An Automated Debugging Plug-in for Visual Studio Code

Steven Brito Rui Maranhão
Department of Computer Science and Engineering
Instituto Superior Técnico, University of Lisbon
Portugal

***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 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, 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. The plug-in is published in the Visual Studio Marketplace, and a user study was carried out to assess its capabilities. Users 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, they filled a form providing feedback of their experience and also suggestions for future work. The study proves that the extension can be effectively used to locate faults in a program.**

***Index Terms*—Fault Localization, Graphical Debugger, Automatic Testing, Automatic Debugging**

I. Introduction

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. 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 [1]. 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.

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).

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 [2] as the following: SBFL [3], slice-based [4], [5], statistics-based [6], program-state based [7], model-based [8], machine learning-based [9], data mining-based [10], [11] and other miscellaneous techniques.

Although there are several tools which integrate these features [12]–[14], 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.

GZoltar [15] 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 imple-

ments the spectrum-based fault localization (SBFL) technique using the Ochiai algorithm [16], 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 [17], Astor [18], ACS [19], CapGen [20], 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).

The main contributions of this paper 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.

II. Architecture

The extension's architecture is presented in Figure 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 III-A). 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.

III. Building Extensions

VS Code is built using Electron [21], 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 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.

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. The main files which are essential to understanding the extension are *package.json* (extension manifest) and *extension.ts* (entry file).

A. 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

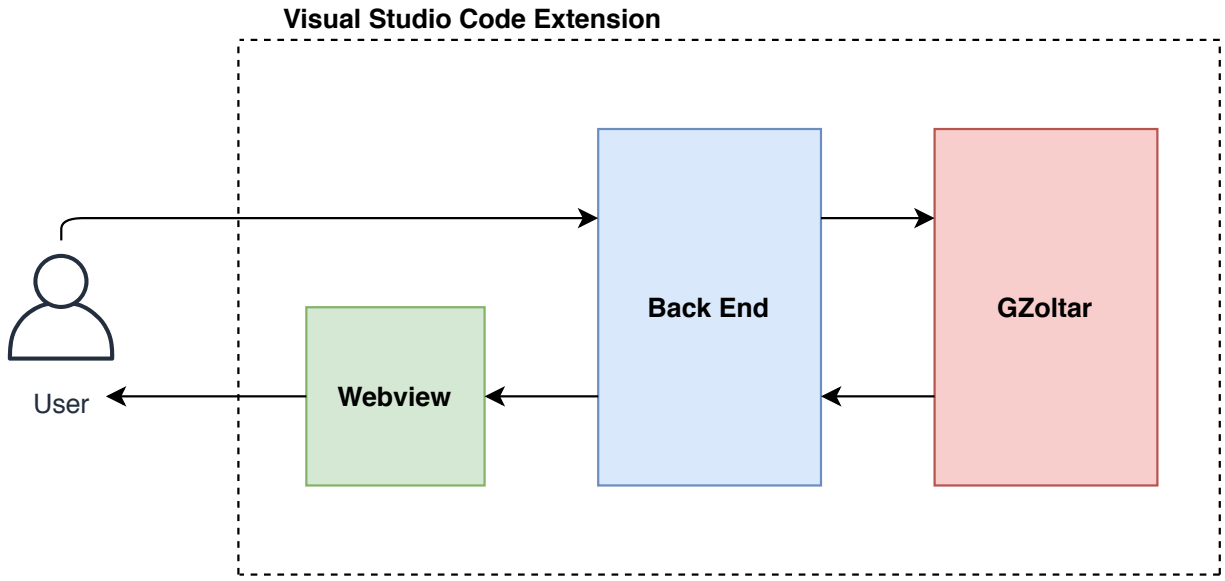

Fig. 1. *Architecture.*

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.

B. 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.

C. 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*.

1) package.json Contribution: As previously mentioned in III-A, 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.

2) 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

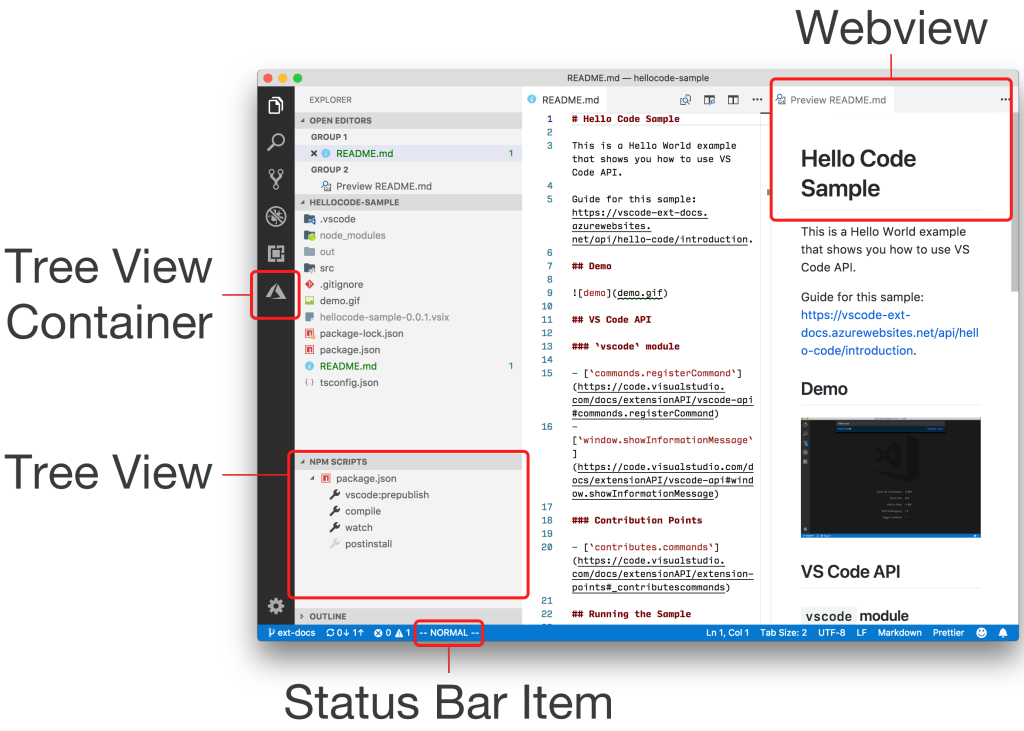

Fig. 2. *VS Code UI.*

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.

3) 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.

4) 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.

D. 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.

E. 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.

1) 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.

2) 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.

IV. Back End

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.

A. Manifest File

As mentioned earlier in III-A, 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 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 [22]. 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.

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.

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. 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.

B. 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* (IV-E), which represents all of the acceptable and currently open Java projects. This instance is passed on to the *GZoltarCommander* (IV-C), 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.

C. 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 IV-A. 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.

D. 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.

E. 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,

Fig. 3. *Suspiciousness icons.*

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.

F. 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.

G. 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. 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 [23], 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.

V. 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. 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.

A. 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.

1) 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, 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. 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.

2) 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.

VI. Results

This section presents the results obtained throughout this project. We will show how the extension's 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.

A. Using the Extension

To access GZoltar's functionalities, we click on GZoltar's icon in the activity bar previously mentioned in IV-A. The menu that is open after that will show every single open and acceptable Java project in the current workspace (shown in Figure 4. To run GZoltar on a specific project, we click on the icon right next to the project's name.

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 5.

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 6). 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 IV-G. 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.

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. The icon's colors are the same as the ones represented in the charts.

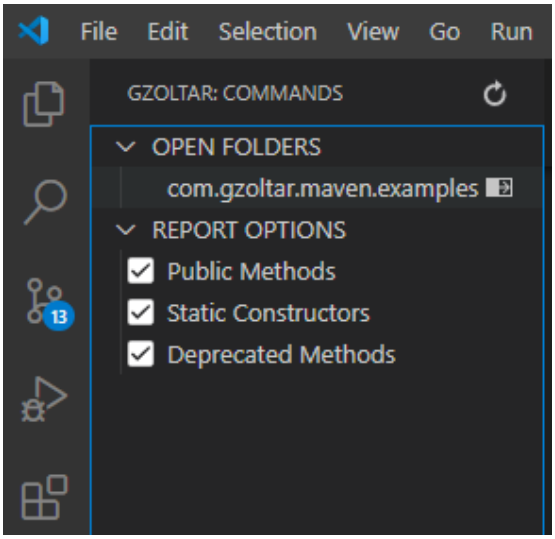

Fig. 4. *GZoltar Commands.*

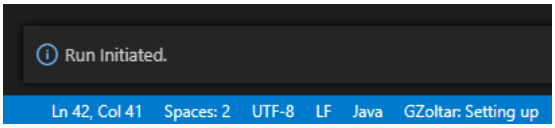

Fig. 5. *GZoltar Status Bar Update.*

B. Feedback

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.

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 [24]. Most of the users were able to find

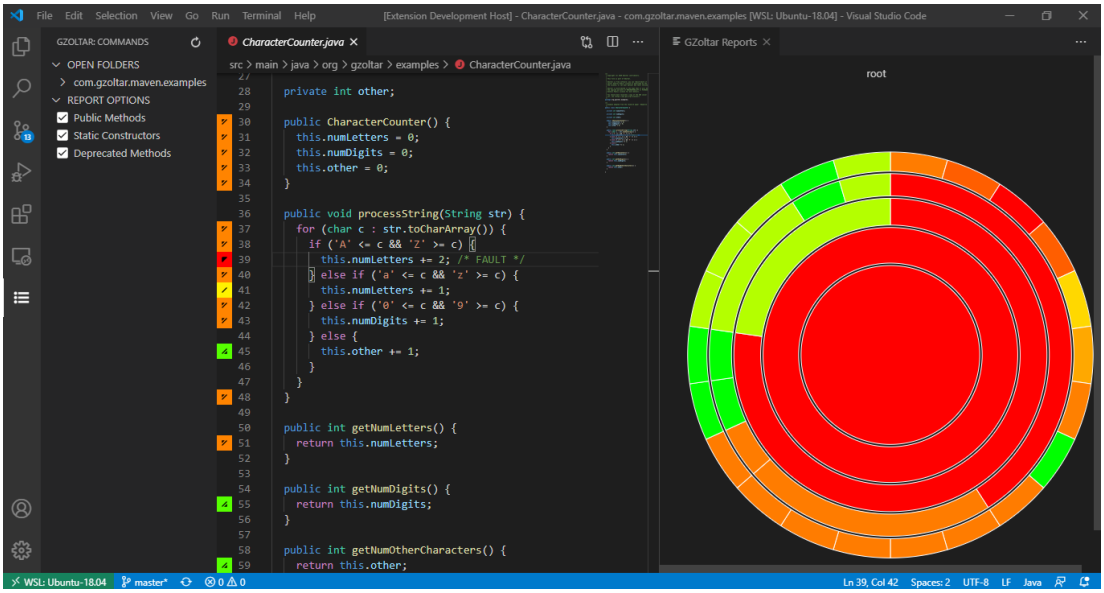

Fig. 6. *GZoltar Result.*

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 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%).

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.

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.

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

VII. Related Work

There have been several tools developed that pose a solution to the problems initially mentioned.

Tarantula [25] 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.

Jaguar [26] 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.

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 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. Finally, it can also mark each line with information about the failure probability of that code block. 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.

VIII. Conclusion

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.

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.

A. 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.

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.

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.

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.

References

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

[2] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[3] 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*, vol. 10, no. 3, pp. 171–194, 2000.

[4] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 1981, pp. 439–449.

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

[6] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Acm Sigplan Notices*, vol. 40. ACM, 2005, pp. 15–26.

[7] S. M. Daniel S. Wilkerson. (2001) Delta tool. [Online; accessed October 30, 2019]. [Online]. Available: http://delta.tigris.org/

[8] R. Reiter, "A theory of diagnosis from first principles," *Artificial intelligence*, vol. 32, no. 1, pp. 57–95, 1987.

[9] W. E. Wong and Y. Qi, "Bp neural network-based effective fault localization," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 04, pp. 573–597, 2009.

[10] 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*. Springer, 2008, pp. 548–559.

[11] 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*. ACM, 2005, pp. 396–399.

[12] 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*, vol. 61, no. 1, pp. 149–169, 2011.

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

[14] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Transactions on software engineering*, vol. 32, no. 10, pp. 831–848, 2006.

[15] 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*. ACM, 2012, pp. 378–381.

[16] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[17] 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*. IEEE Press, 2017, pp. 660–670.

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

[19] 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)*. IEEE, 2017, pp. 416–426.

[20] 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*. ACM, 2018, pp. 1–11.

[21] Github. (2013) Electron. [Online; accessed October 30, 2019]. [Online]. Available: https://electronjs.org/

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

[23] M. Neiva. (2010) Coloradd. [Online; accessed October 21, 2020]. [Online]. Available: https://www.coloradd.net/

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

[25] 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*. IEEE, 2002, pp. 467–477.

[26] 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)*. IEEE, 2018, pp. 404–409.