An Automated Debugging Plug-in for Visual Studio Code

Steven Carlos Lopes Brito

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Rui Filipe Lima Maranhão de Abreu

Examination Committee

October 2020

Resumo

Uma das fases mais complicadas durante o desenvolvimento de software é a realização de testes e depuração. Pode facilmente se tornar numa tarefa muito cansativa e cara, sem mencionar a alta probabilidade de erros. Como tal, vários métodos foram desenvolvidos para melhorar essa tarefa, automatizando este processo o máximo possível, melhorando assim a qualidade do produto final. O GZoltar é uma *framework* para automatização de testes e localização de falhas para projetos Java, integrando-se perfeitamente com testes JUnit. Além disso, a *framework* fornece feedback intuitivo sobre falhas de código utilizando diferentes técnicas de visualização que mostram a distribuição de erros ao longo do código. Atualmente, está disponível como uma interface de linha de comandos, *ant task*, *plug-in* para o Maven e, finalmente, como um *plug-in* para o Eclipse. Nos últimos anos, a popularidade do Eclipse tem vindo a diminuir em comparação com outros IDEs e editores de código (por exemplo, IntelliJ IDEA e Visual Studio Code). O Visual Studio Code é um editor de código desenvolvido pela Microsoft para Windows, Linux e macOS. Inclui suporte para depuração, controle de Git e GitHub, realce de sintaxe, acabamento inteligente de código, *snippets* e refatorização de código. Ultimamente tem crescido em popularidade, visto que é considerado leve e flexível em várias linguagens. O objetivo principal desta tese é desenvolver uma extensão que ofereça as funcionalidades do GZoltar para o Visual Studio Code, visando apaziguar os desenvolvedores que desejam usar a *framework*, mas que já não têm tanto o interesse pelo Eclipse como antes, ou nunca tiveram nenhuma interação prévia com o IDE.

Palavras-chave: Localização de falhas, Debugger Gráfico, Testes Automáticos, Debugging Automático.

Abstract

One of the most cumbersome phases of software development is testing and debugging. It can easily become a very tiring and expensive task, not to mention extremely prone to errors. As such, several methods have been developed to improve this task by automating the whole process as much as possible, thus improving the overall quality of the end product. GZoltar is a framework for automatic testing and fault localization for Java projects, integrating seamlessly with JUnit tests. Additionally, the framework provides intuitive feedback about code faults by using different visualization techniques, which showcase the error distribution along the code base. Currently, it is available as a command line interface, ant task, maven plug-in, and finally as an Eclipse plug-in. In the last couple of years, Eclipse's popularity has been decaying in comparison to other IDEs and code editors (e.g., IntelliJ IDEA and Visual Studio Code). Visual Studio Code is a source-code editor developed by Microsoft for Windows, Linux and macOS. It includes support for debugging, Git control and GitHub, syntax highlighting, intelligent code completion, snippets, and code refactoring. Lately it has been rising in popularity, as it is considered lightweight and flexible across several languages. The main objective of this thesis is to develop an extension offering the GZoltar functionalities in Visual Studio Code, which aims to appease developers who want to use the framework but are not as fond of using Eclipse as before, or have never had any previous interaction with the IDE.

Keywords: Fault Localization, Graphical Debugger, Automatic Testing, Automatic Debugging.

Acknowledgments

In first place, I would like to thank my family for giving me the opportunity and support needed to carry out this thesis and accomplish it adequately. Without them, this would have been difficult.
I would like to express my gratitude to my supervisor Prof. Dr. Rui Maranhão for all his support and guidance throughout this new experience which made me very scared initially, but progressed smoothly as it went on.
Furthermore, I would also like to thank Prof. Dr. José Campos for helping me understand how to effectively use GZoltar. Despite not being my advisor, he helped as much as he could and was always available when I needed.
Finally, I would like to thank my friends for also helping me with the project, encouragement and support.

Contents

List of Figures

Chapter 1

Introduction

GZoltar [1] is a framework for automatic testing and fault localization for Java projects. It integrates seamlessly with JUnit tests, and provides intuitive feedback about code faults by using different visualization techniques. Its toolset implements the spectrum-based fault localization (SBFL) technique using the Ochiai algorithm [2], which is known to be among the best for fault localization. Currently, it is available as a command line interface, ant task, maven plug-in, and finally as an Eclipse plug-in. GZoltar is already widely used in many systems such as ssFix [3], Astor [4], ACS [5], CapGen [6], among others. However, in the last couple of years, other IDEs have surpassed Eclipse in terms of popularity, for instance, IntelliJ IDEA and VS Code. As such, the framework will not have as much exposure as before, due to the declining use of the IDE. For such a reason, we propose adapting the framework as a plugin in a new code editor in hopes of catering to a wider public.

VS Code is a relatively recent lightweight, cross-platform code editor developed by Microsoft for Windows, Linux and macOS. It contains a plethora of useful features right out of the box: IntelliSense, which provides code completion, variable/parameter info, imported modules, etc.; built-in debugger; built-in Git commands, allowing the developer to review file differences, stage, commit, push and pull from the editor; highly customizable, meaning it is possible to create/install extensions to add languages, themes, debuggers or other types of features. Overall, the combination of these features makes a powerful tool for developers.

Considering this, we deemed necessary to provide GZoltar's features as a plugin in VS Code (called extension in this environment).

1.1 Motivation

This section contains the project motivation. It discusses the relevance of the debugging process in the software development life cycle, along with the difficulties that accompany it. It also presents the main problem of debugging tools: the lack of a powerful graphical debugger. There are already many tools based on different techniques and concepts, which aim to alleviate this process. However, very few possess the capabilities of a graphical debugger tool.

1.1.1 About Debugging

Debugging, defined as the process of finding and resolving problems within a computer program, is an intricate process which comprises multiple tactics to try and solve these problems. This set of tactics, which can usually be found in debuggers, includes interactive debugging (a.k.a., step-by-step), unit testing, code coverage, integration testing, among many others. It is a crucial phase during the life cycle of any software development process, allowing the developers to be aware of existing problems in their code, as well as to give them the chance to hone the product as much as possible. However, the process itself can easily become a very tiresome task.

One of the many difficulties present in debugging lies in the ability to reproduce the error, or even knowing where it originated. This is a non-trivial task, since several factors can make it difficult to reproduce the problem, as well as doing it in a time efficient manner. For this purpose, many of the techniques created offer a solution to this by automating the process as much as possible (automated testing), and by trying to pinpoint the fault's origin (fault localization).

Besides being extremely costly in large systems, debugging is also very much needed in software dependant systems where a malfunction or fault can cause deaths, injuries or even environmental harm [7]. These are called safety-critical systems, and are heavily computer-based. Naturally, a failure that has happened in a personal computer may result in a loss of files and/or progress in a project, but a failure in a safety-critical system such as nuclear power plants or airplanes can cause injuries to lots of people, so clearly there is a need to ensure not only mechanisms to tolerate errors in those systems, but also to guarantee the quality of the software being used in them.

1.1.2 About Automated Techniques

Automated testing can ease this problem by automating some repetitive but necessary tasks in a testing environment. This is beneficial for large projects that either require testing the same areas repeatedly, or simply have a large amount of test cases, such that it would be too arduous for someone to do manually.

Fault localization techniques identify the reason for the fault that can explain the bug or error encountered. Debugging is a strenuous task to execute single-handedly, meaning that automated techniques are clearly preferred over manual ones. These techniques are usually classified by the approaches they are based on, categorized in [8] as the following: SBFL [9], slice-based [10, 11], statistics-based [12], program-state based [13], model-based [14], machine learning-based [15], data mining-based [16, 17] and other miscellaneous techniques.

Although there are several tools which integrate these features [18–20], many of them lack a visualization tool to provide intuitive feedback. As previously stated, large projects are difficult to maintain and test as they grow larger in size. A visual report of the faults and defects found in the code base can easily extenuate this task, given that it is a more straightforward approach than simply looking at plain text indicating the several errors found.

1.2 Topic Overview

The debugging process is clearly an important phase in software development. Not only does it allow us to detect problems we might not have even known otherwise, but it can also give insight on how to better improve the end product. The many techniques and tools created for debugging proves it is still as relevant nowadays. When we consider that a system's fault could potentially cause harm to people, it further cements the necessity of better tools. However, because of the ever growing code base in software projects, and with projects becoming more and more complex each time, it is empirical for the tools that we use to also adapt to this change, but still work in the same way. This is something being achieved by automated techniques.

Automated testing and fault localization techniques improve this process, but both come with their own set of drawbacks. Given the repetitive nature of debugging, automating parts of it can potentially save lots of time. Fault localization on the other hand, despite the many techniques available, some tools only present the results/fault cause in clear text or some sort of hierarchy with lines of code. The problem with this becomes increasingly more apparent in larger projects, as it is much harder to analyze hundreds of lines of code that might be suspicious instead of a certain type of visualization that condenses this information and presents it to the user in a manner that is easier to comprehend. This encompasses the problem with most debuggers, the lack of a visual tool to grasp the entirety of the project, and immediately spot the areas where there are faults and/or are linked to faulty segments of the code. This brings about the topic of graphical debuggers.

Graphical debuggers, i.e., visualization tools that provide visual aide do in fact exist, albeit in short numbers, but are still present nonetheless. Yet, the currently available ones do not provide an integrated environment that allows the developer to localize and correct software faults at the same place. This forces the programmer to constantly switch between applications, which will most certainly lead to a loss of productivity. Most tools that are integrated in IDEs do not offer powerful visualizations, and even external tools can become quite a hassle to work with. They must provide as much information to the user as possible, with the least amount of trouble required to understand it. There is a need for a solution that will work in the same environment the developer is working in, and without much trouble installing.

1.3 Objectives

Taking into account the several aforementioned techniques, it is clear that a tool providing those features would certainly be useful to greatly improve and reduce the time spent while debugging/testing software [21]. Therefore, this project aims to port GZoltar's features to a plug-in (extension) in VS Code as a means of making it more accessible to new users. GZoltar's main features, which are its views that indicate fault localization, are readily available as out-of-the-box extension that can be found in VS Code's marketplace. This marketplace (also integrated in the editor) contains every published extension, from which it is fairly simple to access and install. The goal is to make the extension as easy to use as possible.

The extension itself will be responsible for identifying open projects/folders as Java projects, and then present GZoltar's functionalities to the user. The interface should indicate clearly how many of the open folders are Java projects (since it is possible to work with several projects simultaneously in VS Code) and allow the user to call upon its features only on the acceptable projects. When those requirements are met, it is possible to request an analysis for the selected Java project. Shortly after this, the visualizations are presented to the user. These consist of hierarchical localization views, where a component is connected to its child components or its parent ones. It is possible to zoom in inside a component (assuming there are small components inside), and also to zoom out to get a wider view of the project. The deepest level from which we are able to zoom in can be a single line of code, while the broadest level is the root of the project. This allows the developer to have a clear perception about the components at any level, be it a smaller and more specific section or a broader one. Figure 1.1 shows a graphical representation of this concept.

The views provided by GZoltar consist of 3 hierarchical visualizations, namely sunburst, bubble hierarchy and treemap partition. A sunburst chart consists of an inner circle surrounded by rings of deeper hierarchy levels. The angle of each segment is either proportional to a value or divided equally under its parent node, and each segment may be colored according to which category or hierarchy level they belong to. A treemap chart shows the visualization of hierarchical data in the form of nested rectangles. Each level of a tree structure is depicted as a colored rectangle which contains other rectangles, where the area is proportional to its value. A bubble hierarchy chart consists of a hierarchical data visualization through bubble nesting. Each bubble represents a category, and inside are many specific sub-parts of that same category. An example of each chart can be seen in Figures 1.2, 1.3 and 1.4 respectively.

1.4 Contribution

In this dissertation, we show that the key to effective debugging is the use of a fault localization tool with powerful visualizations, which allow the user to navigate through every section of the project, with code segments containing colored labels ranging from red (indicating high suspicion, likely cause of the fault) to green (having no suspicion at all). With this set of views, we can easily find the faults faster, or at least predict where the actual fault might be. The main contributions of this thesis are:

- We introduce the adaptation of a fault localization tool as a plug-in in Visual Studio Code. The plug-in is built to be as easy to use as possible, with hopes of ensuring the best debugging experience possible.
- The plug-in is published in Visual Studio Marketplace, the aggregator for all plug-ins related to Visual Studio, Visual Studio Code, and Azure DevOps. Being easily accessible in a public place allows it to be discovered by more users.
- We have conducted a user study to obtain feedback from users who have never interacted with GZoltar before. This experiment serves to prove the effectiveness of the plug-in, and also to know about ways to further improve and develop it.

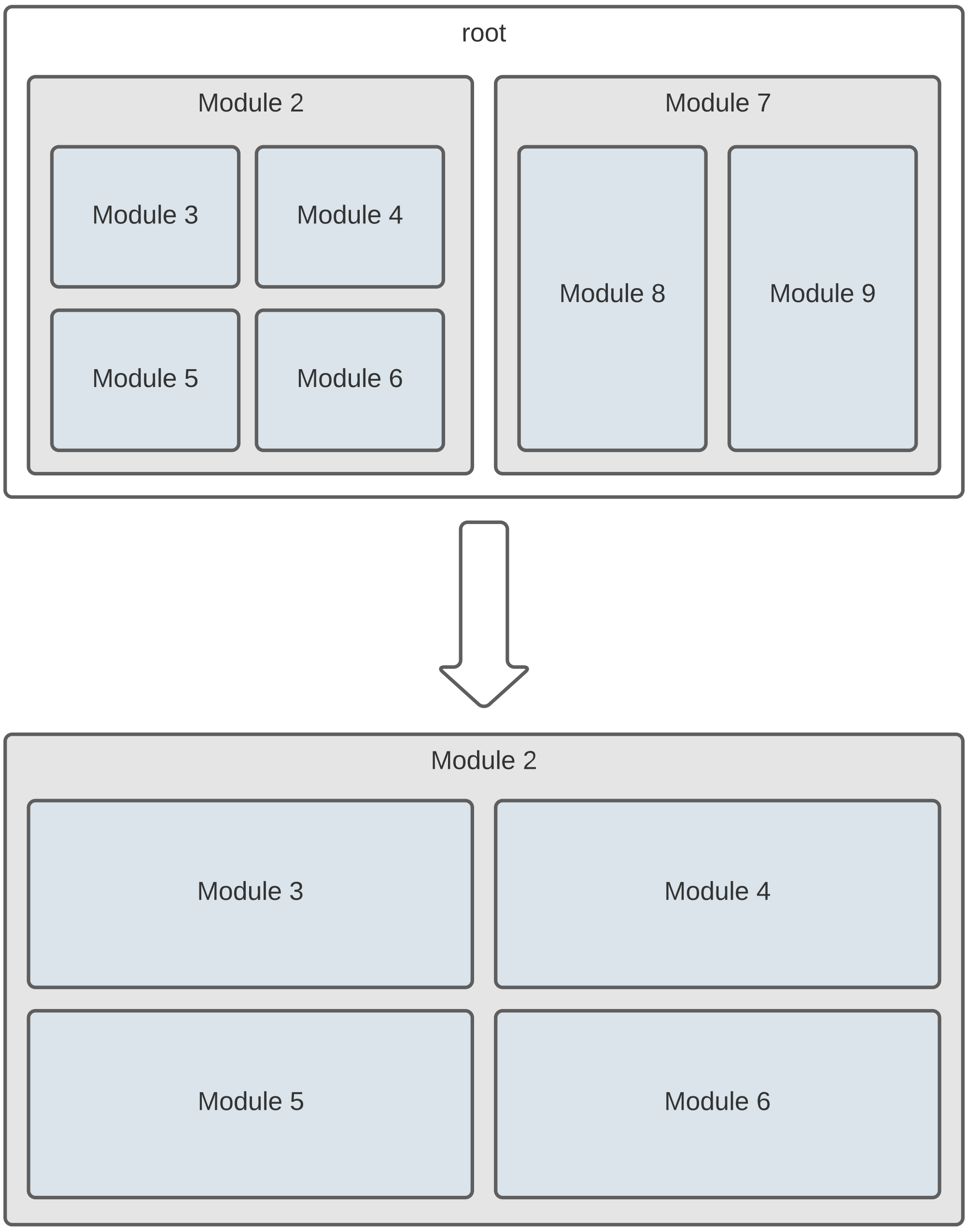


Figure 1.1: ***GZoltar's Hierarchical View.*** *It is possible to view the project in its entirety and navigate to specific sections.*

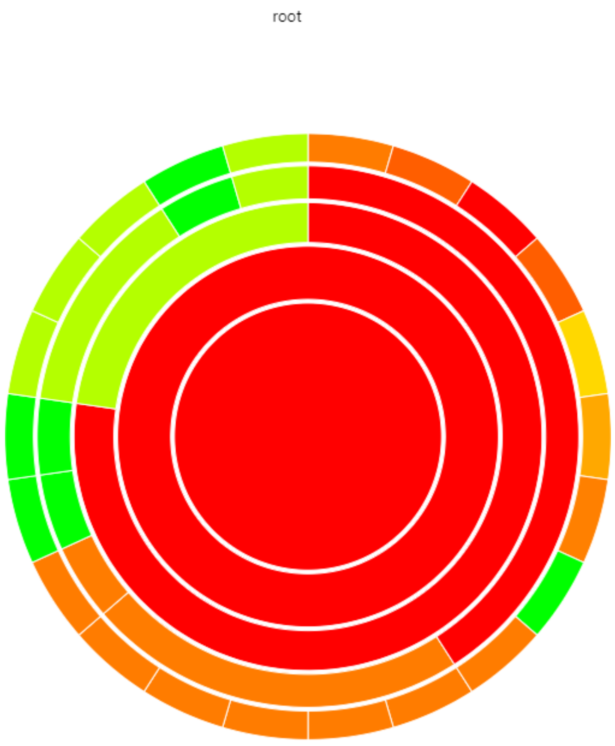


Figure 1.2: *Sunburst chart.*

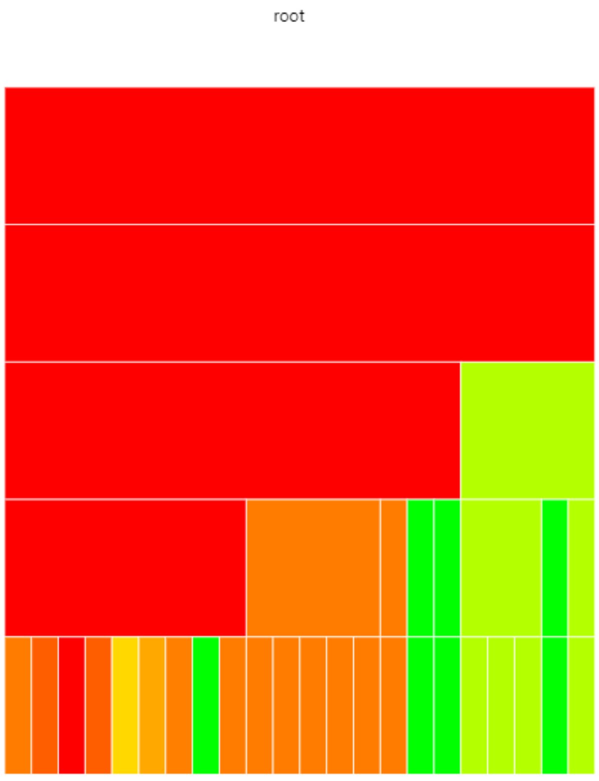


Figure 1.3: *Treemap chart.*

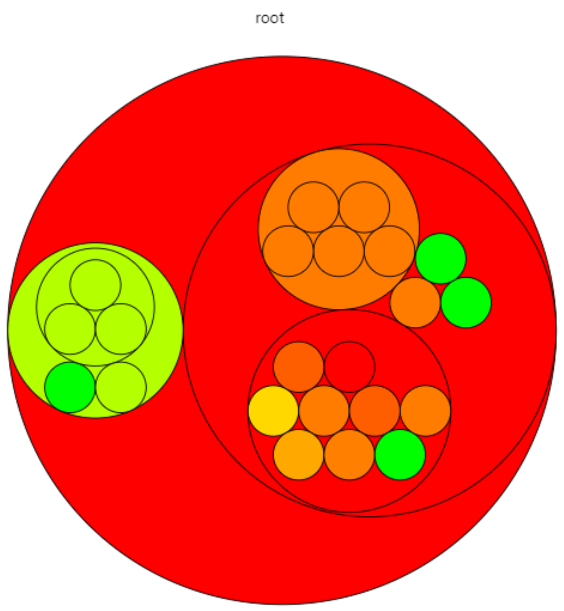


Figure 1.4: *Bubble hierarchy chart.*

1.5 Thesis Outline

This section presents the document structure. Apart from Introduction, the remainder of this document has four more chapters organized in the following way:

Chapter 2 - Related work and state-of-the-art technologies. This chapter analyzes tools and technologies that solve some of the problems previously mentioned, along with the advantages and disadvantages of each one, and why there is still need for further improvement.

Chapter 3 - Details of the implementation. This chapter details the development process of a VS Code extension, the implementation of our solution and how to make the extension public by publishing it to the Visual Studio Marketplace.

Chapter 4 - Results and evaluation. This chapter presents the results obtained throughout this project. It shows the extension in practice, how it can be installed, and also a user study that was carried out to obtain feedback about the extension's usability.

Chapter 5 - Conclusions. This chapter summarizes everything that was done in this project, the accomplishments, and also some ideas for future work.

Chapter 2

State of the Art

This chapter describes the many tools and technologies we have found that pose, in one way or another, a solution to the problems previously mentioned. There are many different techniques used in their implementation, each with a varying degree of complexity to it. Some of them may not have been recently developed, but are still held in high regards in the context of fault localization.

As previously stated, nowadays there are several debugging techniques available to aid developers. Each technique has its own set of difficulty associated, with different types of effectiveness. The most typical ones, manual debugging tools, consist of step-by-step executions of test cases to try to localize the fault. This problem will be aggravated by the project's dimension (among other factors) as it becomes extremely difficult to manually search for the bug in a large-scale project. Furthermore, this depends on the developer's own experience to identify or at least prioritize which section of the code is likely to be faulty. Our main focus will be on automated fault localization tools with visualizations that rely on information about test case executions and their results, to speculate about where the fault might be. Several studies have been made to ascertain their credibility [8, 22, 23], which prove their usefulness in this situation. We will briefly explore some of these fault localization techniques, and then analyze examples of graphical tools which use these techniques since many of them solve some aspects of the problem in question, but still lack certain features we deem useful. At the end of this chapter, we will be comparing each of these tools' graphical capabilities, summarizing both their advantages and disadvantages, indicating what they are lacking in order to be considered a better visual aide.

2.1 Spectrum-Based Techniques

Spectrum-based techniques [9], use information obtained from a program spectrum. A program spectrum is an agglomerate of several types of data that provides a specific view on a software's dynamic behavior. Naturally, these types of data are collected at run-time and typically consist of a number of flags or counters for the different parts of a program. As a result, collecting data for a program spectrum is a light-weight analysis [24], thus contributing to SBFL techniques' reliability.

In [2] their focus is solely on block-hit spectra, but this approach can easily be generalized to other types of program spectra, such as path-hit spectra, data-dependence-hit spectra, etc.) A block-hit spectrum contains a flag for every block of code in a program, indicating whether that block was executed in a certain run or not. A block of code consists of a statement, where individual statements of a compound statement are not distinguished, but the cases of a switch statement are.

Given *M* runs of a program, the hit spectra of these runs constitute a binary matrix, where the columns correspond to *N* different parts of the program (blocks in this case). Another column vector, the error vector, contains information about which runs contain an error. This vector is thought to represent a hypothetical part of the program responsible for all the observed errors. SBFL consists in identifying the part whose column vector resembles the error vector the most. Figure 2.1 shows the vectors used in SBFL.

$$\begin{array}{ccc} & N \text{ parts} & \text{errors} \\ M \text{ spectra} & \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1N} \\ x_{21} & x_{22} & \cdots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M1} & x_{M2} & \cdots & x_{MN} \end{bmatrix} & \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix} \\ & \begin{matrix} s(1) & s(2) & \ldots & s(N) \end{matrix} & \end{array}$$

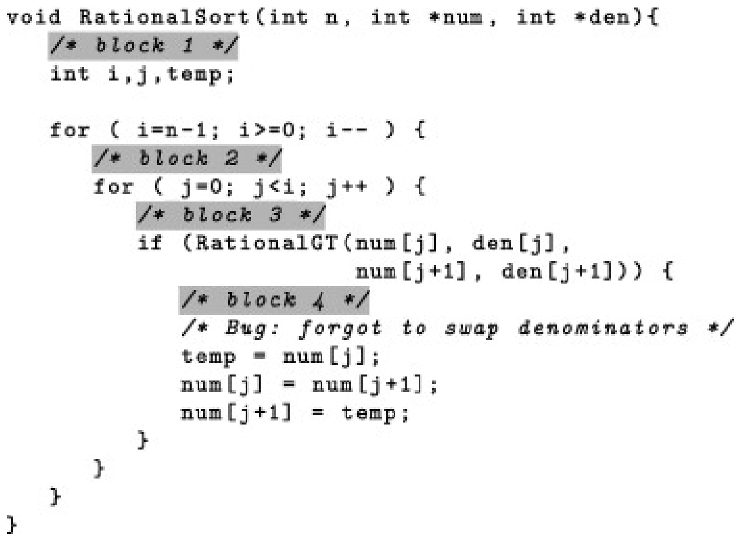
Figure 2.1: *Spectrum-based fault localization vectors.*

In the field of data clustering, to find resemblances between vectors of binary, nominally scaled data such as the columns in program spectra matrices, similarity coefficients are used [25]. The calculated similarity coefficients rank parts of the program with respect to their likelihood of containing the faults. This is used under the assumption that if any part of the software's vector is highly similar to the error vector, then there is a high probability that that part caused the detected errors. The Ochiai coefficient used in the molecular biology domain [26] serves as the underlying coefficient used in GZoltar:

$$s_o(j) = \frac{a_{11}(j)}{\sqrt{(a_{11}(j) + a_{01}(j)) * (a_{11}(j) + a_{10}(j))}}$$

where $a_{pq}(j) = |\{i \mid x_{ij} = p \wedge e_i = q\}|$, and $p, q \in \{0, 1\}$, with $x_{ij} = p$ indicating whether the block j was touched ($p = 1$) in the execution of run i or not ($p = 0$). Additionally, $e_i = q$ indicates whether a run i was faulty ($q = 1$) or not ($q = 0$).

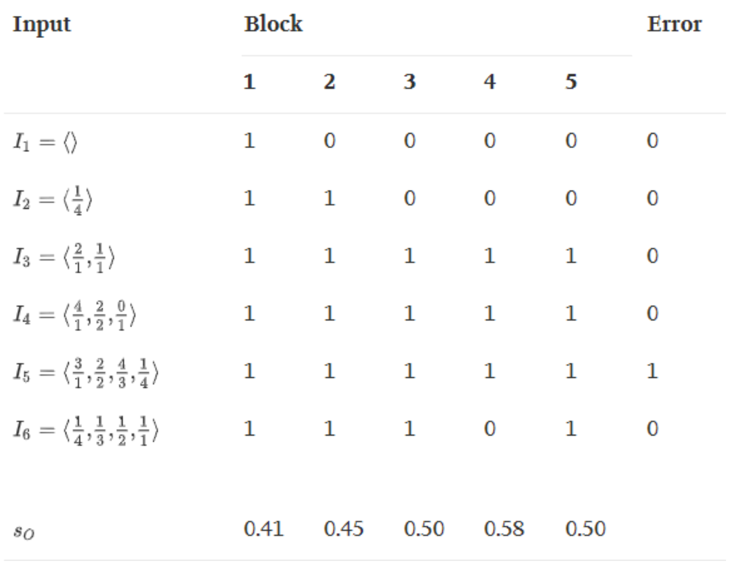
To illustrate these concepts, consider the C function in Figure 2.2. It uses the bubble sort algorithm to sort a sequence of n rational numbers whose numerators and denominators are stored in the parameters num and den respectively. There is a fault in the code within the body of the if statement: only the numerators of the rational numbers are swapped while the denominators maintain their original order. But still, some errors can go unnoticed and do not automatically lead to failures. For example, if the sequence $\langle \frac{4}{1}, \frac{2}{2}, \frac{0}{1} \rangle$ is used, an error occurs after swapping the first two numerators. However, this error is overshadowed by the later swapping actions, and the final sequence ends up being sorted correctly.



```
void RationalSort(int n, int *num, int *den){
    /* block 1 */
    int i,j,temp;

    for ( i=n-1; i>=0; i-- ) {
        /* block 2 */
        for ( j=0; j<i; j++ ) {
            /* block 3 */
            if (RationalGT(num[j], den[j],
                           num[j+1], den[j+1])) {
                /* block 4 */
                /* Bug: forgot to swap denominators */
                temp = num[j];
                num[j] = num[j+1];
                num[j+1] = temp;
            }
        }
    }
}
```

Figure 2.2: *Faulty code for sorting rational numbers.*

To put the similarity coefficients into action, suppose that we apply the RationalSort function to the input sequences $I_1, ...I_6$ shown in Figure 2.3. The block hit spectra for these runs are shown in the central part of the table (‘1’ denotes a hit), where block number five corresponds to the body of the RationalGT function. I_1, I_2 and I_6 are already sorted and lead to passed runs. I_3 is not sorted, but no error occurs since the denominators happen to be equal. I_4 is the example previously mentioned, where an error occurs during the execution but goes undetected. Lastly, for the sequence I_5, the program fails, since the calculated result is different from what is expected, which is a clear indication that an error has occurred. For this sequence, the calculated similarity coefficients listed at the bottom of the image correctly identify block 4 as the most likely location of the fault.



Input	**Block**					**Error**
	1	**2**	**3**	**4**	**5**	
$I_1 = \langle\rangle$	1	0	0	0	0	0
$I_2 = \langle\frac{1}{4}\rangle$	1	1	0	0	0	0
$I_3 = \langle\frac{2}{1},\frac{1}{1}\rangle$	1	1	1	1	1	0
$I_4 = \langle\frac{4}{1},\frac{2}{2},\frac{0}{1}\rangle$	1	1	1	1	1	0
$I_5 = \langle\frac{3}{1},\frac{2}{2},\frac{4}{3},\frac{1}{4}\rangle$	1	1	1	1	1	1
$I_6 = \langle\frac{1}{4},\frac{1}{3},\frac{1}{2},\frac{1}{1}\rangle$	1	1	1	0	1	0
s_O	0.41	0.45	0.50	0.58	0.50	

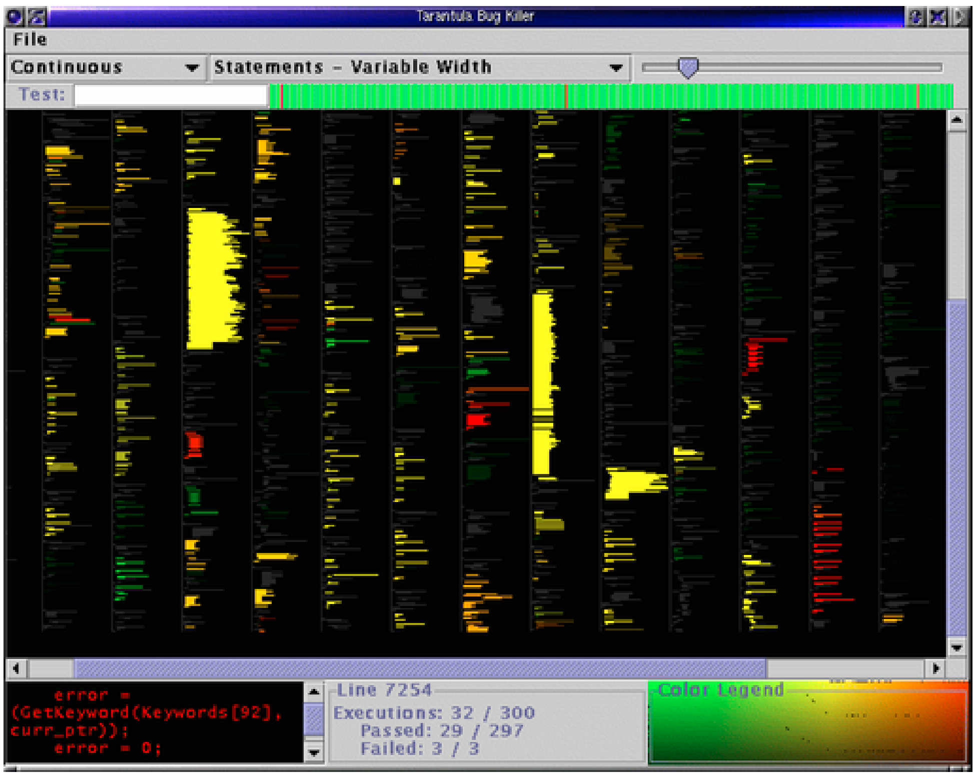
Figure 2.3: *SBFL applied to six runs of the RationalSort program.*

2.1.1 Tarantula

Tarantula [27] is fault localization tool for programs written in the C programming language. It is a visualization tool which allows the user to inspect potentially faulty statements present in failed tests by visually mapping each program's statement in the outcome of an executed test suite. Tarantula provides the developer a global view of the source code, making use of a color and brightness component to showcase the different results, depending on the test results.

The color component depends on the percentage of passed/failed test cases. If a higher percentage of passed test cases executes a statement, it will appear more green. However, if a higher percentage of failed test cases executes that statement, it will appear more red. In the event that both percentages are equal, the statement will appear yellow. The brightness component illustrates the percentage of coverage by either passed or failed test cases, meaning that a statement will either be drawn at full brightness if all test cases execute it, or completely dark otherwise. Figure 2.4 shows Tarantula's overall view.

Despite being a well-known automatic debugging tool, it does not work with unit testing framework and does not integrate with any IDE. This proves to be a clear inconvenience for developers who wish to use the tool without having to rely on extra software. Additionally, it lacks a hierarchical view, and its interface looks dated when compared to modern debugger interfaces, which might decrease the appeal to some developers.



```
    error =
(GetKeyword(Keywords[92],
curr_ptr));
    error = 0;
```

```
Line 7254
Executions: 32 / 300
    Passed: 29 / 297
    Failed: 3 / 3
```

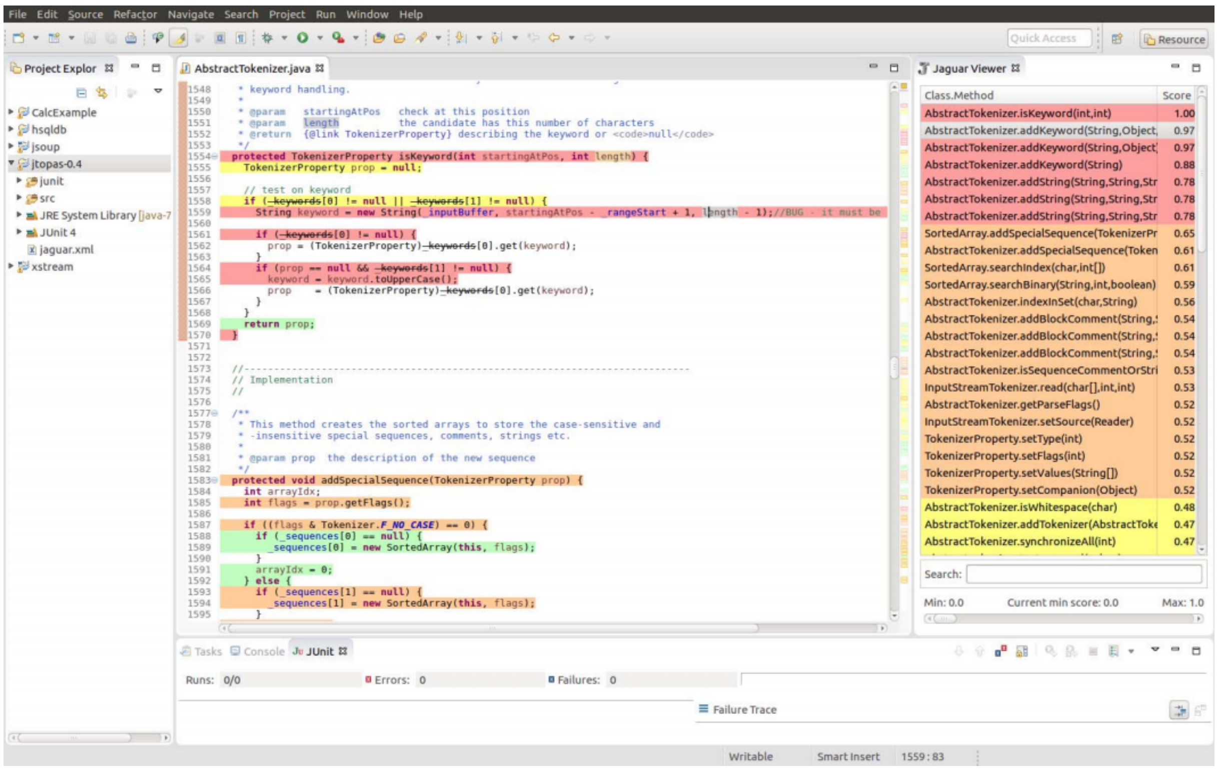
Figure 2.4: *Tarantula.*

2.1.2 Jaguar

Jaguar [28] is a fault localization tool for Java programs, available as an Eclipse plug-in and a command line tool. It uses SBFL techniques based on both data and control-flow spectra. Most SBFL techniques use control-flow spectra (which take into account statements and branches) due to the low cost associated. Since data-flow spectrum subsumes control-flow, it can provide more information to better assist during the fault location process. However, this implies a higher run-time overhead compared to using control-flow only. Nevertheless, recent studies show that with large and long-running programs both spectra can be used with acceptable overhead.

In terms of visual assessment, Jaguar provides visual information with Jaguar viewer. There are four colors used to represent suspicious, with red (danger) being used on the most suspicious entities, orange (warning) to those with high suspicion, yellow (caution) to moderate suspicion and finally green (safety) to label the least suspicious ones. The viewer shows a list of all the suspicious statements differently colored, with the possibility of choosing among seven types of views. Figure 2.5 shows the Jaguar View's list of methods within the Eclipse IDE.

Although Jaguar makes use of data-flow spectra, one of its limitations come from define-use associations. For example, it cannot collect data when the definition and use of a variable occur in the same block, due to limitations of other tools used. Lastly, there was not a data-flow spectrum evaluation, given that they used only control-flow spectrum in the study, which does not lead to conclusive results relating to data-flow spectra usage.



```
   * keyword handling.
   *
   * @param   startingAtPos   check at this position
   * @param   length          the candidate has this number of characters
   * @return  {@link TokenizerProperty} describing the keyword or <code>null</code>
   */
  protected TokenizerProperty isKeyword(int startingAtPos, int length) {
    TokenizerProperty prop = null;

    // test on keyword
    if (~~_keywords~~[0] != null || ~~_keywords~~[1] != null) {
      String keyword = new String(_inputBuffer, startingAtPos - _rangeStart + 1, length - 1);//BUG - it must be

      if (~~_keywords~~[0] != null) {
        prop = (TokenizerProperty)~~_keywords~~[0].get(keyword);
      }
      if (prop == null && ~~_keywords~~[1] != null) {
        keyword = keyword.toUpperCase();
        prop    = (TokenizerProperty)~~_keywords~~[0].get(keyword);
      }
    }
    return prop;
  }


  //---------------------------------------------------------------------------
  // Implementation
  //

  /**
   * This method creates the sorted arrays to store the case-sensitive and
   * -insensitive special sequences, comments, strings etc.
   *
   * @param prop  the description of the new sequence
   */
  protected void addSpecialSequence(TokenizerProperty prop) {
    int arrayIdx;
    int flags = prop.getFlags();

    if ((flags & Tokenizer.F_NO_CASE) == 0) {
      if (_sequences[0] == null) {
        _sequences[0] = new SortedArray(this, flags);
      }
      arrayIdx = 0;
    } else {
      if (_sequences[1] == null) {
        _sequences[1] = new SortedArray(this, flags);
      }
```

Class.Method	Score
AbstractTokenizer.isKeyword(int,int)	1.00
AbstractTokenizer.addKeyword(String,Object,	0.97
AbstractTokenizer.addKeyword(String,Object)	0.97
AbstractTokenizer.addKeyword(String)	0.88
AbstractTokenizer.addString(String,String,Str	0.78
AbstractTokenizer.addString(String,String,Str	0.78
AbstractTokenizer.addString(String,String,Str	0.78
SortedArray.addSpecialSequence(TokenizerPr	0.65
AbstractTokenizer.addSpecialSequence(Token	0.61
SortedArray.searchIndex(char,int[])	0.61
SortedArray.searchBinary(String,int,boolean)	0.59
AbstractTokenizer.indexInSet(char,String)	0.56
AbstractTokenizer.addBlockComment(String,!	0.54
AbstractTokenizer.addBlockComment(String,!	0.54
AbstractTokenizer.addBlockComment(String,!	0.54
AbstractTokenizer.isSequenceCommentOrStri	0.53
InputStreamTokenizer.read(char[],int,int)	0.53
AbstractTokenizer.getParseFlags()	0.52
InputStreamTokenizer.setSource(Reader)	0.52
TokenizerProperty.setType(int)	0.52
TokenizerProperty.setFlags(int)	0.52
TokenizerProperty.setValues(String[])	0.52
TokenizerProperty.setCompanion(Object)	0.52
AbstractTokenizer.isWhitespace(char)	0.48
AbstractTokenizer.addTokenizer(AbstractToke	0.47
AbstractTokenizer.synchronizeAll(int)	0.47

Search:

Min: 0.0   Current min score: 0.0   Max: 1.0

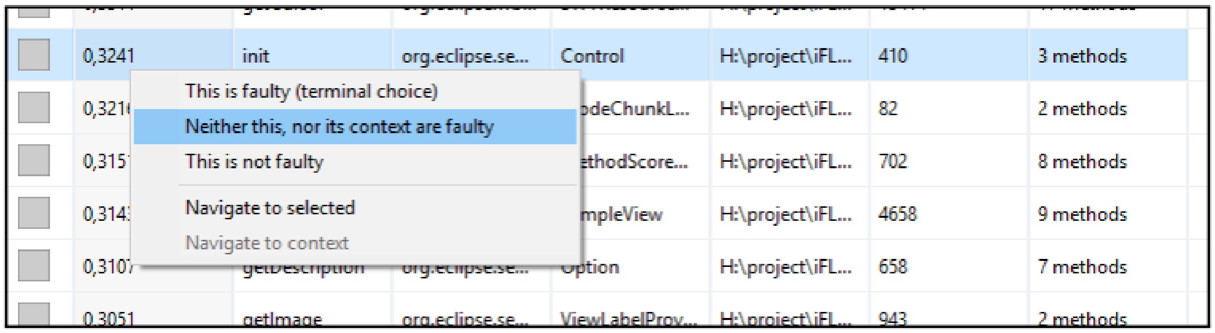
Runs: 0/0   Errors: 0   Failures: 0

Figure 2.5: *Jaguar tool with a method list.*

2.1.3 Interactive Fault Localization

iFL [29] is a fault localization tool for Java programs available as an Eclipse plug-in. Like any fault localization tool, it shows a list of the potential faulty statements in the code. Then, the developer interacts with the tool (by giving appropriate feedback) so that it can recalculate the ranking list, hoping to reach the faulty statement sooner.

The process begins by calculating an initial ranking list using any SBFL approach. The developer will then analyze the elements and give one of four possible answers: the fault is found; the element is not faulty, neither its context; the element is not faulty but the fault is somewhere within the context; or neither of the above. If the fault is found, the process will terminate. If the fault is not found, the tool will readjust the ranking and show the newly formed list so the process can continue. Finally, if none of the other options apply, the developer will continue looking at the next elements on the list. Figure 2.6 shows an example of a menu with all the options mentioned above.



0,3241	init	org.eclipse.se...	Control	H:\project\iFL...	410	3 methods
0,321			deChunkL...	H:\project\iFL...	82	2 methods
0,315			thodScore...	H:\project\iFL...	702	8 methods
0,314			mpleView	H:\project\iFL...	4658	9 methods
0,310	getDescription	org.eclipse.se...	Option	H:\project\iFL...	658	7 methods
0,3051	getImage	org.eclipse.se...	ViewLabelProv...	H:\project\iFL...	943	2 methods

This is faulty (terminal choice)
Neither this, nor its context are faulty
This is not faulty
Navigate to selected
Navigate to context

Figure 2.6: *iFL with a suspicious items list.*

2.1.4 VIDA

VIDA [30] is an Eclipse plug-in for Java programs with JUnit test cases. It follows a testing-based fault-localization approach which measures a statement's suspiciousness based on execution information. The program's statements are then ranked according to their suspicions. The suspicion comes from failed test cases, i.e., a statement's involvement in failed test cases imply a higher suspicion. After having a list with the highest-ranking suspicious statements, the developer has to set a breakpoint on one of them. Then, based on whether the developer's chosen breakpoint was faulty or not, VIDA will re-examine the suspicions and provide a new list. This may impact the effectiveness of the tool, since it relies on the developer's estimation of suspicious statements.

Figure 2.7 shows an example of the program outline generated by VIDA. It uses long lines (in the middle of the window) in different color to show statements with various suspicions. A black line indicates a statement with large suspicion, whereas a light grey line denotes a statement with small suspicion. The breakpoint candidates are addressed by blue lines. On the left boundary of the window, the short lines denote the position of the existing breakpoints. The color of these short lines denote the programmer's previous estimation on these breakpoints. Specifically, VIDA uses a red line to denote that the variables' values at the corresponding breakpoint have been estimated to be wrong by the programmer, a green line denotes the variables' values at the corresponding breakpoint have been estimated to be correct, and a yellow line denotes that the programmer is not confident with their estimation at the breakpoint.

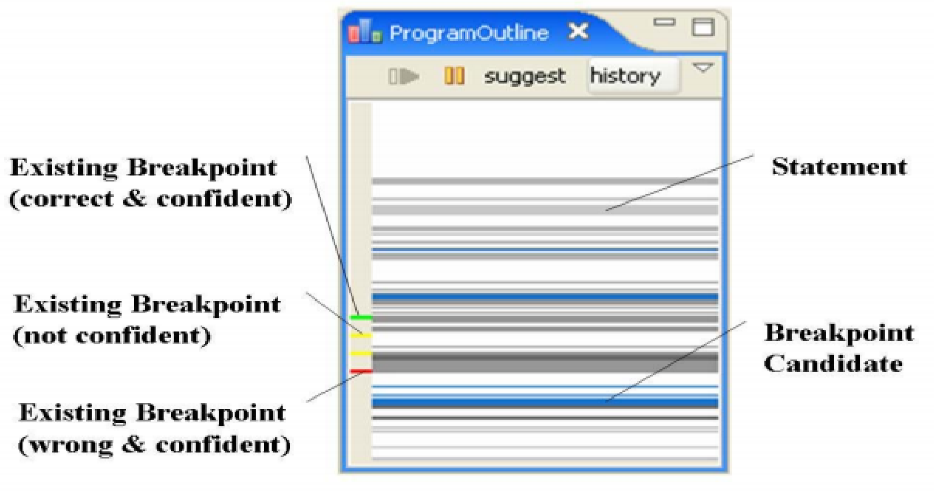

ProgramOutline
suggest
history
Existing Breakpoint (correct & confident)
Statement
Existing Breakpoint (not confident)
Breakpoint Candidate
Existing Breakpoint (wrong & confident)

Figure 2.7: *VIDA's program outline.*

2.2 Slice-Based Techniques

Program slicing [10, 11] is a technique based on the idea of reducing the program into a form that contains only its relevant parts, i.e., by removing parts that do not affect the program in any way, the resulting slice will still have the same behavior as the original program. Static slicing [10] proves to be advantageous when searching for bugs, given that a smaller domain implies less places to search. Dicing, introduced by Weiser and Lyle in [31], is defined as the difference between a variable's static slices. This was used to further reduce the search domain for possible locations of a fault. However, one problem that arises from this technique comes from pointer variables. These variables can make data-flow analysis inefficient because dereferencing pointer variables introduces large data sets that need to be stored. Another problem is that the slice for a given variable contains all the executable statements that could affect the variable, and as a result, it might generate a dice with statements that should not be included. To solve this problem, dynamic slicing [32, 33] is necessary instead of static slicing.

Dynamic slicing differs from static slicing in the sense that it is entirely defined on the basis of a computation. With it comes two main advantages, namely with dynamic data structures and arrays which can be handled more precisely, thus reducing the size of a slice.

Alternatively, execution slicing [34] can be used to locate program bugs. It focuses on statements executed by a specific input, as opposed to statements that could affect the variable based on any inputs. Yet, despite having all these different techniques, there is also the possibility of the bug not even being present in the dice. On the offchance that the bug is actually in the dice in question, there may be too much code in the dice that needs to be examined, since slices are usually lengthy and hard to understand.

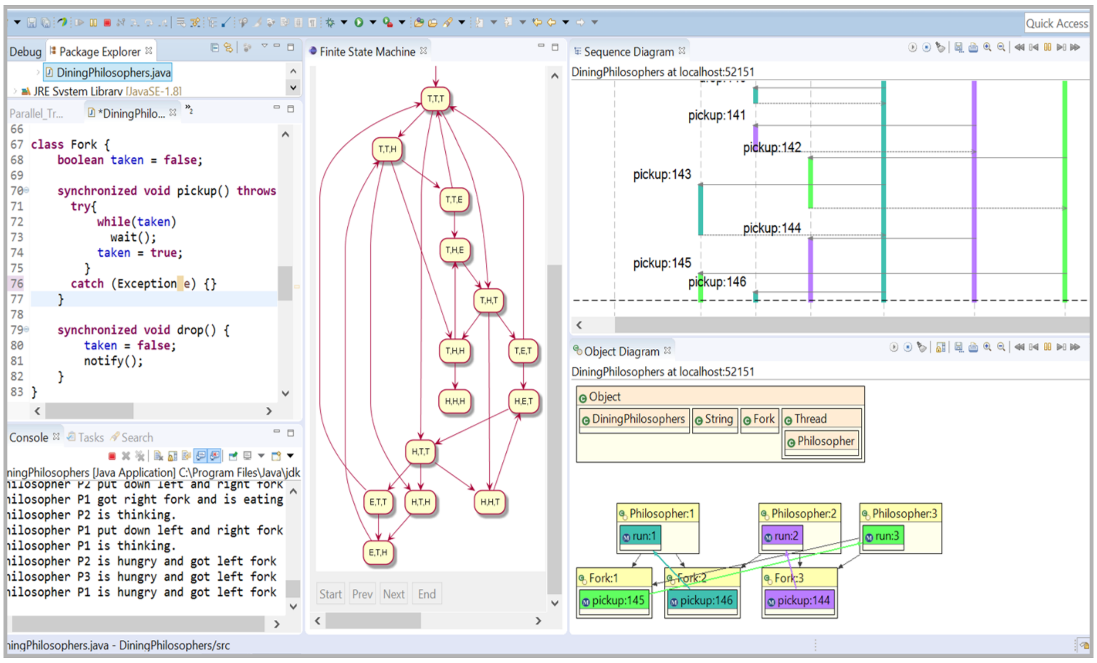
2.2.1 JIVE

JIVE [35] is an interactive execution environment for Eclipse that provides visualizations of Java program execution at different levels of granularity. It facilitates program understanding and interactive debugging, featuring multiple, customizable views of object structure; representation of execution history via sequence diagrams; interactive queries on runtime behavior; forward and reverse interactive execution.

It supports a declarative approach to debugging by providing an extensible set of queries over a entire program's execution history, not just over the stack of outstanding calls. Queries are formulated using the source code or the diagrams, and the results are shown in a tabular format and also as diagram annotations.

JIVE allows the programmer to step backwards during a program's execution. This can save a great deal of time and effort since the usual scenario would be to re-execute the program until the point of error. The programmer can also jump directly back to any previous point in the execution history to observe the object diagram at that point.

Lastly, JIVE depicts both the runtime state and call history of a program in a visual manner, as shown in Figure 2.8. The runtime state is visualized as an enhanced object diagram, showing object structure as well as method activations in their proper object contexts. The call history is depicted as an extended sequence diagram, with each execution thread shown in a different color, clarifying the object interactions that occur at runtime. The diagrams are scalable and can be filtered to show only information pertinent to the task at hand. JIVE also supports a state diagram view, which is useful for programs that exhibit a repetitive behavior.



```

class Fork {
    boolean taken = false;

    synchronized void pickup() throws
      try{
          while(taken)
            wait();
          taken = true;
        }
      catch (Exception e) {}
    }

    synchronized void drop() {
        taken = false;
        notify();
    }
}
```

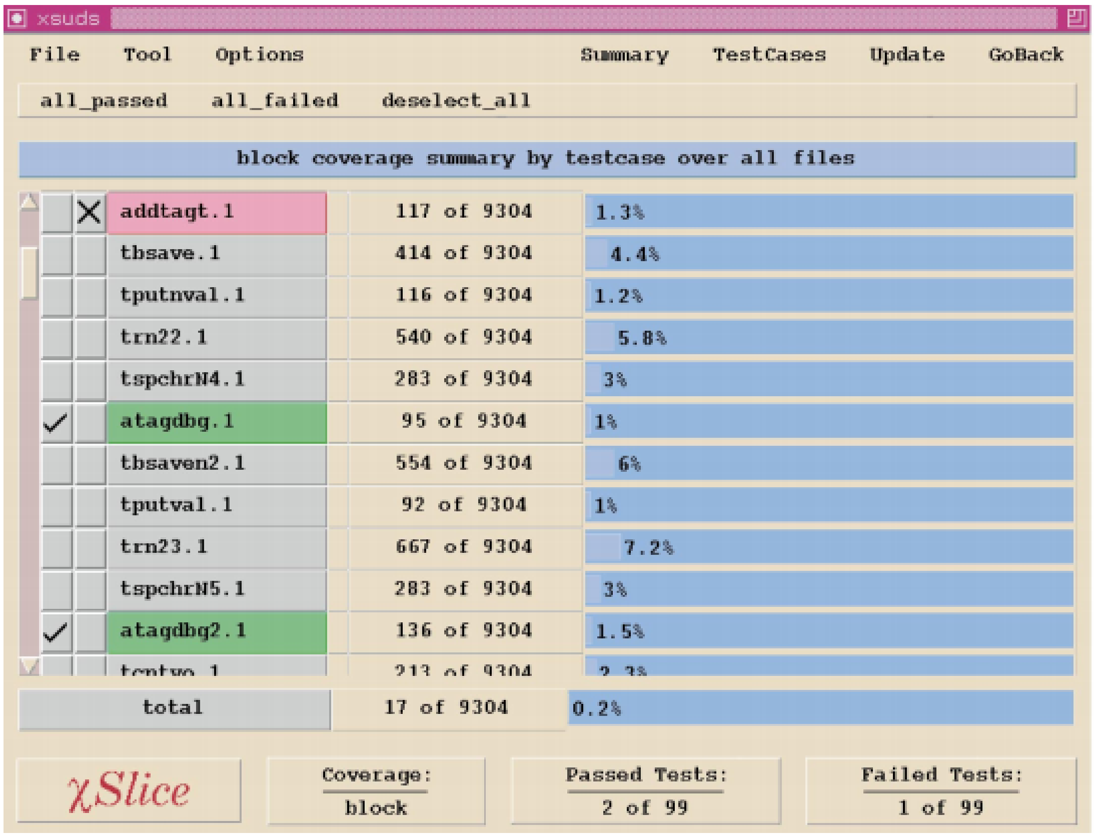
```
ningPhilosophers [Java Application] C:\Program Files\Java\jdk
hilosopher P2 put down left and right fork
hilosopher P1 got right fork and is eating
hilosopher P2 is thinking.
hilosopher P1 put down left and right fork
hilosopher P1 is thinking.
hilosopher P2 is hungry and got left fork
hilosopher P3 is hungry and got left fork
hilosopher P1 is hungry and got left fork
```

Figure 2.8: *JIVE visualizations.*

2.2.2 χSlice

χSuds tool suite [36] is a software understanding and diagnosis system, containing various techniques that can be used for program understanding, debugging and testing for C and C++ programs. Similar to other systems that monitor test coverage, χSuds first creates a representation of the program's control graph, laying out its structure. As the user runs various test cases, it stores an execution trace, which records how many times each test has exercised a particular software component (function, block, decision, or dataflow association). An effective use of χSuds requires only for the user to have a basic understanding of the program's features and be able to identify the features each test case exercises.

Among the several techniques present in the tool suite, χSlice is a debugging tool which uses a program's execution slice to help locate errors in the code. Assuming all previously available test cases in a project are bug-free, when adding new test cases it will only analyze the code in the new slices, and not the old. After running the test cases and the error is presumably detected, χSlice will highlight in red the code section containing the bug. Like most tools, it does not give a clear overview of the project nor an easy way to navigate through the errors found. Figure 2.9 and Figure 2.10 show χSlice's interface.



block coverage summary by testcase over all files

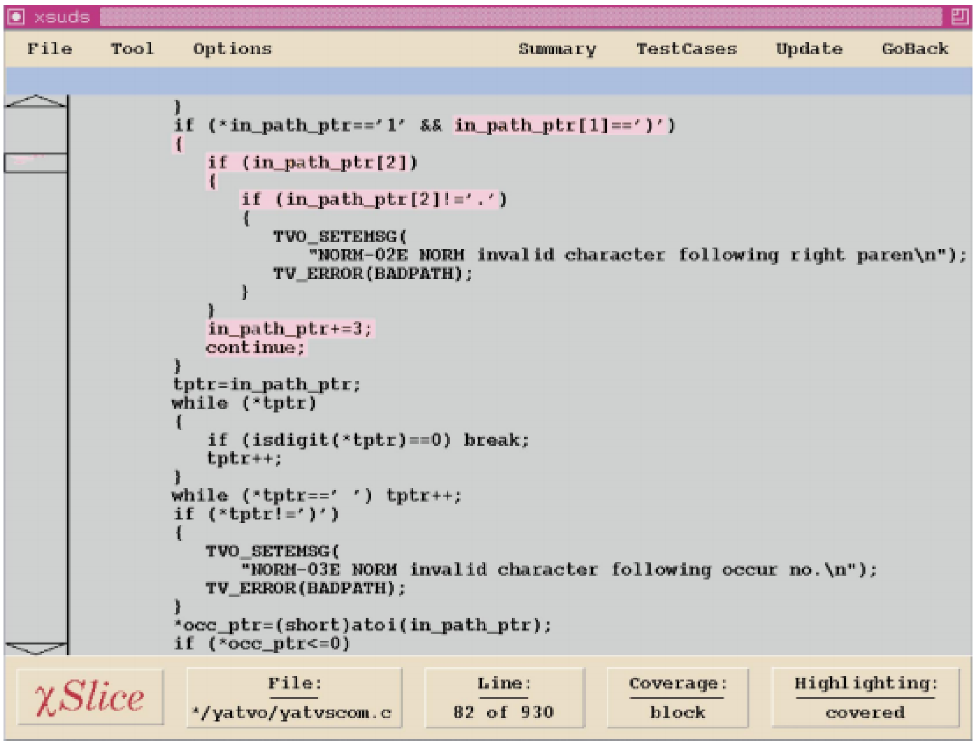
	✗	addtagt.1	117 of 9304	1.3%
		tbsave.1	414 of 9304	4.4%
		tputnval.1	116 of 9304	1.2%
		trn22.1	540 of 9304	5.8%
		tspchrN4.1	283 of 9304	3%
✓		atagdbg.1	95 of 9304	1%
		tbsaven2.1	554 of 9304	6%
		tputval.1	92 of 9304	1%
		trn23.1	667 of 9304	7.2%
		tspchrN5.1	283 of 9304	3%
✓		atagdbg2.1	136 of 9304	1.5%
		tcntwo.1	213 of 9304	2.3%
total			17 of 9304	0.2%

Coverage: block

Passed Tests: 2 of 99

Failed Tests: 1 of 99

Figure 2.9: *eXVantage tool displaying test cases.*



```
}
if (*in_path_ptr=='1' && in_path_ptr[1]==')')
{
   if (in_path_ptr[2])
   {
      if (in_path_ptr[2]!='.')
      {
         TVO_SETEMSG(
            "NORM-02E NORM invalid character following right paren\n");
         TV_ERROR(BADPATH);
      }
   }
   in_path_ptr+=3;
   continue;
}
tptr=in_path_ptr;
while (*tptr)
{
   if (isdigit(*tptr)==0) break;
   tptr++;
}
while (*tptr==' ') tptr++;
if (*tptr!=')')
{
   TVO_SETEMSG(
      "NORM-03E NORM invalid character following occur no.\n");
   TV_ERROR(BADPATH);
}
*occ_ptr=(short)atoi(in_path_ptr);
if (*occ_ptr<=0)
```

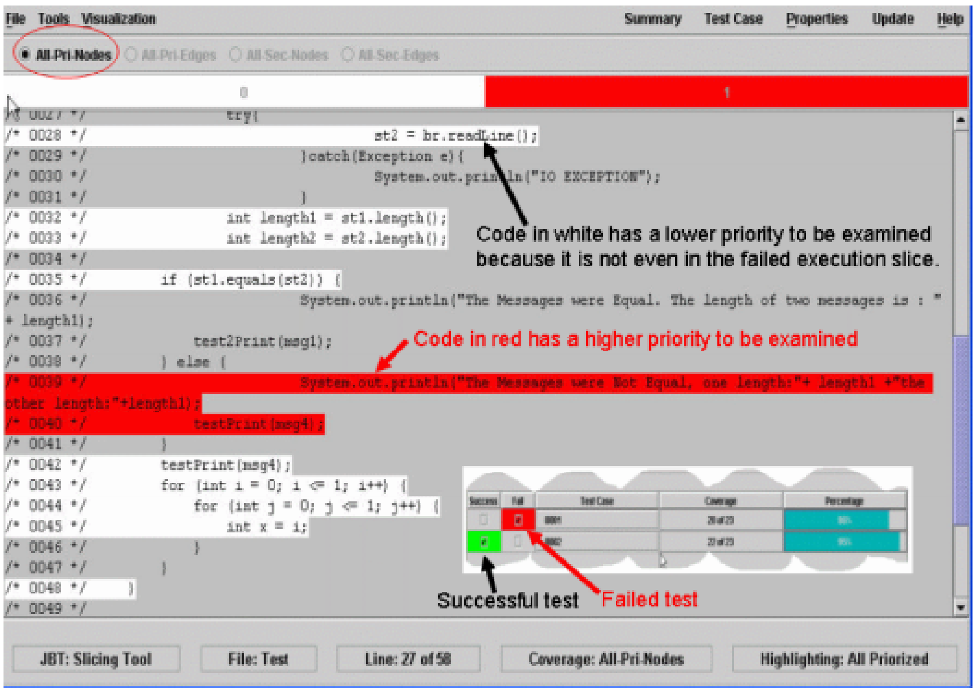
Figure 2.10: *eXVantage tool displaying source code.*

2.2.3 eXVantage

eXVantage [37] is a tool suite for code coverage testing, debugging, performance profiling for Java programs. Their approach is based on the idea that code executed in successful test cases are less likely to contain any fault, and code that is repeatedly executed in failed test cases is more likely to contain the fault in question. Considering this, the starting point for locating the fault is the execution slice from the failed tests, i.e., the execution dice obtained from subtracting the successful slice from the failed slice. Figure 2.11 shows the tool displaying code in red, which indicates high priority, and also code in white, indicating lower priority since it is not even in the failed execution slice.

2.3 Model-Based Techniques

Model-based diagnosis [14] consists of creating a model that can be used to represent the behavior of a system, typically coming from its description. This model can be used as a point of reference in what concerns the system's correct behavior, that is, by observing the system's actual behavior, any discrepancies obtained from comparing it to the model's behavior may indicate a potential fault. Still, this comes from the assumption that the model is without errors, indicating a correct model of the system. When talking about model-based software fault localization, the roles are reversed. The model reflects the behavior of the incorrect program, while test cases defined by the developer/tester indicate the desired results. The differences between them are used to extract model elements that presumably behaved differently than intended, and thus contain the fault that explains the misbehavior.


File Tools Visualization
Summary Test Case Properties Update Help
All-Pri-Nodes All-Pri-Edges All-Sec-Nodes All-Sec-Edges
0
1
Code in white has a lower priority to be examined because it is not even in the failed execution slice.
Code in red has a higher priority to be examined
Successful test
Failed test
JBT: Slicing Tool
File: Test
Line: 27 of 58
Coverage: All-Pri-Nodes
Highlighting: All Priorized

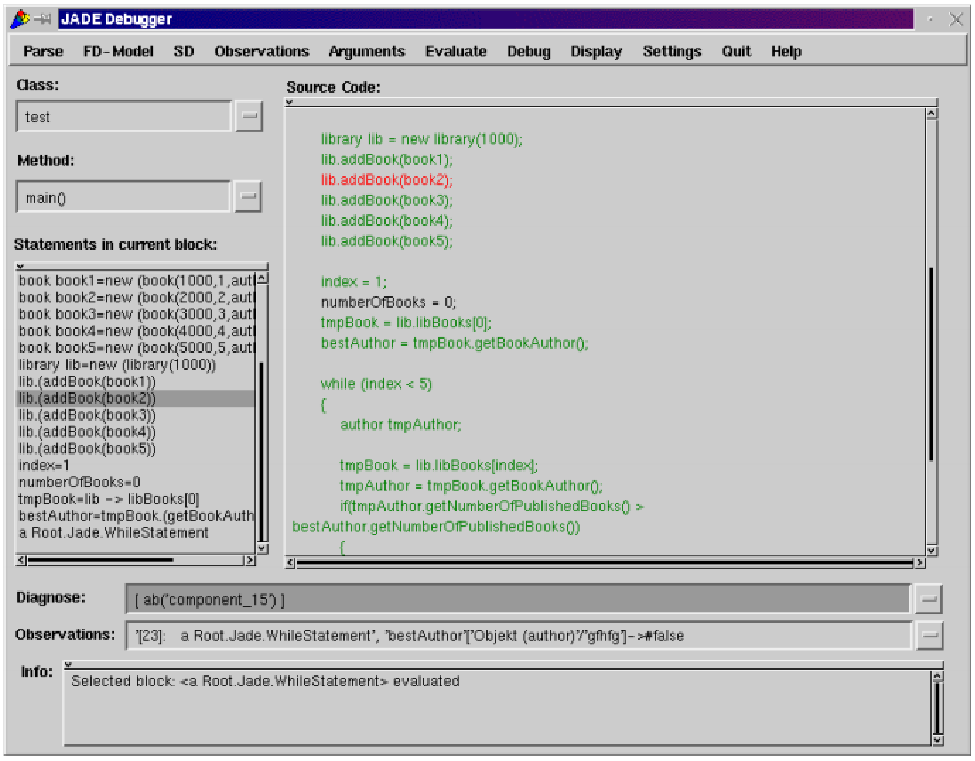
```
                    try{
                                st2 = br.readLine();
                          }catch(Exception e){
                                System.out.println("IO EXCEPTION");
                          }
                    int length1 = st1.length();
                    int length2 = st2.length();

            if (st1.equals(st2)) {
                          System.out.println("The Messages were Equal. The length of two messages is : "
+ length1);
                test2Print(msg1);
            } else {
                          System.out.println("The Messages were Not Equal, one length:"+ length1 +"the
other length:"+length1);
                testPrint(msg4);
            }
            testPrint(msg4);
            for (int i = 0; i <= 1; i++) {
                for (int j = 0; j <= 1; j++) {
                    int x = i;
                }
            }
        }
```

Figure 2.11: *eXVantage tool displaying source code.*

2.3.1 Jade

Jade [38] is a model-based debugger for Java programs. It uses artificial intelligence to create models derived from the source code of the program without any additional specifications, except for Java semantics. Using the method previously described, the Java program in case is compiled to an internal representation, and along with a set of model fragments (logical description of parts of a model) are converted to logical models for diagnosis by a converter. After building it, the model is used in conjunction with test cases by the diagnosis engine to obtain the diagnoses indicating where the possible faults are located, and even repair suggestions. The user interface presents the outcomes produced by the diagnosis engine to guide the user through the debugging process. This may include stepping through either blocks or bodies of functions, depending on the results obtained.

Debugging starts by loading a method into Jade and performing a diagnosis step. If it results in a single diagnosis, it checks whether the statement is a loop or a conditional statement. If it matches any of those cases, then the bug must be inside either the condition expression or the block it contains. If the condition is correct, the debugger will load the block to proceed with the debugging. Otherwise, if the condition or loop statement is wrong, then the debugging is finished as the error has been found. When the result of the diagnosis is not a single diagnosis, the debugger will discriminate between the candidates using measurement selection (selection of variables that should be examined at a specific location within the program). This is achieved by asking the user for values of variables for given locations inside the program, and depending on those answers, diagnoses can be removed until only one diagnosis remains. Figure 2.12 shows Jade's user interface.



Class: test

Method: main()

Statements in current block:

```
book book1=new (book(1000,1,aut
book book2=new (book(2000,2,aut
book book3=new (book(3000,3,aut
book book4=new (book(4000,4,aut
book book5=new (book(5000,5,aut
library lib=new (library(1000))
lib.(addBook(book1))
lib.(addBook(book2))
lib.(addBook(book3))
lib.(addBook(book4))
lib.(addBook(book5))
index=1
numberOfBooks=0
tmpBook=lib -> libBooks[0]
bestAuthor=tmpBook.(getBookAuth
a Root.Jade.WhileStatement
```

Source Code:

```
    library lib = new library(1000);
    lib.addBook(book1);
    lib.addBook(book2);
    lib.addBook(book3);
    lib.addBook(book4);
    lib.addBook(book5);

    index = 1;
    numberOfBooks = 0;
    tmpBook = lib.libBooks[0];
    bestAuthor = tmpBook.getBookAuthor();

    while (index < 5)
    {
      author tmpAuthor;

      tmpBook = lib.libBooks[index];
      tmpAuthor = tmpBook.getBookAuthor();
      if(tmpAuthor.getNumberOfPublishedBooks() >
bestAuthor.getNumberOfPublishedBooks())
      {
```

Diagnose: [ab('component_15')]

Observations: '[23]: a Root.Jade.WhileStatement', 'bestAuthor'['Objekt (author)'/'gfhfg']->#false

Info: Selected block: <a Root.Jade.WhileStatement> evaluated

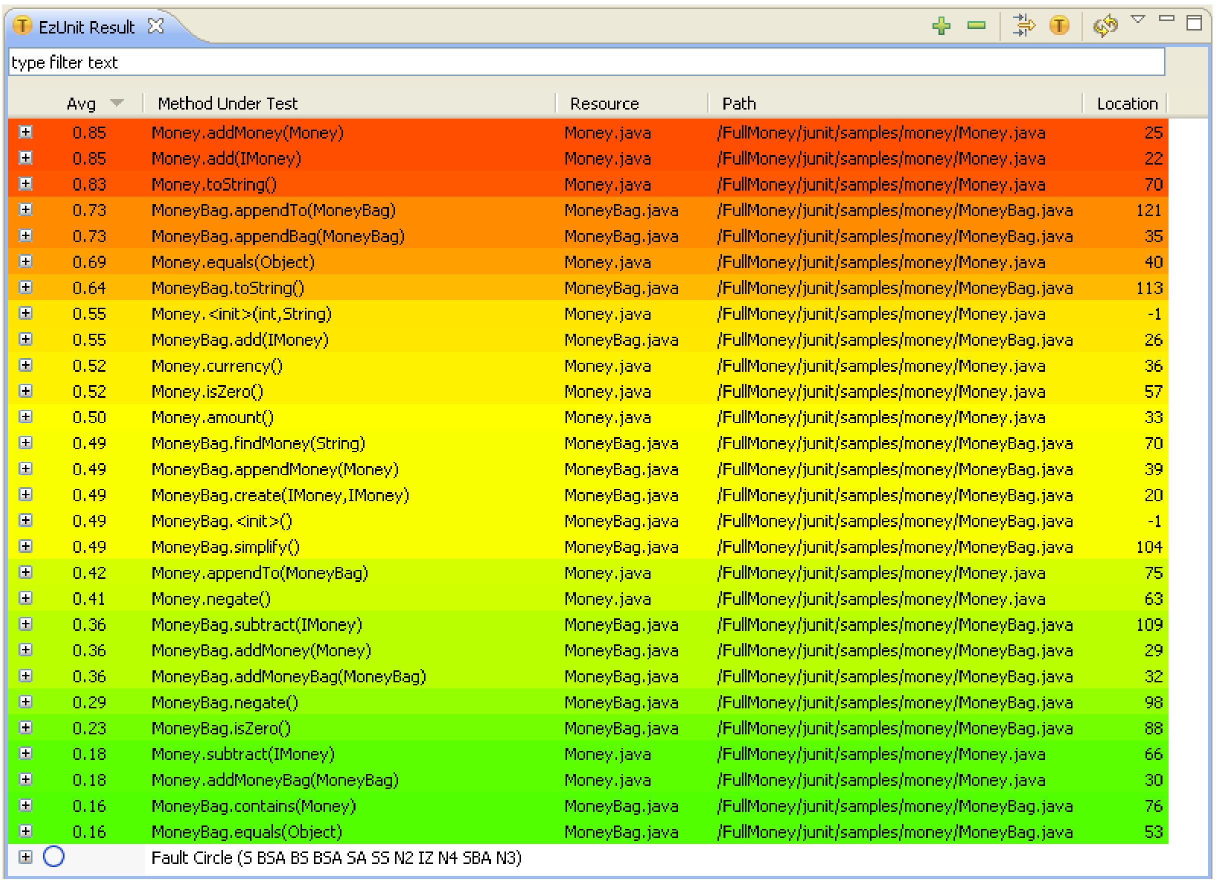
Figure 2.12: *Jade user interface.*

2.4 Miscellaneous Techniques

There are many tools which do not fit into any of the categories mentioned above, due to their focus being on specific testing scenarios, or because it contains the combination of many of those techniques in a single tool.

2.4.1 EzUnit

EzUnit is a tool which integrates with Eclipse and links JUnit test failures to locations in the source code. It creates a list with several code blocks and their failure probability, where each line is highlighted with a color that represents the severity of the probability (e.g., green for low probability or red for high probability). This can be seen in Figure 2.13. This tool also provides a graph view of all the methods calls in a test case, since possible fault locations are usually restricted to methods called by one or more failed unit tests. Figure 2.14 exemplifies this graph. Finally, it can also mark each line with information about the failure probability of that code block, as seen in Figure 2.15. Overall, EzUnit is one of the most complete graphical debuggers out there, being completely integrated with the IDE, and providing many options to help the developer.


EzUnit Result

type filter text

Avg	Method Under Test	Resource	Path	Location
0.85	Money.addMoney(Money)	Money.java	/FullMoney/junit/samples/money/Money.java	25
0.85	Money.add(IMoney)	Money.java	/FullMoney/junit/samples/money/Money.java	22
0.83	Money.toString()	Money.java	/FullMoney/junit/samples/money/Money.java	70
0.73	MoneyBag.appendTo(MoneyBag)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	121
0.73	MoneyBag.appendBag(MoneyBag)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	35
0.69	Money.equals(Object)	Money.java	/FullMoney/junit/samples/money/Money.java	40
0.64	MoneyBag.toString()	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	113
0.55	Money.<init>(int,String)	Money.java	/FullMoney/junit/samples/money/Money.java	-1
0.55	MoneyBag.add(IMoney)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	26
0.52	Money.currency()	Money.java	/FullMoney/junit/samples/money/Money.java	36
0.52	Money.isZero()	Money.java	/FullMoney/junit/samples/money/Money.java	57
0.50	Money.amount()	Money.java	/FullMoney/junit/samples/money/Money.java	33
0.49	MoneyBag.findMoney(String)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	70
0.49	MoneyBag.appendMoney(Money)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	39
0.49	MoneyBag.create(IMoney,IMoney)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	20
0.49	MoneyBag.<init>()	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	-1
0.49	MoneyBag.simplify()	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	104
0.42	Money.appendTo(MoneyBag)	Money.java	/FullMoney/junit/samples/money/Money.java	75
0.41	Money.negate()	Money.java	/FullMoney/junit/samples/money/Money.java	63
0.36	MoneyBag.subtract(IMoney)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	109
0.36	MoneyBag.addMoney(Money)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	29
0.36	MoneyBag.addMoneyBag(MoneyBag)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	32
0.29	MoneyBag.negate()	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	98
0.23	MoneyBag.isZero()	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	88
0.18	Money.subtract(IMoney)	Money.java	/FullMoney/junit/samples/money/Money.java	66
0.18	Money.addMoneyBag(MoneyBag)	Money.java	/FullMoney/junit/samples/money/Money.java	30
0.16	MoneyBag.contains(Money)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	76
0.16	MoneyBag.equals(Object)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	53
	Fault Circle (S BSA BS BSA SA SS N2 IZ N4 SBA N3)			

Figure 2.13: *EzUnit list.*

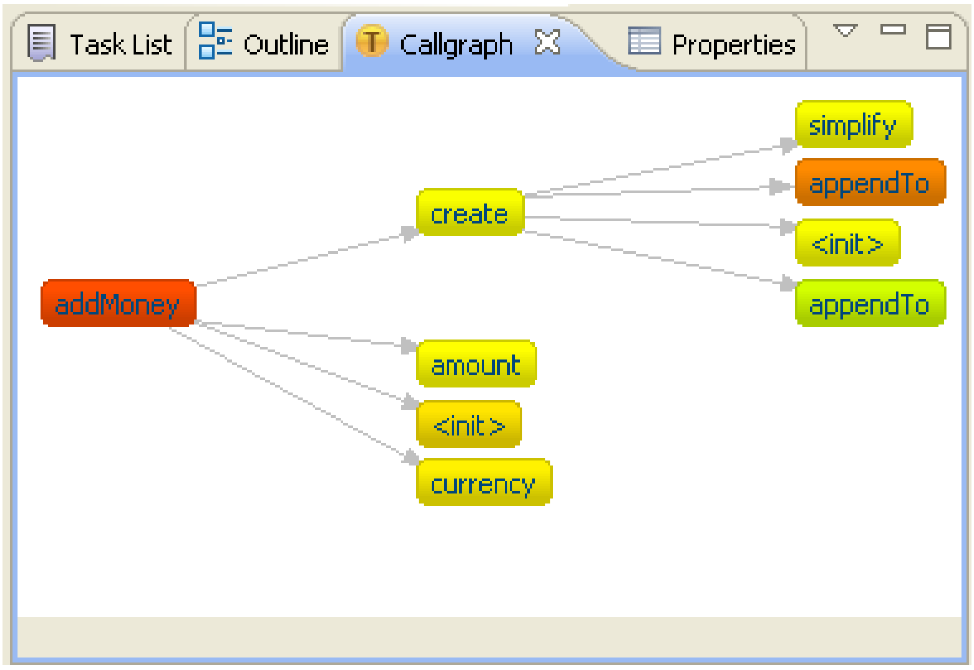
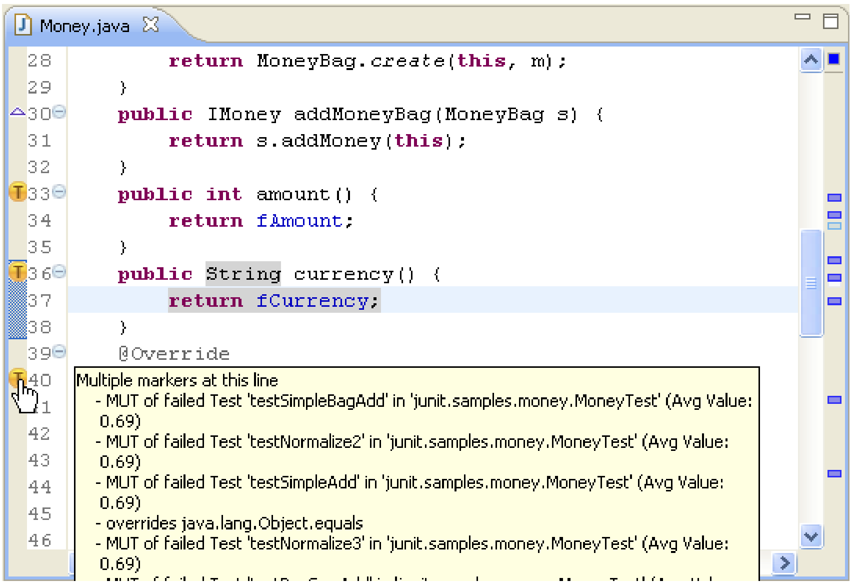


Figure 2.14: *EzUnit graph view.*



```
        return MoneyBag.create(this, m);
    }
    public IMoney addMoneyBag(MoneyBag s) {
        return s.addMoney(this);
    }
    public int amount() {
        return fAmount;
    }
    public String currency() {
        return fCurrency;
    }
    @Override
```

Multiple markers at this line
- MUT of failed Test 'testSimpleBagAdd' in 'junit.samples.money.MoneyTest' (Avg Value: 0.69)
- MUT of failed Test 'testNormalize2' in 'junit.samples.money.MoneyTest' (Avg Value: 0.69)
- MUT of failed Test 'testSimpleAdd' in 'junit.samples.money.MoneyTest' (Avg Value: 0.69)
- overrides java.lang.Object.equals
- MUT of failed Test 'testNormalize3' in 'junit.samples.money.MoneyTest' (Avg Value: 0.69)

Figure 2.15: *EzUnit integration with the code editor.*

2.4.2 Whyline

Whyline [39] is a standalone interactive tool for Java programs. Instead of speculating the whereabouts of the problem, it allows the developer to select questions about the program, such as *why did* or *why didn't* something happen, to which the tool provides possible explanations. Whyline achieves this by recording the program's execution traces (e.g., class files executions, sequence of events occurred in each thread, etc.), along with static analysis and call graphs to determine explanations for each problem in the code. Figures 2.16 and 2.17 illustrate Whyline's functionalities.

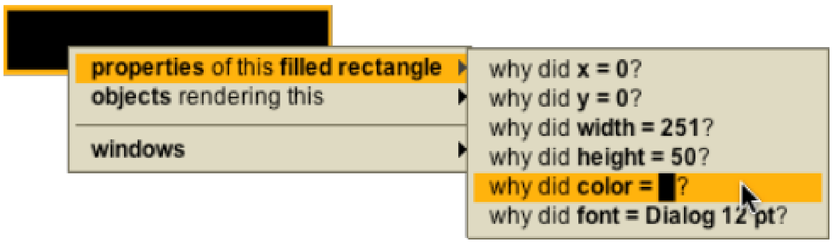
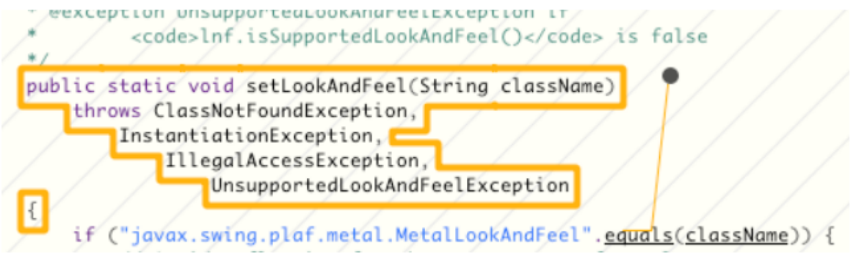


Figure 2.16: *Whyline interface showing questions about an object's properties.*



```
 * @exception UnsupportedLookAndFeelException if
 *        <code>lnf.isSupportedLookAndFeel()</code> is false
 */
public static void setLookAndFeel(String className)
    throws ClassNotFoundException,
        InstantiationException,
            IllegalAccessException,
                UnsupportedLookAndFeelException
{
    if ("javax.swing.plaf.metal.MetalLookAndFeel".equals(className)) {
```

Figure 2.17: *Source code highlighting.*

2.5 Tools Overview

To better understand each of the properties of the tools that were previously presented, a comparative table can be seen in Table 2.1. Each individual debugging tool has its own set of advantages, but most of them lack a capability that we deem necessary for the debugging process. Only GZoltar ticks all of the marks, which is why we felt the need to further advance its integration with other IDEs/code editors.

Although each tool provides a unique feature relying on its visual aide, most of them lack a hierarchical visualization. This means that the user is unable to have an overview of the project being tested, making it more difficult to clearly spot any faulty areas of the code or identifying any suspicious module. Regardless, most tools provide some way of showing failure probability, which is to say, indicating to the user whether a line or segment of the code is suspicious. Another useful feature is to indicate how each component is related to one another. If it is known to the user that a certain module is faulty, then visualizing its relations to other modules might certainly provide more insight. Navigating through these modules in the visualizations is also convenient for the developer, as it is an easy way to quickly verify each module for faults, yet only half the tools contains both these features. Lastly, the ideal place for the developer to correct his faults would be in the same place where they find the fault. However, much like the previous features, IDE integration is adopted only by half of them.

In conclusion, despite all of the tools providing graphical output, most of them lack the powerful visualization that could help the developer deal with large projects, either by providing an easy way to navigate through the large amount of information, or by showcasing the project in a clear way. Even the tools with great visualizations already become a hassle to work with if they are not integrated in an IDE, since it forces the developer to constantly switch between applications during the debugging process. All of this will contribute to spending even more time with debugging, which is why we seek to adapt GZoltar in a new code editor, to bridge the gap between functionality and IDE/editor.

	Hierarchical View	**Fail Probability**	**Components Relations**	**Navigation**	**IDE Integration**
Tarantula	-	✓	-	-	-
Jaguar	-	✓	-	-	✓
iFL	-	✓	-	-	✓
VIDA	-	✓	-	-	✓
JIVE	-	-	✓	✓	✓
xSlice	-	✓	-	-	-
eXVantage	-	✓	-	✓	-
Jade	-	✓	-	-	-
EzUnit	-	✓	✓	✓	✓
Whyline	-	-	-	-	-
GZoltar	✓	✓	✓	✓	✓

Table 2.1: Current Graphical Tools Comparison

Chapter 3

Implementation

The extension's architecture is presented in Figure 3.1. The solution comprises of three main components: the GZoltar command-line interface, which executes the test cases' analysis and returns the coverage results; the back end, responsible for accepting user requests and interacting with GZoltar to obtain the analysis results. These results are then used to build the webview; lastly, the webview, a panel that can render HTML inside VS Code, presents the test cases' results in the form of three different hierarchical charts to the user. When the extension is activated, the user can begin using it by requesting the back end for an analysis of the open Java project(s), granted that certain requirements are met (more details about this can be found in 3.1.1). The back end will then send the request to the GZoltar CLI to execute the analysis and obtain the results. Having those in hand, it will also instrument all classes to obtain coverage results. Afterwards, these are sent to the back end so that they can be processed to create a webview.

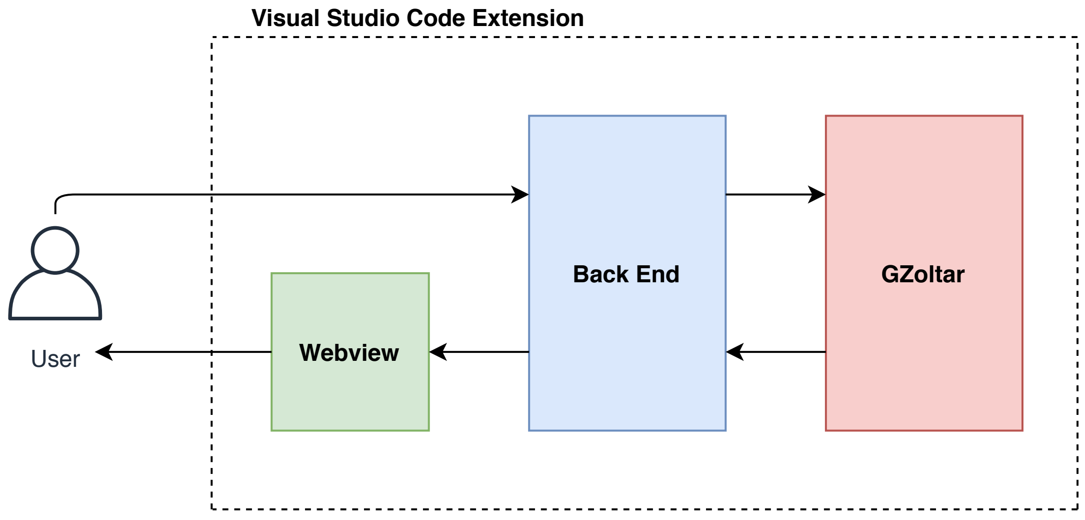
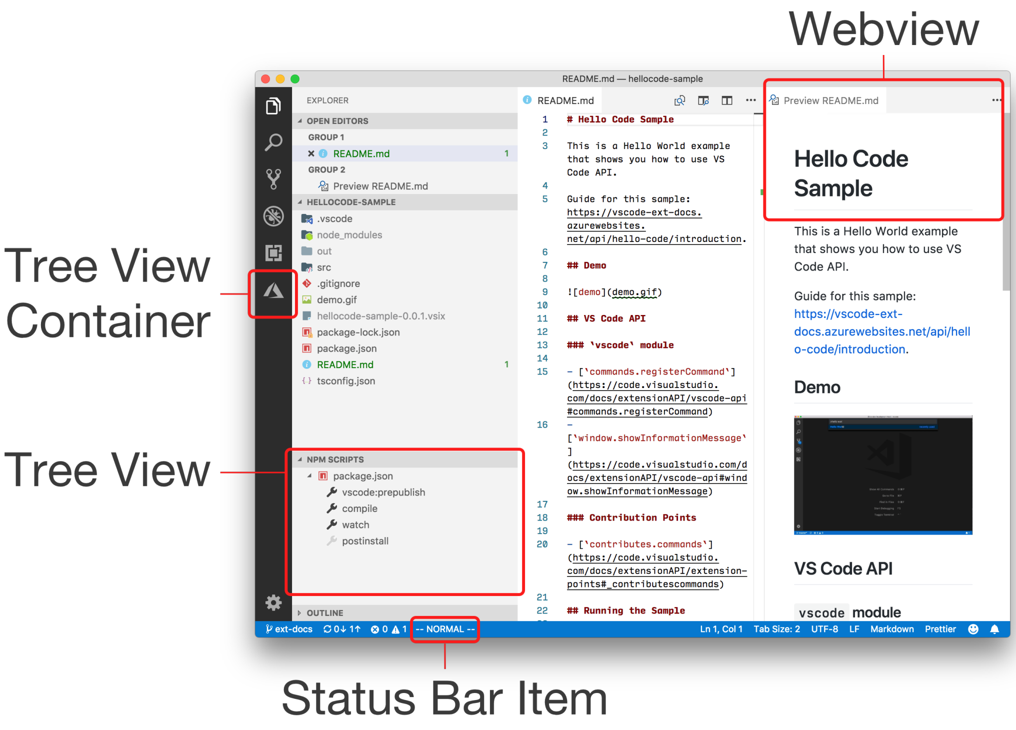


Figure 3.1: *Architecture.*

3.1 Building Extensions

VS Code is built using Electron [40], an open source framework developed by Github. It allows for the development of desktop GUI applications using web technologies, specifically Node.js and the Chromium rendering engine. This means that applications built with Electron are able to use HTML, CSS and JavaScript. The extensions themselves may be built using either JavaScript or TypeScript, which is a typed superset of the former. The API provided is quite extensible, with almost every part of VS Code being highly customizable. Figure 3.2 shows the different parts of its interface that can be customized. The tree view container allows for the addition of new icons, sitting along with the default view containers already added by VS Code. By clicking on the added icon, a tree view is opened with new views integrated into the extension. For each view situated in the tree view, an action with a specific behavior may be attributed. The status bar item can also be used to provide additional information regarding the extension. Lastly, a component called webview allows the extension to open tabs containing HTML, CSS and JavaScript code.



```
# Hello Code Sample

This is a Hello World example
that shows you how to use VS
Code API.

Guide for this sample:
https://vscode-ext-docs.
azurewebsites.
net/api/hello-code/introduction.

## Demo

![demo](demo.gif)

## VS Code API

### `vscode` module

- [`commands.registerCommand`]
(https://code.visualstudio.
com/docs/extensionAPI/vscode-api
#commands.registerCommand)
-
[`window.showInformationMessage`
]
(https://code.visualstudio.com/d
ocs/extensionAPI/vscode-api#wind
ow.showInformationMessage)

### Contribution Points

- [`contributes.commands`]
(https://code.visualstudio.
com/docs/extensionAPI/extension-
points#_contributescommands)

## Running the Sample
```

This is a Hello World example that shows you how to use VS Code API.

Guide for this sample: https://vscode-ext-docs.azurewebsites.net/api/hello-code/introduction.

Figure 3.2: *VS Code UI.*

VS Code extensions are created using two tools available in the Node.js package manager, Yeoman and the VS Code Extension Generator. Yeoman is an open source scaffolding tool for web applications. As such, it can be used to generate project templates, manage package dependencies, among other tasks. The VS Code Extension Generator is installed along with Yeoman, so that the generator can create a base folder structure containing a rough implementation of an extension. To start a new project, run the generator using Yeoman and it will prompt some questions regarding the extension, as seen in Figure 3.3.

```
yo code

# ? What type of extension do you want to create? New Extension (TypeScript)
# ? What's the name of your extension? HelloWorld
### Press <Enter> to choose default for all options below ###

# ? What's the identifier of your extension? helloworld
# ? What's the description of your extension? LEAVE BLANK
# ? Initialize a git repository? Yes
# ? Which package manager to use? npm
```

Figure 3.3: *Yeoman.*

Following that example, a folder called helloworld is created with the structure presented in Figure 3.4. The main files which are essential to understanding the extension are *package.json* (extension manifest) and *extension.ts* (entry file).

```
.
├── .vscode
│   ├── launch.json     // Config for launching and debugging the extension
│   └── tasks.json      // Config for build task that compiles TypeScript
├── .gitignore          // Ignore build output and node_modules
├── README.md           // Readable description of your extension's functionality
├── src
│   └── extension.ts    // Extension source code
├── package.json        // Extension manifest
├── tsconfig.json       // TypeScript configuration
```

Figure 3.4: *Extension folder structure.*

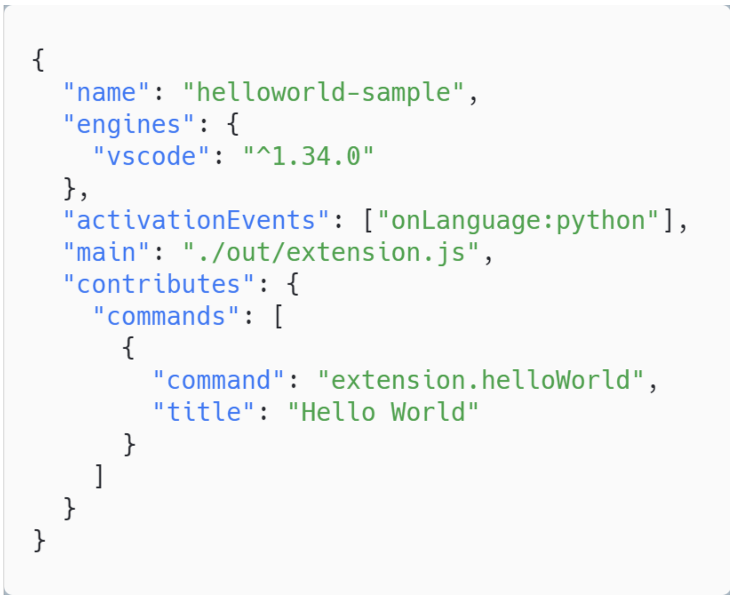
3.1.1 Extension Manifest

Each VS Code extension must have a package.json as its Extension Manifest. The package.json file contains a mix of Node.js fields such as scripts and dependencies and VS Code specific fields such as publisher, activationEvents and contributes. Some of the most important fields of the manifest are the following:

- *name* and *publisher*, since VS Code uses <*publisher.name*> as the unique identifier for the extension.
- *main* indicates the extension's entry point.
- *engines.vscode* specifies the minimum version of VS Code API that the extension depends on.
- *activationEvents* is a set of JSON declarations. The extension becomes activated when the activation event happens. Examples of activation events include: having a file of a certain programming language present in the folder (e.g., *"onLanguage:python"*); having a file that matches a pattern (e.g., *"workspaceContains:**/pom.xml"*); on start up, meaning that the extension will always be activated regardless of the projects open.

- *contributionPoints* are a set of JSON declarations that are made in the contributes field of the package.json Extension Manifest. The extension registers Contribution Points to extend various functionalities within VS Code, such as: commands, contributing the UI for a command consisting of a title and (optionally) an icon, category, and enabled state. It is also possible to specify when the command should be enabled (e.g., when a file with a certain type is open on the editor); language, contributing the definition of a language. This will introduce a new language or enrich the knowledge VS Code has about a language. This allows the developer to define a language identifier which could be used as an activation event, or even associate file name patterns.

To ensure a better user experience, it is best to be as specific as possible when creating activation events, since the user may already have several extensions already installed on their device, and adding another extension's initialization may hinder VS Code's start up. As such, the last activation event (on start up) must only be used when no other activation events combination works. Figure 3.5 shows an example of an extension manifest.



```
{
  "name": "helloworld-sample",
  "engines": {
    "vscode": "^1.34.0"
  },
  "activationEvents": ["onLanguage:python"],
  "main": "./out/extension.js",
  "contributes": {
    "commands": [
      {
        "command": "extension.helloWorld",
        "title": "Hello World"
      }
    ]
  }
}
```

Figure 3.5: *Extension manifest example.*

3.1.2 Entry File

The extension entry file exports two functions, *activate* and *deactivate*. *activate* is executed when the registered activation Events happen. This is the main entry point, and it is where any and all sorts of preparation for the extension must happen. In this method, it is possible to register commands indicated in the manifest by assigning behaviors to them. We can also create and update tree views, update the status bar, and basically control any section of the extension. *deactivate* allows us to clean up before the extension becomes deactivated. For many extensions, explicit cleanup may not be required, and the *deactivate* method can be removed. However, if an extension needs to perform an operation when VS Code is shutting down or the extension is disabled or uninstalled, this is the correct method to do so.

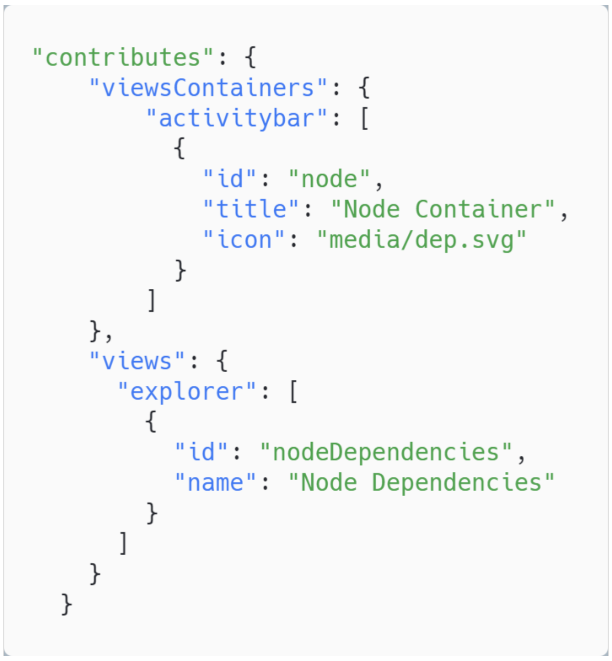
3.1.3 Tree View Container

VS Code's Tree View API allows extensions to show content in the sidebar in Visual Studio Code. This content is structured as a tree and conforms to the style of the built-in views of VS Code. In order to add a treeview to the extension, the following steps are required: contribute the treeview in the *package.json* file, create a *TreeDataProvider*, and register the *TreeDataProvider*.

package.json Contribution

As previously mentioned in 3.1.1, the *contributionPoints* allows us to register many functionalities in an extension, with the treeview being one of them. To do that, the *contributes.views* Contribution Point must be used. However, to create a treeview container, an entry to the *contributes.viewsContainers* is also required. This is necessary because the treeview itself is essentially an agglomerate of elements organized in a tree-like fashion, so they could be placed in any containers, including the default ones already included in VS Code.

A treeview container entry is composed of an id, the title for the container and an icon, which will appear on VS Code's sidebar. The treeview entry requires the id of the container it will appear on, an id for the treeview itself and a name too. Figure 3.6 shows an example of a container in the extension manifest.



```
"contributes": {
    "viewsContainers": {
        "activitybar": [
          {
            "id": "node",
            "title": "Node Container",
            "icon": "media/dep.svg"
          }
        ]
    },
    "views": {
      "explorer": [
        {
          "id": "nodeDependencies",
          "name": "Node Dependencies"
        }
      ]
    }
  }
```

Figure 3.6: *Treeview container example.*

Tree Data Provider

VS Code needs data to display in the newly created view, so next we must provide data to the view that was registered. Firstly, a *TreeDataProvider* must be implemented. Providers are always of a generic type T, which needs to extend from the class *TreeItem*. A *TreeItem* represents a treeview item down to its core elements, with the most relevant fields being the label, which is what is shown in the tree, and the collapsible state, meaning that if this item has any children, it can be opened or closed to reveal them. As such, a provider needs to be of a certain type of *TreeItem* to be fully implemented. There are two methods in a *TreeDataProvider* that are required:

- *getChildren(element?: T): ProviderResult<T[]>* - this method returns the children for the given element, or root if no element is passed. Basically, when a treeview is created it will be empty at first, and as such, the given element will be nonexistent. A verification to this case must be made, so that we can appropriately decide what children are to be returned. On the first scenario where there are no elements, we return the ones which will populate the tree initially. Then, when there are actual items, if the element requires children (in the context of the extension and its underlying business logic), then we must return them. Depending on the case, some elements may or may not have children, so we must always verify that we are assigning children to the correct elements.
- *getTreeItem(element: T): TreeItem | Thenable<TreeItem>* - this returns the UI representation of the element that gets displayed in the view. If no changes are needed to be made regarding the element, then this method simply returns the element received.

Registering the TreeDataProvider

The final step is to register the previously created data provider to the view. This can be achieved in the following two ways:

- *vscode.window.registerTreeDataProvider* - registers the tree data provider by providing the data provider and the view id registered in the manifest file.
- *vscode.window.createTreeView* - create the treeview by providing the registered view id and the data provider. It is similar to the method described above, with the difference being that this will give access to the treeview itself, which we can use for performing other view operations.

Updating Tree View Content

Once created, a treeview's items will remain static unless we add the capability to update them. This can be done by using the *onDidChangeTreeData* event in the provider.

- *onDidChangeTreeData?: Event<T | undefined | null>* - the type T must be the same one that the provider uses. This is a function that represents an event to which you subscribe by calling it with a listener function as argument. This has to be paired with an event emitter that can be fired to signalize the event change.
- *EventEmitter<T | undefined>* - the type T must be the same one that the provider uses. An event emitter can be used to create and manage an event for others to subscribe to. One emitter always owns one event.

3.1.4 Status Bar Item

A status bar item is a status bar contribution that can show text and icons and run a command on click. This can be used to provide additional information about the extension, such as lengthy background work that is being executed to signify the users that the extension is still working, despite not being able to show results right away. The status bar item can be created in the following way:

- *vscode.window.createStatusBarItem* - creates a status bar item by indicating its alignment (left or right) and the priority. A higher value means the item should be shown more to the left.

3.1.5 Webview

The webview API allows extensions to create fully customizable views within VS Code. Webviews can be used to build complex user interfaces beyond what VS Code's native APIs support. A webview is similar to an HTML iframe within VS Code that the extension controls. A webview can render almost any HTML/CSS/JavaScript content in this frame, and it communicates with extensions using message passing. Despite the possibilities that webviews provide, they should be used sparingly and only when VS Code's native API is inadequate. They are resource heavy and run in a separate context from normal extensions, so if the functionality can only exist within VS Code and is important enough that the high resource cost is disregarded, then webviews are the correct tools to use. A webview can be created in the following two ways:

- *vscode.window.createWebviewPanel* - creates and shows a new webview panel by indicating: the view type (a string that identifies the type of the webview panel); the title; the show options (where to show the webview in the editor, e.g., on the active column, on a new column beside the active one, etc.); and the panel options, which contain settings for the panel. Examples include keeping the content even when the panel is no longer visible, enabling scripts, specify paths from which the webview can load local resources, among many others.

- *vscode.window.registerWebviewPanelSerializer* - registers a webview panel serializer, i.e., a webview that can automatically be restored when VS Code restarts. This is done by indicating the view type and a *WebviewPanelSerializer*. This is explained in more detail further ahead.

Scripts and Message Passing

Webviews can also run scripts, but JavaScript is disabled by default for security reasons. This can be easily re-enabled by passing in the *enableScripts: true* option in the panel options when creating a webview. An extension can send data to its webviews using *webview.postMessage()*. This method sends any JSON serializable data to the webview. The message is received inside the webview through the standard message event *window.addEventListener('message', listener)*. Webviews can also pass messages back to their extension. This is accomplished using the *postMessage* function on a special VS Code API object inside the webview. To access the VS Code API object, the method *acquireVsCodeApi* must be called inside the webview. This function can only be invoked once per session. The instance of the VS Code API returned by this method should be held onto, and hand it out to any other functions that wish to use it. To send a message to the extension the method *vscode.postMessage()* can be used to send JSON serializable data.

Persistence

Webviews that are created by the *createWebviewPanel* method are destroyed when the user closes them or when the *.dispose()* method is called. The contents of webviews however are created when the webview becomes visible and destroyed when the webview is moved into the background. Any state inside the webview will be lost when the webview is moved to a background tab. Scripts running inside a webview can use the *getState* and *setState* methods to save off and restore a JSON serializable state object. This state is persisted even if the webview content itself is destroyed when a webview panel becomes hidden. The state is destroyed when the webview panel is destroyed. By implementing a WebviewPanelSerializer, the webviews can be automatically restored when VS Code restarts. The serialization builds on *getState* and *setState*, and can only be enabled if the extension registers a *WebviewPanelSerializer* for the webviews. To achieve this, the method *registerWebviewPanelSerializer* can be called to register the serializer.

3.2 Back End

The back end section's diagram can be seen in Figure 3.7. The main file from the extension (*extension.ts*) serves as the entry file, and its *activate* method will be called when the activationEvents conditions defined in the manifest file are met. Then, it will create an instance of *FolderContainer* (a representation of all the acceptable Java projects that are open in the workspace) and use it to create the *GZoltarCommander*, which is responsible for having all of the commands that will be run in the extension. The commander holds the implementation for all of the features available to the user, and as such, handles the creation of the webview that will present the main HTML containing the fault reports. Additionally, it also serves as the representation of the tree view container, since that is where the main features of the extension will reside.

3.2.1 Manifest File

As mentioned earlier in 3.1.1, the manifest file contains fields such as the activation events for the extension and commands. This section will explain in detail the main fields of the GZoltar extension's package.json.

Starting with activation events, the extension must only be activated when in the presence of at least one Java project. In order to run GZoltar, many conditions must be satisfied before we can successfully perform an analysis on a project. Specifically, we need the folders containing the source and test files, and the project's dependencies. As such, the extension will only be accepting projects with a build

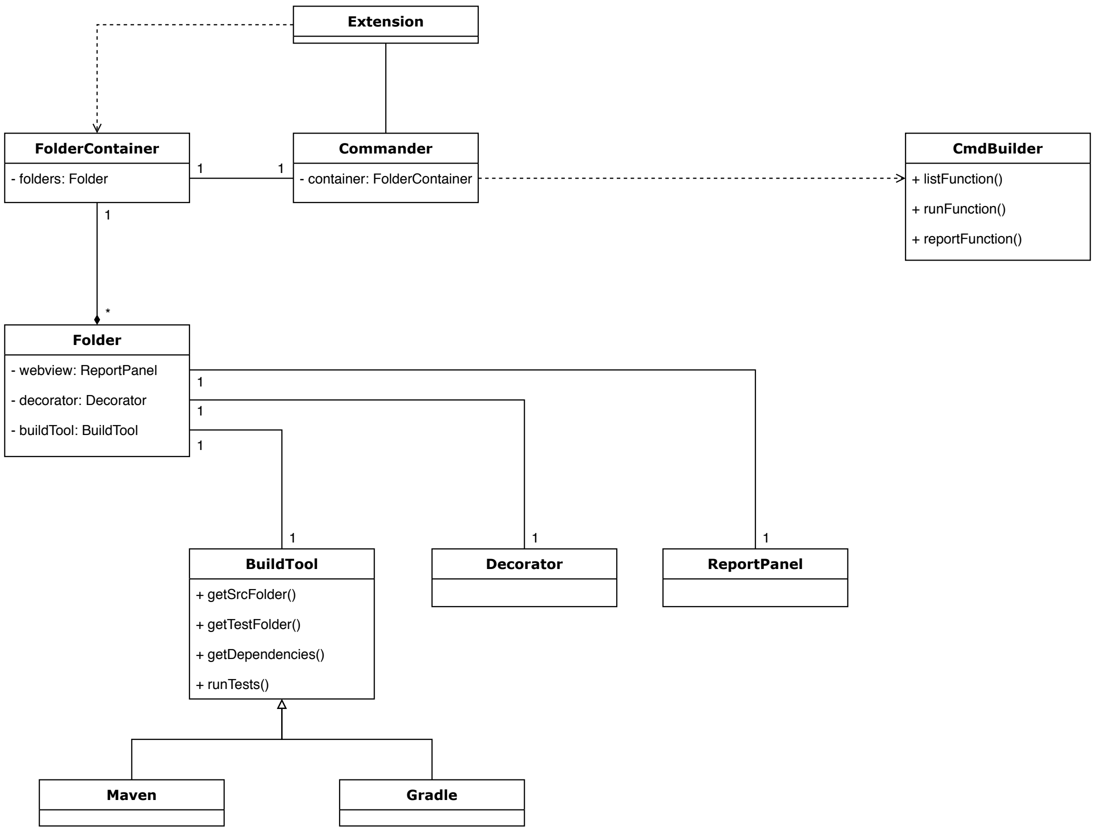
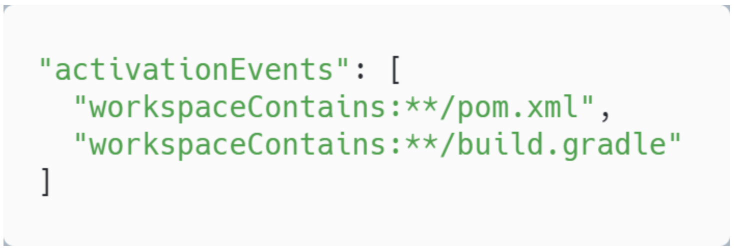


Figure 3.7: *Back end class diagram.*

automation tool, which is capable of achieving every single one of those conditions. This is largely due to the fact that GZoltar's capabilities are meant to be used on projects considerably large in size, to the point where debugging starts to become harder and more time consuming than it already is. A project lacking a build tool is likely to have a small test case suit, and an automated fault localization tool is not as necessary. Using a build tool means that a project's folder structure will be the same in most cases, so the folders containing the necessary files are in fixed folders following a specific naming convention, facilitating the automating process of the extension. Our build tools of choice are Maven and Gradle, since both remain the most used automation tools for Java projects [41]. To identify a project with Maven or Gradle, a file named pom.xml or build.gradle need to be present at the root of the project, respectively. Thus, the activation events for the extension need to be able to recognize at least one of those files before activating the extension itself. Figure 3.8 shows the activation events.



```
"activationEvents": [
  "workspaceContains:**/pom.xml",
  "workspaceContains:**/build.gradle"
]
```

Figure 3.8: *Activation events for the extension.*

Next on the manifest file, it is necessary to declare in the *contributes* field the activity bar for GZoltar.

This activity bar will present the user with every open and acceptable Java project, and allow them the possibility to run an analysis over that it, as well as other miscellaneous commands. All of these commands must also be declared in the *contributes* field. The activity bar's presentation in the VS Code user interface is presented in Figure 3.9 and its declaration in the manifest is in Figure 3.10.

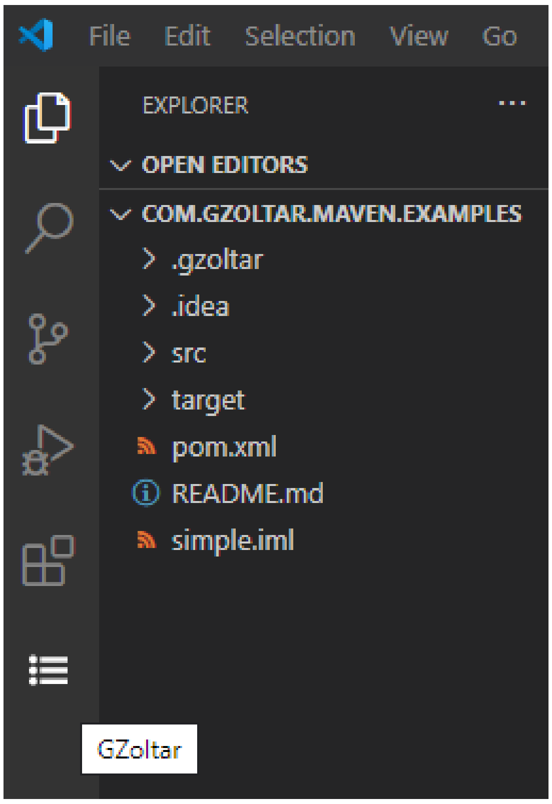


Figure 3.9: *GZoltar's activity bar.*

```
"contributes": {
  "viewsContainers": {
      "activitybar": [
          {
              "id": "gzoltar-commands",
              "title": "GZoltar",
              "icon": "media/dep.svg"
          }
      ]
  }
  ...
}
```

Figure 3.10: *Activity bar's declaration.*

For the sake of simplicity, GZoltar's extension will provide three commands to the user. These are meant to be simple commands with a clear function, so as to not make the extension's interface too complex. This allows for an easier understandable interface, with the purpose of being more accessible to new users. The commands' declaration can be seen in Figure 3.11. They are the following:

- **Run:** performs an analysis on the selected Java project and presents the results in HTML to the user. A new analysis is performed every single time this command is executed. Although it is possible to be notified when a file is changed due to VS Code's API (thus granting the possibility to optimize this process, and only execute it again when files are changed), it is not possible to detect file changes on disk, i.e., changes triggered by another application or even from VS Code's own API.

- **Refresh:** refreshes the interface to show if new projects have been added/removed. VS Code allows users to create a workspace (a project that consists of one or more projects) giving them the possibility to work on multiple projects at once. Whenever a project is added/removed, the interface should be automatically updated to reflect this change. However, in the off-chance that it is not, this command acts as a fail-safe measure.

- **Reset:** cleans the configuration folder for each project open with the extension. As we have mentioned before, GZoltar needs several conditions to be able to execute an analysis correctly, and even after that, it creates a multitude of files as a result. So, to organize this conglomerate of files and folders, we create an invisible configuration folder for each acceptable Java project that is open within the extension. This folder will have the necessary jars and dependencies to run GZoltar, and the results of the executions. In case of a mishap in a configuration folder, this command allows the user to start a new one with a clean slate.

```
"commands": [
  {
    "command": "gzoltar.refresh",
    "title": "Refresh"
  },
  {
    "command": "gzoltar.run",
    "title": "Run GZoltar"
  },
  {
    "command": "gzoltar.reset",
    "title": "Reset Configuration"
  }
]
```

Figure 3.11: *Commands' declaration.*

3.2.2 Entry File

The main file is the entry way for the extension's execution. The extension will be running when the method *activate* is called. Initially, it asserts that in the currently open workspace, there is at least one open acceptable Java project. Due to the activation event defined in the manifest file, it is technically impossible for the extension to enter the *activate* method without there being a single open project. Still, VS Code's API demands that a verification be made before accessing any of the folders residing in the workspace. These folders are then processed and stored in a newly created instance of *FolderContainer* (3.2.5), which represents all of the acceptable and currently open Java projects. This instance is passed on to the *GZoltarCommander* (3.2.3), the holder of the commands' implementation and the tree view container. This method also contains a listener that will update the *FolderContainer* every time a project is added/removed. Lastly, it registers every command available in the extension by using the commander's implementations.

3.2.3 GZoltar Commander

The commander is responsible for providing the elements in the tree data provider and also updating its elements when a change occurs. The elements in question are the open Java projects. It also contains the implementation of the three commands previously mentioned in 3.2.1. The run command is the most intricate of the three, due to the many steps involved in executing GZoltar. Firstly, it ensures the configuration folder for the selected project exists, and obtains the necessary dependencies. To execute GZoltar, a command-line interface, it uses the *CommandBuilder* which contains methods that simplify the creation of these commands, and returns them as formatted strings ready to use. With each command ready, the commander notifies in the status-bar what the extension is doing at all times, to keep the user updated in case a certain command is taking a long time to execute. In the end, it will create a *ReportPanel* and present it to the user with the results obtained from the execution, and also a *Decorator*, which indicates in the text editor the severity on each line of code.

3.2.4 Command Builder

In order to run GZoltar, there are three main methods that must be executed in succession. The first one is the *listTestMethods*, which obtains the names of all the test methods in the test case suite. The next one is *runTestMethods* that uses the previous results to execute the test cases and obtain the coverage. The last one, *faultLocalizationReport* uses the data obtained from the previous method to generate the reports that are used in the webview and decorator.

3.2.5 Workspace

The *FolderContainer* serves as a container for every open Java project while the extension is active. Its only purpose is to add/remove project folders, and retrieve the specific folder when required. The representation of a singular project folder is the *Folder* class. It is responsible for holding the webview, decorator and build tool pertaining to this project. The build tool contains information such as the folder names for the source and test classes, and also how to get the project's dependencies.

3.2.6 Report Panel

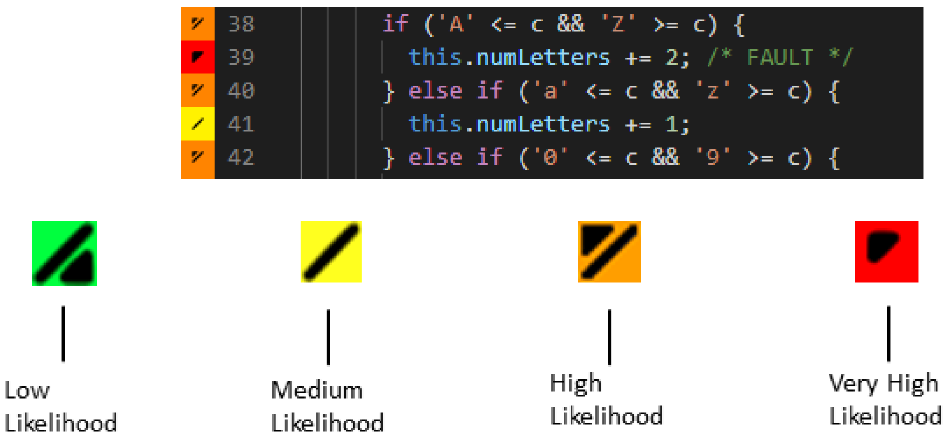
The report panel contains the HTML visualizations that are presented to the user. Since the visualizations have levels organized in a hierarchy, it is possible to double click on each level to zoom in. Each level represents a segment in the code, meaning that at the highest level, the whole chart represents the root of the project, and at the deepest level, it represents a single line of code. It also displays the path at the top of the chart, as we progress through it. We can click at the final level of the chart to open the file referenced by that line of code.

3.2.7 Decorator

The decorator indicates in the text editor the severity on each line of code. After GZoltar runs an analysis on the test suite and presents the webview to the user, it obtains information regarding the lines of code and their suspiciousness levels. Then, depending on the highest level of suspiciousness, the rest is also calculated. Meaning that if the highest level is 0.1, then that line of code is considered highly suspicious, whereas in a project where the highest level is 0.9, a line of code with 0.1 is not as suspicious.

There are four levels of suspicion used to rank each statement, as seen in Figure 3.12. Red is used for very high likelihood, orange for high likelihood, yellow for medium likelihood, and green for low likelihood. These icons are color coded, but they also have an internal symbol. These symbols come from the organization called ColorADD [42], whose aim is to help color-blind people determine which color is which, by creating a set of icons that are used to represent all primary and secondary colors. This way, even if the user has difficulties distinguishing between the initial colors, the internal symbols identify the correct and intended meaning behind them.

After the webview is presented to the user, every time a file is open in the text editor, the decorator is set to show the suspiciousness icons on the sidebar before each line of code. This will persist even after the webview is closed. If a new analysis is made, then the decoration will also be updated to reflect the changes made.



```
if ('A' <= c && 'Z' >= c) {
    this.numLetters += 2; /* FAULT */
} else if ('a' <= c && 'z' >= c) {
    this.numLetters += 1;
} else if ('0' <= c && '9' >= c) {
```

Figure 3.12: *Suspiciousness icons.*

3.3 Publishing the Extension

In order to make the extension readily available to other users so that they may find it, download and use its features, we have to publish it to the VS Code Extension Marketplace. The marketplace is where we can find all of the published extensions that we may wish to install on our computers.

Extensions are published using *vsce*, short for Visual Studio Code Extensions, which is a command-line tool for packaging, publishing and managing VS Code extensions. It can also search, retrieve metadata, and unpublish extensions. Figure 3.13 shows an example of vsce being used. When inside an extension folder, it is possible to package it into a *VSIX* file, or publish it given the specified publisher ID (explained in more detail further below). A *VSIX* package is a file that contains one or more Visual Studio extensions, together with the metadata Visual Studio uses to classify and install the extensions. That metadata is contained in the *VSIX* manifest and the [Content_Types].xml file. A VSIX package may also contain additional files to provide localized setup text, and may contain additional *VSIX* packages to install dependencies.

```
$ vsce package
# myExtension.vsix generated
$ vsce publish
# <publisherID>.myExtension published to VS Code MarketPlace
```

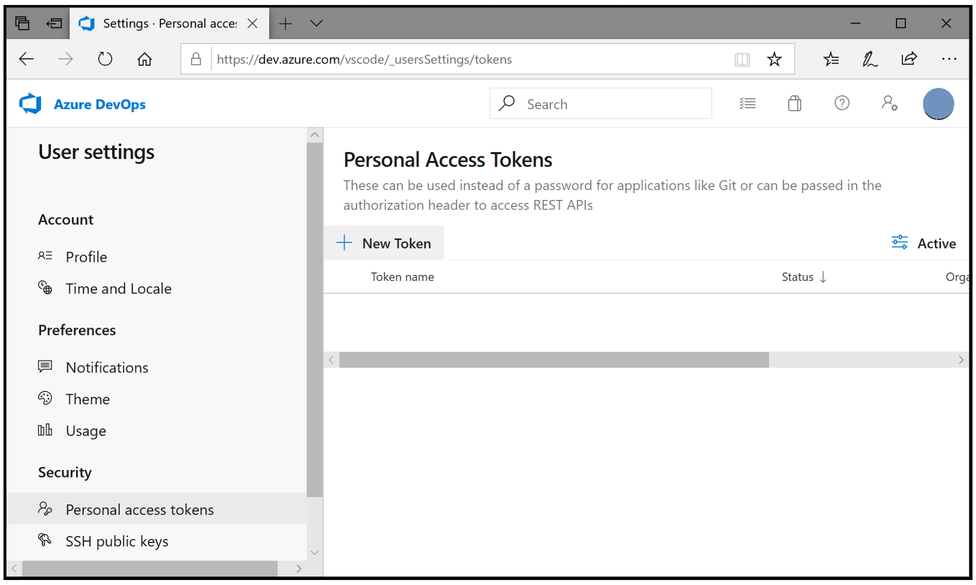
Figure 3.13: *Packaging and publishing an extension with vsce.*

3.3.1 Publisher ID

As previously stated, we need a publisher ID to publish the extension. The published extension needs an entity that is responsible for publishing it, i.e., an "owner" of sorts. This entity is responsible for managing the extension's versions, and how it will be presented to the public. To create the publisher ID, one of the two ways explained below can be used.

Using vsce

vsce can only publish extensions using Personal Access Tokens. A personal access token (PAT) is used as an alternate password to authenticate into Azure DevOps. Since VS Code is part of the tools in Microsoft's environment, it is possible to use a PAT obtained from the Azure DevOps services. First, we have to make sure we have an Azure DevOps organization, which is used to connect groups of related projects. After creating a group and a personal token associated with it (shown in Figure 3.14), it is now possible to create a publisher. The extension also needs to include the publisher name in the package.json file. With the PAT in hand, we can use vsce to create a publisher as shown in Figure 3.15. vsce will remember the provided PAT for future references to this publisher. However, while this is still a viable method to do so, it is considered deprecated and is soon to be removed, so it is not recommended to create a publisher using vsce.



Personal Access Tokens

These can be used instead of a password for applications like Git or can be passed in the authorization header to access REST APIs

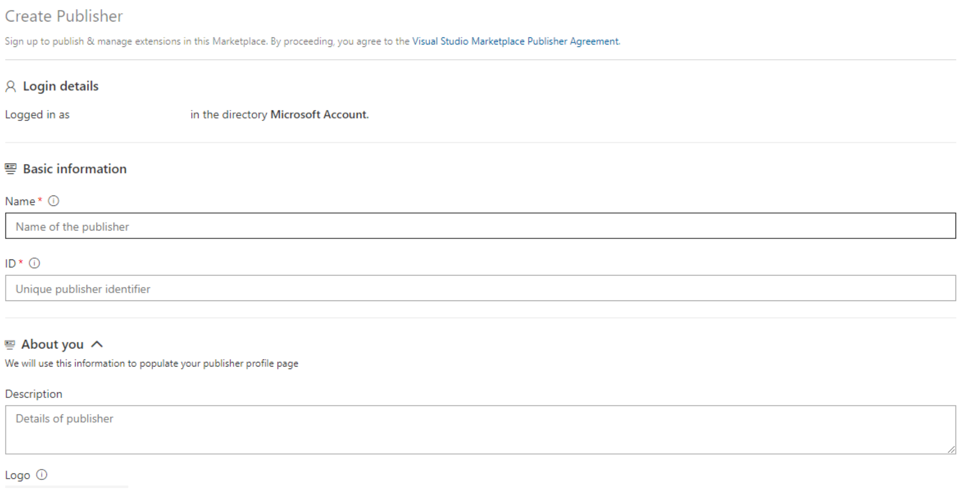
Figure 3.14: *Creating a Personal Access Token in Azure DevOps.*

```
$ vsce create-publisher (publisher name)
# Publisher human-friendly name: (t) New Publisher
# Email: ...
```

Figure 3.15: *Creating a Publisher using vsce.*

Using the Marketplace

By logging in the Visual Studio Marketplace, it is possible to manage publishers and extensions associated with them. To create a publisher, we only need to input its name and ID as obligatory fields. The rest, such as the company website or even source code repository are optional fields. With the publisher created, we can now create extensions by uploading a *VSIX* file. A few moments after uploading it, the extension will be live and ready to be downloaded by other people. Figures 3.16 and 3.17 show a snippet of how to create a publisher and upload an extension.



Create Publisher

Sign up to publish & manage extensions in this Marketplace. By proceeding, you agree to the Visual Studio Marketplace Publisher Agreement.

Login details

Logged in as in the directory Microsoft Account.

Basic information

Name *

Name of the publisher

ID *

Unique publisher identifier

About you

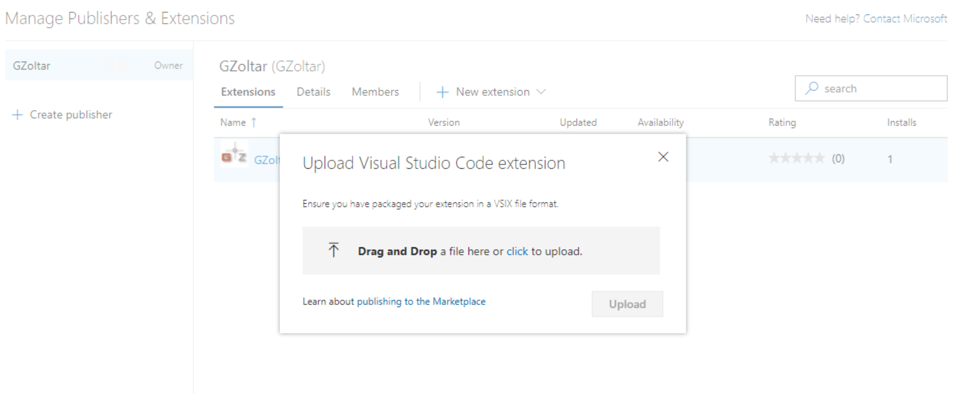
We will use this information to populate your publisher profile page

Description

Details of publisher

Logo

Figure 3.16: *Creating a Publisher using Visual Studio Marketplace.*



Manage Publishers & Extensions

Need help? Contact Microsoft

GZoltar Owner

Create publisher

GZoltar (GZoltar)

Extensions Details Members New extension

search

Name	Version	Updated	Availability	Rating	Installs
GZol				(0)	1

Upload Visual Studio Code extension

Ensure you have packaged your extension in a VSIX file format.

Drag and Drop a file here or click to upload.

Learn about publishing to the Marketplace

Upload

Figure 3.17: *Uploading an extension to the Visual Studio Marketplace.*

Chapter 4

Results and Evaluation

This chapter presents the results obtained throughout this project. We will show how the extension can be found and installed in the Visual Studio Marketplace, how its features can be accessed and what the final product looks like. We will also present an evaluation of the results achieved by conducting a user study. The goal of this evaluation is to measure the extension's usability and efficiency with users that have never had any previous interaction with GZoltar. They were given a project and a limited amount of time to find a fault that was previously injected in the source code. In the end, the users filled a form (present in Appendix A) providing feedback of their experience and also suggestions for future work.

4.1 Extension Showcase

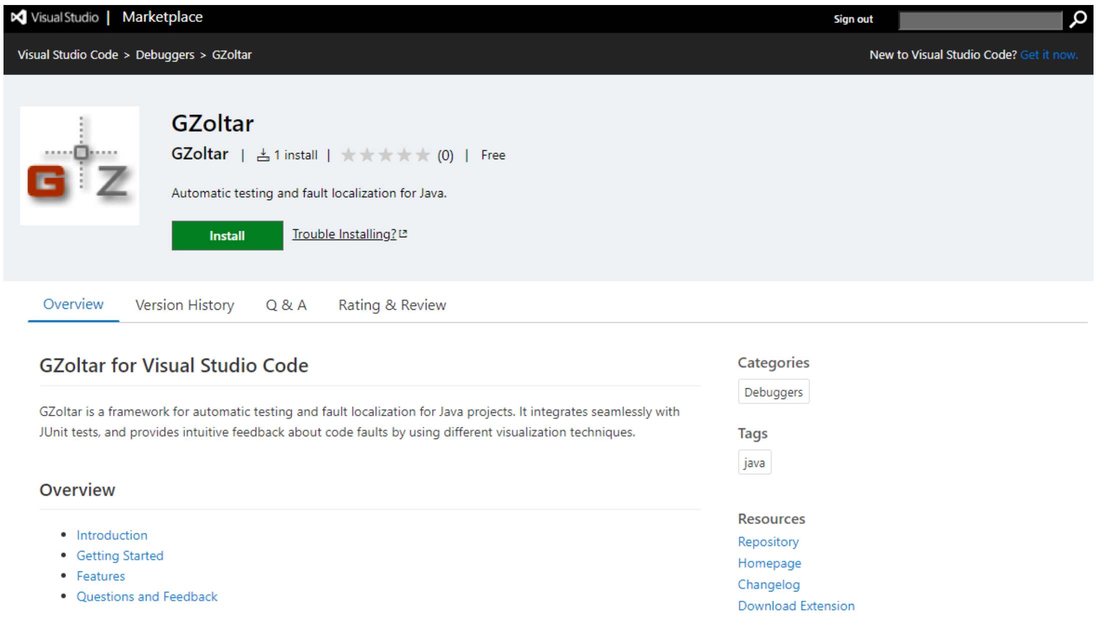
This section will present the several ways to install the extension from the marketplace, and also how it works. We will be running GZoltar on a small project just to illustrate how it is supposed to work, and what should be presented to the user.

4.1.1 Installing

There are two ways the extension can be installed. The first one is by heading directly to the Visual Studio Marketplace and searching for GZoltar. Then, it is possible to install it from there, as shown in Figure 4.1. The other way is to install it from VS Code itself. By heading to the extensions tab and searching GZoltar, the extension will also show up and be available for installation. This can be seen in Figure 4.2.

4.1.2 Using the Extension

To access GZoltar's functionalities, we click on GZoltar's icon in the activity bar previously mentioned in 3.2.1. The menu that is open after that will show every single open and acceptable Java project in the current workspace (shown in Figure 4.3. To run GZoltar on a specific project, we click on the icon right next to the project's name.



Automatic testing and fault localization for Java.

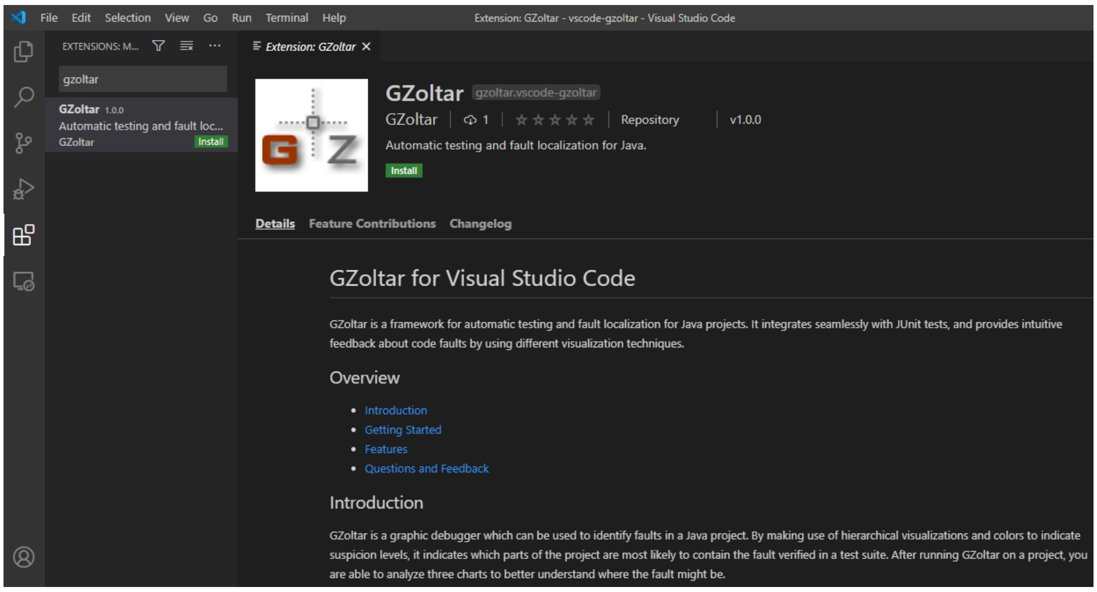
GZoltar for Visual Studio Code

GZoltar is a framework for automatic testing and fault localization for Java projects. It integrates seamlessly with JUnit tests, and provides intuitive feedback about code faults by using different visualization techniques.

Overview

- Introduction
- Getting Started
- Features
- Questions and Feedback

Figure 4.1: *GZoltar in the VS Marketplace.*


File Edit Selection View Go Run Terminal Help
Extension: GZoltar - vscode-gzoltar - Visual Studio Code
EXTENSIONS: M...
gzoltar
GZoltar 1.0.0
Automatic testing and fault loc...
GZoltar
Install
Extension: GZoltar
GZoltar gzoltar.vscode-gzoltar
GZoltar | 1 | Repository | v1.0.0
Install
Details
Feature Contributions
Changelog

Automatic testing and fault localization for Java.

GZoltar for Visual Studio Code

GZoltar is a framework for automatic testing and fault localization for Java projects. It integrates seamlessly with JUnit tests, and provides intuitive feedback about code faults by using different visualization techniques.

Overview

- Introduction
- Getting Started
- Features
- Questions and Feedback

Introduction

GZoltar is a graphic debugger which can be used to identify faults in a Java project. By making use of hierarchical visualizations and colors to indicate suspicion levels, it indicates which parts of the project are most likely to contain the fault verified in a test suite. After running GZoltar on a project, you are able to analyze three charts to better understand where the fault might be.

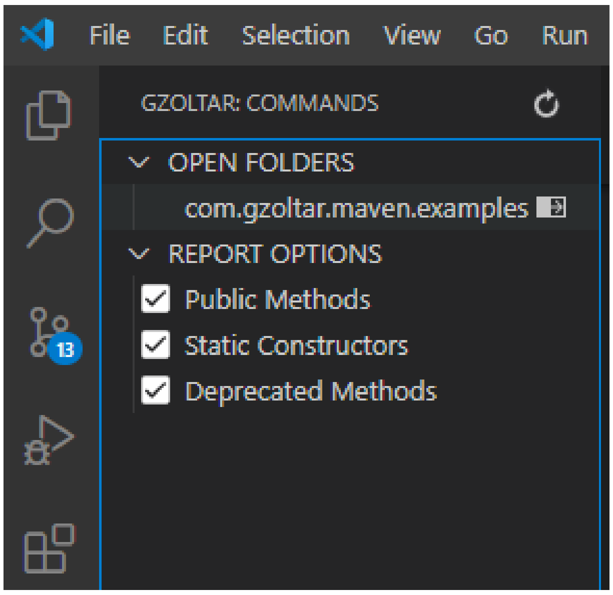
Figure 4.2: *GZoltar in VS Code.*

While GZoltar is running, it is possible to see on the lower right corner of the status bar what phase it is currently on. This is to help the user by indicating that the extension is doing background work, on the off chance that the analysis takes some time to complete. This can be seen in Figure 4.4.

After it is done running, a new tab will open on the right side of the editor. The newly opened tab will show the results of the analysis in the form of a chart (Figure 4.5). The colors in the charts indicate the likelihood of a certain code segment being suspicious. The color coordination is the same as the one already shown in 3.2.7. It is possible to navigate through the chart by double clicking on each color coded segment. Right clicking on the chart will reset it back to its original state. Clicking on an edge segment (which corresponds to a single line of code) will open the file associated with that line of code.

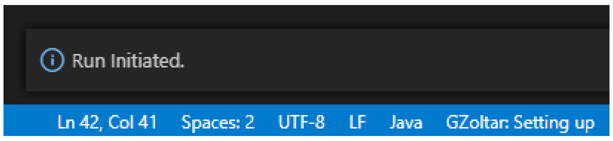
We can also change the visualization that is currently being shown. On the activity bar menu, under the project that was just used to perform an analysis, we can choose one of the three charts to change the visualization that we want to see. This is shown in Figure 4.6.

Lastly, when a file is opened after performing an analysis, it will show an icon to indicate the level of suspiciousness for each line of code (Figure 4.7). The icon's colors are the same as the ones represented in the charts.



✓ Public Methods
✓ Static Constructors
✓ Deprecated Methods

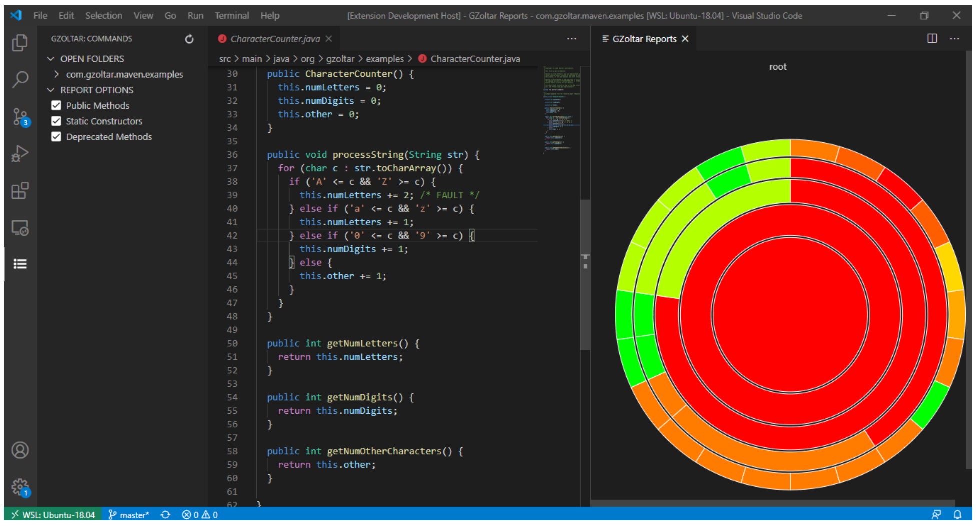
Figure 4.3: *GZoltar Commands.*



Run Initiated.

Ln 42, Col 41   Spaces: 2   UTF-8   LF   Java   GZoltar: Setting up

Figure 4.4: *GZoltar Status Bar Update.*

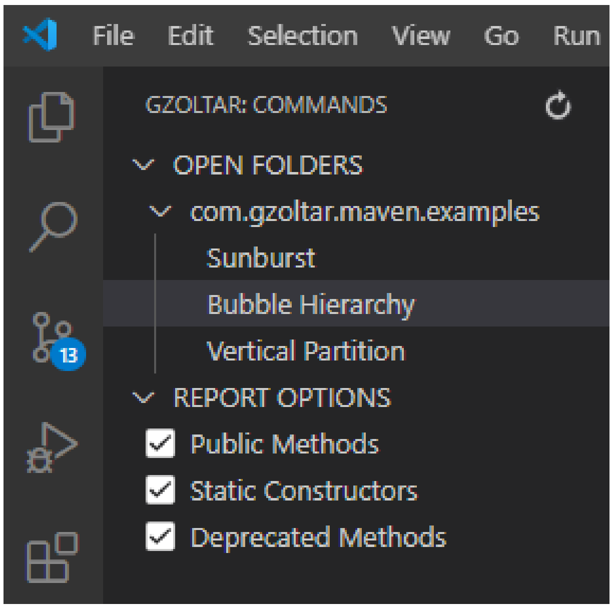


[Extension Development Host] - GZoltar Reports - com.gzoltar.maven.examples [WSL: Ubuntu-18.04] - Visual Studio Code

✓ Public Methods
✓ Static Constructors
✓ Deprecated Methods

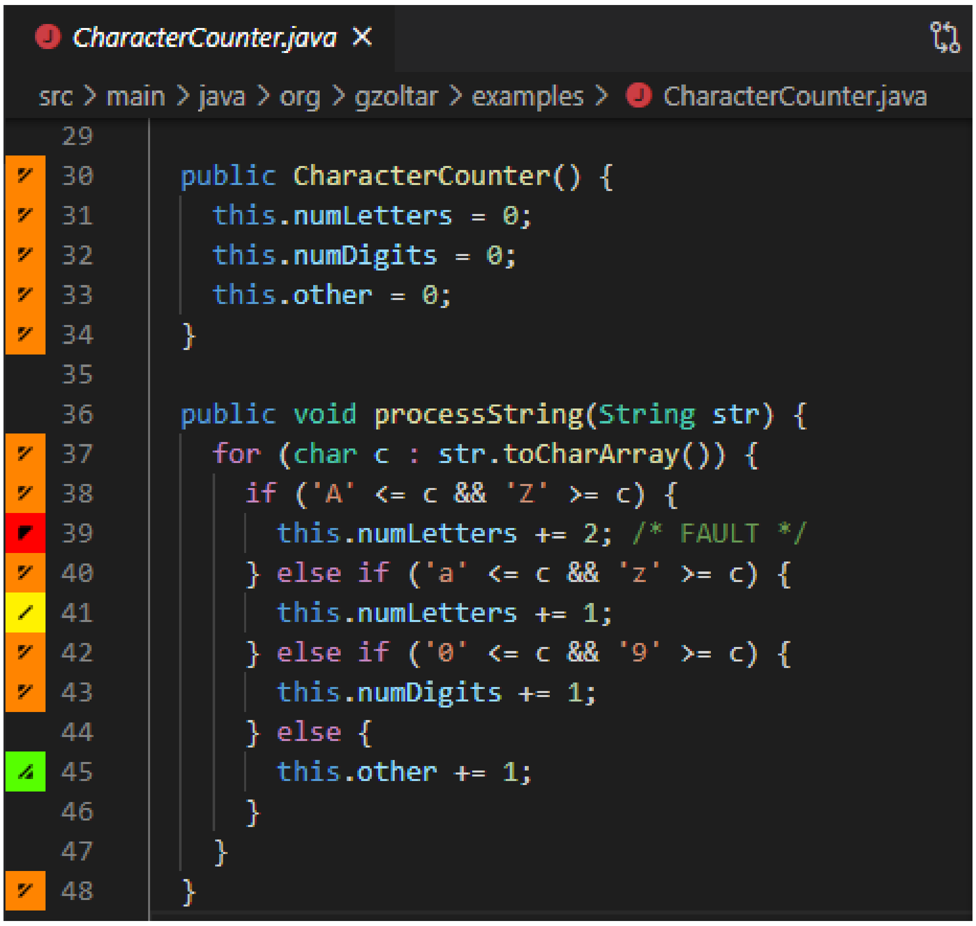
```java
  public CharacterCounter() {
    this.numLetters = 0;
    this.numDigits = 0;
    this.other = 0;
  }

  public void processString(String str) {
    for (char c : str.toCharArray()) {
      if ('A' <= c && 'Z' >= c) {
        this.numLetters += 2; /* FAULT */
      } else if ('a' <= c && 'z' >= c) {
        this.numLetters += 1;
      } else if ('0' <= c && '9' >= c) {
        this.numDigits += 1;
      } else {
        this.other += 1;
      }
    }
  }

  public int getNumLetters() {
    return this.numLetters;
  }

  public int getNumDigits() {
    return this.numDigits;
  }

  public int getNumOtherCharacters() {
    return this.other;
  }

}
```

Figure 4.5: *GZoltar Result.*



✓ Public Methods
✓ Static Constructors
✓ Deprecated Methods

Figure 4.6: *GZoltar Visualizations.*


CharacterCounter.java
src > main > java > org > gzoltar > examples > CharacterCounter.java

```java

public CharacterCounter() {
    this.numLetters = 0;
    this.numDigits = 0;
    this.other = 0;
}

public void processString(String str) {
    for (char c : str.toCharArray()) {
        if ('A' <= c && 'Z' >= c) {
            this.numLetters += 2; /* FAULT */
        } else if ('a' <= c && 'z' >= c) {
            this.numLetters += 1;
        } else if ('0' <= c && '9' >= c) {
            this.numDigits += 1;
        } else {
            this.other += 1;
        }
    }
}
```

Figure 4.7: *Open file showing suspicion levels.*

4.2 User Study

In order to validate the usefulness of the extension's current version, nine users were selected to test the efficiency of the interactive visualizations. We recorded the time that each user took to finish the testing and debugging task. At the end of this process, each user filled a form with the feedback of their experience and some suggestions for future work. This usability test was important to test the efficiency of the extension and also to create guidelines to improve future developments of this tool, thus providing some insight regarding future versions.

4.2.1 Users Description

The number of users picked to carry out the evaluation must be sufficiently enough, such that we are able to get conclusive evidence of the extension's usability and overall usefulness of its features. As such, the number we choose is based on J. Nielsen's work on usability and user tests [43]. According to Nielsen et al., the number of users needed to test a small software project should be 9. That number should be enough to identify any main usability issues.

The group is composed by MSc students in Computer Engineering from the Computer Science and Engineering Department of the University of Lisbon, and the Computer Science Department of the University of Porto. The users were picked randomly, aged less than 25 years old and from all genders. An introductory survey was made to know the level of experience with programming languages, IDEs, etc. The programming languages that the users were familiarized with the most were Java (100%), C# (88.9%), JavaScript (55.6%), TypeScript (33.3%), C (22.2%), with Python and Assembly both at 11.1%. The most used IDE was IntelliJ IDEA at 100%, with Eclipse at 44.4% and Visual Studio at 77.8%. The most used software testing framework was JUnit at 100%. Most of the users have not used automatic fault detection tools, with only one user having used one before. The most used debugging techniques are both breakpoints and console output at 100%. A summary of these questions can be found in Figure 4.8.

4.2.2 Experiment Conditions

Each user tested the extension on their own computer. While most users with Linux/MacOS were able to immediately use the extension, users with Windows systems had to install a Windows subsystem for Linux and the extension *Remote - WSL* (which can be found in the VS Code marketplace). This extension allows the user to open any folder in the subsystem for Linux. This is necessary due to a limitation found within GZoltar, since it contains a fault which does not allow it to correctly execute its functions when operating on a Windows computer. Furthermore, given that we are using GZoltar strictly as a black box (using its services as an API and not interacting with its inner modules), this fault remains untreated for now.

Each user was asked to debug a faulty version of a medium sized project containing a suite of JUnit tests. A fault was injected randomly in the project, making some test cases fail. The users had no previous contact with this project nor its tests, to better verify the extension's usefulness. A short explanation was given to each user to explain the process and how GZoltar works. Afterwards, they each had 20 minutes to try and localize the fault. In the end, they had to fill a survey with questions based on their experience with the tool.

4.2.3 Feedback

The time limit for this task was 20 minutes, with its main purpose being on obtaining feedback about the extension's usability and usefulness, since its effectiveness has been proven in previous studies [21]. Most of the users were able to find the fault (77%), while the rest that were not able to find it did manage to pinpoint the fault's most likely localization. Since all the users had previous experience with VS Code, they were comfortable with the environment.

The survey had a section for the users to answer about their experience with the debugging session, and also give their feedback. Many of the questions regarding the usability and interface of the extension were made using a scale from 1 (very poor) to 5 (very good).

A majority of the users (55.5%) stated the extension to be very easy to understand at first. However, when asked how hard it was to learn how to effectively use it, the responses were evenly split between moderately easy and moderately hard.

Everyone agreed that the icons/buttons are moderately intuitive, claiming that while they are not bad, there is definitely room for improvement. Regarding the information that was presented to the users, when asked if it was being presented in a clear and understandable manner, the responses were mostly on the positive side (55.6%). These questions' responses are summed in Figure 4.9.

Following up on the extension's performance, an overwhelming majority rated its responsiveness as good (77.8%) or very good (22.2%), claiming that the interaction with the visualizations and the source code went smoothly.

Once more, most of the users (66.7%) believed the extension to be helpful in finding the bug, while others did not think it was as crucial. Additionally, most of them (88.9%) agree that the extension does require a bit of user experience, meaning that the extension's usefulness increases along with the user's experience. Overall, the users reported the session as a positive experience, as far as debugging sessions go. The charts for these answers are in Figure 4.10.

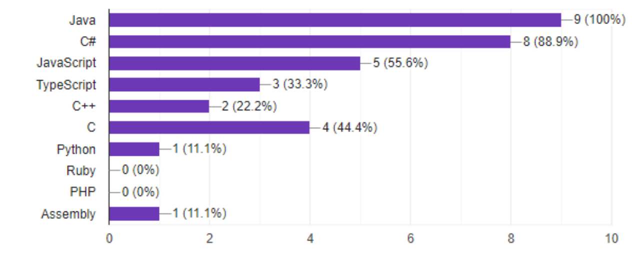
When asked about the concepts related to GZoltar, all of the users agreed that automatic debugging is very important. They also rated visual debuggers to be very necessary, as well as debuggers integrated into an IDE.

The survey also had an open question for the users to indicate any issues found during the session, or suggestions they might have. Some suggested that changing the zoom from double click to a single click might improve the user experience. Others suggested different or more intuitive icons for the interface. Regardless, none of them had any issues with the debugging task. The final summary of the survey's answers are in Figure 4.11.

Thus, this experiment validates the initial hypothesis that GZoltar's interactive visualizations can help developers to find faults in a short period of time.

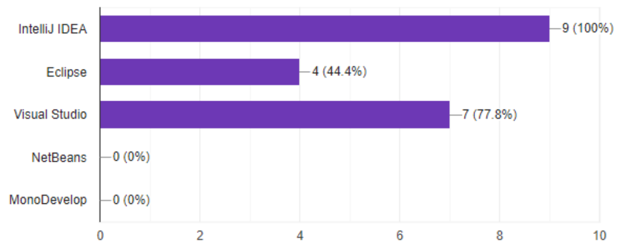
What programming languages have you used in the past or frequently use?

9 responses



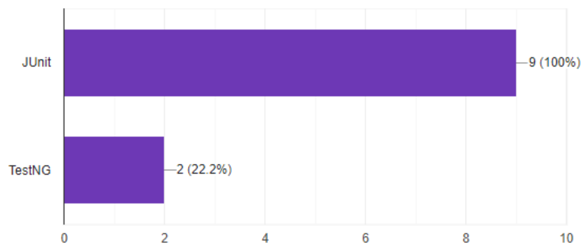
What Integrated Development Environments (IDE) have you used in the past or frequently use?

9 responses



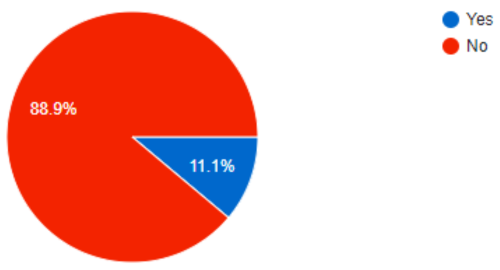
What software testing frameworks have you used in the past or frequently use?

9 responses



Have you ever used automatic fault detection tools?

9 responses



What software debugging techniques have you used in the past or frequently use?

9 responses

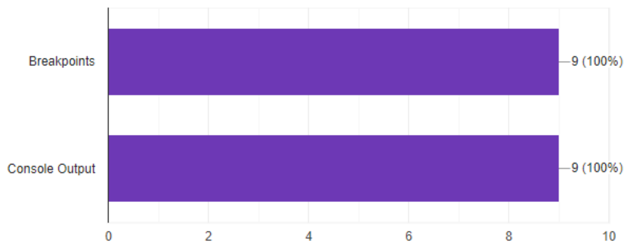


Figure 4.8: *First set of survey questions.*

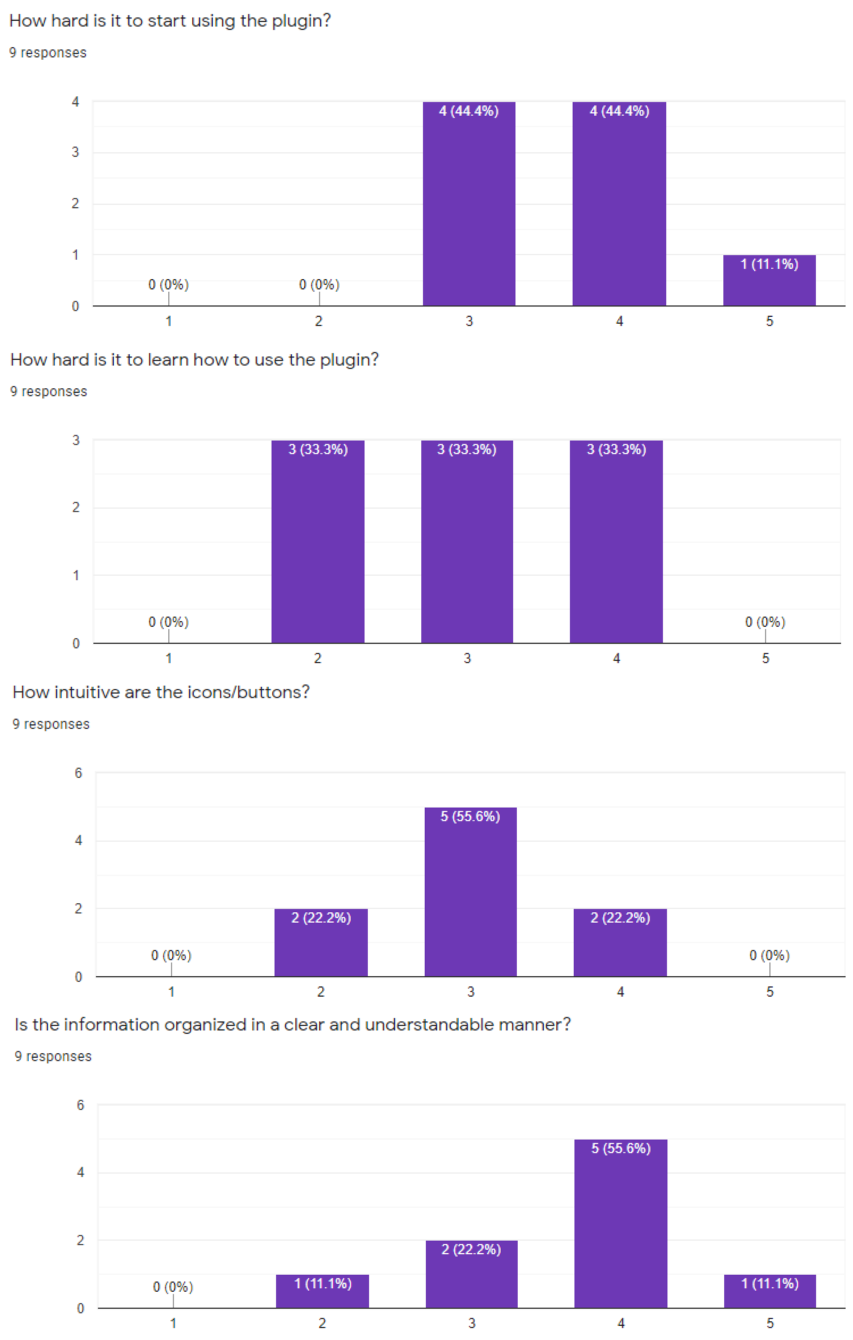
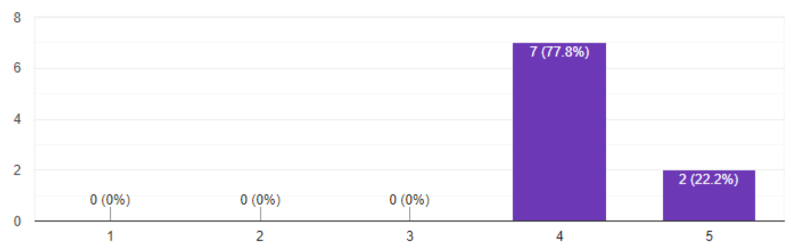


Figure 4.9: *Second set of questions about the extension's interface.*

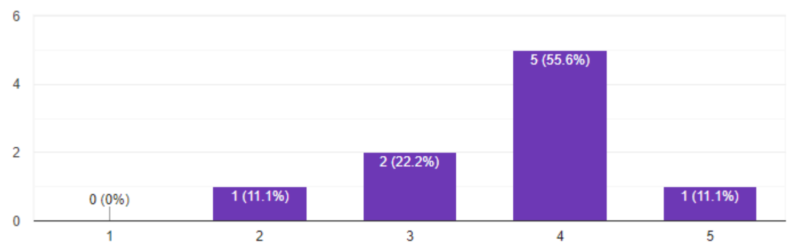
Is the plugin responsive?

9 responses



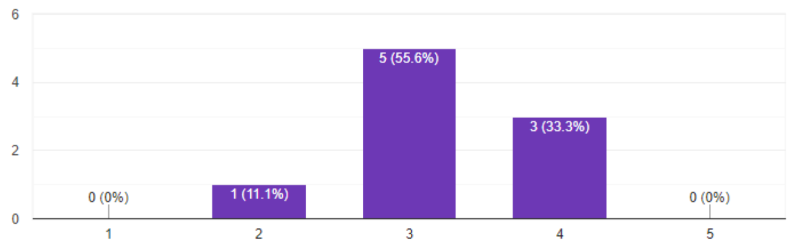
How effective was the plugin in finding the bug/understanding where the fault might be?

9 responses



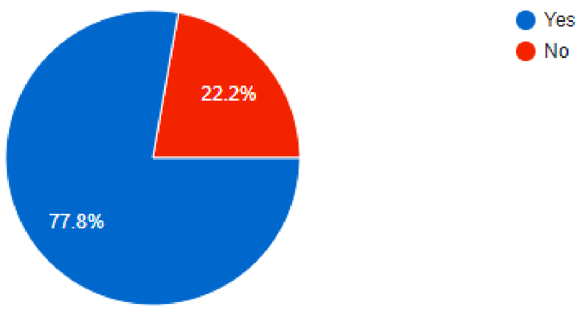
Does the plugin depend on the user's previous experience?

9 responses



Were you able to find the bug?

9 responses



What would you rate your overall experience?

9 responses

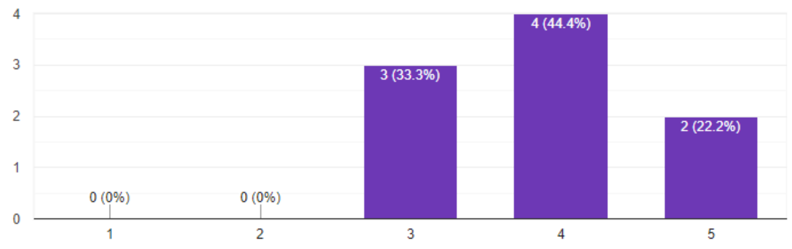


Figure 4.10: *Third set of questions about the extension's capabilities.*

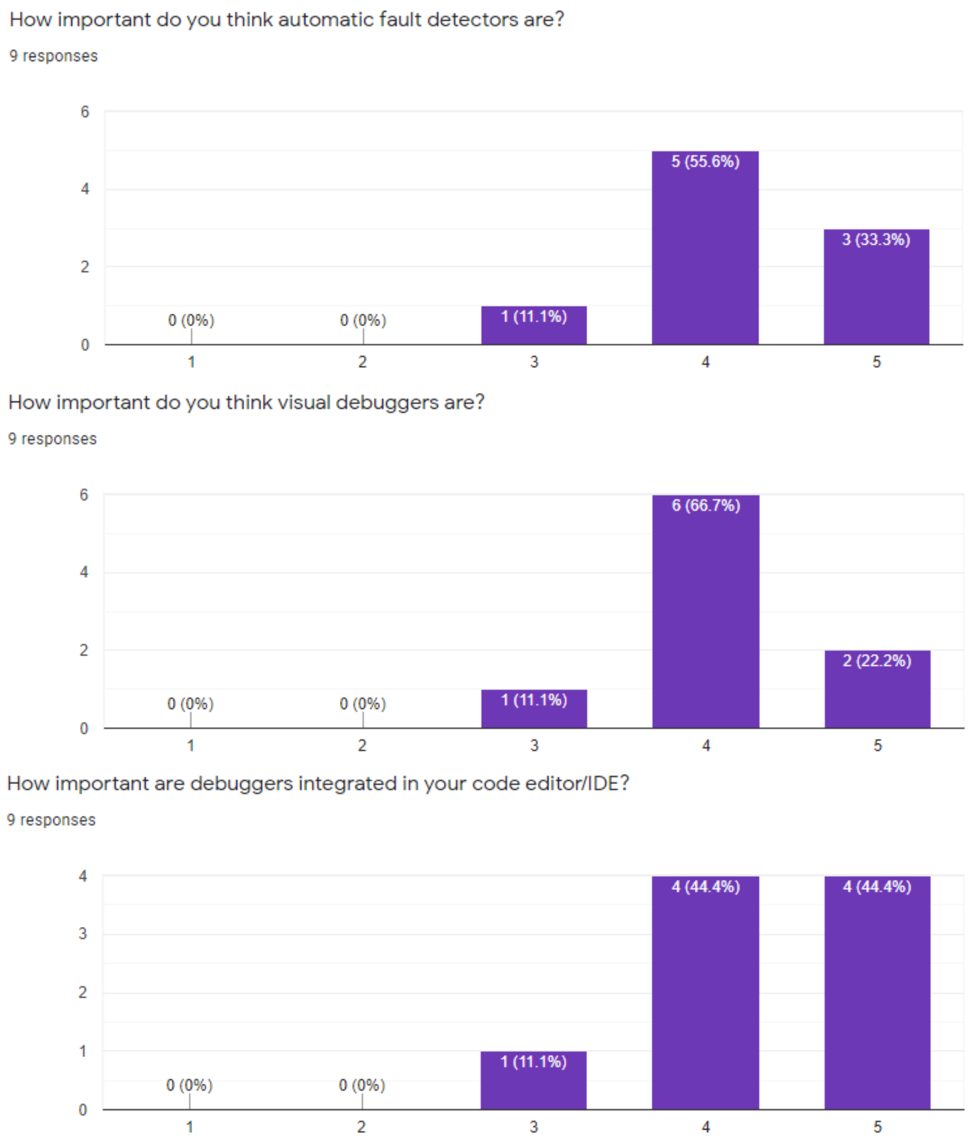


Figure 4.11: *Final set of questions about automatic and visual debugging.*

Chapter 5

Conclusions

At the beginning of this document, we explored the concept of debugging and its intricacies. That is, the many techniques associated, its relevance in software development and the potential risks if not handled correctly. Debugging is clearly very important for any type of software, especially in safety-critical systems, so having more options to explore this field is beneficial to developers. Considering this, automated and fault localization tools help automate this process, but most tools are incomplete. Most debugging tools and techniques reported usually lack a powerful visualization tool, or on the off chance that they have one, it is typically in the form of a list with potentially faulty statements. Even if that is not the case, some of those tools themselves are dated, so the graphical capabilities look antiquated and are probably harder to interpret. The effect that a graphical debugger can have in the process of software development is clear and very much needed, as stated by several studies made on the subject. Thus, there is an evident need for a tool with already established results and visualizations proven to be effective, which is why we are aiming to provide GZoltar in a more recent development environment in hopes of being more widely adopted by the community.

As such, we intended to port GZoltar (an automatic fault localization tool) to Visual Studio Code, a code editor that has gained a lot of traction in the recent years. The extension provides an interface from which the user can obtain a global view of the project and immediately spot the places where it is most likely to contain the fault. The extension's functionalities will be present to the user when it identifies the open folder to be a Java project. From there, the user can request to construct the views from the current set of test cases, and visualize them in HTML form in a webview panel. Finally, an evaluation was carried out as a user study to determine whether the extension meets with users' expectations, and proves to be helpful in a debugging situation.

5.1 Work Summary

The GZoltar plug-in for Visual Studio Code (also called an extension) was written in TypeScript, and created using the command-line version of GZoltar. The extension communicates with the CLI to generate reports, and presents the results to the user in the editor in special tabs called webviews, which render

HTML. The user can also explore the files that were deemed suspicious by GZoltar, and they will be marked with symbols in lines to indicate the level of suspiciousness. Lastly, the extension was published to the Visual Studio Code Marketplace.

5.2 Future Work

The initial goals set for this project were achieved, however there are always new ideas to further improve the extension. Since GZoltar is open-source, the extension itself is also open for change by anyone willing to contribute.

5.2.1 Differently Color Coded Graphics

GZoltar relies heavily on colors to indicate the suspiciousness levels of the lines of code in the project. It also makes use of symbols to assist colorblind people in distinguishing the colors. But still, the graphics that are presented in the webviews do not have those symbols, and contain only four colors: green, yellow, orange and red. An improvement to this would be to allow the user to select the color scheme they desire, to better fit with either their personal preference, or to suit the colors they are able to perceive.

5.2.2 More Build Tools

The extension currently only accepts Java projects using Maven or Gradle. Although those are the most popular build tools, it would attract more people if it also accepted more build tools.

5.2.3 Windows Integration

The current CLI of GZoltar only works immediately in UNIX based systems. Which means, to be able to use GZoltar in a Windows operating system, it is necessary to use a UNIX-based CLI along with the Remote-WSL VS Code extension to open folders in the Windows subsystem for Linux. This adds a couple of extra steps for Windows users to use GZoltar in VS Code, so it would benefit them if this still seemingly unknown problem were to be fixed to appease all users.

Bibliography

[1] J. Campos, A. Riboira, A. Perez, and R. Abreu. Gzoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 378–381. ACM, 2012.

[2] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[3] Q. Xin and S. P. Reiss. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 660–670. IEEE Press, 2017.

[4] M. Martinez and M. Monperrus. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 441–444. ACM, 2016.

[5] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. Precise condition synthesis for program repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 416–426. IEEE, 2017.

[6] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1–11. ACM, 2018.

[7] J. C. Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th international conference on software engineering*, pages 547–550, 2002.

[8] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[9] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, 2000.

[10] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[11] F. Tip. *A survey of program slicing techniques.* Centrum voor Wiskunde en Informatica Amsterdam, 1994.

[12] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Acm Sigplan Notices*, volume 40, pages 15–26. ACM, 2005.

[13] S. M. Daniel S. Wilkerson. Delta tool, 2001. URL `http://delta.tigris.org/`. [Online; accessed October 30, 2019].

[14] R. Reiter. A theory of diagnosis from first principles. *Artificial intelligence*, 32(1):57–95, 1987.

[15] W. E. Wong and Y. Qi. Bp neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 19(04):573–597, 2009.

[16] S. Nessa, M. Abedin, W. E. Wong, L. Khan, and Y. Qi. Software fault localization using n-gram analysis. In *International Conference on Wireless Algorithms, Systems, and Applications*, pages 548–559. Springer, 2008.

[17] T. Denmat, M. Ducassé, and O. Ridoux. 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*, pages 396–399. ACM, 2005.

[18] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham. Effective software fault localization using an rbf neural network. *IEEE Transactions on Reliability*, 61(1):149–169, 2011.

[19] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *Proceedings of the 28th international conference on Software engineering*, pages 272–281. ACM, 2006.

[20] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on software engineering*, 32(10):831–848, 2006.

[21] C. Gouveia, J. Campos, and R. Abreu. Using html5 visualizations in software fault localization. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–10. IEEE, 2013.

[22] H. A. de Souza, M. L. Chaim, and F. Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *arXiv preprint arXiv:1607.04347*, 2016.

[23] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007.

[24] P. Zoeteweij, R. Abreu, R. Golsteijn, and A. J. Van Gemund. Diagnosis of embedded software using program spectra. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, pages 213–220. IEEE, 2007.

[25] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Inc., USA, 1988. ISBN 013022278X.

[26] A. d. S. Meyer, A. A. F. Garcia, A. P. d. Souza, and C. L. d. Souza Jr. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l). *Genetics and Molecular Biology*, 27(1):83–91, 2004.

[27] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 467–477. IEEE, 2002.

[28] H. L. Ribeiro, H. A. de Souza, R. P. A. de Araujo, M. L. Chaim, and F. Kon. Jaguar: a spectrum-based fault localization tool for real-world software. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 404–409. IEEE, 2018.

[29] G. Balogh, F. Horváth, and Á. Beszédes. Poster: Aiding java developers with interactive fault localization in eclipse ide. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 371–374. IEEE, 2019.

[30] D. Hao, L. Zhang, L. Zhang, J. Sun, and H. Mei. Vida: Visual interactive debugging. In *2009 IEEE 31st International Conference on Software Engineering*, pages 583–586. IEEE, 2009.

[31] M. Weiser and J. Lyle. Experiments on slicing-based debugging aids. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, pages 187–197, Norwood, NJ, USA, 1986. Ablex Publishing Corp. ISBN 0-89391-388-X. URL `http://dl.acm.org/citation.cfm?id=21842.28894`.

[32] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPlan Notices*, volume 25, pages 246–256. ACM, 1990.

[33] B. Korel and J. Laski. Dynamic program slicing. *Information processing letters*, 29(3):155–163, 1988.

[34] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*, pages 143–151. IEEE, 1995.

[35] P. V. Gestwicki and B. Jayaraman. Jive: Java interactive visualization environment. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 226–228, 2004.

[36] H. Agrawal, J. L. Alberi, J. R. Horgan, J. J. Li, S. London, W. E. Wong, S. Ghosh, and N. Wilde. Mining system tests to aid software maintenance. *Computer*, 31(7):64–73, 1998.

[37] W. E. Wong and J. Li. An integrated solution for testing and analyzing java applications in an industrial setting. In *12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 8–pp. IEEE, 2005.

[38] C. Mateis, M. Stumptner, D. Wieland, and F. Wotawa. Jade-ai support for debugging java programs. In *ictai*, page 62, 2000.

[39] A. J. Ko and B. A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 151–158. ACM, 2004.

[40] Github. Electron, 2013. URL `https://electronjs.org/`. [Online; accessed October 30, 2019].

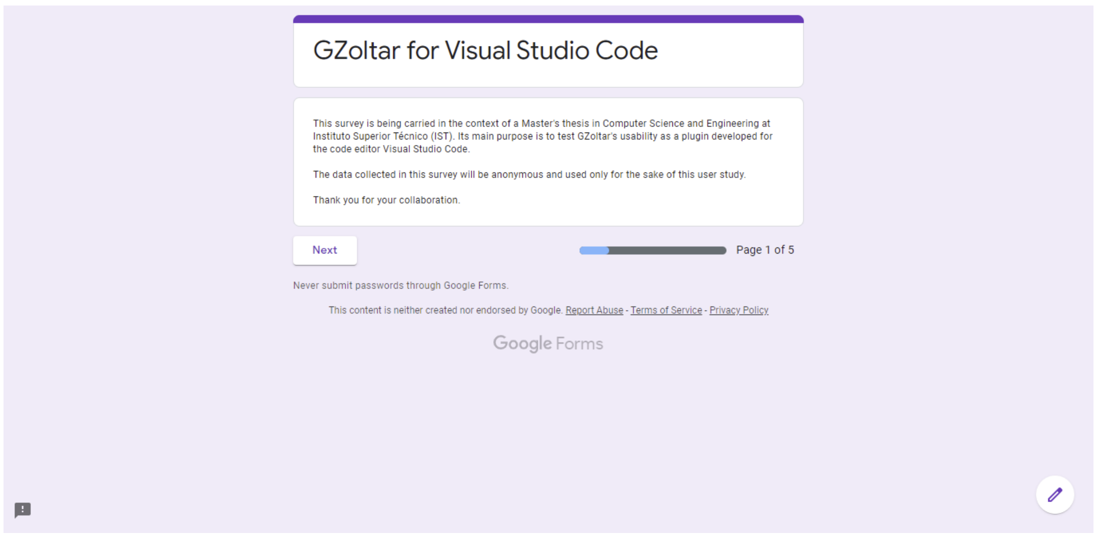
[41] Perforce. Coloradd, 2020. URL `https://www.jrebel.com/blog/2020-java-technology-report#build-tool`. [Online; accessed October 22, 2020].

[42] M. Neiva. Coloradd, 2010. URL `https://www.coloradd.net/`. [Online; accessed October 21, 2020].

[43] J. Nielsen and T. K. Landauer. A mathematical model of the finding of usability problems. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, pages 206–213, 1993.

Appendix A

Survey



GZoltar for Visual Studio Code

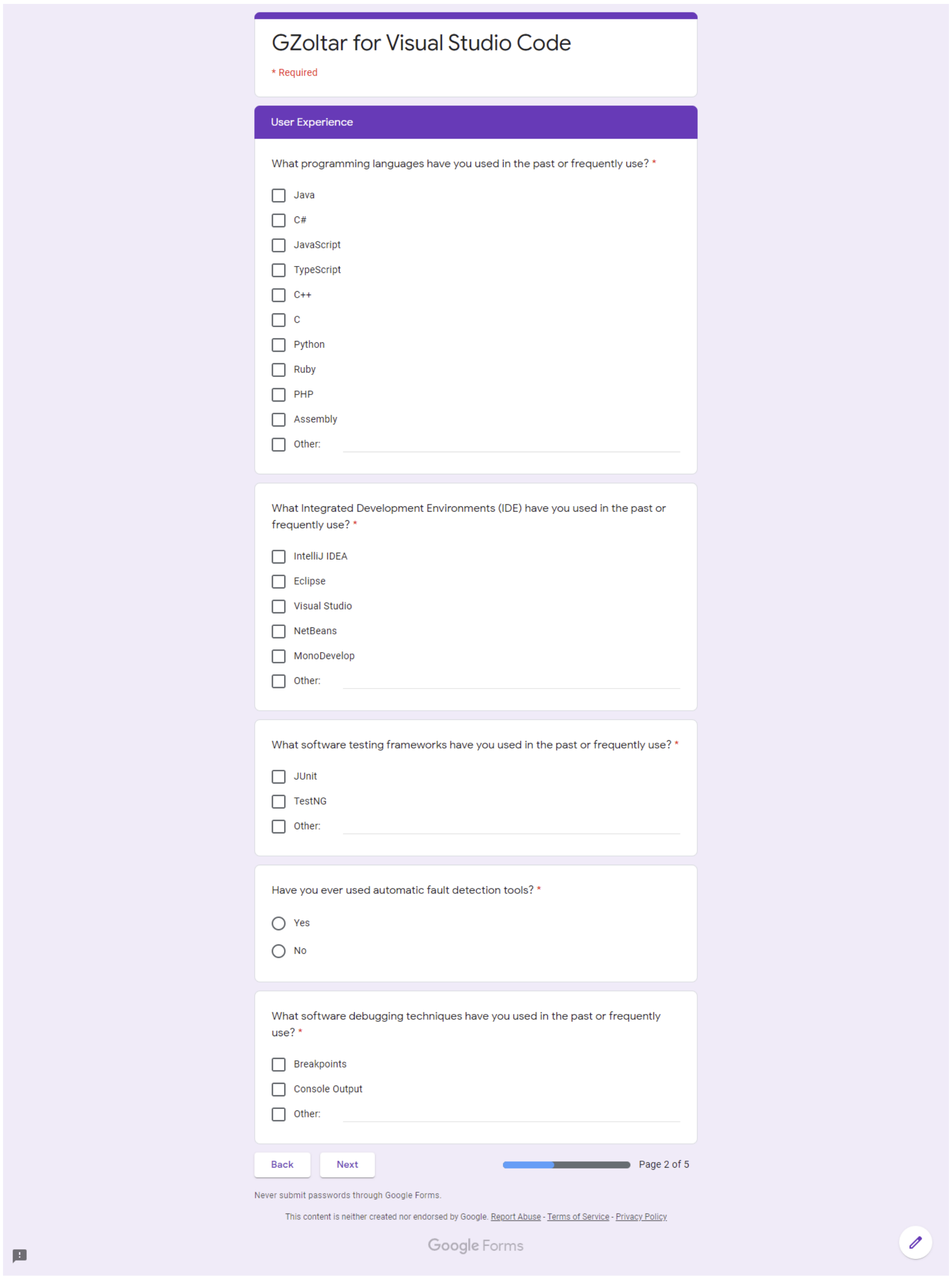
This survey is being carried in the context of a Master's thesis in Computer Science and Engineering at Instituto Superior Técnico (IST). Its main purpose is to test GZoltar's usability as a plugin developed for the code editor Visual Studio Code.

The data collected in this survey will be anonymous and used only for the sake of this user study.

Thank you for your collaboration.

Next
Page 1 of 5
Never submit passwords through Google Forms.
This content is neither created nor endorsed by Google. Report Abuse - Terms of Service - Privacy Policy
Google Forms

Figure A.1: *Introduction.*



GZoltar for Visual Studio Code

* Required

User Experience

What programming languages have you used in the past or frequently use? *

- ☐ Java
- ☐ C#
- ☐ JavaScript
- ☐ TypeScript
- ☐ C++
- ☐ C
- ☐ Python
- ☐ Ruby
- ☐ PHP
- ☐ Assembly
- ☐ Other:

What Integrated Development Environments (IDE) have you used in the past or frequently use? *

- ☐ IntelliJ IDEA
- ☐ Eclipse
- ☐ Visual Studio
- ☐ NetBeans
- ☐ MonoDevelop
- ☐ Other:

What software testing frameworks have you used in the past or frequently use? *

- ☐ JUnit
- ☐ TestNG
- ☐ Other:

Have you ever used automatic fault detection tools? *

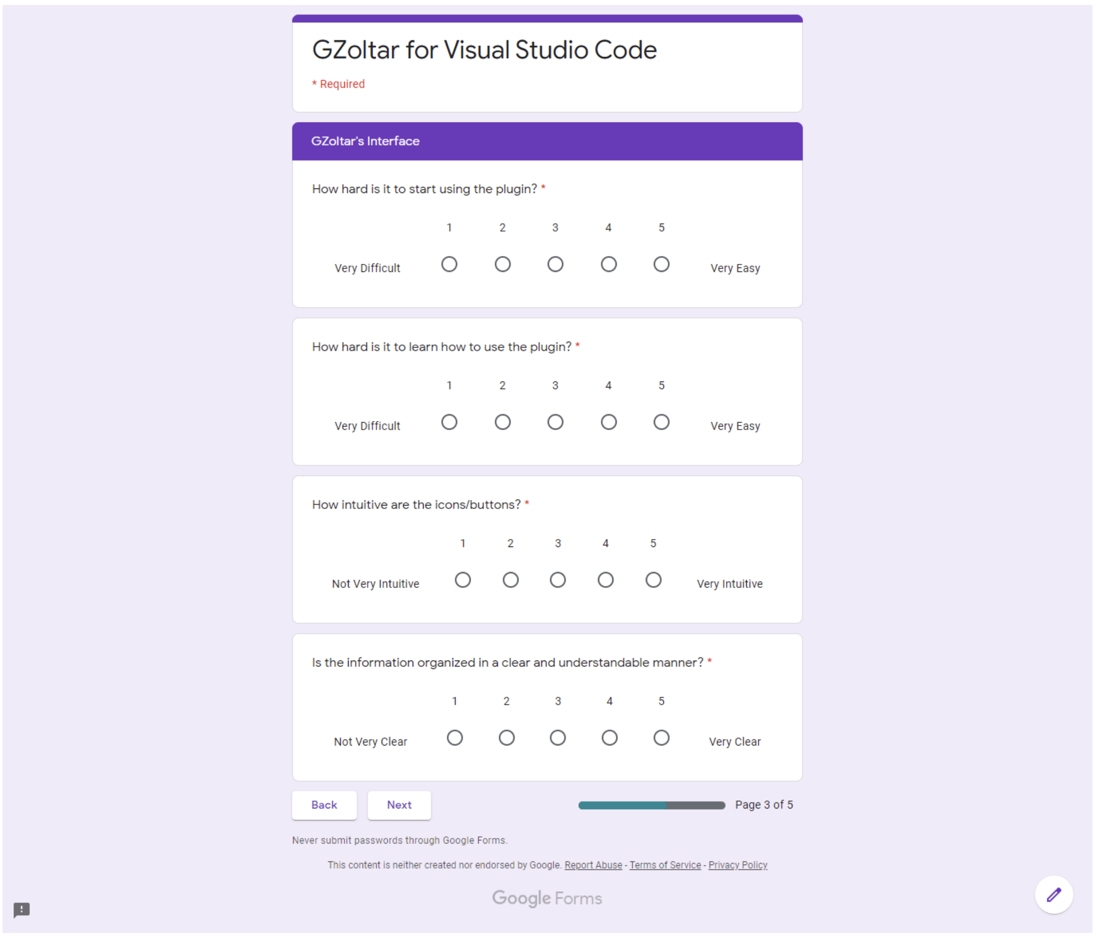
- ○ Yes
- ○ No

What software debugging techniques have you used in the past or frequently use? *

- ☐ Breakpoints
- ☐ Console Output
- ☐ Other:

Back
Next
Page 2 of 5
Never submit passwords through Google Forms.
This content is neither created nor endorsed by Google. Report Abuse - Terms of Service - Privacy Policy
Google Forms

Figure A.2: *User Profile.*


GZoltar for Visual Studio Code

* Required

GZoltar's Interface

How hard is it to start using the plugin? *

	1	2	3	4	5	
Very Difficult	○	○	○	○	○	Very Easy

How hard is it to learn how to use the plugin? *

	1	2	3	4	5	
Very Difficult	○	○	○	○	○	Very Easy

How intuitive are the icons/buttons? *

	1	2	3	4	5	
Not Very Intuitive	○	○	○	○	○	Very Intuitive

Is the information organized in a clear and understandable manner? *

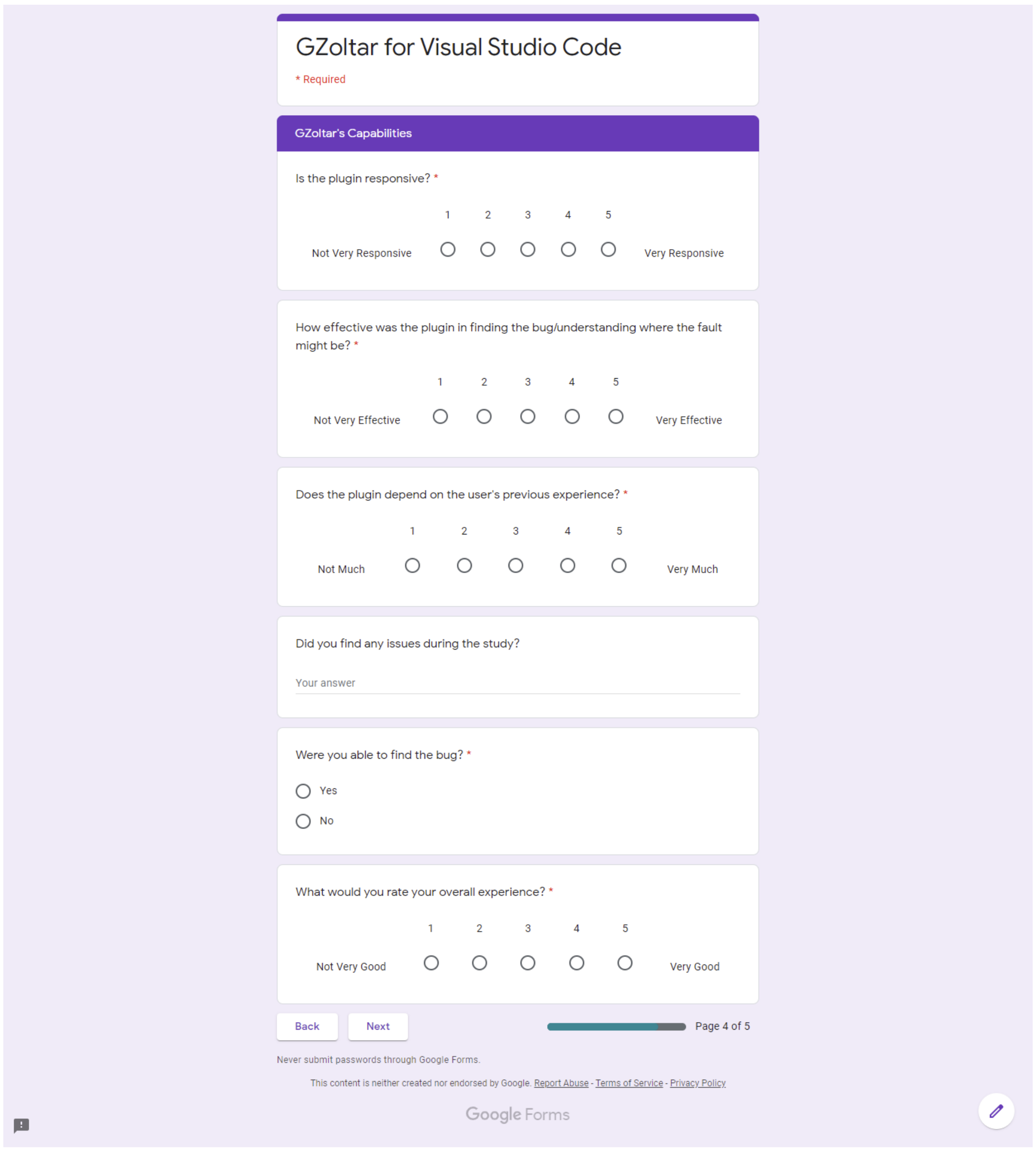
	1	2	3	4	5	
Not Very Clear	○	○	○	○	○	Very Clear

Never submit passwords through Google Forms.

This content is neither created nor endorsed by Google. Report Abuse - Terms of Service - Privacy Policy

Google Forms

Figure A.3: *GZoltar Interface.*



GZoltar for Visual Studio Code

* Required

GZoltar's Capabilities

Is the plugin responsive? *

	1	2	3	4	5	
Not Very Responsive	○	○	○	○	○	Very Responsive

How effective was the plugin in finding the bug/understanding where the fault might be? *

	1	2	3	4	5	
Not Very Effective	○	○	○	○	○	Very Effective

Does the plugin depend on the user's previous experience? *

	1	2	3	4	5	
Not Much	○	○	○	○	○	Very Much

Did you find any issues during the study?

Your answer

Were you able to find the bug? *

○ Yes

○ No

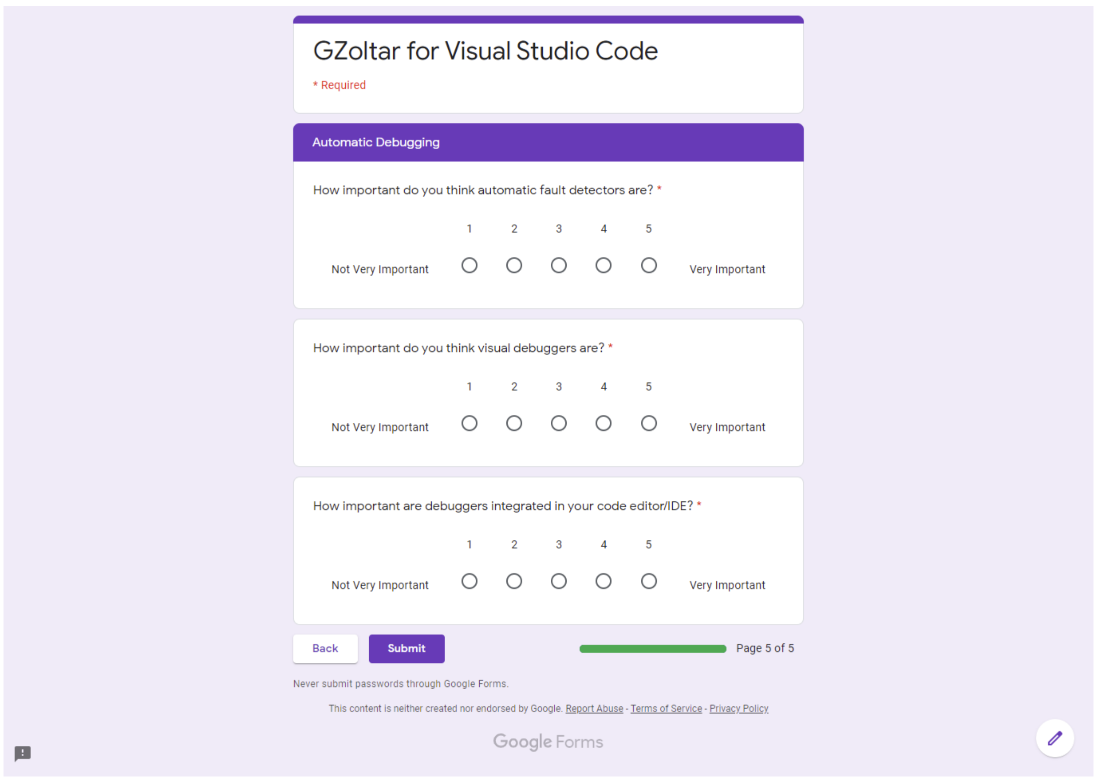
What would you rate your overall experience? *

	1	2	3	4	5	
Not Very Good	○	○	○	○	○	Very Good

Never submit passwords through Google Forms.

This content is neither created nor endorsed by Google. Report Abuse - Terms of Service - Privacy Policy

Figure A.4: *GZoltar Capabilities.*



GZoltar for Visual Studio Code

* Required

Automatic Debugging

How important do you think automatic fault detectors are? *

	1	2	3	4	5	
Not Very Important	○	○	○	○	○	Very Important

How important do you think visual debuggers are? *

	1	2	3	4	5	
Not Very Important	○	○	○	○	○	Very Important

How important are debuggers integrated in your code editor/IDE? *

	1	2	3	4	5	
Not Very Important	○	○	○	○	○	Very Important

Never submit passwords through Google Forms.

This content is neither created nor endorsed by Google. Report Abuse - Terms of Service - Privacy Policy

Google Forms

Figure A.5: *Automatic Debugging Concepts.*