# Remote Java Debugger Using Java Platform Debugger Architecture

**Robert Dunsmore**

2718125

Dr Simon Jones

*April 2020*

**Dissertation submitted in partial fulfilment for the degree of Bachelor of Science with Honours Software Engineering**

Computing Science and Mathematics
University of Stirling

## Abstract

Debugging is a tedious and time-consuming task that all developers encounter at some stage. Traditionally, developers have two approaches that they can use to locate and rectify errors. The first approach is using additional output statements within normal execution. These statements are typically used to identify specific variable values or other desired information at a specified location. Statements are added to every location of interest. These outputs are then used in conjunction to identify the bug or narrow down its location. In some project scopes this can be extremely useful. However, with large scale projects, the level of effort that is required rises exponentially for little reward. This is commonly the case as these statements must be added and removed from multiple locations.

Alternatively, 'Interactive debuggers' offer developers mass amounts of information about their debuggee program as well as complete control of the execution. Developers can specify exact locations to halt execution of the debuggee. Upon reaching this location, the interactive debugger provides extremely detailed information about the state of the debuggee at the current point of execution. Using the information provided, developers can watch real-time changes to the variables or information that is of interest during execution. The amount of information offered by interactive debuggers can be particularly useful in certain circumstances. However, it is more than likely that developers are only interested in specific information. Unfortunately, interactive debuggers do not filter out undesired information. Interactive debugging sessions can be considerably long, as blocks of executable code will only run when instructed to do so. This approach can become even more time consuming with larger projects, as the level of interaction required by the debugger to execute the remaining code correlates with the scale of the project.

This research project explores the feasibility of a remote scripted debugger. The prototype of this project looks at the possibility of combining both common approaches into a simple but effective debugging tool for Java developers. The aims of this research project are to construct a debugger which allows Java developers to debug their application without editing source code; to debug remotely or locally; to only report back specific information, while reducing the overall time of their debugging sessions. The prototype aims to employ user-defined scripts to identify the desired information as well as the location in which to extract this information. Encapsulating this information using a script allows the debugger to perform as required without the need of source code changes. To reduce the time of debugging sessions, the debugger will automatically start and stop execution at the specified locations without requiring additional interaction from the user. Using the script information, the returning output of the debugger will be filtered to only display the specified information.

The potential of remote scripted debuggers has been recognised by this research project through the construction of an initial prototype exclusively for Java developers. This solution was built upon through the use of the Java Platform Debugging Architecture (JPDA) which supplies the core communication between debugger and debuggee. Users can connect to the target debuggee in various ways and can begin to interact with the debuggee once a connection is established. An XML script can be uploaded which outlines the exact locations of interest as well as specified consequent information for the debugger to output. The debugger will extract the script information and apply it internally, logging the specific information when an identified location of interest is hit. These log files only contain this specified information and their corresponding location of execution. This use of the script eliminates the need to edit source code. Execution of the target debuggee is automatically controlled by the prototype, consequently eradicating unnecessary interactions required from the user. This therefore reduces the overall time spent debugging.

## Attestation

I understand the nature of plagiarism, and I am aware of the University's academic misconduct policy.

I certify that this dissertation reports original work by me during my University project except for the following:

- The JDI connector arguments in Section JDI Connectors 5.1 was taken from [1].
- The implemented debugger structure discussed in Sections 5.3 and 5.4 were derived from example code provided from Dr Simon Jones.
- Inspiration for the format and layout of this dissertation was taken from Rory Taylor's 'The Scripted Debugging of Java Program via the Java Platform Debug Architecture' in September 2017.

**Signature** Robert Dunsmore **Date** *14 April 2020*

## Acknowledgements

Firstly, I would like to thank Dr Simon Jones for highlighting this interesting research topic, as well has his ongoing support and guidance throughout. As well as all of the participants who endured the prototype testing.

Secondly, I want to share my gratitude to the person who helped make completing this dissertation possible, Molly Ross. Without whom, I know that this monumental task would never have come to fruition. Thank you, for always being there for me. From keeping me calm and focussed, to making a delightful dinner every night.

Finally, to my family, in particular my parents, Meg and Robert. For not only supporting me financially but also by encouraging me through my academic career, for which I will be forever grateful.

# Table of Contents

## List of Figures

## List of Tables

# 1 Introduction

Debugging programs can be tedious and difficult, however, it is an essential component of programming [2] [3]. Debugging applications can take a considerable length of time [2] [3], which is a valuable commodity in any project. Debugging is a task that every developer will face, no matter the type, size or complexity of the project. Even with current debugging techniques [3], the process is still slow and time intensive. Not every debugging session will have access to a display output, log files or even local access to the debugging project. There are currently limited automatic debugging tools for local or remote debugging [4] [5] .

This project aims to reduce the amount of time developers spend debugging their applications, by allowing for prepared or custom-built diagnostic scripts to automate the debugging process, locally or remotely.

## 1.1 Background

The term 'debugging' was first coined in 1945 by Grace Murray Hopper, a developer on the Harvard Mark computer, who found a moth inside a relay within the machine which was causing an issue [6]. This gave birth to the term 'bug' and the removal of this bug was the first debugging of a computer system.

From here, debugging at this time mainly consisted of data dumps or outputs limited to basic printer and display lights [2]. The only technique available at this point was the hit-or-miss approach that did not yield a reliable result. As technology advanced and high-level programming languages like C & COBOL became the standard [2], this gave developers the ability to store their programs in individual files. Symbol files were used to map variable names to their assigned memory addresses at run time. Giving developers the ability to dump specific memory locations with the knowledge of what variable this memory address contained, this type of debugging was coined as 'Symbolic Debuggers' [7].

Debuggers were then developed alongside programming technology, as developers always needed to debug their code. The next big revolution in debugging was the introduction to interactive debuggers [2].This new technology provided developers with the ability to add breakpoints, use stepping features and all the common features we use today. This technology was then included into Integrated Development Environment (IDE's) which allowed for one environment that could control code production, compiling as well as run and debugging features [3].

## 1.2 Scope and Objective

The purpose of this project's prototype is to evaluate the feasibility and usefulness that a script-based debugging tool may have for Java developers.

This prototype is to be a standalone debugging tool which developers can utilise during or on completion of their implementation. The premise of the prototype is to encapsulate the best available debugging methods and techniques into one tool. The prototype aims to harness the power and functionality of an IDE debugger, without requiring modification of the source code as well as eradicating irrelevant and unnecessary information, cluttering the user-interface. Essentially creating a hybrid debugging technique. Users will interact primarily with the target debuggee using a diagnostic script. The script specifies the types of interaction that are of interest to the developer as well as identifying the data to be included in the debuggers output.

Reducing the time that a developer spends debugging may not come from providing a new innovative debugging technique, but rather re-envisioning the way we interact with debuggers.

Interactive debugging is an extremely powerful technique commonly used by developers' [8], however, all available functionality is rarely required for common and frequent bugs. Interactive debuggers can examine the state of the Java virtual machine (JVM) at every line of execution, although commonly, developers are only interested in a specific section of the application at one time. Using the interactive debugging technique requires the developer to continuously interact with the debugger to resume the debuggee's execution, even after the code of interest has been executed.

The debugging scripting technique used in this prototype aims to reduce debugging time by allowing developers to specify the information relevant to them for that current debugging session. Using this script information, the debugger would only act when required; reducing the time spent unnecessarily interacting with the debugger. Filtering the output information and storing it in a log file provides a centralised location for the relevant information, eliminating the task of manually locating key data.

## 1.3 Project Achievements

On completion of the project, a functioning prototype has been made available to java developers. The prototype contains a graphical user interface (UI) that facilitates the ability to connect to a target debuggee and execute a valid diagnostic script. Available to the user are multiple connection options depending on the selected connection type, allowing for a connection that suits the user's current situation.

Uploaded script files are parsed to extract the user-specified commands which are used during execution; these are stored locally. For the script file to be valid, it must contain at least one breakpoint and line number command. Once the user instructs the debugger to begin execution, the internal script information is applied to the debuggee. The debugger halts execution based on information extracted from the script file. When the debugger hits a specified location, any identified variables of interest contained within the script are logged. If the script does not identify any variables, the debugger will log all variables at the current location.

The UI allows the user to reconnect to the target debuggee using the same connection information. This feature increases the debugger's effectiveness, as users do not have to repeat the connection process to continue their debugging session. This feature is dependent on the connection type used. Apart from the required click to attempt the reconnection, no other input is required from the user when using a local connection. Remote connectors require the user to relaunch the debuggee but are not required to supply the connection details.

The solution created for this research project has resulted in a finalised prototype which demonstrates the potential of scripted debuggers for Java developers. The prototype illustrates the time that can be saved by automating the debugging prosses, by employing a hybrid of common debugging techniques.

## 1.4 Dissertation outline

This dissertation will explore the background and associated inadequacies of current debugging tools. Highlighting these shortcomings led to the design of specific components.
These designs were then gradually applied to the implementation, establishing the finalised prototype. This completed solution then underwent user testing, the results of which have been examined, influencing the overall evaluation. These chapters have been outlined below:

- ***State of the Art:*** Critical discussion of existing research relevant to this project.
- ***Prototype Design***: Analysis and justification of decisions made in prototype design.
- ***Final Prototype***: Critical explanation of functionality of the final prototype.
- ***Prototype Implementation***: Comprehensive justification of implementation decisions.

- ***Prototype Testing***: Discussion and analysis of tests performed and corresponding results.
- ***Conclusion***: The overall successes and shortcomings of the project with recommendations for future development.

# 2 State of The Art

## 2.1 Debugging

Debugging is a crucial part of the development cycle that every project will be subject to [9]. There are various techniques that developers can use to debug their application, which may increase efficiency such as top-down or bottom-up debugging [9]. However, these techniques do not guarantee success or reduce debugging time.

Testing and debugging are both error detection techniques, however, the difference between testing and debugging is blurred [10]. The two methods identify errors in different ways and have different consequences when errors are identified.

### 2.1.1 Differences in Techniques

| Testing | Debugging |
|---|---|
| Identify failures of implemented code | Supply solutions to failed code |
| Executed by tester | Executed by programmer or developer |
| No need for design knowledge | Requires some knowledge of design |
| Executed inhouse or outsourced | Executed inhouse only |
| Different levels of testing | Different types of bugs |

*Table 1 Differences between Testing and Debugging*

Table 1 Differences between Testing and Debugging above highlights the differences found between these two error detection techniques both of which offer valuable insights into the functionality of the current implementation. Debugging detects errors during development, while testing is performed at the end of the development cycle. Even though debugging can be classed as testing, the two are fundamentally distinct techniques which are not comparable [10].

### 2.1.2 Most Used Debugging Features

As stated previously, the introduction of interactive debuggers provided developers with extremely valuable features which are still some of the most common methods to debug applications today [11]. The features discussed below are the most commonly utilised within interactive debuggers.

#### 2.1.2.1. Breakpoints

Breakpoints are used to stop the debuggee at a specified location, allowing developers to examine the state of their application [2]. Breakpoints allow developers to examine variables or view frames within the call stack when the breakpoint is hit. Breakpoints can be set at any line of executable code; however, they cannot be set on the namespace, class declarations or method signatures [12]. Once a breakpoint is hit, standard step-into, step-over and return debugging functions can be used to resume code execution. Breakpoints can be set on method calls, allowing normal code execution until the specified method is called, stopping the execution and allowing examination [13]. Breakpoints are the foundation of a debugger, as without a specified location to halt execution, no valuable information can be gathered. This prototype utilises breakpoints as a crucial component of the finalised debugger.

#### 2.1.2.2. Method Entry/Exit Breakpoints

These features use the same principle as breakpoints but can be set at specific methods. This may be more efficient, as a breakpoint can be set on a method and if the developer wishes, they can step into the method to debug line by line [14]. While this technique warrants a more thorough process, increasing the overall time of the debugging session can detract developers from implementing this feature. It is for this reason that this feature has been excluded, as time saving is a core aspect of this project.

#### 2.1.2.3. Variables

Knowing which values are stored in different variables at particular states of execution is extremely valuable. This information can be used to ensure that particular values are as expected in this state. If they are not, then this can identify a logical error, not a syntactical or semantical [15]. Every type of variable can be examined, including user defined variables. Similar to breakpoints, identifying variable values is a fundamental aspect of a useful debugger. The absence of variable values essentially makes the task of locating a bug near impossible. It is for this reason that the debugger logs variable values at specified breakpoint locations, which was a critical capability for this project.

#### 2.1.2.4. Stepping

Stepping is a combination of actions that are used to control the flow of execution [2]. The features below are broken down individually and can be used in conjunction or independently. This prototype did not require any stepping functionality, as the core aim of the project is to reduce the time of debugging sessions. As stepping, similarly to method breakpoints, requires the user to identify to the debugger when to continue execution, line by line, this therefore increases the overall time of debugging sessions. All below stepping functions were not implemented as the prototype automatically continues execution until the next breakpoint is hit.

##### *2.1.2.4.1 Step-out*

The step-out function is used when the debugger has stepped into a method call that the developer wishes to ignore, returning them to the location of the initial method call [16]. This feature is useful when stepping into has not identified the issue and normal execution flow is required.

##### *2.1.2.4.2 Step-into*

The step-into feature is used to follow the execution of methods to accomplish the original method [16]. If the executed code does not contain a method call, then similar action to 'step-over' feature will occur. If the code contains a method call, the debugger will jump to the method signature and continue debugging from that point. This will continue until all classes and methods involved have been executed, then returning to the origin of that method call. Alternatively, if the developer has satisfied their curiosity, an alternative stepping feature may be more useful.

##### *2.1.2.4.3 Step-over*

This feature of the stepping allows the current line of execution to be skipped, moving to the next executable line [16]. If the current line includes a method call, the method will be executed and return without debugging each line. A useful feature when developers want to control the flow of execution but are not interested in debugging working code or each individual line.

##### *2.1.2.4.4 Return*

The final stepping feature is used to return from a method which has been stepped into. Even though execution has returned from the method, the remainder of the code inside the method will be executed normally. This stepping feature can reduce the amount of unnecessary interactions with the debugger [2].

#### 2.1.2.5. Watchpoints

This feature allows developers to place special breakpoints on class attributes. Watchpoints are fired when a modification of the attribute occurs. This can be useful when investigating logical errors that involve this attribute [17]. Normal stepping features can also be used in conjunction with watchpoints, however, not all JVMs support this feature.

## 2.2 Current Debugging Software

Various debugging tools exist for the Java language, one of which is directly built into the language and various additional third-party tools, some of which serve multiple languages. The project prototype was heavily influenced by the functionality offered by the Java Platform Debugger Architecture (JPDA). The Java Debugger (JDB) and third-party tools help identify areas in which the project prototype could deliver more efficiently and effectively, predominantly the speed at which these current technologies execute.

### 2.2.1 Java Platform Debugger Architecture

The JPDA is a set of interfaces and protocols designed to provide a foundation for third-party developers to develop debugging tools [18]. The JPDA facilitates all the debugging features available in the Java language, it is primarily intended for desktop users and contains three interfaces, seen in Table 2 JPDA Interfaces below. The structure of the JPDA can be seen in Figure 1 JPDA Overview.

| | **Java Virtual Machine Tools interface (JVMTI)** | **Java Debug Interface (JDI)** | **Java Debug Wire Protocol (JDWP)** |
|---|---|---|---|
| **Defines** | All the services that a virtual machine must supply to be debugged | requests and information at user code level | the format of information |
| **Component** | Backend | Frontend | Communication channel |
| **Role** | Debuggee | Debugger | Handles transfer of information between the backend and frontend |

*Table 2 JPDA Interfaces*

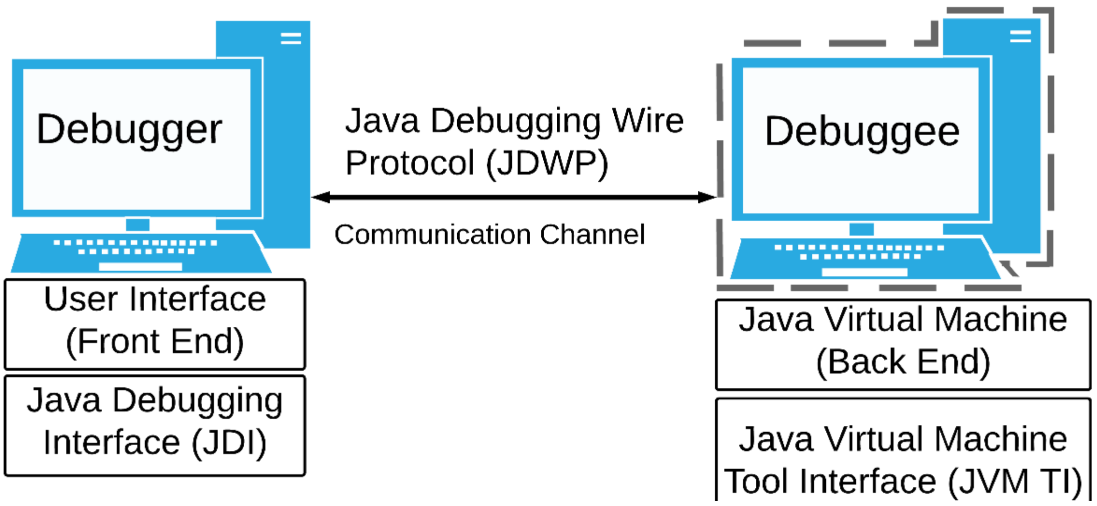

Debugger
Java Debugging Wire Protocol (JDWP)
Communication Channel
Debuggee
User Interface (Front End)
Java Debugging Interface (JDI)
Java Virtual Machine (Back End)
Java Virtual Machine Tool Interface (JVM TI)

*Figure 1 JPDA Overview*

The JPDA is widely used, being the chosen architecture in numerous academic studies [19] [20] [21]. The JPDA is the fundamental debugging architecture built into the Java language, restricting the functionality as it is strictly for Java applications only.

This architecture allows developers to focus on their debugger implementation, as they do not have to consider individual computer configurations. The JPDA is divided into two parts; a frontend (JDI) and a backend (JVMTI) with a communication channel (JDWP) between them to transport data. The frontend is responsible for making the connection to the backend, which can be established in three ways [18], seen Table 3 JPDA Connection Types below.

| Listening | Attaching | Launching |
|---|---|---|
| The frontend waits and listens for a connection from the backend | The frontend attaches to the already running backend | The frontend launches the backend process and shares memory locations to the virtual machine |

*Table 3 JPDA Connection Types*

The JDI interface is the highest layer of the JPDA [22] it facilitates debugging features. All debugging aspects performed by a debugger will be achieved through the use of this interface. Establishing a connection, making requests, controlling execution and much more is accessed through the JDI interface. This architecture was selected for this project as the JPDA already includes fundamental aspects of a debugger, reducing the complexity and overall time of implementation.

### 2.2.2 Java Debugger

The Java Debugger (JDB) is a simple command line debugger. The construction of which was to demonstrate the capability of the JPDA; facilitating the ability to inspect target JVMs that were connected locally or remotely. JDB is supplied as standard within the Standard Edition Development Kit (JDK) with no requirements to external libraries [23]. The JPDA facilitates the interaction and communication between JVMs. The command line interface enables developers to specify which command to fire at the target JVM.

JDB provides the capability to accomplish the most used debugging features, using key words, seen in Table 4 JDB Commandsbelow.

| Key Words | Description |
|---|---|
| help / ? | Lists JDB commands, with a small description of the commands |
| run | Runs the debuggee application and will stop once a defined breakpoint/watchpoint is hit |
| print | Outputs Java objects or primitive values |
| dump | Dumping primitive values are treated as a print, dumping objects; will output that objects attribute values |
| threads | Lists all running threads |
| stop | Adds breakpoints at a specific location |

*Table 4 JDB Commands*

These key words are the commands that JDB uses to perform the debugging actions [23]. However, these commands require additional syntax and parameters to perform. Parameters include the name of the target class, the line number, method name or variable name. The syntax includes ‘in’, ‘at’ and ‘all’. These syntax operators indicate to the JDB which kind of action the command should perform. Attempting to use the JDB to perform the required actions of the debugger would add an unnecessary additional layer of complexity to the prototype, therefore increasing the overall time of the debugging session. The functionality offered by the JDB was avoided by directly utilising the JPDA in the project's prototype.

### 2.2.3 Industry Debuggers

Many mainstreams IDEs such as IBM Eclipse, use aspects of the JPDA when debugging java applications [24]; all the features previously discussed are included in these debuggers. The example shown in Appendix 1 – Eclipse Debugger View has been taken from the Eclipse IDE for java developers.

The screenshot shown in Appendix 1 – Eclipse Debugger View is a simple 'Hello World' java program which holds three string variables and displays the concatenation of the strings. This project has been launched in a debugging state which has switched to the Eclipse debugger view [25].

- Breakpoint
  - The first break point was added to line nine in the source code. When Eclipse hits this breakpoint, the view changes to the debugging view. Execution is halted at this location.
- Variables
  - The variable pane shows two string variables; 'args', which is used by the main method and 'JPDA' which has been initialised and displayed to the console.
- Stepping Features
  - This is where the developer can control execution flow. The continue/return feature is greyed out as there is no method call that has been stepped-into or second breakpoint, so the debugger has nowhere to return to as it is already at the top level of execution.
- Output
  - This is a console output that will display the program output. As seen from the screenshot, only one string has been outputted, meaning that the breakpoint has been hit and is now waiting for the instruction from the developer.

Eclipse is using the JPDA to accomplish the debugging tasks on the application. As this example was stored locally on the machine, the connection type was of a shared memory location. Eclipse launches the program independently in a debugging state, facilitating the interaction through its debugging interface.

IDE debuggers are extremely powerful tools that offer a wide range of control and actions that can be performed on the debuggee. These debuggers, however, require developers to continually interact with the debugger to identify when to resume execution. This requirement increases the overall time of the debugging session considerably, as instructions need to be continually fired at the debugger. Little inspiration was taken from this technique as this project aims to reduce the time of debugging sessions by automatically resuming execution, without the need of interaction from the developer.

## 2.3 Diagnostic Scripts

This project aims to radically reduce the amount of time developers spend debugging their application by executing diagnostic scripts, which will return valuable information about the state of the application. The project will provide generic scripts that developers can use for a quick result. The project will also allow developers to customize the provided script or build complete custom scripts to debug their application more specifically.

Extensible Markup Language (XML) [26] is a markup language which is user defined and used to transport data [27]. XML is not an executable language but was first built to transport data between systems. XML uses tags similarly to HTML; however, XML does not have any defined tags; all tags are user defined. This makes the language extremely flexible as it can be manipulated to accomplish the required task.

As stated previously, XML on its own is not an executable language, however there are XML scripting libraries. Apache Jelly is a scripting and processing engine that can parse XML files into runnable scripts through XML and Java [28]. Jelly is an extremely powerful scripting tool that can be used to perform a mass number of different tasks. Jelly works cooperatively with other programming features such as ANT, Maven, workflow and many more [28].

Jelly allows for dynamic variables within an XML to be modified during runtime. This feature would be useful when trying to connect to the debuggee machine, as this script could supply the debuggee with debugging parameters when attaching or launching the debuggee JVM. Jelly could also be used to run diagnostic scripts as the XML script can call methods within the debugger VM to perform required tasks [28].

Apache Jelly does offer extremely powerful features that the debugger could eventually utilise, but this would require a considerable investment of time to understand how to incorporate its functionality. The debugger can achieve the same result using a simple XML document. Due to the time constraints on the project, it was decided not to use Apache Jelly for these reasons.

# 3 Prototype Design

Designs for this prototype are built upon the functionality offered by the JPDA. This architecture was selected as it has been extremely effective within the industry, as aspects of the JPDA have been incorporated into Oracle JDeveloper [29], IntelliJ IDEA [30] and IBM Eclipse [24] built-in debuggers.

## 3.1 Requirements

The requirements for this project were formed from the initial project outline, as well as through further discussion with Dr Simon Jones. The description of this project highlighted the interesting research potential that a scripted debugger could provide. The requirements of the project were broken down into functional and non-functional, insuring a concise set of requirements for the completed project prototype.

### 3.1.1 Functional

These requirements directly impact the type of actions that the prototype of this project should facilitate once completed. These functional requirements are seen below.

1. Connect to a target debuggee, see Table 3 JPDA Connection Typesfor more details.
    1.1. Attach
    1.2. Listen
    1.3. Launch
2. Upload diagnostic scripts
3. Execute script commands on target debuggee
4. Log debugging variable information

The reason that multiple connection possibilities are important is that it makes the debugger more adaptable for individual developers' circumstances, as well has highlighting the potential of remote debuggers.

The prototype must have the ability for users to communicate the types of actions they would like to be performed. This is why the prototype must facilitate the ability to upload and execute script commands.

In order for a debugger to be of any value to the developer, it must offer helpful information such as variable values. This is why the prototype logs these values at a centralised location to allow for the developer to pinpoint this variable information.

### 3.1.2 Non-Functional

These requirements do not directly correspond to prototype functionality, but rather requirements that should be considered during construction.

1. Graphical User Interface (UI)
    1.1. Quick production expansion
2. Customisable scripts
    2.1. Contain at least one breakpoint
    2.2. Contain at least one-line number
3. Readable log files
    3.1. Debugger log
    3.2. Debuggee log

Providing a UI for users to interact with the debugger increases the likely hood of a successful debugging session. An additional benefit is that new users will easily understand the actions they must perform, such as establishing a connection to the debuggee, in addition to identifying the non-essential actions.

Customisable scripts would again increase the flexibility of the debugger, by allowing users to write specific scripts for their application, rather than just providing a generic script that could be applied to every debuggee. For the script to be of any use, it must specify a location to halt execution for information to be logged. It is for this reason that in order for the script to be valid, it must contain at least one breakpoint and line location.

Individual log files make it easier for the developer to locate the desired information for both the debugger and debuggee. Therefore, decreasing the overall time of the debugging session by alleviating the task of manually locating this information.

The final project application will be compared against both sets of requirements to determine the quality and success of the complete prototype.

## 3.2 Top-level

Using the specified requirements above, top-level diagrams were created, visualising the required core functionality. During design development, Appendix 2 – Debugger Use Case Diagram and Appendix 3 – Debuggee Use Case Diagra were referred to consistently, which influenced many implementation decisions. Individual diagrams were created for the debugger and debuggee, solidifying the top-level design.

### 3.2.1 Debugger Use Case

All functionality is dependent upon the target's connection status. Without a connection, the debugger cannot begin a debugging session. Once a connection has been established, the user will have full access to all functionality.

The debugger UI will facilitate the ability to establish a connection in three different approaches, supported by the JPDA, which can be seen in Appendix 2 – Debugger Use Case Diagram. Although, the debugger can only support a single connection to a debuggee at one time, which can only be established via one of these approaches. A connection must be established before the UI will display the available actions the user can perform.

Once connected, the primary actions of uploading and executing a script are made available to the user. Uploading a script will cause the debugger to extract the breakpoints, watchpoints and any specified variables. This information is then stored internally to be used later when interrogating the debuggee.

Execution of the script causes the debugger to use the extracted script information to interrogate the debuggee at specified locations. Identified variable values are then logged at these locations.

The alternative action of retrieving the line numbers of the debuggee is also made available once a connection is established. This action does not place any constraints on the debuggee or any impact on the log file, as it is an independent action.

### 3.2.2 **Debuggee Use Case**

Once again, Appendix 3 – Debuggee Use Case Diagra illustrates that a debugging session cannot begin until a connection is established. The debuggee has various opportunities to establish a connection, however, the ability to launch the debugger is not available. Line number requests made from the debugger are independent from script requests, as they are intended to provide aid with script development. All other debugging requests will be contained within the script.

On execution of the script, the specified breakpoint locations are applied. The debuggee begins the main execution. On reaching a specified breakpoint location, the variables and values of the current state of execution is examined, relevant information is then logged and written to the log file. Debuggee execution is automatically resumed. The cycle continues until the debuggee's execution has completed.

## 3.3 **User Interface**

JavaFX was chosen to create the UI as it allows for rapid development using FXML (JavaFX layout) documents to create the UI. Third party provider, Gluon, offers Scene Builder; a powerful visual layout tool that automatically generates valid FXML files [31]. These documents can be additionally styled using external Cascading Style Sheet (CSS) files, allowing for a custom design. The JavaFX API provides quick and easy implementations of UI elements. JavaFX makes use of the Model View-Controller (MVC) architecture which means that the user interface can be manipulated during runtime. When the controller of the user interface executes a method that affects the model, it is immediately reflected by the view [32].

JavaFX provides quick and powerful UI elements that the debugger could take advantage of, improving the overall efficiency of a debugging session. 'Alerts' [33] provide simple to implement alert prompts to the user for displaying information, prompting a decision or input. These alerts will allow the debugger to gather information from the user outside of the script file.

The JavaFX 'FileChooser' [34] dialog supports the ability to select a single file or multiple files anywhere on the hosting operating system. This would allow the user to visually select the desired script file. The file chooser can be implemented with a few lines of code and can have the CSS styles applied to it.

Using JavaFX allows for the main focus of implementation to be on the core aspects of the debugger, rather than losing valuable time on implementing a comprehensive UI.

## 3.4 **JPDA Integration**

The JPDA libraries that Oracle supply provide a solution to complex debugging and JVM managing problem. This consequently makes the implementation slightly inconvenient, as the JPDA is inherently abstract to facilitate all JVM configurations. The provided solution far outweighs the inconveniences that it presents.

The JDI interface is the main form of contact between the debugger and JPDA, however, the JDI will internally call other fundamental libraries. The high level of abstraction in the JDI construction creates a considerable amount of interface interweaving.

The JDI interface used to represent objects in the debuggee's VM is the *Mirror* interface [35]. This interface is the root element for the JDI package and can be used as a proxy for the entire JVM or individual aspects. Implementation is left to the discretion of the client of the interface. The *Mirror* interface has 84 sub interfaces, some of which are additional super interfaces.

The *VirtualMachine* interface that extends *Mirror* is used to represent the composite state of the debuggee JVM [36]. Subsequent JDI objects representing internal aspects of the debuggee that extend *Mirror* are direct or indirect extensions of the *VirtualMachine* implementation [36]. Within the architecture of the prototype, there will be one instance of the *VirtualMachine* interface that will encapsulate the entirety of the target JVM, this object will be used to retrieve *Events* from the debuggee.

The JDI structure solves the fundamental timing problems that can occur when debugging JVM's. As the debugger and debuggee are running independently, the problem of synchronisation becomes a concern. The problem of consistency is inherently resolved by the JDIs strict request and event approach. Within each target JVM there is exactly one *EventRequestManager* [37] which is responsible for processing the debugger request events. These events are actions that the debugger commands from the debuggee. The events requested by the debugger are *EventRequest* [38] objects that use the *EventRequestManager* to be enforced upon the target debuggee. The *EventRequestManager* waits for completion of this request, ensuring that no other requests are expected from the debuggee until completion of the current request. On completion, the target debuggee generates a corresponding *EventSet* [39]. The *EventSet* contains *Event* objects related to that request and places them on the *EventQueue* [40] of the debuggee.

The debugger will constantly extract these *EventSet*(s) from the debuggee *EventQueue* while still connected. Depending on the types of *Events* contained within the *EventSet,* the debugger will perform the desired operations. The JDI controls the flow of the debuggee JVM by halting execution when the debugger is examining the events contained in the current *EventSet*. This is how the JDI can ensure that the independent JVMs are synchronised.

This implementation will be used by the debugger to request breakpoint events, watchpoint events and many more event requests. Once the requests have been satisfied and the debugger is no longer examining the returning *Events,* debuggee execution can resume. This approach ensures that while the debugger is extracting information, execution of the debuggee is halted and information can be gathered with the guarantee that the debuggee has not modified the data.

## 3.5 Class Diagram

A preliminary class diagram was created to visualise the potential structure of the prototype, seen in Appendix 4 - Initial Class Diagram.

The *MainController* class represents the UI that will, gather the connection information used to establish a connection, as well as providing the core interaction with the user for uploading and executing script files. There will be one single instance of the *MainController* class that will control interactions between the user and debugger, as well as between the debugger and debuggee. The *ConnectionManager* has the responsibility of establishing the connection to the target JVM, using the gather connection information from the *MainController*. The type of connection to be attempted will be determined by evaluating the supplied connection information. Only one type of connection can be attempted at any one time. There will only ever be one debuggee connected for a given debugging session. A successful connection will result in an instance of the *Debuggee* class being created, to encapsulate the target JVM.

On creation of the *MainController,* a single instance of the *XMLScriptReader* is instantiated, to provide the ability to extract the script information. When a script is uploaded, the *XMLScriptReader* is called to extract the information and apply it to the internal instance of the *Script* object. All of the extracting script information is stored within this *Script* object. Throughout the debugger's life, the user can upload multiple script files. However, only one of the scripts can be internally represented at one time.

The class diagram, seen in Appendix 4 - Initial Class Diagram, was created is as an initial concept of design that evolves with each version of the prototype.

## 3.6 Script Design

XML's popularity made for an easy decision when considering which scripting language to use. The popularity of XML should mean that developers will at least be familiar with XML syntax. XML provides more than enough functionality for this research project, as the prototype will initially have limited debugging commands. Another benefit of XML's esteem is that XML parsers already exist for the Java language, eradicating the need for a custom parser for the script.

The parser will follow the Document Object Model (DOM) [41]. The script file will be broken down into DOM objects based on the node elements contained within the script. These objects are linked together in a tree structure, which can be traversed bidirectionally to interact with script information. This technique will allow for rapid expansion of script functionality in later development.

Capability of the script will be limited to key components, illustrating the potential of the research project. The script will facilitate five tags users can use to specify actions; they expect to be enforced upon the debuggee. These tags are the core structure of the script, any expansion of script functionality will extend this foundation. Table 5 Acceptable Script Tags below show the tags intended to be the command instructions available to the user.

| Tag Name | Tag Description | Required |
|---|---|---|
| script | The root tag of the script file; all other tags are child nodes of this tag | Yes |
| breakpoint | To indicate a break point request | Yes |
| watchpoint | To indicate a watch point request | No |
| lineNumber | Child node of breakpoint and watchpoint; used to indicate the location to halt execution | Yes |
| variableName | Child node of breakpoint and watchpoint; used to indicate variables of interest to be outputted | No |

*Table 5 Acceptable Script Tags*

The debugger will not offer any UI functionality for development of the diagnostic script, as this is far beyond the scope of the research project. The user will develop the script using their preferred text editor. Flexibility is fundamental in allowing for quick and easy script development during ongoing debugging sessions. To facilitate this, attributes relating to breakpoints will relate to all breakpoint requests. This gives users various ways to perform requests that will result in the same outcome.

The script must include at least one *lineNumber* attribute contained within a *breakpoint* tag, otherwise the debuggee will execute as normal, gathering no relevant debugging information. The breakpoint tag contains one or all desired breakpoint line locations. The specified locations are treated as individual breakpoint requests. Similarly, variables of interest can be identified in one breakpoint tag or scattered across multiple requests.

Once created, the XML script can be uploaded after a connection to the debuggee has been established. A template script will be provided with the debugger installation to illustrate the script structure, as well as providing a simple example of a diagnostic script. An example can be seen in Figure 2 Example Scriptbelow.

```
<?xml version="1.0" encoding="UTF-8"?>
<script>
        <breakpoint>
                <lineNumber>29</lineNumber>
                <lineNumber>35</lineNumber>
                <variableName>variable1</variableName>
        </breakpoint>
        <breakpoint>
                <lineNumber>55</lineNumber>
                <variableName>variable2</variableName>
                <variableName>variable3</variableName>
        </breakpoint>
        <watchpoint>
                <variableName>variable4</variableName>
        </watchpoint>
</script>
```

*Figure 2 Example Script*

Using this example, the script informs the debugger to request that the debuggee set breakpoints at source code location 29, 35 and 55. Four variables of interest have been identified within the script, three variables relating to breakpoints and one associated with a watchpoint. The breakpoint variables will be the only variables logged once debuggee execution reaches a specified breakpoint location. The location at which the value was read will be logged to identify where in execution this value was recorded from. All of the identified variables; ‘variable1’, ‘variable2’ and ‘variable3’ will have their values logged at every breakpoint location. Variables that do not exist at a breakpoint location will be ignored. A watchpoint on the variable ‘variable4’ is being requested, indicating the debugger to log any changes made to this variable the location of the modification.

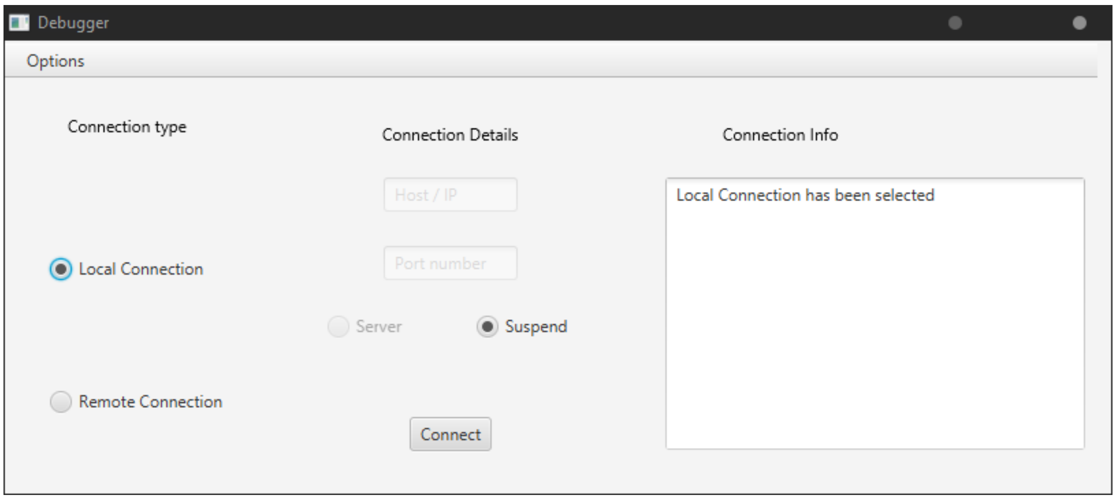
# 4 Final Prototype

The current prototype illustrates the potential of scripted debuggers, by providing a reasonable implementation of an automatic scripted debugger. The implemented debugger facilitates the ability to connect to a target debuggee, internalise script instructions, apply instructions to the debuggee and log results.

The prototype generates two log files; one for the debugger, which logs actions such as result of connection attempts, connection restarts etc. The second log file contains information relating to the debuggee. The information logged consists of variable names, values and the location at which the value was read.

Once the connection to the target JVM has been established, the ability to interact with the debuggee is made available to the user. Currently the prototype has limited interaction functionality as the project simply aims to highlight the potential of scripted debuggers.

## 4.1 Connection Page



Local Connection has been selected

*Figure 3 Debugger Connection Page*

When launched, the prototype displays the connection page. From this page, the user can provide the required connection details needed to attempt a connection to the target debuggee. To establish a connection, a 'Connection Type' must be selected. The user has the ability to establish a connection in three ways. One establishes a local connection by launching a debuggee on the same operating system (OS), sharing the same resources. Alternatively, two remote connections are available. One to listen for a debuggee's connection and one to attach to a debuggee.

When a local connection type is selected, the debugger automatically disables the unnecessary UI elements when attempting this type of connection. As the debuggee file is stored locally, the user is not required to provide an IP address or port number. Instead, the file path of the debuggee is required. To gather this information, the debugger utilises the JavaFX file chooser. The file chooser visually represents the OS's file structure to allow the user to visually navigate to the target debuggee file. The file chooser is presented to the user when they actuate the 'Connect' button seen in Figure 3 Debugger Connection Page. The ability to suspend execution of the debuggee's main thread is always available to the user before establishing a local connection. Suspending the debuggee is a valuable constraint that will most likely be used in every situation, the option however, was left to the discretion of the user to increase the flexibility of the debugger. The debugger utilises the JDI *Launching* connector [42] to perform this type of connection.

The debugger determines the type of remote connection the user is attempting by evaluating the 'Server' toggle. This toggle determines if the debugger is the server in this session. A server in a remote debugging session is the host, the IP address of which is used by the target (client) to locate the server. The port identifies where the server will accept communication. A server waits for a remote target to attempt to connect via the specified connection details. Having the 'Server' toggle selected will cause the debugger to act as the server. Once again, the debugger's UI automatically reacts by disabling the IP address input field, as the debugger will use the IP address of the local machine. The debugger disables the 'Suspend' toggle as this constraint must be set before the debuggee is launched. If the connection type is remote, the debugger cannot ensure this constraint is applied. The port input field is still active as it is required to identify which port to listen for a connection. The debugger utilises the JDI *SocketListen* connector [43] to perform this type of connection.

Alternatively, deselecting the 'Server' toggle when attempting a connection indicates that the debugger should attempt to connect to the specified server (the waiting debuggee) using the supplied information. The user is required to provide the IP address of the host and port number in order for the debugger to attempt the attachment. Supplied information is validated once the user clicks the ‘Connect’ button. For the IP address to be valid it must conform to the IPv4 format. One exception that the debugger accepts as a valid IP is ‘localHost’, this exception causes the debugger to use the address of the local machine. This allows for the communication channel to be established as if it was a remote connection, however, both parties are contained within the same local environment. Integers are the only input values accepted for the port number. The JDI connector used to perform this type of connection is the *SocketAttach* connector [44].

For remote connections, the debugger requires the user enter the name of the class of the remote debuggee. This information is required to be used later when filtering class prepare events. This value must exactly match the name of the target class or the debugger will fail to locate the desired class. The debugger is also reliant on the debuggee to apply the suspend contract; to halt execution once a connection is established. Without this constraint, debugger will not have control over the debuggee’s execution, making debugging impossible.

A connection is only attempted once the user clicks the ‘Connect’ button. Input is validated before attempting to apply the supplied values to the relevant JDI connector arguments. For local connections, the chosen file extension is validated. The only acceptable file extension is ‘.class’ files. As class files are the compiled version of the debuggee, the debugger can launch a JVM to execute the target class.

Additional options are available for configuration via this page, including the log location and log level. A custom log location will result in both debugger and debuggee logs being written to the specified location. The log level will limit the type of information written to the log file, adding an extra layer of control to the generated output. As the results of actions performed on this page are related to the debugger, any relevant information will be added to the debugger log file.

## 4.2 **Interactive Page**

Once a connection has been successfully established to the target debuggee, the interactive page is displayed to the user, seen in Figure 4 Debugger Interactive Page below. From this page, the user can retrieve the line numbers from the target debuggee, upload diagnostic scripts and execute scripts. To increase the effectiveness of the debugger, the user can reconnect to the same target with the same connection information. This option was implemented to allow the user to execute multiple diagnostic scripts on the same target, without reperforming the connection stage. Reconnecting to a debuggee that initially used a local connection type does not require any interaction from the user. Connecting using either of the remote connections requires the user to attempt to reattach the debuggee.

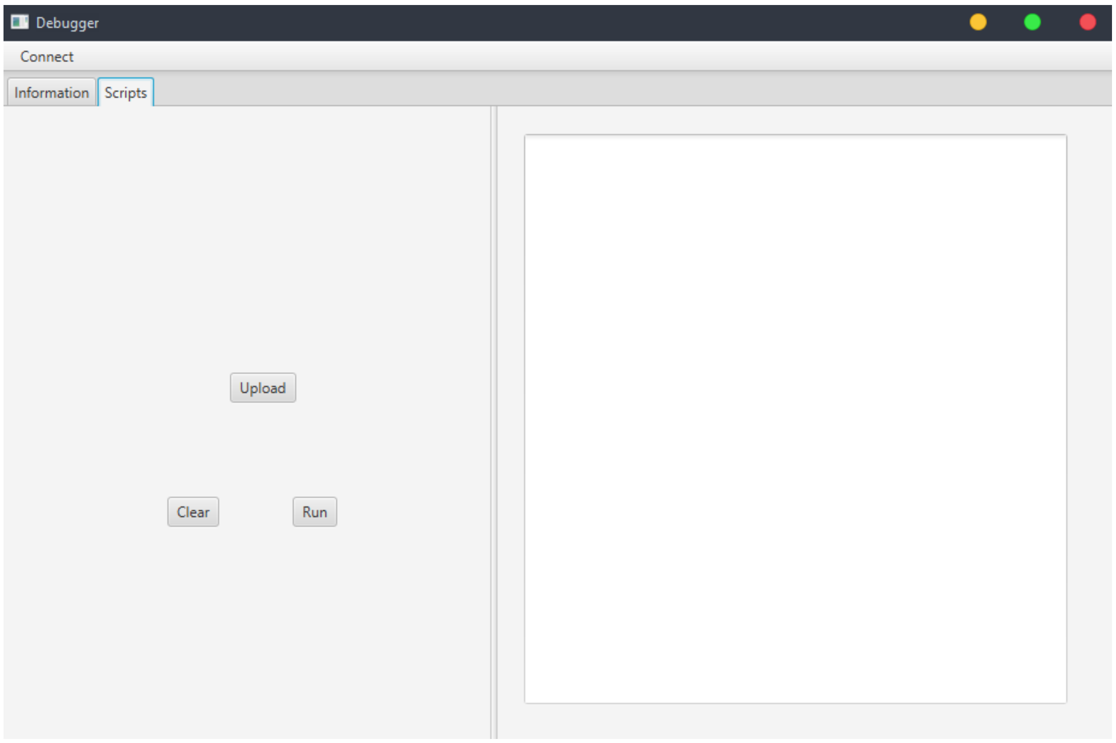


*Figure 4 Debugger Interactive Page*

The ability to view source code line number information from the target debuggee allows developers to design diagnostic scripts for applications that they might not have direct access to. Once users of the debugger have this information, they can create a diagnostic script based on the line numbers relating to the target debuggee. On execution of the script, the results are written to a log file, the user can then examine this file to discover the specific information outlined in the script.

Diagnostic scripts can be uploaded and executed from this page. The supplied script must be an XML file, with specific tags that the debugger accepts. On a successful upload, the user is notified, and the script can be executed on the target JVM. The debuggee will begin its execution under the control of the script instructions. Information relating to the debuggee is then written to the debuggee log file and similarly with the debugger.

## 4.3 Debugging Example

An example of how the debugger prototype can identify logical errors is based on the debuggee code in Appendix 5 – Debuggee Example. The simple debuggee program attempts to count the number of specified letters in a given word. In this example it attempts to count the number of time 'g' appears in 'Debugging'. The debuggee was compiled with debugging commands enabled by including '-g' in the command line arguments. The command used to compile the debuggee was *'javac -g Debuggee.java'*. When the debuggee code is executed, an output of '*G was found 0 times.*' is displayed, which is incorrect. In this example, the debugger used a local connection type to launch the debuggee.

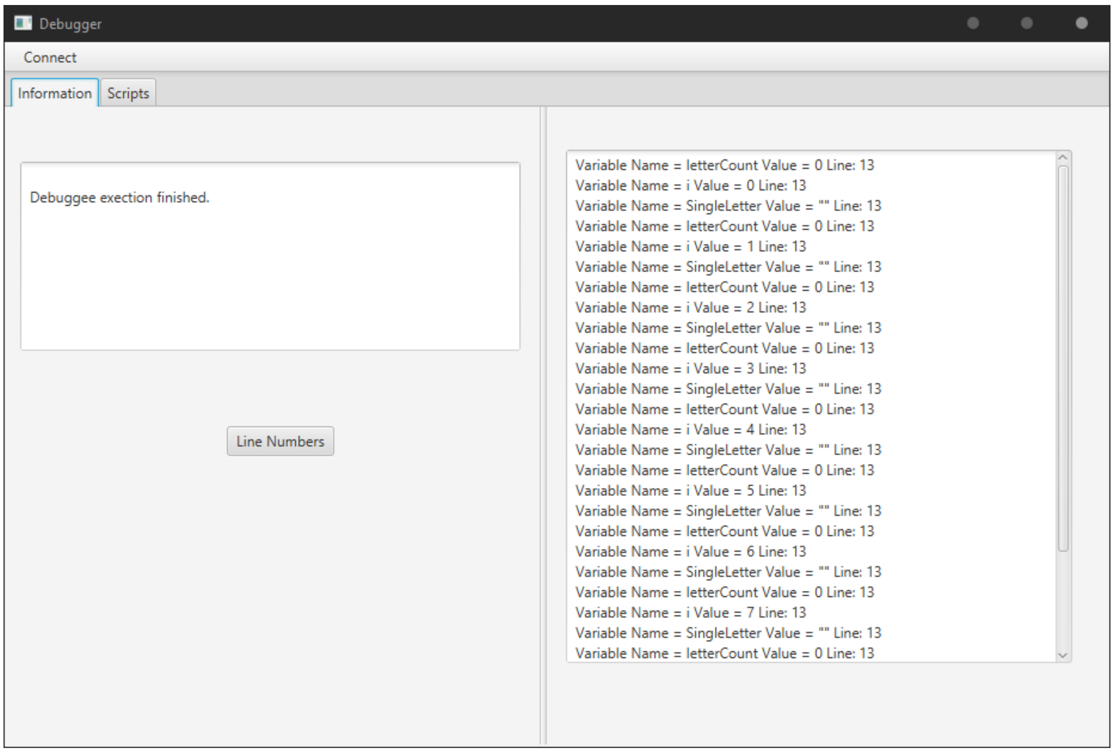
Once the connection was established, the corresponding script file was uploaded. The script that was used can be seen in Figure 5 Debuggee Example Script below which identifies one breakpoint location.

```
<?xml version="1.0" encoding="UTF-8"?>
<script>
        <breakPoint>
                <lineNumber>13</lineNumber>
                <variableName>i</variableName>
    <variableName>SingleLetter</variableName>
    <variableName>letterCount</variableName>
        </breakPoint>
</script>
```

*Figure 5 Debuggee Example Script*

The breakpoint is set on the '*if*' statement that evaluates the current letter against the desired letter. The script also identifies three variables, 'singleLetter', 'letterCount' and 'i'. The variable 'singleLetter' holds the value of the current letter being evaluated, the expected result of which should reflect each letter of the supplied word (Debugging). 'letterCount' holds the number of times the specified letter (g) was identified. 'i' is the for-loop counter holding the number of letters that have been evaluated.

Figure 6 Debuggee Example One Results below shows the output generated by the debugger on execution of the supplied script.



```
Debuggee exection finished.
```

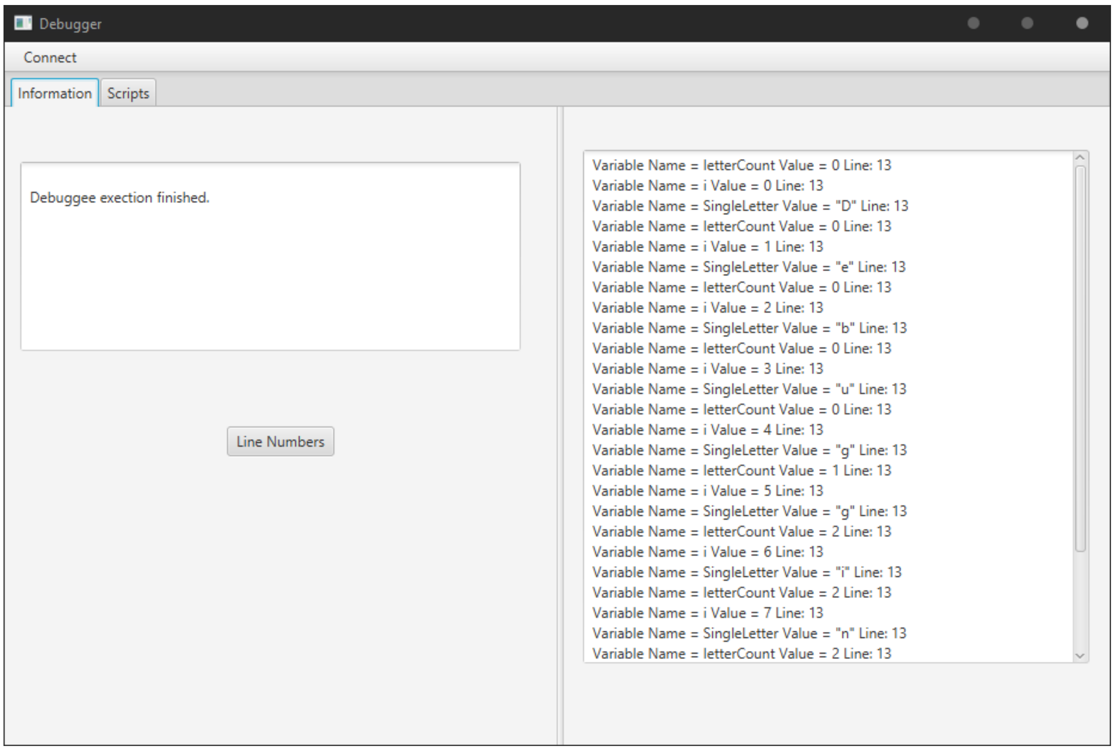
```
Variable Name = letterCount Value = 0 Line: 13
Variable Name = i Value = 0 Line: 13
Variable Name = SingleLetter Value = "" Line: 13
Variable Name = letterCount Value = 0 Line: 13
Variable Name = i Value = 1 Line: 13
Variable Name = SingleLetter Value = "" Line: 13
Variable Name = letterCount Value = 0 Line: 13
Variable Name = i Value = 2 Line: 13
Variable Name = SingleLetter Value = "" Line: 13
Variable Name = letterCount Value = 0 Line: 13
Variable Name = i Value = 3 Line: 13
Variable Name = SingleLetter Value = "" Line: 13
Variable Name = letterCount Value = 0 Line: 13
Variable Name = i Value = 4 Line: 13
Variable Name = SingleLetter Value = "" Line: 13
Variable Name = letterCount Value = 0 Line: 13
Variable Name = i Value = 5 Line: 13
Variable Name = SingleLetter Value = "" Line: 13
Variable Name = letterCount Value = 0 Line: 13
Variable Name = i Value = 6 Line: 13
Variable Name = SingleLetter Value = "" Line: 13
Variable Name = letterCount Value = 0 Line: 13
Variable Name = i Value = 7 Line: 13
Variable Name = SingleLetter Value = "" Line: 13
Variable Name = letterCount Value = 0 Line: 13
```

*Figure 6 Debuggee Example One Results*

The debuggers output shows that the 'singleLetter' and 'letterCounter' variables never change, but 'i' is increasing, as expected. This indicates that the 'singleLetter' assignment code is not being executed as expected. When examining the 'singleLetter' variable assignment statement, this leads to the realisation that the code responsible for extracting the current letter from the supplied word is being assigned the same value every time. The defective code and the solution can be seen below.

- Defective code
    - singleLetter = checkWord.substring(1,1)
- Solution
    - singleLetter = checkWord.substring(i,i+1)

As the debuggee source code has changed, the code must be recompiled; using the same command described above. Once completed, the connection to the debugger can be established as previously stated. The output of the same script file can be seen in Figure 7 Debuggee Example One Solution Results, which shows that the assignment statement is working as intended and the correct count is displayed.



Debuggee exection finished.

```
Variable Name = letterCount Value = 0 Line: 13
Variable Name = i Value = 0 Line: 13
Variable Name = SingleLetter Value = "D" Line: 13
Variable Name = letterCount Value = 0 Line: 13
Variable Name = i Value = 1 Line: 13
Variable Name = SingleLetter Value = "e" Line: 13
Variable Name = letterCount Value = 0 Line: 13
Variable Name = i Value = 2 Line: 13
Variable Name = SingleLetter Value = "b" Line: 13
Variable Name = letterCount Value = 0 Line: 13
Variable Name = i Value = 3 Line: 13
Variable Name = SingleLetter Value = "u" Line: 13
Variable Name = letterCount Value = 0 Line: 13
Variable Name = i Value = 4 Line: 13
Variable Name = SingleLetter Value = "g" Line: 13
Variable Name = letterCount Value = 1 Line: 13
Variable Name = i Value = 5 Line: 13
Variable Name = SingleLetter Value = "g" Line: 13
Variable Name = letterCount Value = 2 Line: 13
Variable Name = i Value = 6 Line: 13
Variable Name = SingleLetter Value = "i" Line: 13
Variable Name = letterCount Value = 2 Line: 13
Variable Name = i Value = 7 Line: 13
Variable Name = SingleLetter Value = "n" Line: 13
Variable Name = letterCount Value = 2 Line: 13
```

*Figure 7 Debuggee Example One Solution Results*

# 5 Prototype Implementation

The finalised implementation of this solution, as discussed in section 4, stems from the design patterns above, in section 3. A rapid iterative prototyping methodology was chosen for this research project. Using this methodology allowed the prototype to be refined and expanded upon for each iteration. The finalised solution was the result of refinement and expansion of multiple iterations of previous prototype versions.

## 5.1 JDI Connectors

The initial iteration constructed the foundations of establishing a connection to a target debuggee. Understanding an unfamiliar API structure can be challenging enough, but even more so with a high level of abstraction like the one in the JDI. Various attempts were made to integrate the connection functionality offered by the JDI into the prototype, firstly by launching a target debuggee. Understanding this process allowed for expansion to contain two additional means of establishing a connection.

Using the provided *Connector* [1] interface from the JDI, a communication channel could be established between the debugger and target debuggee. Connecting JDI implementations follow a similar structure, however, depending on the type used, their arguments for connecting differ. The JPDA documentation constructed in Table 6 JDI Launching Connector Arguments illustrates the arguments of the launching connector [45].

| | Arguments | Description |
|---|---|---|
| **Command Line Launching Connector** | home | The location of the JDK used to invoke the target JVM. |
| | options | Standard java debugging options to apply to the target JVM. |
| | main | Applications main class and any required command line arguments for the class. ***Required*** |
| | suspend | Flag to indicate if the target JVM should be suspended before the main class is loaded. |
| | quote | The character to concatenate the text of the command line operation. ***Required*** |
| | vmexe | The type of executable to invoke the JVM with. ***Required*** |

*Table 6 JDI Launching Connector Arguments*

The launching connector facilitates the largest number of configurable arguments when invoking a target JVM, however, not all arguments are required to establish a connection. The JDI provides default arguments that can be modified for each connector type. The finalised prototype implementation does not facilitate the ability to change every available connector argument. Arguments eligible for modification are seen in Table 7 JDI Launching Connector Configurable Arguments.

| Configurable Connector Argument | Description |
|---|---|
| main | The locations of the compiled java file to be debugged. Using JavaFX as planned facilitated a quick concise implementation of a file picker, allowing users to visualise, locate and select the required file. The debugger used the standard Java ‘File’ class to represent the selected file. From this, a string representation of the file’s location can be stored. The stored string is manipulated to conform to the structure that the JDI expects, containing only the name of the class. |
| options | Using the stored file location, the string is manipulated again to identify the directory of the specified class. Before the argument is set, the string is concatenated with ‘-cp’ immediately before it. This additional string is a standard Java command used by low level Java code to indicate the class path. |
| suspend | Using a simple javaFX toggle button allows the user to specify whether the debuggee should be suspended. The value of the toggle is evaluated to determine the correct string representation of the Boolean value, expected by the JDI. |

*Table 7 JDI Launching Connector Configurable Arguments*

An example of the backend command used to launch a target debuggee, seen below. The example is based on a compiled Java file selected at ‘C:\Users\Desktop\HelloWorld.class’ with the suspend toggle selected.

```
G:\Java\jdk11\bin\java -cp C:\Users\Desktop\HelloWorld.class
-Xdebug-Xrunjdwp:transport=dt_shmem,address=localHost,suspend=y HelloWorld
```

JDI provides six connector implementations to establish a communication channel. Two launching connectors, two socket connectors and two shared memory connectors. The finalised prototype implements three out of these six connection types; one launching connector as described above and two socket connectors. The socket connectors used are the attach and listen connectors described below [45].

| | Arguments | Description |
|---|---|---|
| **Socket Listening Connector** | timeout | The amount of time to wait for a connection to be established |
| | localAddress | The transport address of the host (debugger) ***Required*** |
| | port | The port number of the host to attach to ***Required*** |

*Table 8 JDI Listening Connector Arguments*

Significantly less configurable arguments are available using the socket or shared memory connectors. These arguments are unavailable due to the fact both connector types rely on already established JVMs to attach or be attached to. This dependency removes the ability to specify low level commands, as the JVM has already established these arguments [45].

| | Arguments | Description |
|---|---|---|
| **Socket Attaching Connector** | timeout | The amount of time to wait for a connection to be established |
| | localHost | The transport address of the host (debuggee) to attach to ***Required*** |
| | port | The port number of the host to attach to ***Required*** |

*Table 9 JDI Attaching Connector Arguments*

The valid values accepted by the UI will be applied to the relevant connector arguments to be used in the connection attempt. When a *SocketListen* connector is used the only argument that can be changed is the port number. The 'Timeout' argument is internally set by the debugger to last two minutes, allowing enough time for the connection to be established. Exceeding the allocated time will report a connection failure and the debugger will no longer accept or connect to any targets. The 'LocalHost' argument is set to the address of the local machine that the debugger is running on.

## 5.2 Diagnostic Script

Implementation of the diagnostic script successfully followed the design discussed in section 3.6. The DOM parser configuration was effectively used to represent the node elements of the script in a tree structure. Node lists offered by the DOM parser are used to store a collection of nodes with a given name. The current implementation generates a node list for elements named *breakPoint*.

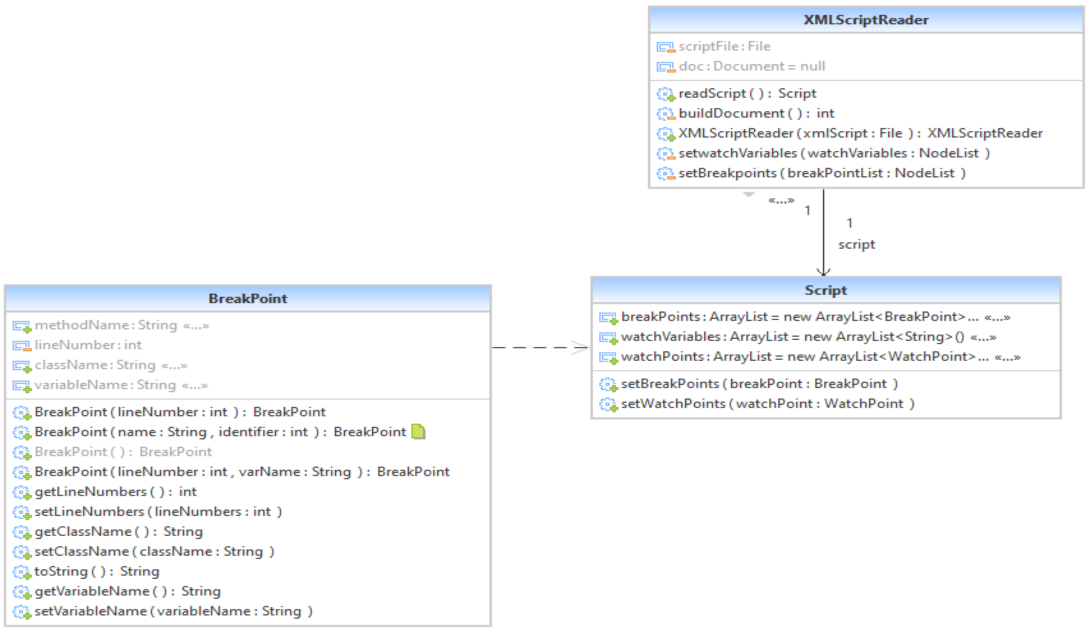
Once the list has been validated, it is iterated over to examine individual breakpoint elements. Attributes relating to breakpoints are identified when evaluating the list elements. A secondary node list is created to contain the child elements of the current breakpoint element. Iterating over this secondary list identifies the attributes relating to all break point requests. Current implementation only supports two attributes: *LineNumber* and *variableName*.

- *lineNumber*
    - Corresponds to a source code location to add a breakpoint
- *variableName*
    - Identifies a variable of interest. This variables value is logged at each breakpoint location

The script is validated by determining if the supplied script contains at least one *breakPoint* element and one *lineNumber* child element. Firstly, by ensuring that the breakpoint node list contains at least one element. An empty list indicates an invalid script. The debugger examines each breakpoint element in turn, searching for the first instance of the line number child element. The script only has to identify a single breakpoint location to be deemed valid. Identifying the first instance of the *lineNumber* element is enough to validate the script. Invalid scripts are not accepted, and the user is prompted to select a different file. Valid scripts have their information extracted and stored locally. Once all of the information has been obtained, it is internally stored within an entity class to represent the script as an object.

A *Script* object was created to represent the extracted script data. The script object contains a list of the identified variable names to be outputted as well as a list of breakpoints.

Each breakpoint within the script is an instance of a *BreakPoint* object. The decision to represent breakpoints as objects was made to facilitate future expansion of the prototype.


XMLScriptReader
scriptFile : File
doc : Document = null
readScript ( ) : Script
buildDocument ( ) : int
XMLScriptReader ( xmlScript : File ) : XMLScriptReader
setwatchVariables ( watchVariables : NodeList )
setBreakpoints ( breakPointList : NodeList )
«...»
1
1
script
Script
breakPoints : ArrayList = new ArrayList<BreakPoint>... «...»
watchVariables : ArrayList = new ArrayList<String>() «...»
watchPoints : ArrayList = new ArrayList<WatchPoint>... «...»
setBreakPoints ( breakPoint : BreakPoint )
setWatchPoints ( watchPoint : WatchPoint )
BreakPoint
methodName : String «...»
lineNumber : int
className : String «...»
variableName : String «...»
BreakPoint ( lineNumber : int ) : BreakPoint
BreakPoint ( name : String , identifier : int ) : BreakPoint
BreakPoint ( ) : BreakPoint
BreakPoint ( lineNumber : int , varName : String ) : BreakPoint
getLineNumbers ( ) : int
setLineNumbers ( lineNumbers : int )
getClassName ( ) : String
setClassName ( className : String )
toString ( ) : String
getVariableName ( ) : String
setVariableName ( variableName : String )
*Figure 8 Debugger Script Structure*

The structure shown in Figure 8 above illustrates that the *XMLScriptReader* internalises the script instructions by building the DOM of the supplied script. The *XMLScriptReader* only represents an individual *Script* object. The *Script* object is dependent on the *BreakPoint* class to represent individual breakpoints. The *Script* object holds a list of breakpoint objects, which are instantiated with a line number identified in the script. The *BreakPoint* class has extra attributes that the current implementation does not utilise. These attributes have been added in order to accommodate later expansion of implemented script constraints. For example, when the debugger starts making breakpoint requests to the debuggee, it uses the list of breakpoints contained within the *Script* object. As the breakpoints themselves are objects, their attributes could be used to add more specific constraints to individual breakpoints.

Both of the examples seen in Figure 9 Implementable Script Examplesbelow will result in the same output from the debugger. This is due to the fact that both scripts identify the same breakpoint locations, as well as the desired variables to be outputted.

```
<?xml version="1.0" encoding="UTF-8"?>
<script>
 <breakpoint>
   <lineNumber>29</lineNumber>
   <lineNumber>35</lineNumber>
  </breakpoint>
 <breakpoint>
<variableName>variable1</variableName>
<variableName>variable2</variableName>
</breakpoint>
</script>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<script>
  <breakpoint>
    <lineNumber>29</lineNumber>
    <lineNumber>35</lineNumber>
   <variableName>variable1</variableName>
   <variableName>variable2</variableName>
  </breakpoint>
</script>
```

*Figure 9 Implementable Script Examples*

The corresponding *Script* object generated from parsing these scripts would follow the structure illustrated in Figure 10.

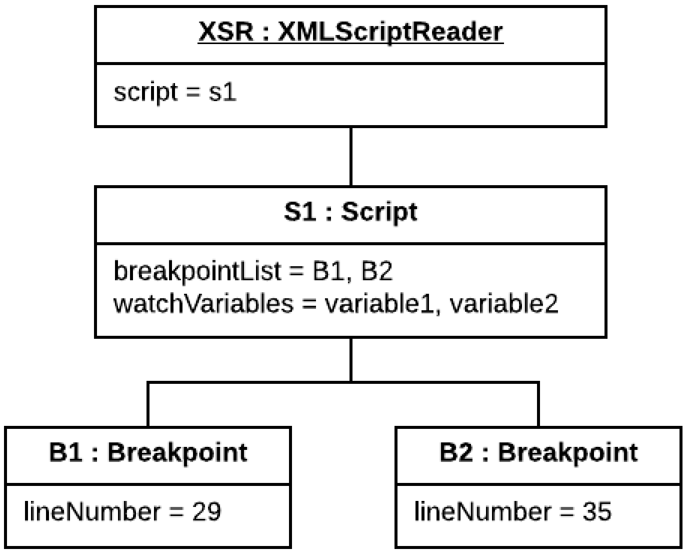


*Figure 10 Example of Internalised Script*

The Script object created by the *XMLScriptReaded* contains two collections, one for breakpoint objects and one for string representations of variable names. The breakpoint collection contains two individual breakpoint objects that store individual locations.

## 5.3 Event Dispatching

Once a reliable connection could be established consistently and script information could be internalised, the next iteration aimed to retrieve incoming events from the debuggee. Incoming events are evaluated and dispatched to the relevant section of the debugger, as specific events require particular actions. Incoming events are stored within an *EventSet* contained on the *EventQueue* as discussed earlier in section 4.4. *EventSets* are removed individually from the *EventQueue* using supplied JDI methods. A JDI *EventIterator* provides the ability to remove individual events from the current *EventSet*. Depending on the current event, the debugger will perform the required action.

Events of interest, discussed as they appear to the debugger, seen below in Table 10 JDI :

| Event Name | Event Description | Event Handler |
|---|---|---|
| *VMStartEvent* [46] | Receival of this event indicates that the debuggee has successfully executed fundamental low-level system code. This event occurs before the main thread executes, meaning no application code has been performed yet. This event will always occur, even if not explicitly requested. | *createClassRequest* |
| *ClassPrepareEvent* [47] | Identifying this event informs the debugger that the specified class has been loaded but not yet executed. | *setBreakpoints* |
| *BreakpointEvents* [48] | This event appears to the debugger when the debuggee has reached a line of | *checkVariables* |

| | | |
|---|---|---|
| | execution that is associated with breakpoint request. | |
| *VMDeathEvent* [49] | This event is the last event to be generated by the debuggee after the completion of its executable code. This event is used to indicate to the debugger that the connection will close as the debuggee has completed execution and is shutting down. Control is still managed by the debugger, until the debugger resumes the remaining shutdown code of the debuggee. | The dispatcher indicates to the rest of the system that the connection has been closed. |
| *VMDisconnectEvent* [50] | Receiving this event indicates to the debugger that the debuggee has disconnected unexpectedly and its remaining code has not been executed. This event can be caused by numerous processes, all of which have the same consequence; communication is lost, and the debugger now has no control. | The dispatcher indicates to the rest of the system that the connection has been closed. |

*Table 10 JDI Events*

The event dispatcher continually tries to remove events while the connection to the debuggee is still established. A *VMDeathEvent* or a *VMDisconnectEvent* will cause the connection to be closed, stopping the debugger from iterating over the debuggee's *EventQueue*.

## 5.4 Event Handling

Dispatched events are handled by the corresponding event handlers. Some event handlers produce requests that influence the next incoming event, meaning that the request must be made before this future event occurs. The majority of the handler methods require that the event dispatcher supplies the current *Event* object, this ensures all of the relevant information about the debuggee at its current stage of execution is available. Figure 11 below illustrates the flow of execution when handling events.

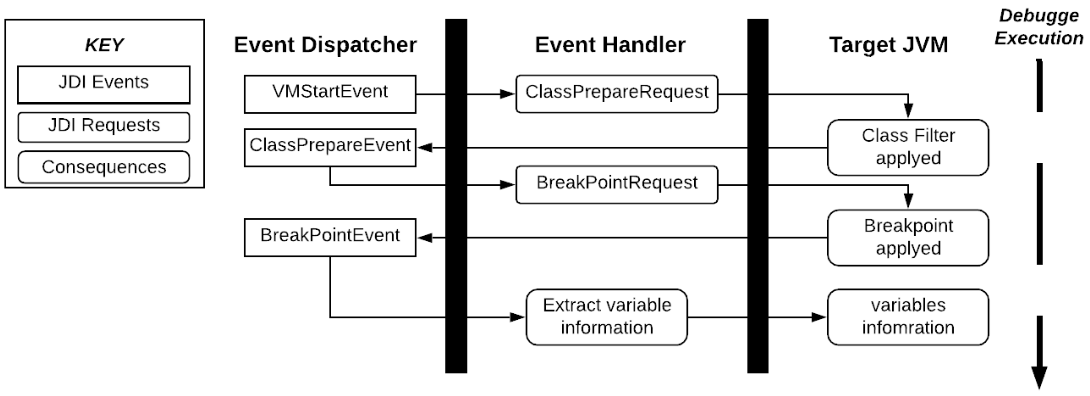


*Figure 11 JDI Event and Request Flow of Execution*

*VMStartEvent*(s) indicates that enough low-level system code has been executed and that the debuggee is now available to receive requests [46]. The dispatcher utilises the *createClassRequest* handler, that uses the JDI *ClassPrepareRequest* [51] to be notified when the debuggee loads an identified class. This is the only event handler that does not need the current dispatched event. To make a *ClassPrepareRequest*, the name of the target class is required. The class name used is that of the selected class file employed to establish the connection or the name of the class supplied when attempting a remote connection.

Once the debuggee loads the target class, it will generate the corresponding *ClassPrepareEvent*, which will be caught by the debugger via the *EventQueue*. To handle this type of event, the dispatcher calls the *setBreakpoints* method. Using the generated event, information can be extracted to apply *BreakPointRequest*'s [52] on the loaded class. The event supplied from the dispatcher is used to locate the events origin within the debuggee. From here, the line locations of the debuggee class are compared against the location of the individual breakpoints. Matching locations will result in a breakpoint request being set at this position of executable code.

Debuggee execution is then resumed until a *BreakPointEvent* occurs. *BreakPointEvent*'s are dealt with by the *checkVariables* handler. The first thing that the handler does is identify if the script contains any variable names. When a script does contain a specified variable, the handler attempts to identify the variables in the current class, by first searching the target classes static variables. This search is only done once when handling the first *BreakPointEvent*. As the static variables are fix locations in memory, the fields of the class can simply be requested via the *Event* object. The response list of this request is iterated over, to compare the names of static variables with the identified variable names contained within the script. Matching names would result in that field being added to the *watchFields* list. Then any static variables contained in the list would be included in the log.

Identifying local variables was slightly more complicated due to the inherent interweaving of interfaces within the JDI. Understanding how to extract the required information proved to be more complex than anticipated. To access local variables, multiple pieces of information were required. A JDI *ThreadReference* [53] was necessary to obtain an encapsulated instance of the thread responsible for the *BreakPointEvent* within the debuggee. Using this thread reference, a JDI *StackFrame* [54] was used to encapsulate the current stack frame related to the thread reference. From here, the JDI can be requested to supply a list of the visible instance variables of this stack frame. A similar process used to identify the static variables was used to find the matching local variable names. By requesting a list of the variables visible to that frame, it could be evaluated against the script variables. A map was created to store the identified variables and their values. Once again values contained within the map were outputted.

There is a possibility that no variable names are supplied within the script, this would be identified when handling the first breakpoint. Three Java Enums [55] are utilised to indicate what information is to be logged. Table 11 Script Enums below describes what they identify to the debugger.

| **Enum Name** | **Enum Description** |
|---|---|
| *ALL* | Identifies that no variable name(s) were specified in the supplied script. All variables should be logged. |
| *VARIABLES* | Identifies that the script contains at least one variable name, but the name does not correspond to a static variable. A specified local variable is to be logged. |
| *BOTH* | Identifies that the script contains at least two variable names. One relating to a static variable & one relating to a local variable. A specified static & local variable is to be logged. |

*Table 11 Script Enums*

## 5.5 Overall Implementation Structure

The structure of the prototype has evolved with each iteration growing slightly more complex, while still staying modular enough that review or expansion is easily achievable. As the chosen implementation is modular in design, the issue of accessing required information from independent components of the prototype became prevalent. Evaluating the potential solutions lead to the realisation that a simple but highly effective entity class could be used to hold current target JVM information. The *ConnectionBundle* class seen in Appendix 6 – Final is the entity class responsible for holding data relating to the supplied connection information. The connection information is collected via the UI and is stored in an instance of a *ConnectionBundle* object. This object is passed to multiple other sections of the debugger, as the connection information is required to perform certain actions. As this simple approach demonstrated to be extremely effective, it was chosen to represent the information relating to the target JVM (*Debuggee*) seen in Appendix 6 – Final . Once again, this object was passed around to sections that required this information. The decision was made the debugger to internally access the log functionality. In order for the debugger to access this functionality, it must utilise the debuggee reference, as the debuggee object will always be instantiated once a connection is attempted.

Appendix 6 – Final shows the overall structure of the finalised prototype. The depicted structure has evolved over multiple iterations and can be broken down into the following components in Table 12 Prototype Components.

| **Component Name** | **Corresponding Classes** |
|---|---|
| Connection | • ConnectionManager<br>• Connection<br>○ ConnectionListen<br>○ ConnectionAttach<br>○ ConnectionLaunch |
| UI | • Main<br>• ConnectionController<br>• MainController |
| Script | • XMLScriptReader<br>• Script<br>• Breakpoint |
| Events | • EventDispatcher<br>• EventHandler |
| Function / Entity | • ConnectionBundle<br>• ConnectionEnums<br>• DebuggingStageEnums<br>• Log<br>• UserNotification<br>• InformationBuilder |

*Table 12 Prototype Components*

# 6 Prototype Testing

Constant internal testing was prevalent throughout the development of the prototype due to the chosen implementation tactic. For each iteration of the prototype, it underwent white box testing to identify and rectify any errors. A major hindrance in the prototype's development was the lack of knowledge of the JDI's structure. The result of which caused the development of fundamental aspects of the debugger to be slow. Countless techniques were utilised to understand the structure of frequently used JDI interfaces, as well as the kind of information required by the JDI to perform a desired action. Once the basic understanding was grasped, development began to expand rapidly.

The three available connector options visible to the user have been thoroughly tested throughout development. This functionality was introduced within the first iteration as it is the foundation of the debugger. A connection can be constantly established by all three connectors provided that the supplied connection information is present and correct. Exceptions thrown by failed connection attempts are logged within the debugger log file for evaluation.

When testing the core debugging functionality, various areas became visibly problematic when attempting to extract debuggee information. The problem became prevalent when using a debuggee that contained multiple classes. Evidently, the problem lay within the structure of the debuggee, as well as how low-level Java code internalises classes and objects. To identify local or instance variables, the stack frame of the thread responsible for the event was required. The debugger failed to retrieve information contained within an additional class as it does not evaluate every frame on the stack. This conclusion was reached due to the fact that two separate debuggee examples containing the same code but arranged in a different structure, yielded different results. In light of this problem, the decision was made to use the previous version of the debuggee as the project simply aims to illustrate the potential of scripted debuggers.

## 6.1 User Feedback

Once implementation was complete, a small group of Java developers were asked to participate in a simple worksheet and provide feedback. This worksheet requested these individuals to establish a connection to a debuggee, upload a script and execute the script. The worksheet also requested that these participants repeat this for all three connection types. They were then asked to modify the script and execute it on a connection type of their choice. They were asked to provide feedback on each individual debugging session, as well as their opinions on the practicality of scripted debuggers overall.

Each participant was supplied with seven files in order to perform the testing. Firstly, the worksheet seen in Appendix 8 – Worksheet. Secondly, a Java file *HelloWorld.java* (Appendix 7 – User Test (Debuggee)) to act as the debuggee. In addition, an executable version of the debugger (Debugger.jar) was also included, as well as a corresponding script file. Three '.bat' files, including 'compile.bat' to compile the debuggee with the required debugging features enabled, 'attach.bat' to launch the debuggee in a waiting state for the debugger to connect, and finally, 'listen.bat', which attempts to connect to the waiting debugger. Participants were told to attempt the worksheet without any external guidance to evaluate the debuggers usability. However, the test was primarily intended to evaluate the opinions that other Java developers may have on the potential effectiveness of scripted debuggers.

### 6.1.1 **User Feedback results**

All participants were able to complete the required tasks in a reasonable time. Most of the participants needed additional explanation at the beginning of the worksheet in order to clarify the required connection between the debugger and debuggee. Once doing so, the participants did not require any additional interaction or guidance. Each participant specified that the expected results for all three connection types were the same as there actual results, indicating that the debugger was working as intended. However, one participant did have problems executing the debugger jar file. This was due to the Java environment on their local machine, as the Java compiler version was different to the Java version. This participant removed their current Java environment and installed the Java 8 JDK, doing so resolved this issue.

Some participants found the structure of the script confusing, as identified variables were applied to every breakpoint location specified in the script. Participants gave suggestions on features that could increase the functionality of the script. The main suggestion was to allow users to specify variables relating to a specific breakpoint. For example, a program contains two integer values, 'minValue' and 'maxValue', and a breakpoint is set at line '4' and line '8'. If the user wanted to know the 'minValue' at line 4 and the 'maxValue' at line 8, this is not possible with the current implementation, as the debugger would return both of the variable values at both breakpoint locations.

Participants did find the script structure to be flexible and easy to expand upon during the ongoing debugging session. As the debuggee can reconnect to the debugger multiple times throughout the session, numerous variations of the script can be executed, which the participants responded well to.

Overall, the participants did agree that scripted debuggers do have great potential as a useful debugging technique. The majority of participants did say that they would use a scripted debugger tool if it was as robust as the widely available interactive debuggers.

# 7 Conclusion

## 7.1 Summary

This research project aimed to reduce the amount of time developers spend debugging their applications, by facilitating the ability for custom-built diagnostic scripts to automate the debugging process, locally or remotely. Upon conclusion of the project, the finalised prototype offers the ability to:

- Connect to a target debuggee via three means:
  - Launch
    - The connection specifics are gathered through the UI and the target debugger's compiled file is then selected by a file picker
  - Socket Attach
    - The connection specifics are gathered through the UI and used to attempt the remote connection
  - Socket Listen
    - The connection specifics are gathered through the UI and used to attempt the remote connection
- View source code line numbers to assist script development
- Upload XML Script files via a file picker
- Execute script commands
- Display relevant variable information
- Log relevant debugger and debuggee events separately

Script functionality in the current state is limited to only breakpoint commands due to the time constraints of the project. However, the breakpoint components use two child elements to specify breakpoint locations and the variables of interest when handling *BreakPointEvents*. Each child element specification was applied to every breakpoint identified via the *lineNumber* child element. For more implementation information, see section 5.2.

Control over execution is managed by the debugger but facilitated by the JDI, as it provides simple 'suspend' and 'resume' methods to start and stop the main thread of the debuggee. This functionality is fundamental in reducing the overall time of debugging sessions, as the debugger does not require additional interaction from the user to identify when to resume execution.

Fundamentally, the prototype can be broken down into independent sections that collectively encapsulate the debugger. The first section is communication, which lays the foundation for any debugging session as both parties must be able to communicate in order for information to be gathered. Secondly are script details, as the debugger must be able to internally comprehend the actions specified by the user's script. Lastly is execution control, which is the capability to apply scripted command to the correct time of execution and ensure that consistent and reliable reads are taken from the debuggee.

Various aspects of the debugger's initial design were removed due to the strict time constraint. Watchpoint functionality never came to fruition due to the unforeseen complication when interacting with the JDI interface. Watchpoints were excluded to allow for more time to be invested into implementing breakpoints, as breakpoints are a more commonly used debugging feature. As the design was constructed with the intention to include watchpoints, aspects to support integration of this feature are already present within the debugger in later development.

## 7.2 Evaluation

Overall, this research project highlights the clear benefits offered by scripted debuggers. Evidently, a scripted debugger can reduce the total debugging time by internalising the supplied command information via the script. The script information was utilised to specify the locations at which to control the debuggee's execution, as well as the variables to output. By supplying the debugging parameters before beginning execution, this eradicates the need for additional interaction, as the debugger is already aware of where to look and what to look for.

Scripted debuggers would be an ideal tool for identifying logical errors, as variable results at multiple locations can be collected, stored and evaluated against the expected results. Scripted debuggers can execute this without the need for modification to the source code or manually stepping through executable code. As well as eradicating the tedious task of searching for the desired information.

Conventional interactive debuggers still have their place within the industry and always will. The amount of information and control that they provide over a debuggee is incredibly extensive, allowing for absolute control over every aspect of the debuggee, which a scripted debugger could not achieve. However, as stated in section **Error! Reference source not found.**, the aim of this project was to create a hybrid of debugging techniques. This prototype has accomplished this aim, by employing the JDI to facilitate the core underlying implementation of communication and interaction between both parties. The script identifies the variables to output, the equivalent of simple diagnostic statements added to source code.

The end product is a comfortable balance between simple, clear diagnostic statements and valuable features offered by interactive debuggers, offering a powerful and fast debugger in exchange for the level of control and internal detail of the debuggee.

## 7.3 Future Work

Fundamentally, the prototype serves as a good example of the advantages posed by scripted debuggers. However, to become a reliable debugging tool, many aspects need refinement and expansion. Any future work should primarily be focused on exploring the extensive number of *EventRequest* options provided by the JDI, as well as implementing extensive validation checking, due to the high level of abstraction in the JDI, as not all JVM support ever debugging feature [22]. Table 13 Future Work shows some examples of potential requests that could be implemented.

| Potential Future Requests | Purpose |
|---|---|
| *ModificationWatchpointRequest* | Specified fields have data collected about them when they are modified. Class and thread filters can be applied to restrict the events generated by the request [56]. |
| *AccessWatchpointRequest* | Specified fields have data collected about them when they are accessed. Modification is not classified as access. Class and thread filters can be applied to restrict the events generated by the request [57]. |
| *MethodEntryRequest* | When identified methods are invoked, the debugger is notified and examines the state of the incoming arguments (if any). Class and thread filters can be applied to restrict the events generated by the request [56]. |
| *MethodExitRequest* | When identified methods have finished execution but not yet returned, the debugger is notified and |

| | |
|---|---|
| | examines the state of the returning arguments (if any). Class and thread filters can be applied to restrict the events generated by the request [58]. |
| *ExceptionRequest* | Any exception that occurs within the debuggee have their type and location logged. Class and thread filters can be applied to restrict the events generated by the request [59]. |

*Table 13 Future Work*

Development of the script could be expanded upon simultaneously with that of acceptable requests, increasing the overall development time.

Facilitating the ability for the user to set variables would provide another layer of flexibility not offered by conventional interactive debuggers. Once again, the underlying, core implementation is provided by the JDI [22] and could easily be added to the debugger alongside the addition of corresponding script tags.

The issues mentioned by the participants in section 6.1.1 could possibly be addressed by adding an XML attribute on to the tag element, allowing the user to specify the exact class that this breakpoint is intended for. An example of this can be seen in Figure 12 Future Script Work below.

```
<script>
        <breakPoint className="HelloWorld">
                <lineNumber>17</lineNumber>
                <lineNumber>42</lineNumber>
                <variableName>helloObject2</variableName>
                <variableName>x</variableName>
        </breakPoint>
</script>
```

*Figure 12 Future Script Work*

It was intended that this feature be included in the final prototype, as it would have been extremely useful for debugging across multiple classes. The failure to implement this valuable feature was due to the strict time constraints placed on the research project.

An additional component which could lead to valuable functionality for the user is the inclusion of a counter to track the number of times a variable/method is accessed/modified. Countless filtering possibilities could be utilised with great affect when handling large scale applications with hundreds of requests and operations. As an example, if a logical error was occurring somewhere with a large loop, the debugger could evaluate the suspect variables at particular cycles of the loop, providing a much faster solution to current interactive debuggers. The JDI already provides a potential solution, or at least a valid mechanism in which to accomplish this goal via the 'addCountFilter()' which is provided by the *EventRequest*; the super-interface of all *EventRequest* [38].

All in all, the prototype does need additional work in order to become a valid alternative to interactive debuggers, which have been discussed in this section. However, it is not uncertain that this research project in its entirety demonstrates the potential which lies within scripted debuggers.

## References

[1] "Connector," Oracle, [Online]. Available: https://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/com/sun/jdi/connect/Connector.html. [Accessed 26 January 2020].

[2] M. Telles and Y. Hsieh, in *Science of Debugging*, Paraglyph Press, 2001.

[3] A. Afzal and C. Le Goues, "A Study on the Use of IDE Features for Debugging," in *International Conference on Mining Software Repositories*, Gothenburg, Sweden, 2018.

[4] X. Xie, Z. Liu, S. Song, Z. Chen, J. Xuan and B. Xu, "Revisit of automatic debugging via human focus-tracking analysis," *16: Proceedings of the 38th International Conference on Software Engineering,* pp. 808-819, May 2016.

[5] S. Rahman and D. Kourkouzelis, "Method and apparatus for automatic debugging technique". United States of America Patent US8191074B2, 2007.

[6] Yash, "History of software Bugs and Debugger," 12 January 2011. [Online]. Available: http://www.ksyash.com/2011/01/178/. [Accessed 9 September 2019].

[7] J. Hennessy, "Symbolic Debugging of Optimized Code," *Transactions on Programming Languages and Systems,* vol. 4, no. 3, pp. 323-344, 1982.

[8] M. Beller, N. Spruit and A. Zaidman, "How developers debug," PeerJ, Delft, The Netherlands, 2017.

[9] S. Lauesen, "Debugging techniques," *Software: Practice and Experience,* vol. 9, no. 1, p. 1, 1979.

[10] pp_pankaj, "Differences between Testing and Debugging," GeeksforGeeks, 8 May 2019. [Online]. Available: https://www.geeksforgeeks.org/differences-between-testing-and-debugging/. [Accessed 11 November 2019].

[11] J. C. Weber, "Interactive debugging of concurrent programs," *Proceedings of the ACM SIGSOFT/SIGPLAN software engineering symposium on high-level debugging,* vol. 18, no. 8, pp. 107-111, 1983.

[12] "Use breakpoints in the Visual Studio debugger," Microsoft, 28 October 2019. [Online]. Available: https://docs.microsoft.com/en-us/visualstudio/debugger/using-breakpoints?view=vs-2019. [Accessed November 2019].

[13] Tutorialspoint, "Tutorialspoint," 15 September 2019. [Online]. Available: www.tutorialspoint.com/jdb/jdb_breakpoints.html.. [Accessed 15 September 2019].

[14] "Setting Method Breakpoints," IBM Eclipse, [Online]. Available: https://help.eclipse.org/2019-12/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fbreakpoints%2Fref-entry_option.htm&cp%3D1_4_0_6. [Accessed November 2019].

[15] J. Jackson, M. Cobb and C. Carver, "Identifying Top Java Errors for Novice Programmers," in *Proceedings Frontiers in Education 35th Annual Conference*, Indianopolis, IN, USA, 2005.

[16] Y. Srikant and P. Shankar, The Compiler Design Handbook: Optimizations and Machine Code Generation., CRC Press, 2002.

[17] "Debugging with GDB: Set Watchpoints," sourceware, [Online]. Available: https://sourceware.org/gdb/current/onlinedocs/gdb/Set-Watchpoints.html. [Accessed 23 November 2019].

[18] Oracle, "Java Platform Debugger Architecture (JPDA)," Java Documentation, 2016. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/index.html. [Accessed 30 November 2019].

[19] R. Caballero, C. Hermanns and H. Kuchen, "Algorithmic Debugging of Java Programs," *Electronic Notes in Theoretical Computer Science,* vol. 177, no. 1, pp. 75-89, 2007.

[20] D. . J. Murray and . D. . E. Parson, "Automated Debugging In Java Using OCL And JDI," Cornell University, New York, 2001.

[21] Y. Gueheneuc , R. Douence and N. Jussien, “No Java without caffeine: A tool for dynamic analysis of Java programs,” IEEE, Edinburgh, 2002.

[22] “Java Debug Interface,” Oracle, 1 January 2001. [Online]. Available: https://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/. [Accessed 30 November 2019].

[23] “jdb - The Java Debugger,” Oracle, 1 January 2001. [Online]. Available: https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html. [Accessed 30 November 2019].

[24] C. Aniszczyk and P. Leszek, “Debugging with the Eclipse Platform,” IBM, 1 May 2007. [Online]. Available: https://www.ibm.com/developerworks/library/os-ecbug/. [Accessed 2 December 2019].

[25] “Debug View,” Eclipse, 30 November 2019. [Online]. Available: https://help.eclipse.org/2020-03/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fviews%2Fdebug%2Fref-debug_view.htm.

[26] m3schools.com, “https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/index.html,” m3schools.com, [Online]. Available: https://www.w3schools.com/xml/xml_whatis.asp. [Accessed 10 September 2019].

[27] T. Bray, J. Paoli and C. M. Sperberg-McQueen , “Extensible Markup Language,” W3C, Chicago, 1998.

[28] Apache Commons, “Jelly : Executable XML,” Apache Commons, 25 September 2017. [Online]. Available: http://commons.apache.org/proper/commons-jelly/. [Accessed 10 September 2019].

[29] Oracle, “Running and Debugging Java Programs,” Oracle, 2015. [Online]. Available: https://docs.oracle.com/middleware/1212/jdev/OJDUG/run_debug_java.htm#OJDUG2059. [Accessed October 2019].

[30] D. Aderemi, “How to Remotely Debug Application Running on Tomcat From Within Intellij IDEA,” Trifork, 14 July 2014. [Online]. Available: https://blog.trifork.com/2014/07/14/how-to-remotely-debug-application-running-on-tomcat-from-within-intellij-idea/. [Accessed 15 January 2020].

[31] “Scene Builder,” Gluon, 20 March 2018. [Online]. Available: https://gluonhq.com/products/scene-builder/. [Accessed 20 January 2020].

[32] J. Vos, S. Chin, W. Gao, J. Weaver and D. Iverson, Pro JavaFX 2, Apress, 2018.

[33] “Alert,” Oracle, [Online]. Available: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/Alert.html. [Accessed 20 January 2020].

[34] “File Chooser,” Oracle, [Online]. Available: https://docs.oracle.com/javafx/2/ui_controls/file-chooser.htm. [Accessed 20 January 2020].

[35] “Mirror,” Orical , [Online]. Available: https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/Mirror.html. [Accessed 23 January 2020].

[36] “VirtualMachine (Java Debug Interface ),” Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/VirtualMachine.html. [Accessed 23 January 2020].

[37] “EventRequestManager,” Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/request/EventRequestManager.html. [Accessed 23 January 2020].

[38] “EventRequest,” Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/request/EventRequest.html. [Accessed 23 January 2020].

[39] “EventSet,” Oracle, [Online]. Available: https://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/com/sun/jdi/event/EventSet.html. [Accessed 23 January 2020].

[40] "EventQueue," Oracle, [Online]. Available: https://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/com/sun/jdi/event/EventQueue.html. [Accessed 23 January 2020].

[41] "Reading XML Data into a DOM," Oracle, [Online]. Available: https://docs.oracle.com/javase/tutorial/jaxp/dom/readingXML.html. [Accessed 26 January 2020].

[42] "LaunchingConnector," Oracle, [Online]. Available: https://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/com/sun/jdi/connect/LaunchingConnector.html. [Accessed 16 February 2020].

[43] "ListeningConnector," Oracle, [Online]. Available: https://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/com/sun/jdi/connect/ListeningConnector.html. [Accessed 16 February 2020].

[44] "AttachingConnector," Oracle, [Online]. Available: https://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/com/sun/jdi/connect/AttachingConnector.html. [Accessed 16 February 2020].

[45] "Connection and Invocation Details," Oracle, [Online]. Available: https://www.doc.ic.ac.uk/csg-old/java/jdk6docs/technotes/guides/jpda/conninv.html. [Accessed 2 February 2020].

[46] "VMStartEvent," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/event/VMStartEvent.html. [Accessed 5 February 2020].

[47] "ClassPrepareEvent," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/event/ClassPrepareEvent.html. [Accessed 5 February 2020].

[48] "BreakpointEvent," Oracle, [Online]. Available: https://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/com/sun/jdi/event/BreakpointEvent.html. [Accessed 5 February 2020].

[49] "VMDeathEvent," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/event/VMDeathEvent.html. [Accessed 5 February 2020].

[50] "VMDisconnectEvent," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/event/VMDisconnectEvent.html. [Accessed 5 February 2020].

[51] "ClassPrepareRequest," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/request/ClassPrepareRequest.html. [Accessed 5 February 2020].

[52] "BreakpointRequest," Oracle, [Online]. Available: https://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/com/sun/jdi/request/BreakpointRequest.html. [Accessed 5 February 2020].

[53] "ThreadReference," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/ThreadReference.html. [Accessed 5 February 2020].

[54] "StackFrame," Oracle, [Online]. Available: https://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/com/sun/jdi/StackFrame.html. [Accessed 10 February 2020].

[55] "Enum Types," Oracle, [Online]. Available: https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html. [Accessed 15 February 2020].

[56] "ModificationWatchpointRequest," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/request/ModificationWatchpointRequest.html. [Accessed 22 March 2020].

[57] "AccessWatchpointRequest," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/request/AccessWatchpointRequest.html. [Accessed 22 March 2020].

[58] "MethodExitRequest," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/request/MethodExitRequest.html. [Accessed 22 March 2020].

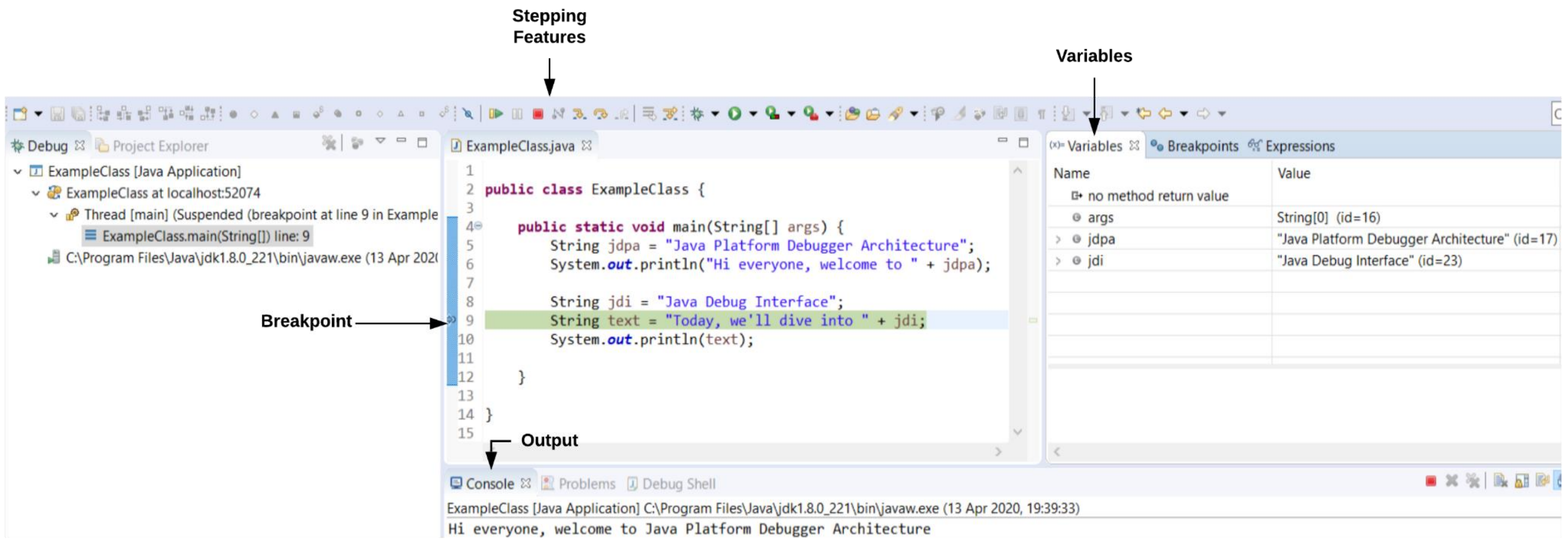
[59] "ExceptionRequest," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/request/ExceptionRequest.html. [Accessed 22 March 2020].

[60] IBM, "Debug Java applications remotely with Eclipse," IBM, 09 December 2008. [Online]. Available: https://www.ibm.com/developerworks/library/os-eclipse-javadebug/index.html. [Accessed October 2019].

[61] "Running and Debugging Java Projects," Oracle, [Online]. Available: https://docs.oracle.com/middleware/12213/jdev/user-guide/running-and-debugging-java-projects.htm#OJDUG4366. [Accessed 10 January 2020].

[62] "MethodEntryRequest," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/request/MethodEntryRequest.html. [Accessed 22 March 2020].

# Appendix 1 – Eclipse Debugger View



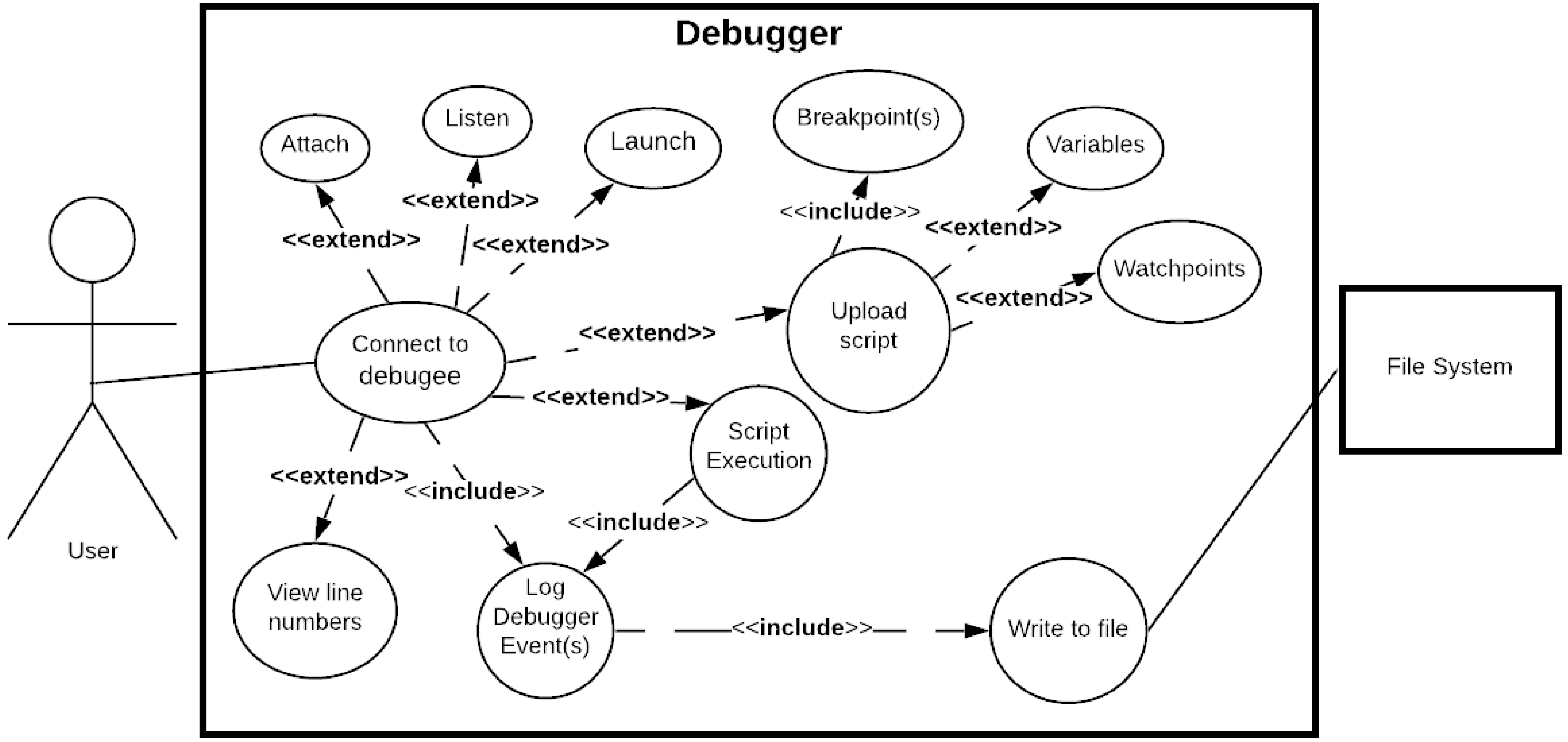
```
ExampleClass [Java Application]
  ExampleClass at localhost:52074
    Thread [main] (Suspended (breakpoint at line 9 in Example
      ExampleClass.main(String[]) line: 9
  C:\Program Files\Java\jdk1.8.0_221\bin\javaw.exe (13 Apr 202
```

```java

public class ExampleClass {

    public static void main(String[] args) {
        String jdpa = "Java Platform Debugger Architecture";
        System.out.println("Hi everyone, welcome to " + jdpa);

        String jdi = "Java Debug Interface";
        String text = "Today, we'll dive into " + jdi;
        System.out.println(text);

    }

}

```

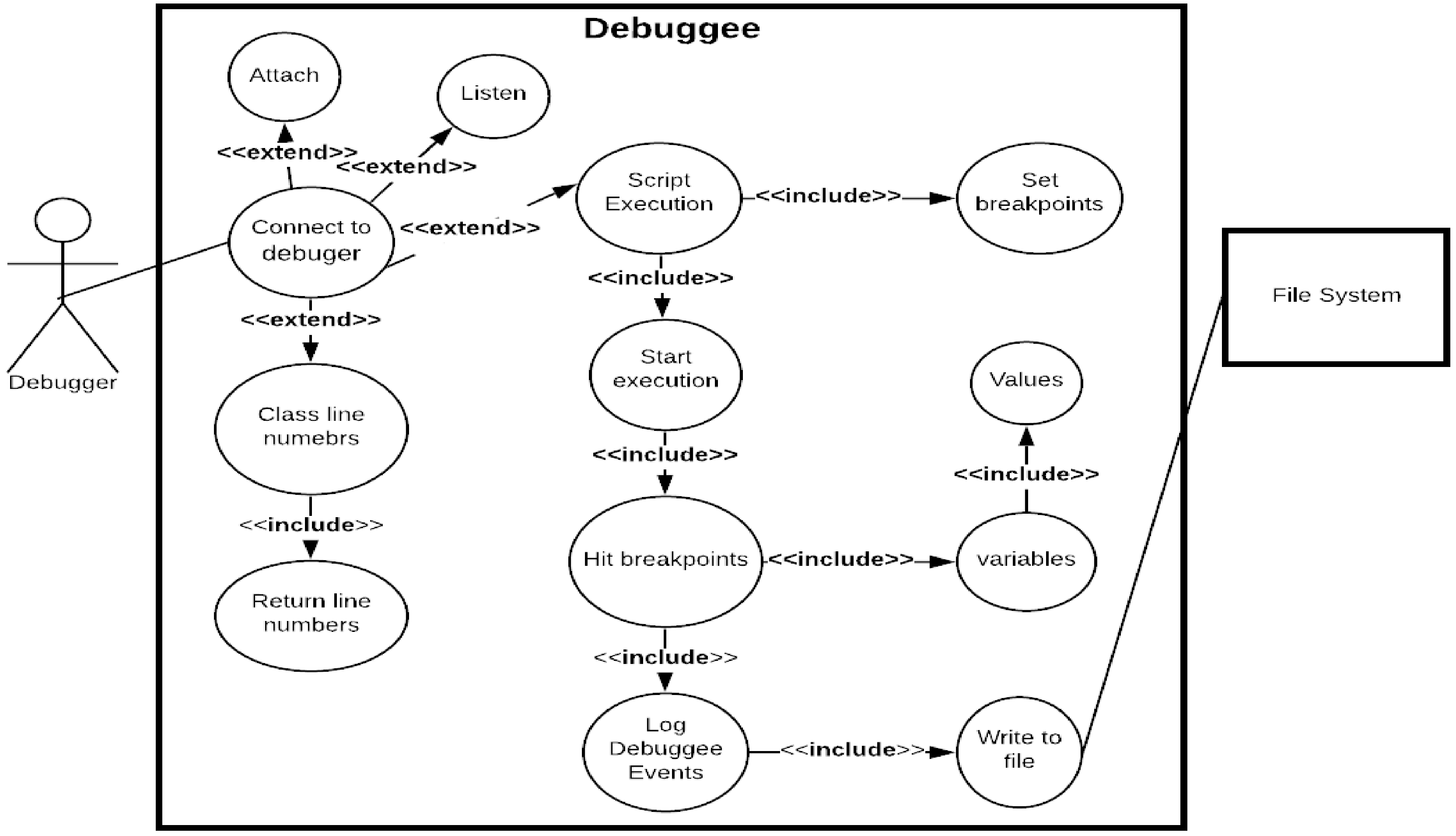
| Name | Value |
|---|---|
| no method return value | |
| args | String[0] (id=16) |
| jdpa | "Java Platform Debugger Architecture" (id=17) |
| jdi | "Java Debug Interface" (id=23) |

```
ExampleClass [Java Application] C:\Program Files\Java\jdk1.8.0_221\bin\javaw.exe (13 Apr 2020, 19:39:33)
Hi everyone, welcome to Java Platform Debugger Architecture
```

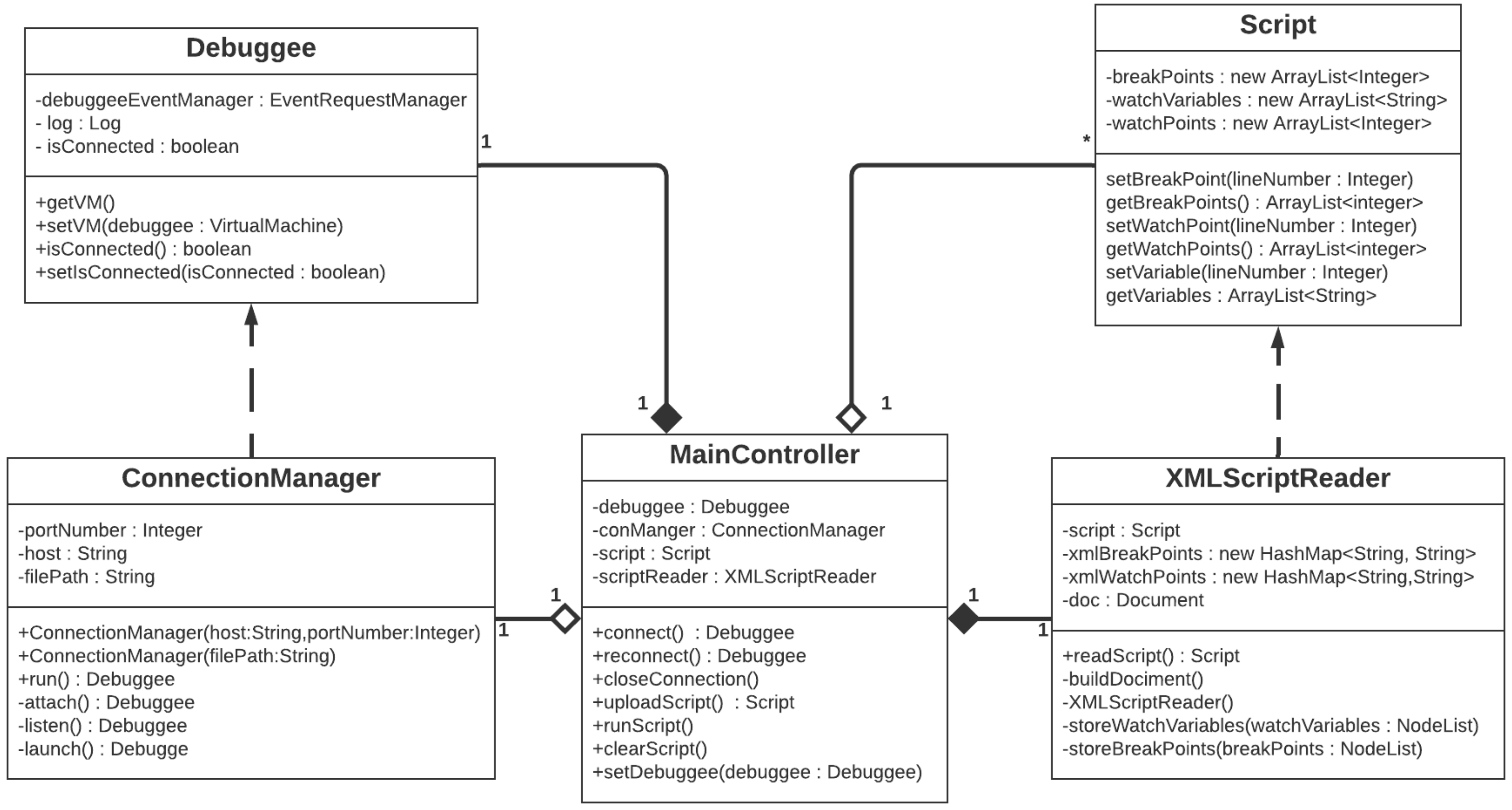
## Appendix 2 – Debugger Use Case Diagram



## Appendix 3 – Debuggee Use Case Diagram



## Appendix 4 - Initial Class Diagram

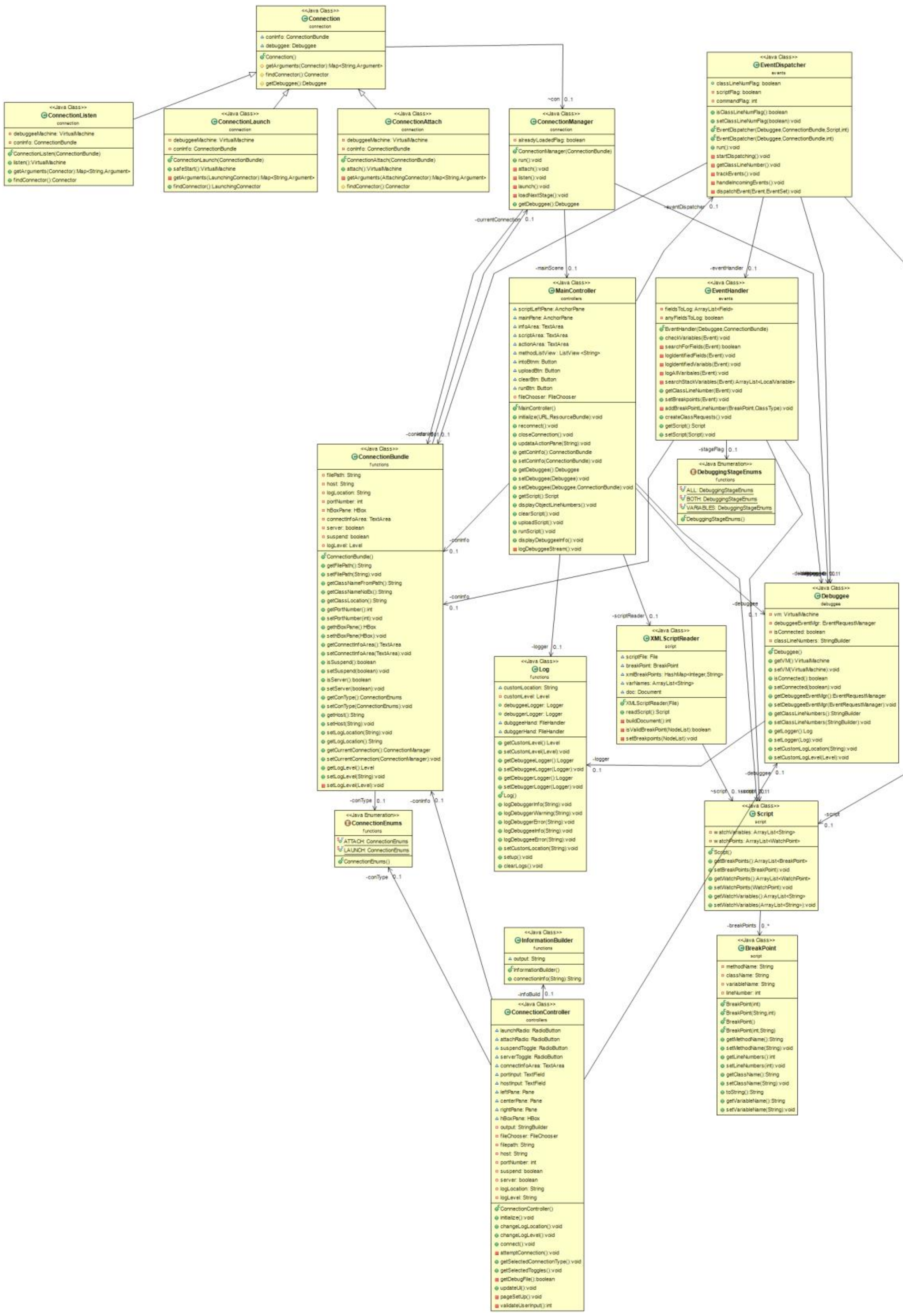


Debuggee
-debuggeeEventManager : EventRequestManager
- log : Log
- isConnected : boolean
+getVM()
+setVM(debuggee : VirtualMachine)
+isConnected() : boolean
+setIsConnected(isConnected : boolean)
Script
-breakPoints : new ArrayList<Integer>
-watchVariables : new ArrayList<String>
-watchPoints : new ArrayList<Integer>
setBreakPoint(lineNumber : Integer)
getBreakPoints() : ArrayList<integer>
setWatchPoint(lineNumber : Integer)
getWatchPoints() : ArrayList<integer>
setVariable(lineNumber : Integer)
getVariables : ArrayList<String>
ConnectionManager
-portNumber : Integer
-host : String
-filePath : String
+ConnectionManager(host:String,portNumber:Integer)
+ConnectionManager(filePath:String)
+run() : Debuggee
-attach() : Debuggee
-listen() : Debuggee
-launch() : Debugge
MainController
-debuggee : Debuggee
-conManger : ConnectionManager
-script : Script
-scriptReader : XMLScriptReader
+connect() : Debuggee
+reconnect() : Debuggee
+closeConnection()
+uploadScript() : Script
+runScript()
+clearScript()
+setDebuggee(debuggee : Debuggee)
XMLScriptReader
-script : Script
-xmlBreakPoints : new HashMap<String, String>
-xmlWatchPoints : new HashMap<String,String>
-doc : Document
+readScript() : Script
-buildDociment()
-XMLScriptReader()
-storeWatchVariables(watchVariables : NodeList)
-storeBreakPoints(breakPoints : NodeList)
1
1
*
1
1
1
1
1

## Appendix 5 – Debuggee Example

```
public class Debuggee {
      public static void main(final String[] args)
      {
         int letterCount = 0 ;
         String checkWord = "Debugging";
         String SingleLetter = "";
         int i;

         for (i = 0 ; i < checkWord.length();i++) {
            SingleLetter = checkWord.substring(1,1);

            if (SingleLetter.equals("g")){
               letterCount++;
            }
         }
         System.out.println("G was found " + letterCount + " times.");
      }
}
```

# Appendix 6 – Final Structure



## Appendix 7 – User Test (Debuggee)

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;

public class HelloWorld2 {
static String Pname = null;
static boolean shouldWait = true;
private String textVal = null;
public static void main(final String[] args) {
HelloWorld2 helloObject = new HelloWorld2();
System.out.println(helloObject.toString());
final JFrame parent = new JFrame();
JButton button = new JButton();
helloObject.setString("Hello World. ");
int x = 4;
String s = "hello";
button.setText("Click me to show dialog!");
parent.add(button);
parent.pack();
parent.setVisible(true);
String name = JOptionPane.showInputDialog(parent, "What is your name?", null);
Pname = name;
shouldWait = false;
System.out.println("Should wait changed to false");
if (name.equals("password")) {
        shouldWait = true;
        System.out.println("Should wait changed to true");
        helloObject.setString("The secret password was entered");
        x = 100;
} else if (name.equals("exeption")) {
        helloObject.throwExeption();
        x = 6;
} else {
        helloObject.setString("The password was entered");
        x = 0;
}
JOptionPane.showMessageDialog(parent, helloObject.getString() + Pname);
System.out.println(helloObject.toString());
System.exit(0);
}

public HelloWorld2() {}
public void setString(String str) {this.textVal = str;}
public String getString() {return this.textVal;}
public boolean getShouldWait() {return shouldWait;}
public String toString() { return "HelloWorld object: " + "shouldWait = " + getShouldWait() + "
helloWorld = " + getString();}
public void throwExeption() {
try {
```

```
Thread t = new Thread(new Runnable() {
public void run() {
                        try {
                                    System.out.println(new NoClassDefFoundError().toString());
                        } catch (Throwable ex) {}
                                    }
                        });
                        t.start();
                        t.join();
            } catch (Throwable th) {
                        // Eat this and don't tell anybody
            }
}
```

# Appendix 8 – Worksheet

## Scripted Debugger Worksheet

This worksheet outlines a simple debugging session on the supplied ‘HelloWorld’ (Debuggee) program. You will be asked to establish a connection to the debuggee, upload the provided ‘Script’ file and execute the script. You will be asked to perform this task three times, once for each connection type. Before beginning the test, you must compile the HelloWorld.java with debugging features available. To do execute the suppled ‘compile.bat’ or open a command prompt in the directory that contains the debuggee and enter:

```
javac -g HelloWorld.java
```

## Local Connection

Make a local connection to the debuggee, insure the ‘Suspend’ toggle is selected. Upload the script and execute. Please fill in the actual results, if the expected results were achieved simply provide ‘yes’, anything outside of the expected results please describe in the ‘Actual result’ section. Any additional comments can be added to the ‘Comments’ section.

| Num | Action | Expected result | Actual result | Comments |
|---|---|---|---|---|
| 1 | Connect to debuggee | Connection page is hidden. Interactive page is displayed | | |
| 2 | Upload Script | Script approved | | |
| 3 | Execute Script | Debuggee executed as normal. Variables: s, x, helloObject and shouldWait are logged and outputted. | | |

Please provide any additional feed-back about making a local connection.

## Remote Connections

When performing either of the remote connection types enter ‘localHost’ as the IP address of the host and ‘8000’ as the port number. When prompted for the name of the class enter ‘HelloWorld’.

### Attach Connection

Before attempting this connection type please execute the supplied ‘Attach.bat’ file. This will launch the debuggee as the server and wait for the debugger to connect. Insure the ‘Server toggle is **not** selected, now repeat the same actions asked of you for a local connection.

| Num | Action | Expected result | Actual result | Comments |
|---|---|---|---|---|
| 1 | Connect to debuggee | Connection page is hidden. Interactive page is displayed | | |
| 2 | Upload Script | Script approved | | |
| 3 | Execute Script | Debuggee executed as normal. Variables: s, x, helloObject and shouldWait are logged and outputted. | | |

Please provide any additional feed-back about making an attached connection:

**Listen Connection**
When attempting this type of connection, provide the required details and ensure the 'Server' toggle is selected. First attempt the connection within the debugger, immediately after, launch the 'Listen.bat'. The Listen bat file will connect the debuggee to the listening debugger. Again, repeat the same actions asked of you for a local connection.

| Num | Action | Expected result | Actual result | Comments |
|---|---|---|---|---|
| 1 | Connect to debuggee | Connection page is hidden. Interactive page is displayed | | |
| 2 | Upload Script | Script approved | | |
| 3 | Execute Script | Debuggee executed as normal. Variables: s, x, helloObject and shouldWait are logged and outputted. | | |

Please provide any additional feed-back about making an attached connection:

## Script Manipulation
Open the provided script file and modify it to contain an additional breakpoint with the line location 59. Add the variable 'Pname' to be included in the debuggers output. Please provide any feed-back you may have relating to the script below. Do you think the script provides a flexible structure? What other things would you like to be able to identify in the script.

## Benefits of Scripted Debugger

Now that you have had some interaction with the debugger, do you think that a scripted debugger could be useful? What other kinds of actions would you expect a scripted debugger to be able to accommodate? Would you use a robust scripted debugger?

# Appendix 9 – Quick Installation Guide

This document will describe the requirements need to launch the debugger prototype and the environment needed for future development. The prototype is dependent on the JavaFX libraries which are no longer part of the standard Java JDK. For the debugger to be launched or for future development these libraries are fundamental. The debugger is also heavily dependent on the external 'tools.jar' library that facilitates the JDI interface.
The packages and classes included are as follows, listed classes are fundamental.

| Package Name | Class/Resource Name |
|---|---|
| application | Main.java |
| Connection | Connection.java<br>ConnectionAttach java<br>ConnectionLaunch java<br>ConnectionListen java<br>ConnectionManager java |
| controllers | ConnectionContoller java<br>MainController java |
| debuggee | Debuggee java |
| events | EventDispatcher.java<br>EventHandler.java |
| functions | ConnectionBundle.java<br>ConnectionEnums.java<br>DebuggingStageEnums.java<br>InformationBuilder.java<br>Log.java<br>UserNotification.java |
| script | BreakPoint.java<br>Script.java<br>XMLScriptReader.java |
| xmlLayouts | connectionLayout.fxml<br>mainLayout.fxml |

Two solutions are available for the JavaFX libraries for both aspects of the installation. The quickest solution is to use Java 8 as the version to execute or develop the debugger, because the necessary JavaFX libraries the debugger is dependent on are included as standard, requiring no additional alterations. The second solutions allow for any Java version to be used but requires that the external libraries are included in the launching arguments or build path for development. The JavaFX libraries can be downloaded separately and attached a long side any JDK version.

The external tools.jar is only necessary for future development as it is already included in the executable version of the debugger, as the tools.jar file was explicitly included in the creation of the jar file. For future development the tools.jar file is only accessible through the JDK and **Not** JRE. It can be found in the 'JDKDirectory/lib/tools.jar.

## Links

- https://www.oracle.com/java/technologies/javase-jdk8-downloads.html
    - includes JavaFX libraries and the tools.jar library.
- https://openjfx.io/
    - JavaFX external libraries