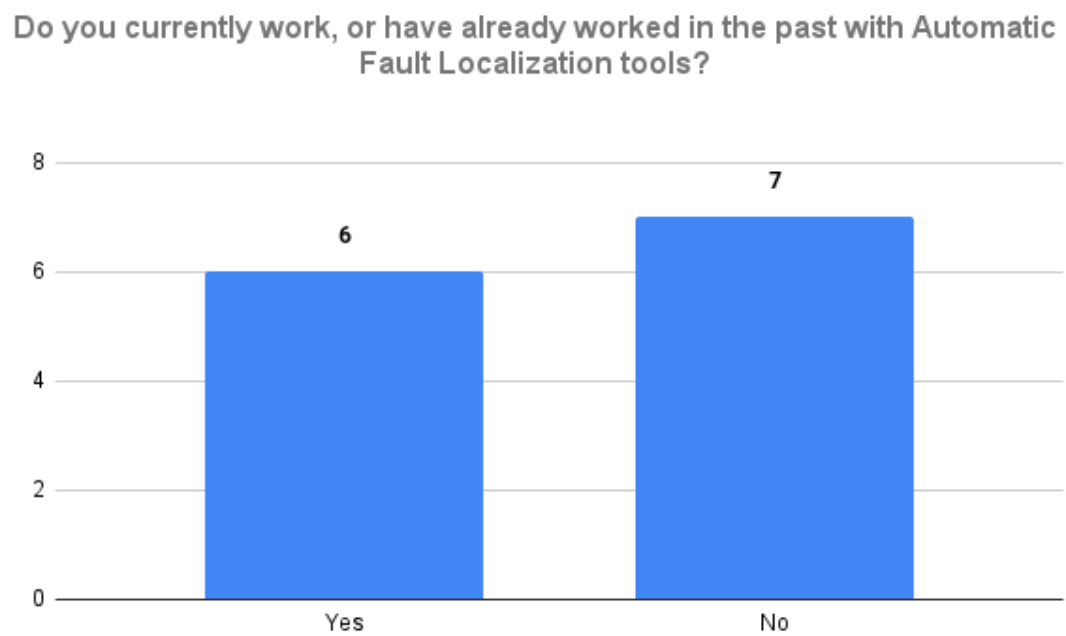


Figure A.11: Demographic Question 4 Results

### A.2.3 User Experience

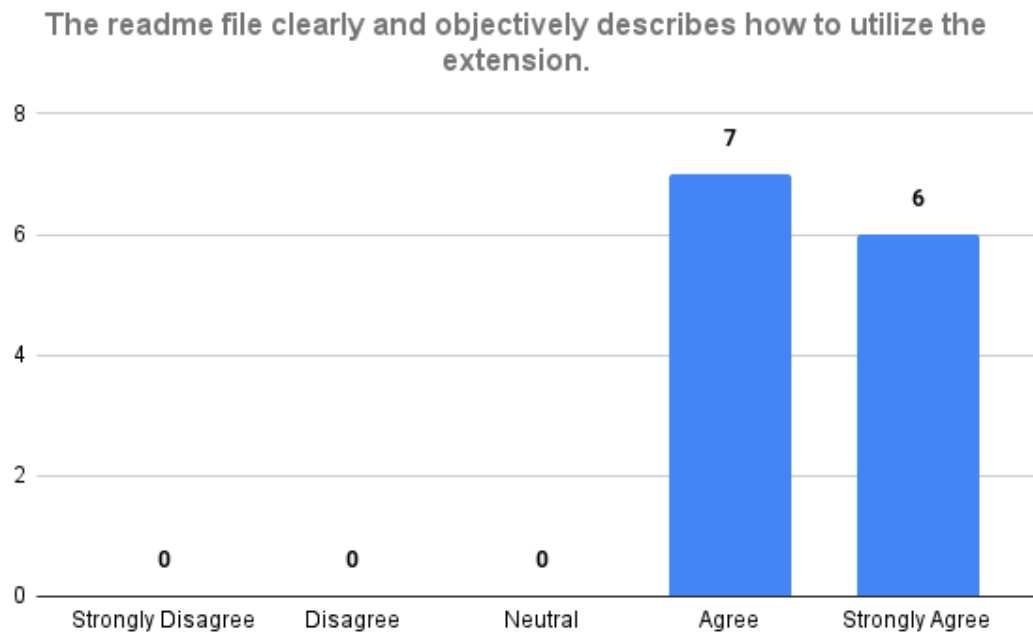


Figure A.12: User Experience Question 1 Results

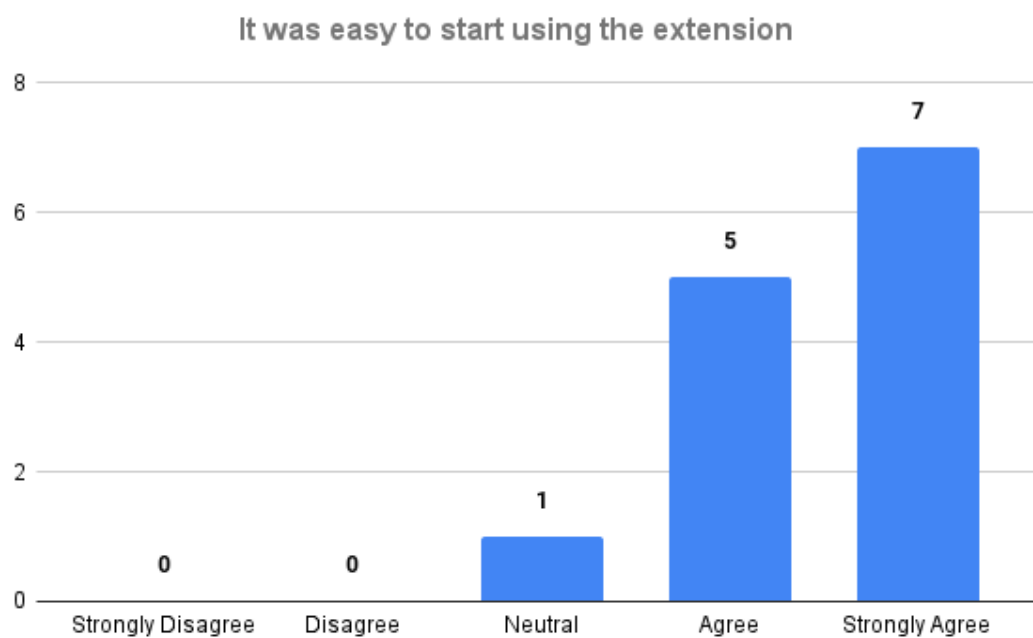


Figure A.13: User Experience Question 2 Results

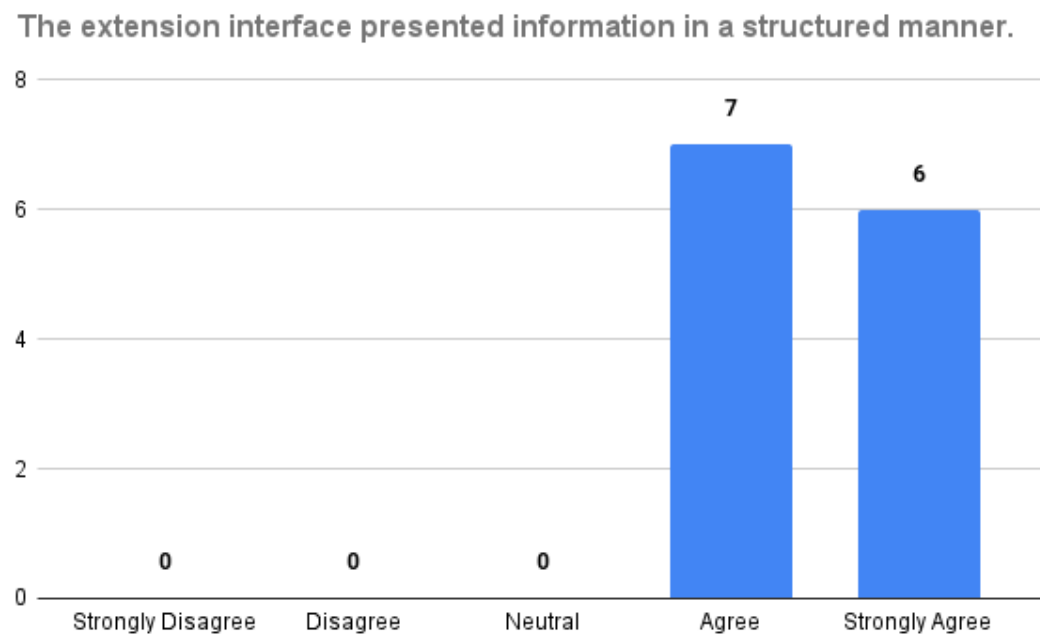


Figure A.14: User Experience Question 3 Results

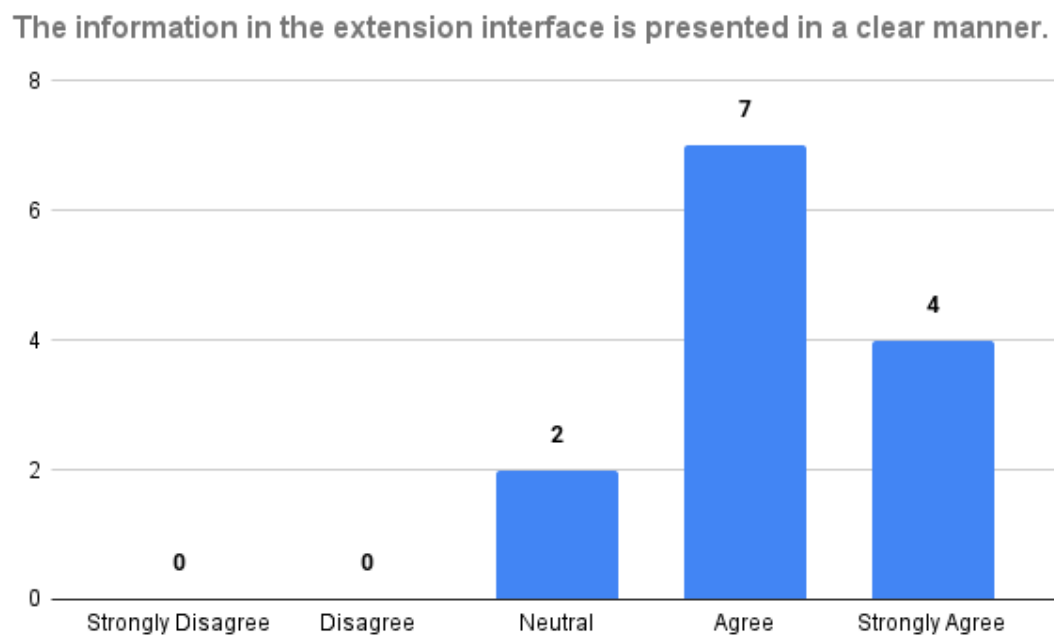


Figure A.15: User Experience Question 4 Results

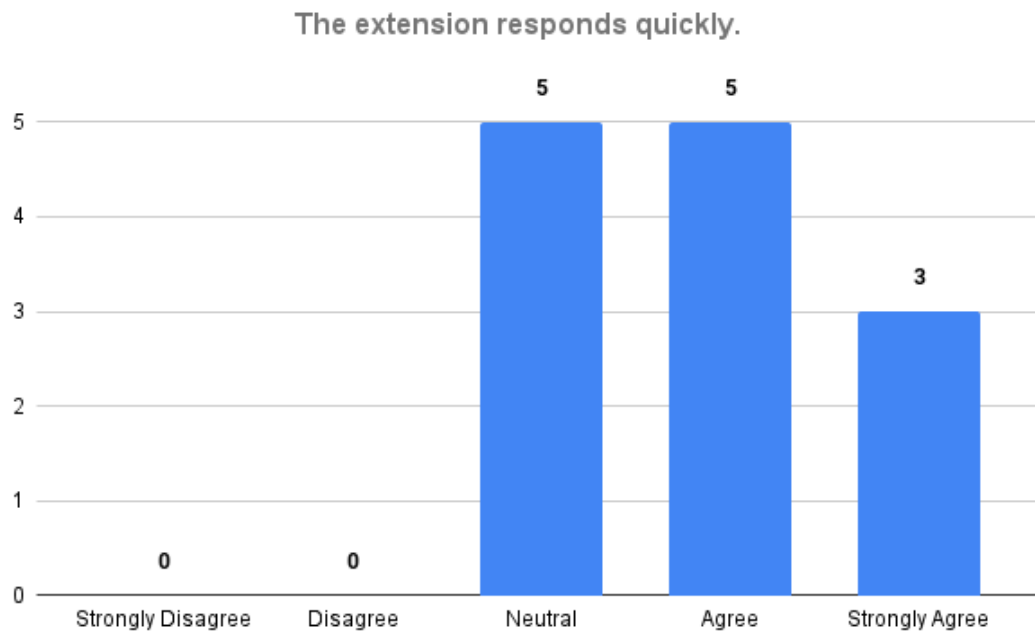


Figure A.16: User Experience Question 5 Results

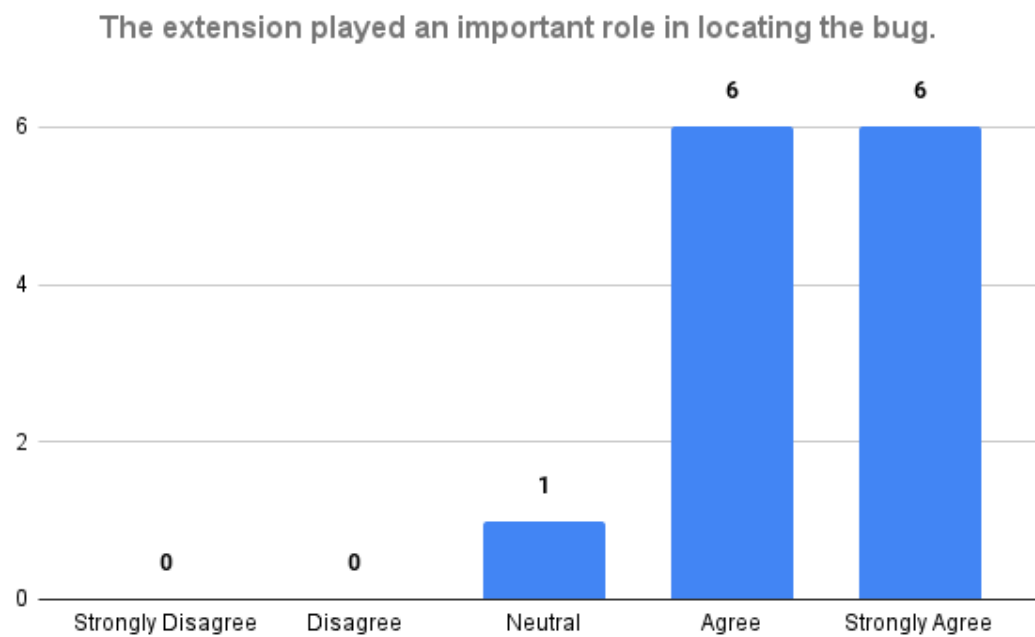


Figure A.17: User Experience Question 6 Results

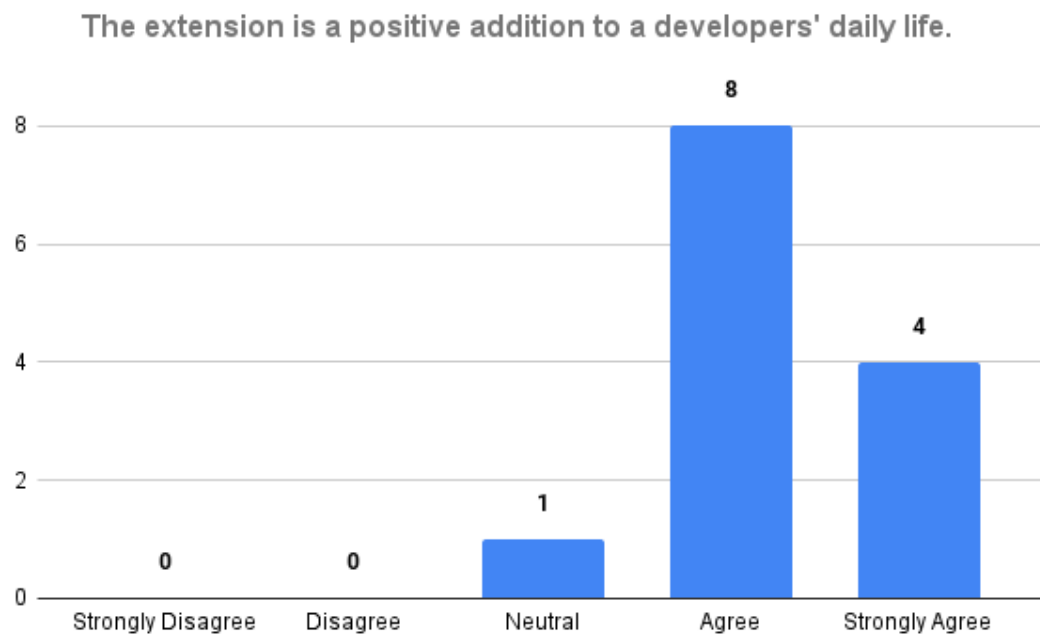


Figure A.18: User Experience Question 7 Results

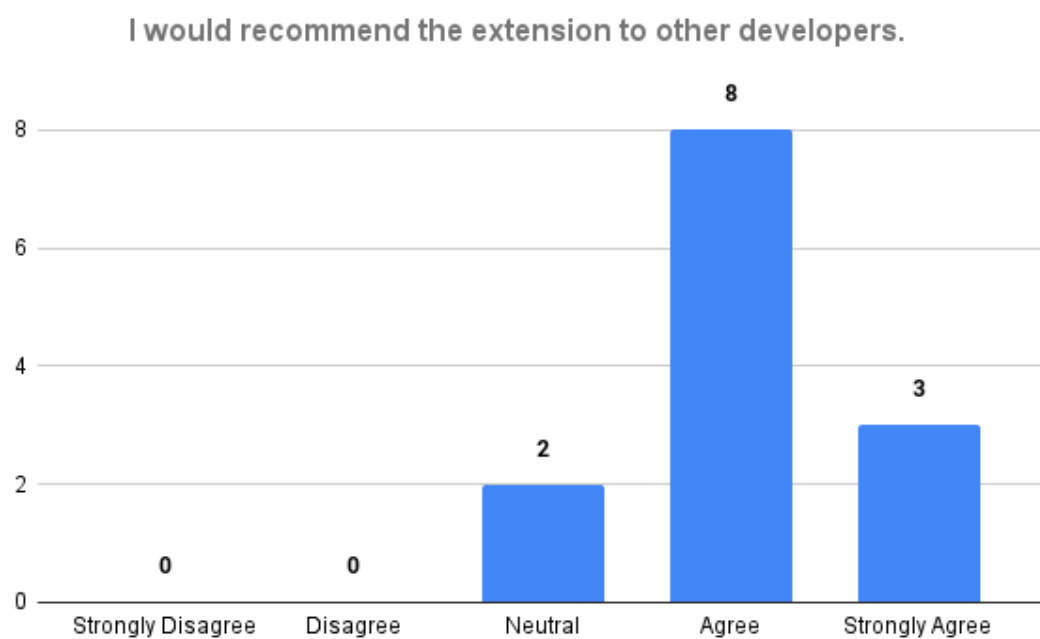


Figure A.19: User Experience Question 8 Results