The Scripted Debugging of Java Program via the Java Platform Debug Architecture

Rory Taylor

September 2017

Dissertation submitted in partial fulfilment for the degree of
Master of Science in Information Technology

Computing Science and Mathematics
University of Stirling

Abstract

Typically, developer debugging their code must utilise one of two approaches. Either simple diagnostic statements are added into the program code itself and the program is executed normally, generating an output of requested variable values to compare with those expected, or the programmer utilises an 'interactive' debugger program. These programs run in real-time and retain control over the target program's execution, pausing it at specified points to allow the user to examine the current state of the target and to iterate individually through the code they have written to locate faults. For large scale programs, however, this is a labour-intensive process that can take up significant time, and requires the user parses through the wealth of information produced from the debugger to target those values that they are interested in. The diagnostic approach requires that these lines of code must be removed after debugging is complete, risking the introduction of new errors.

This project represents an attempt to create a functioning debugger specifically targeting the Java language, which would operate autonomously from the programmer using it. It would attempt to combine the targeted and unobtrusive nature of the 'diagnostic' approach to debugging, reducing the output to only those specifically requested by the programmer, with the real-time execution of an interactive debugger. This would also retain the advantages of the interactive debuggers in not requiring any of the original code to be edited. Instead, it would function automatically using a set of instructions contained within a user-generated script file, through which the programmer would specify their requests and allow the debugger to execute as normal without further input. This produces a report specifically designed to reduce the general flow of information to only those outputs targeted in the original script.

The solution to this task was to utilise the Java Platform Debugger Architecture (the JPDA), Java's integrated libraries designed to facilitate debugging and provide tools to interrogate a target virtual machine from a separate virtual machine. After implementing the connection which allows the debugger to launch the target program from a compiled Java class file, functionality was expanded upon using inspiration drawn from existing interactive IDE debuggers. The final program can request notification of classes being loaded, halt the target's execution at specified points, and extract or set specifically requested variable values at these points. It resumes the target automatically after these points, requiring no user interaction with the debugger outside the creation of a script file to hold the instructions, and automatically reports back on any exceptions which occur in the target. Finally, the program generates a targeted report to log important events and the requested values during runtime.

Attestation

I understand the nature of plagiarism, and I am aware of the University's policy on this.

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

- Inspiration and skeleton code for the EventHandler constant despatcher loop discussed in section 3.1 was taken from Wayne Adams' blog entry on Examining Variables in the JDI, from: *http://wayne-adams.blogspot.co.uk/2011/12/examining-variables-in-jdi.html*.

Signature **Date**

Acknowledgements

My sincere thanks and gratitude are extended to my supervisor, Dr Simon Jones, for the initial inspiration, constant technical assistance, and his ongoing advice and guidance throughout this project.

Alongside this, my heartfelt thanks go to all six of those individuals who suffered through the testing of the project to provide user feedback and also to Wayne Adams for his work and blog documentation of his own implementation of the Java Debug Interface; this proved a far more friendly way to explore small parts of the JDI functionality than Oracle's somewhat dense documentation. From his work, much of my own knowledge was gathered.

Finally, my eternal gratitude is extended to both of my parents, Jennifer and Iain Taylor, and my partner Claire Mottram; as always, their eternal patience and support both financially and personally is irreplaceable.

Table of Contents

List of Figures

1 Introduction

1.1 Overview

With the widespread importance of the debugging process within the software engineering discipline, a plethora of potential tools have been developed to try to assist a programmer with this critical task. For Oracle's Java language, users can choose to utilise an innate command-line debugger, a specialised debug tool or one of the integrated debug views native to many of the development environments (IDEs). Often, a potential user of a given piece of debug software does not require the full suite of functionality that they might offer; very simple debugging is typically targeted at single occurrences within a newly written program and can be output via additional lines of code in the source file with the intent of outputting specific values as the program executes. The full range of functionality offered by specifically developed debuggers have made them the premier choice for any attempts at large-scale testing of created code, despite this, the amount of information and functionality provided by a standard debug implementation, such as those within an IDE[1], can prove overtly complicated for a user seeking to run more simplistic tests or who is interested in simply monitoring their program without searching a wealth of unnecessary information.

As a project grows larger in scale, the debugging process becomes magnitudes more complex [2]. The already slow pace of a typical debug session, which requires iterating through each line of code at runtime and manually altering breakpoints each pass through the program to track down specific problems, slows down even more as the flow of information becomes denser. The greater the complexities of the program at each specific break in execution, the more information a traditional debugger throws at the user. For a programmer, often interested in testing only a very specific portion of the program that has just been modified, this dense information flow can prove overwhelming and slow the process down even further.

1.2 Objectives

The purpose of this project was to produce a debug implementation more suited to the 'middle ground' of possible testing requirements: a debugger designed to assist a programmer in essential testing of their software at the end of the implementation stage. The debugger was intended to be simple and highly targeted in its implementation, without extraneous information or additional clutter, while not requiring the user to edit their original code in any way to access functionality. The user would primarily interact with the program via use of an

external script file, used to specify the precise requirements for the output from the target machine and filter out any information that was not required.

The project was also intended as a proof-of-concept for the potential application of a scripted approach to debugging as a means of semi-automating the process, in the interest of programmer efficiency: by automatically fetching the specific values requested by the programmer and then resuming the target process, the debugger does not require the user to manually gather the information they require and then restart a virtual machine after each pause in execution, or to iterate through each line of code to identify problems. While the project also served to highlight some of the limitations of this approach, each individual debug session proved very fast compared to traditional approaches: only those UI interactions needed by the target program required the programmer to actually engage with the execution, while the automated script-based debugger simply ran in the background and returned the requested information as a report after execution was completed. This allowed the user to engage only with their program itself as it ran, allowing them to potentially identify other problems without the distraction of processing the debug view of an IDE. Originally suggested by Dr Simon Jones as a potential research project, the development of a software implementation for the Java language was designed to explore the potential functionality and usability of such a system, to explore how much of the process could be automated from the script. Much of this explored functionality and the specifications of the project originated as an attempt at streamlining and targeting those options that an experienced programmer might expect from their experiences with other Java debug implementations.

1.3 New Technologies

While some experience of the Java language itself was already possessed, much of the work on the scripted debugger required new technologies to be explored without previous experience. The Java development libraries (the JPDA)[3]**Error! Reference source not found.** dedicated to the debugging process proved the most significant learning curve, with dense documentation and a wealth of potential functionality embedded within; often the actual implementation of these options required entirely unfamiliar code. Similarly, previous exposure to the XML structure and syntax used to construct the script itself was limited only to basic HTML, which has some similarity. Despite this, the facilities for parsing an XML document in Java, and the structural requirements and options for constructing the file itself, were all new and required some dedicated study to understand.

In addition to the technologies used within the project itself, previous projects had only ever been attempted on the beginner-friendly BlueJ [4] development platform and exposure to the debugging process limited by this environment. For a project of this complexity, development was performed on the Eclipse IDE [5] and utilised the debug functionality within this environment for both testing of the project code itself and for inspiration as to potential functionality for the scripted debugger. Research was also required on the debugging process itself, due primarily to a lack of industry experience, to ensure that the essential functionality of the project would align with its ultimate goal of assisting the process.

1.4 Achievements and Dissertation Overview

At the end of the project, the final debugger software offers a user interface to load a script and target a compiled Java file. The debugger launches the specified target program as a separate suspended virtual machine, parses the script to prepare all the user-specified instructions within, then automatically continues the target's execution. It pauses at specified breakpoints from the script file to read or set variable values, before automatically resuming the execution. Then debugger then generates a report after execution is complete, containing: a log of events, all requested variable values, and an automatic report on any exceptions that occur within the target Java program with details on their source location. The program then allows the scripted instructions to be edited and the test run again repeatedly, allowing the programmer to compare the expected internal values with the final output to locate errors.

This dissertation will overview the background of the topic, analyse the problem itself and explore the specifics of implementation for the program. It includes a background search of related software, used to inspire the functionality of the final debugger, the specification of the initial problem and an in-depth analysis of the implementation of the resulting solution, separated into chapters as outlined below:

- **Demonstration**: A detailed look at the essential functionality of the program and what a typical session using the software looks like.
- **Problem Solution:** An overview of the initial approaches to development and design used to create the software in response to the Problem Specification.
- **Event-Request System Design:** An exploration of the internal structure of the Java Debug Interface (JDI) libraries and their use within the final solution.
- **JDI Communications:** Discussion of the specifics of implementing the connection between debugger and target debuggee program, running in separate virtual machines.

- **Script Design:** Analysis of the exact structure of the final script, based on the Extensible Markup Language (XML) structure, and the way in which the debugger parses it.
- **Internal Storage:** An in-depth exploration of the internal storage structure used to hold and translate the instructions from the script itself into JDI-readable objects.
- **Evaluation:** A final analysis of the resulting software and of the feedback gained from a sample set of potential users.

2 State of the Art

2.1 Background

When discussing the tools used to assist the debugging process, a clear definition of the process is important to derive a clear goal for the project. Debugging, as defined by the tutorial site *TechTarget*, is identified as:

> *'In software development, debugging involves locating and correcting code errors in a computer program. Debugging is part of the software testing process and is an integral part of the entire software development lifecycle.'*[6]

The debugging process tends to be preliminary to the actual testing phase in typical software development, usually occurring at the end of the implementation stage [7]. The ultimate goal of any software project is to create a 'reliable product'[8]; to this end, programming workflows employ specialised testing to locate errors in tandem with the debugging process to locate the source of each flagged error [8]. This important distinction, specifically mentioned across several published introductions to the debugging process [9], is the difference between debugging and the actual Testing phase of the Software lifecycle. Testing – typically performed externally to the programming itself – is a sequential process of designing specific test scenarios and comparing the expected action of a program to the actuality, to uncover any potential errors [10]; it is the act of executing a program with the intention of locating defects [11]. Meanwhile, debugging is performed by the programmer to locate the source of any errors in their code; this process is usually constant while the software is being created, or following a period of testing [12]. As a result, the tools available for each stage vary; specifically, debugging requires an open source – or 'glass box' [14]– approach to the underlying code. To debug successfully, the programmer must understand the code and anticipate the expected action at each point of execution. However, the testing phase varies in its specific approach and is designed to test the expected functionality of the program; it does not root through the underlying execution to locate the faults.

A common technique for a programmer at the debugging phase is 'Regression Testing'[14]; here, a fault is identified in the program, the execution traced to locate the fault and the error fixed in the code, before the exact test that initially highlighted the error is repeated again to ensure that the fix was implemented correctly. This is particularly applicable in the case of the project's unique specification; by storing tests scripts out with the tools used to

execute them, repeat regression testing and modification of the original test parameters during the debugging process could be significantly simplified by simply editing or reusing existing scripts. A wealth of potential tools exist for all portions of the testing/debugging phase of development to provide assistance to programmers, particularly in the execution of Regression Testing [14](an exercise prone to human-introduced error without some level of automation), and via traditional text-oriented debuggers which provide insight into the 'dynamic behaviour of running programs' [14] at runtime.

2.2 Debugging Software

A rich variety of debugging tools exist for the Java platform. To explore common functionality and derive possible approaches for the debugger to be implemented within the project, an online opinion list – 'The Definitive List' [15] published on *DZone.com* – and *GitHub*'s top IDE index [16], organised by usage, will be used to derive individual case studies. Specifically identified software to be examined in this section:

- Interactive Development Environment debug options from *Eclipse* [5], *NetBeans* [17] and *IntelliJ IDEA* [18], three of the most popular IDEs for use with the Java language
- Oracle's own Java Debugger (JDB) command-line implementation [19]
- *ChrononDVR*, a 'historical' debugging tool for use with Java that takes snapshots of the internal memory at each stage of execution for the programmer to view later [20]

The JDB serves as a strong reference for debugging via the Java Platform Debug Architecture [3]. A command-line debugger without a user interface, the JDB functions as simple text commands executed upon a target program, either launched from the JDB console or by attaching the debugger to an already executing target [21]. It contains a suite of essential functionality to execute upon the debuggee that offers a clear implementation of the full options offered by the JPDA, including: the option to print out the value of a given variable or object via the "print" command, to request that execution be paused at a specified point via a breakpoint command, and the option to track the currently executing 'thread of control' (defined as a 'section of code executed independently of others within a single program') [22] as a location within the target virtual machine to act upon. As the JDB is an interactive debugger, intended to allow the programmer to act upon executing code in real-time and respond to internal events, it also contains options to respond to Exceptions (error occurrences) that trigger in the target automatically and to 'Step' through lines of code by preference.

Transitioning to the integrated debug tools contained within the IDEs allow a much richer, though more complicated, suite of debug options. IDE debuggers still run interactively in real time, with the programmer using the tools provided to follow their code through the path of execution and to explore the virtual machine [23] as it executes. Beginning with market-leader Eclipse (per the IDE index), the functionality offered is notably richer than the intentionally simple Java Debugger [1]:

- Offers a 'debug perspective'; a specific interface designed to allow the user to see all currently active variables and threads and explore them at will
- Most of the functionality of the JDB: breakpoints are set, via an interactive UI and variables are constantly read out at these points
- Variables do not have to be specifically request; Eclipse displays all fields that are currently on the executing stack (ie. being actively used by the program at that point in execution)
- Fields can also be tagged with Watchpoints: observers which will halt execution if the specified field it targets is accessed or altered by the program
- Breakpoints can also be set on any given Exceptions thrown, on a named Class when it is loaded in the target machine, or on a specified method to trigger when it is called internally
- The internal value of each variable can be manually altered at each break in execution, functionality not provided by the simpler JDB
- The programmer can step through each line of code as before, with the perspective updated in real time at each stage of execution.

Examining IDE competitors, such as JetBrain's IntelliJ IDEA and Oracle Corporation's own NetBeans, suggest that much of the above functionality is considered industry standard and required by the debugger process. IntelliJ offers the same suite of functions, from watchpoints [24] to breakpoints [25], and steps progressively through the code as it executes according to these requests. Netbeans also offers much of the same options, with the addition of enabling the user to take a visual 'snapshot' of current execution for reference to later [26]; these snapshots allow the programmer to progress more efficiently through target execution and store visual references to compare later results if repeating the test. It also offers a Profiler, designed primarily for visual assistance in optimising the program in terms of memory usage and performance [27].

In general terms, these examples offer a solid foundational view of the essentials of a debugger implementation that a Java programmer might be used to using or require for a debug

tool to be useful. However, a final approach to debugging has not yet been explored and remains relevant to the project's goals; ChrononDVR, a Java debugger that 'records' the execution of a program without pausing it for use as a point of reference afterwards. This, uniquely amongst the debuggers previously examined, allows Chronon to execute the target program without pause while still enabling the debugging process to be performed; it offers 'no editing' of the original source code [28] and allows the programmer to examine the action of their program without needing to be actively executing the program, via recordings of previous executions. This allows the code to be altered while the debugging is still in process, running unobtrusively without affecting the target's execution. Chronon also uniquely offers the option to step backwards through a program's execution, allowing the programmer absolute freedom to iterate through the execution and compare it to expected results. It advertises itself as requiring no diagnostic statements (functionality also offered by the IDE environments) and offering an approach to debugging that requires no specified breakpoints to pause execution [29]: the program simply executes, Chronon records the result at each stage of execution and the programmer is free to explore the results at will. While something of an outlier in the debugger field, Chronon is of interest because its unobtrusive approach and attempts to automate the debugging process to some degree are uniquely applicable to the development of any scripted approach, which will require some level of automation itself.

3 Problem Specification

3.1 Limitations of Traditional Approaches

Traditionally, interactive IDE debuggers – which allow the programmer to act upon the program at runtime and act dynamically in response to the information read from their executing code – have remained the widespread solution for the ever-present task of debugging. For programmers seeking to test their work, the possibility of any level of automation of that often time-consuming process is a consideration worthy of exploration. Typically, many of the debuggers targeting the Java language are built utilising the Oracle-designed Java Platform Debugger Architecture (the JPDA): from IDE software such as Eclipse to specialist debugger implementations like the (discontinued) JSwat [30]. Providing a framework and core functionality requirements for these programs, the JPDA consists of several libraries and toolsets to enable one program to seize control of and interrogate running Virtual Machines separate from their own. While the exact methodology and even the specifically utilised libraries vary between implementations, these libraries offer a core set of functionality between them that allows this interrogation without forcing the original code to be edited or altered in any way.

Working with very simplistic debug requirements and for general testing while the program is being written, often programmers simply add diagnostic code manually to request the output of specific, targeted variables to the console. This process requires editing the code throughout to target portions for debugging initially and, once the debugging process is complete, tidying up the code again to remove these outputs. The activity can prove an error-prone one, which can result in additional bugs being inserted into the program after the process designed to root them out has already executed. Console command line tools such as that offered by the JDB [19] do not require the code to be edited but suffer a steeper learning curve than UI-based debug programs and problems of usability and readability.

As the debugger transitions into more complex implementations, such as Eclipse's debug display shown in 0 or Netbeans, much of these outputs are moved away from the code itself and onto the IDE; the IDE constantly reads out values based upon the execution's current position in the code to allow a programmer to explore 'under the hood' of their program as it executes, enabling them to identify problems based upon their own understanding of what should be occurring. However, the flow of information can prove to be unnecessarily dense, with many variables being extraneous or difficult to interpret for the programmer. Often, the programmer is given object references rather than actual values, which can lead to clutter

within the UI without being particularly readable. This can limit easy access to those values which the user is actually interested in. IDE debuggers such as these have other limitations, typically in terms of requiring lengthy debug sessions iterating through lines of code to determine faults throughout execution and in the requirements to manually set breakpoints throughout the code; these points have to then be removed and re-added in the future if other parts of the code need to be tested or a test requires to be repeated..

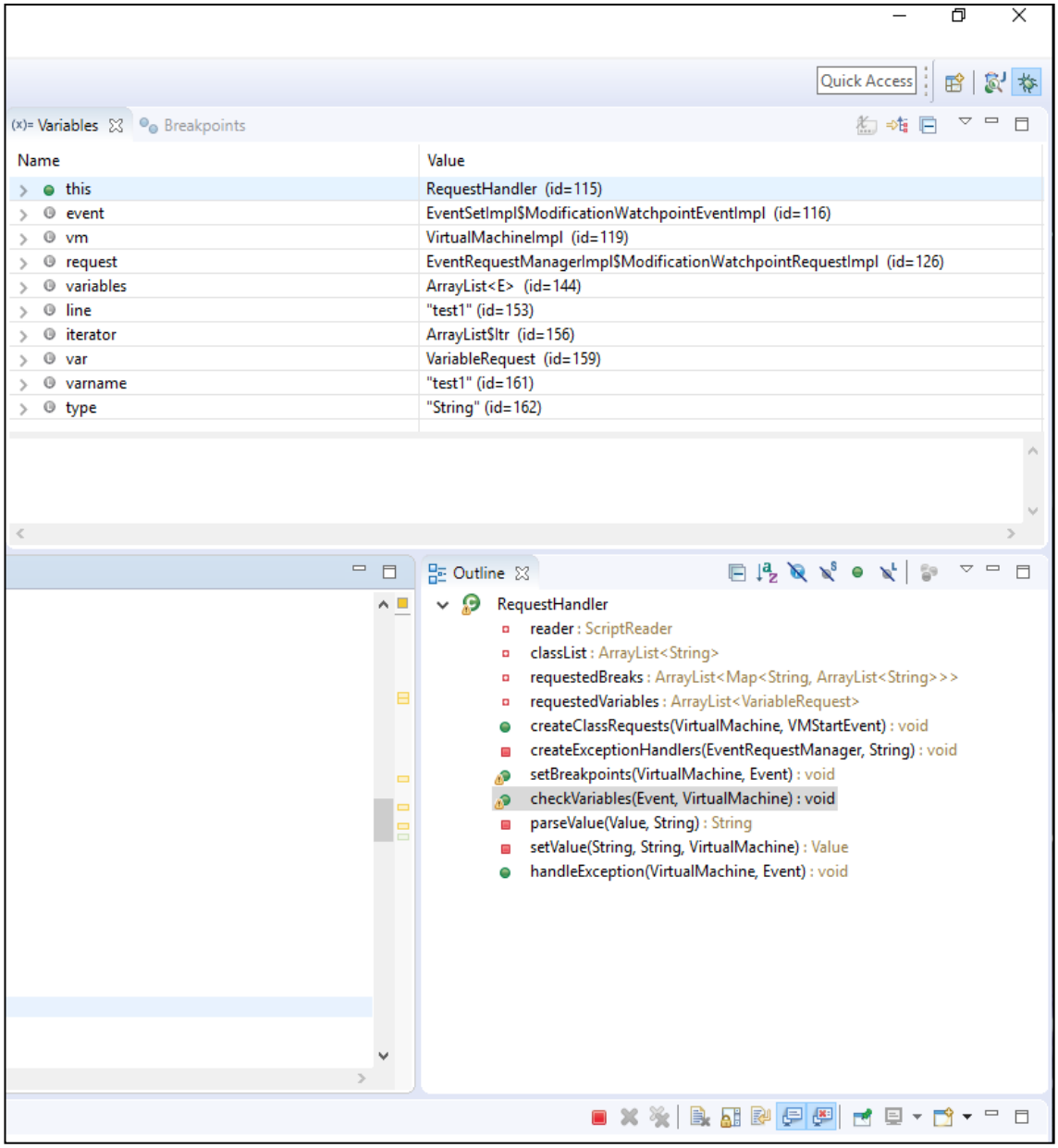

Figure 1. A typical Eclipse IDE debug perspective view of active variables/objects

3.2 Specification of the Scripted Approach

As a result, any attempt at producing a debug solution rooted in scripted automation requires that it attempts to solve some of these limitations. Essential requirements would include attempting to avoid some of the identified problems in other, common debug implementations such as:

- The manual editing of the code itself to insert diagnostic statements and implement outputs.
- Those limitations on accessibility inherent in console-command based debugging, with a clear UI and simple learning curve.
- The frequently (and often unnecessarily) dense flow of information with general debug views such as Eclipse.
- The typically slow, methodical nature of interactive IDE debug sessions, iterating through the code line by line and hunting through the UI to set or remove breakpoints.

Many of these requirements are met innately as a result of the scripted approach; utilising a scripted set of instructions allow the user to save and reuse those same scripts as needed to retest functionality to reproduce a possible fault or to test a function repeatedly after iterating or improving upon the code. The script queuing up instructions internally – assuming the debugger itself allows for it and can execute any required user-generated events within the target virtual machine – would also allow for extended periods of automated debugging without the programmer being required to specifically interact with either program; the final goal of such a program would be to treat a debugger as something akin to a compiler in a large-scale software project, with instruction sets being prepared alongside the code itself before being left to run and the results evaluated.

Similarly, alongside the IDEs already discussed, the scripted debugger should entirely remove the need to manually edit the targeted code by inheriting the functionality to readout targeted variables at specific points of execution. Instead, the script file itself could be simply edited on the fly, altering individual instructions rather than base code to adjust the readouts as required. Entire scripts could be interchanged, stored and potentially combined for later use to execute extended sets of instructions; this approach can prove useful for early ‘white box’ programmer testing, using separate scripts to test individual portions and then combining these scripts for larger-scale testing of the program as a whole.

Final requirements are specified mostly in terms of usability; the scripted debugger should endeavour to allow the user to fine-tune their requirements to avoid the information ‘bloat’

that can occur in typical JDI-based debug readouts, as seen in the IDE debuggers; ideally those variables and the points at which their values are to be fetched should be specified, within the user-generated script, while any additional values which could reasonably be anticipated by that user should be ignored to allow for a uniquely targeted approach to debugging. Secondly, much of the core functionality beneath the 'script level' of the program should be hidden from the user to ensure simplicity in the UI; clarity of the targeted debugger remains of paramount design importance, with users interacting only with the required setup information and the script itself before allowing the program to execute and generate a report.

A possible example of the finished debug program would be to script a simple task to monitor the changes to two instance variables across two separate classes while the program executes normally. Rather than a programmer adding in a console print command to each occurrence of the variable within the code itself, or adding a watchpoint to each field via the IDE and parsing through all of the resulting variable readouts at each break to locate the field they are interested in, the user could instead create a short script file and launch the program through the debugger. He could then run the program normally, without pausing execution at any point, and examine the generated report afterwards to identify any point at which those field's values were not what they were expected to be. The report would be focused, reporting only on those fields specifically requested by the user, for a quick overview of the changes to those fields. The user could then simply edit the script file, adding a new line to request supplementary readouts on other related variables or a new breakpoint, then repeat execution immediately for a new targeted report to aid in narrowing down any problems.

4 Demonstration

When the application is initially launched, a clean user interface (see Figure 2) offers simplistic options to allow a user to locate the source file they wish to debug – referred to henceforth as the target program or target virtual machine – and the prepared script file containing all the instructions to be executed by the debugger. The UI does not enable the editing of the script innately, although it can be edited at any point before launching the debugging process.

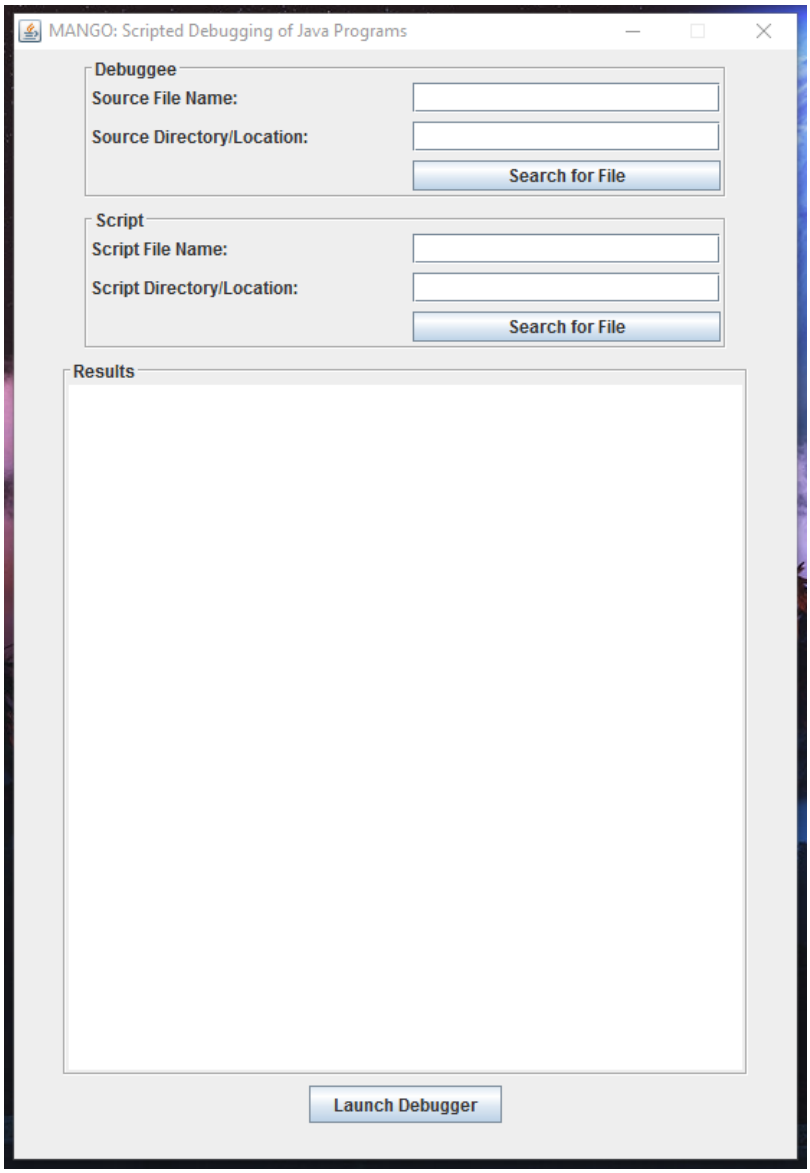

Figure 2. Initial UI Layout for the Debugger

Much of the actual interaction with the instructions is executed through the XML script file, edited in any given text editor and saved as a basic text file for the Debugger to utilise. The program includes a specified Document Type Definition (DTD) file (see Chapter 8.1),

which limits the parser to only those options that the debugger can execute. The DTD, *'!xml-define.txt',* and the accompanying documentation file *'!readme.txt'* together specify the exact structure of the script file and the options available to a programmer with regards to specifying the instruction set to be executed. A third file – *'!template'* – offers a basic pre-made script structure for a two-class program and includes modifiable syntax for all available options:

- breakpoints set by line within a given class
- watchpoints, for either access or modification, set on an instance field within the specified class
- the variables to locate at each break in execution, either to read the value from as a 'get' command or to 'set' their value with a specified one

These files are designed to allow the user to work fluidly and comfortably with the script file itself with little learning required. Once the user is content with the script, they can identify the two target files in the UI itself. This can either be done manually, by simply typing the directory and file name into the provided boxes, or by utilising the "Search for File" buttons in each section to open a basic file browser (Figure 3):

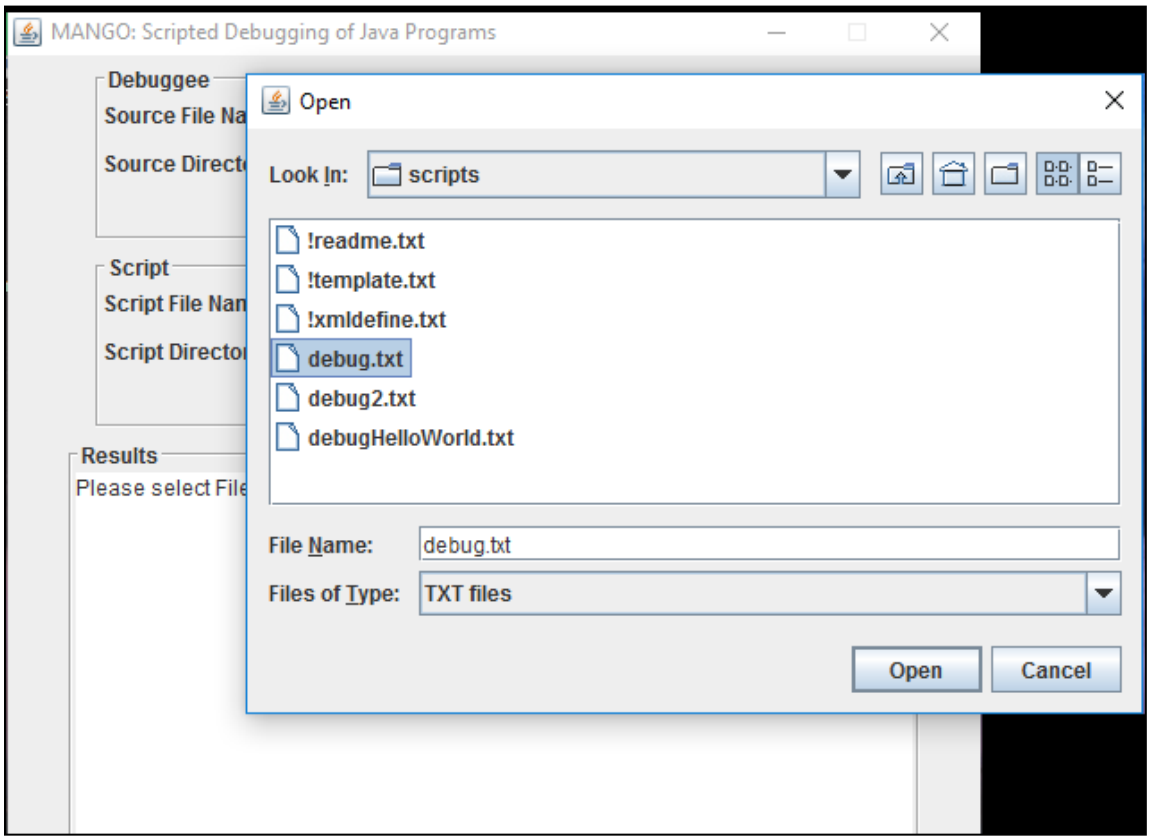

Figure 3. File Browser for the Script .txt file

The browser, after selecting a file, parses the location and displays the file's name and directory in the appropriate boxes for the user to check before launching the debugging process with the bottom button. Basic instructions are present in the 'Results' box at this stage to guide the user and encourage them to confirm the locations before launching (Figure 4).

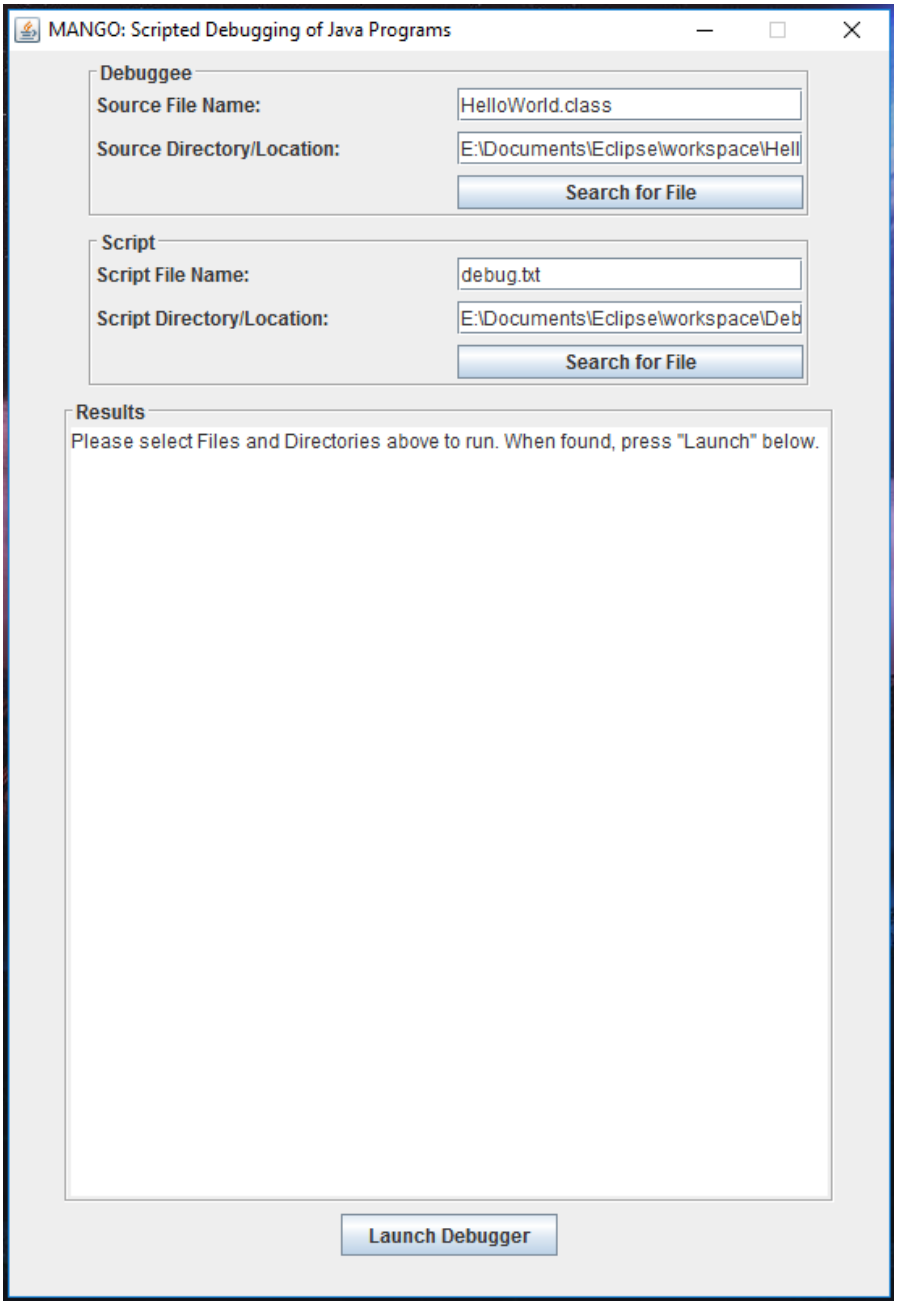

Figure 4. The debugger ready to launch the "HelloWorld" class

When the *'Launch Debugger'* button is pressed, the program begins by parsing the indicated script file and queueing the instructions contained within. At this juncture, any errors in the script's structure or content are indicated to the user so they can be fixed before running the program again. If the script is parsed without problem, the target debuggee is launched and

then suspended to begin utilising the parsed instructions. The internal execution of both instructions and target execution is largely carried out invisibly to the user, though the target program itself will be launched in a new window and remains visible throughout if it includes a user interface element; the debuggee thus remains fully interactive and functional throughout. To indicate to the user that execution of the target program is currently ongoing, the message changes to "Executing the Target Class…" (Figure 5) within the Results area to ensure the user is informed of any activity underlying the program.

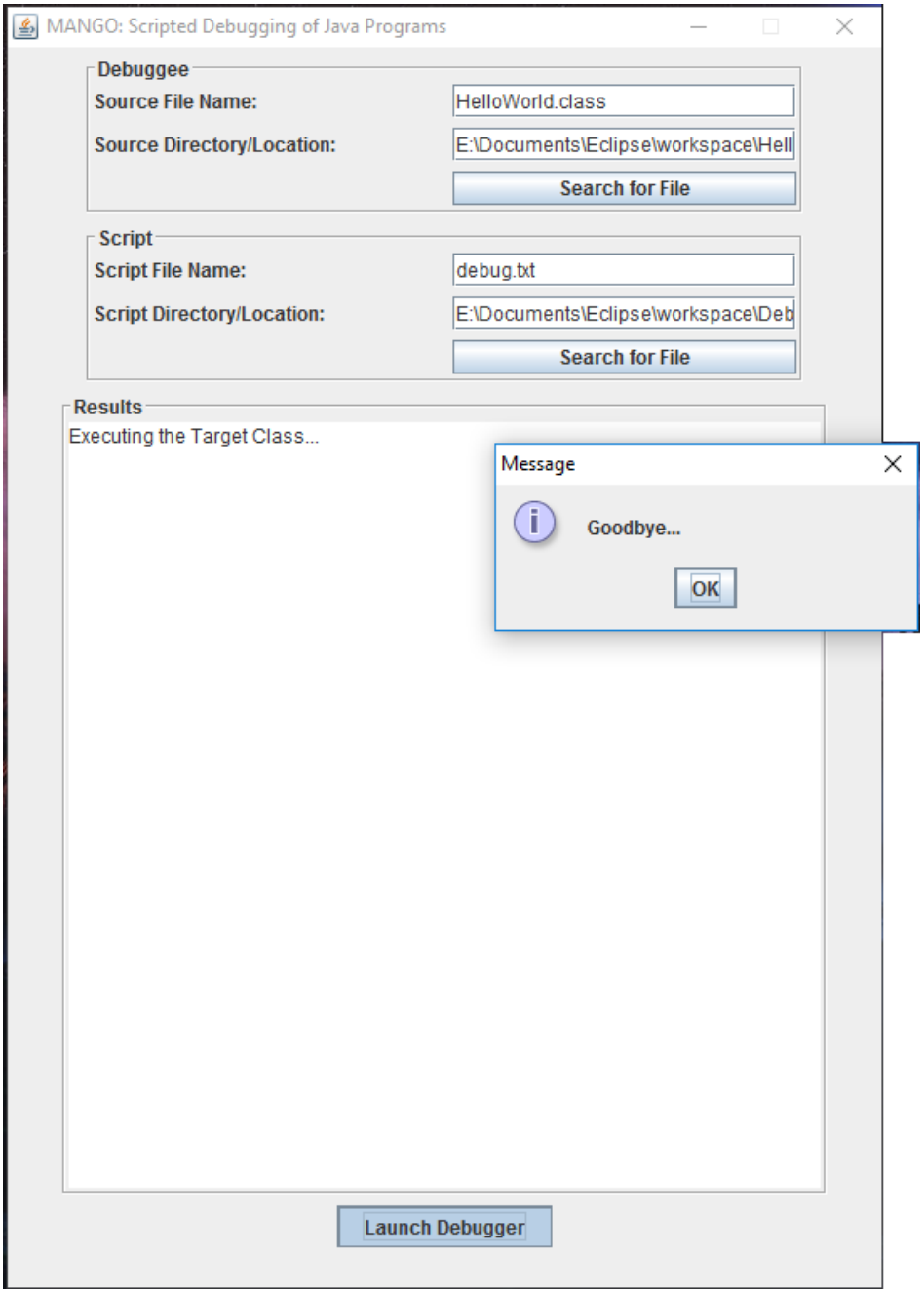

Figure 5. The debugger midway through the execution of 'HelloWorld'

After execution, the results of the scripted instruction are now displayed in the Results area. This area now includes a general log of important events, to help the programmer narrow down any problems in script execution (such as a breakpoint not being triggered correctly due to errors in the script) by following the flow of instruction execution through the process, and displays the requested values of each variable at each given location. The report log is formatted, with breaks between each event and their associated results; an initial log of the preparation and setup of the target program is included which then transitions to the actual results of the requested interrogations during runtime. A scrollbar will appear automatically if the log exceeds the height of the text area, as pictured in Figure 6:

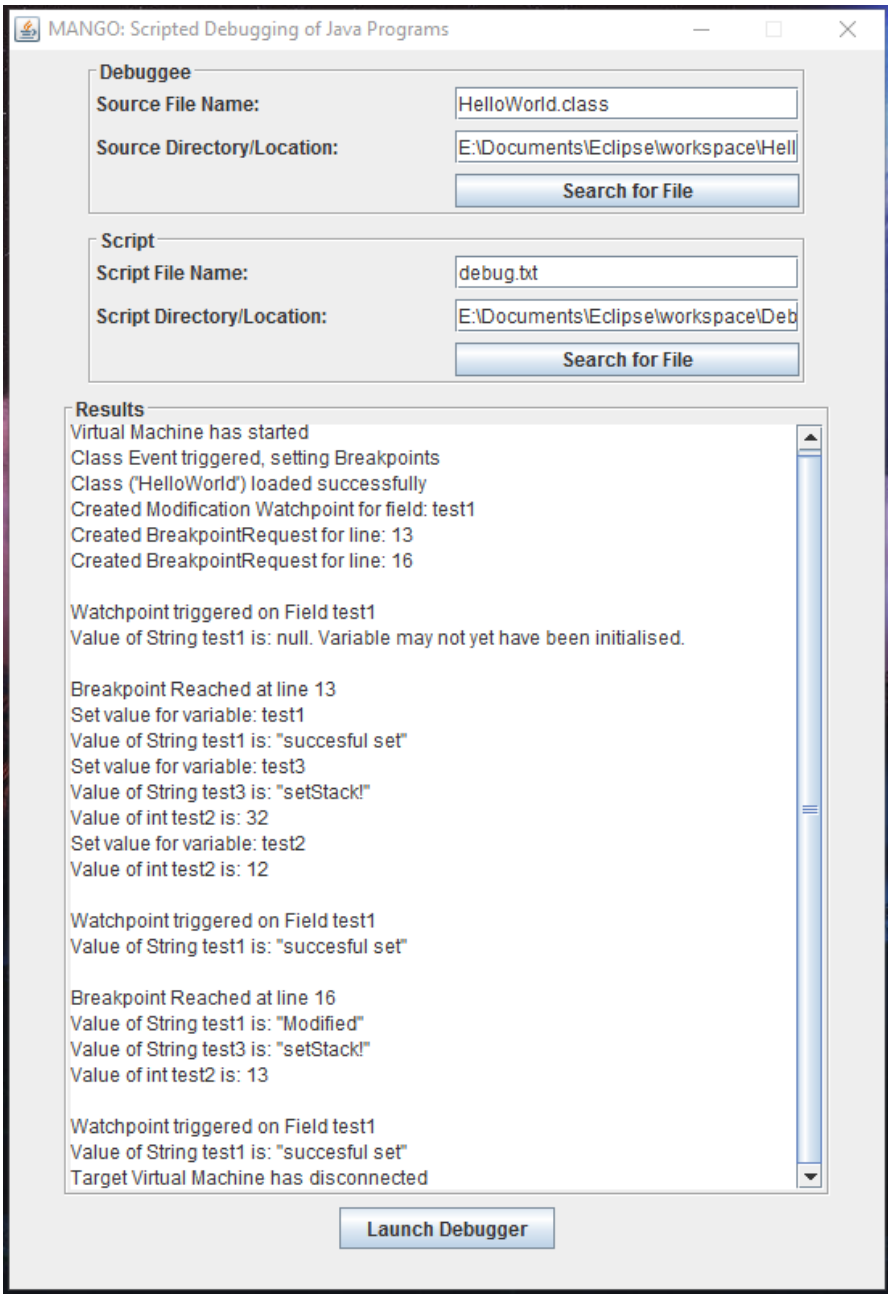

Figure 6. Final results and event log of the Debugger

The log will always end with a final notification of the successful disconnection of the target Virtual Machine, to allow the user to confirm that the program has exited and that the process is not still running in the background. Additionally, any exceptions that occurred within the target program will be reported without an implicit request from the script; this notification will include the location the exception was caught at, the location it originated from, the nature of the exception and indication of the exception going uncaught or being caught by the target program. This functionality was implemented to avoid the user having to pre-empt or predict any possible exceptions that might occur, allowing the debugger to report all such occurrences without specifically requested. These exceptions are internally filtered to avoid overloading the Report with information, such as any exceptions triggered during initial loading of the program's Java library resources. At this point, the script can be safely opened and edited separately to correct problems or request additional readouts; alternatively, a new script can be loaded by simply changing the file name and the process repeated indefinitely to test the debuggee by simply hitting 'Launch' again after adjusting the requirements.

5 Solution Overview

5.1 The Java Platform Debugger Architecture

The JPDA is the official set of libraries from Oracle intended to facilitate both local debugging of Java programs and to form the developmental core of software-based debugger attempts [3]. The JPDA itself comprises multiple libraries, which together form two interfaces which control the actions of the target virtual machine, operating across a communications protocol – the Java Debug Wire Protocol (JDWP) [31] – between bytecode-based virtual machines. These interfaces run on one of two software components, either implemented in the front-end for the *Java Debug Interface* (JDI) libraries or in the back-end, programmed into the target virtual machine (as with the JVM TI, or *Java Virtual Machine Tools Interface*) [32]. This is visualised in Figure 7, below:

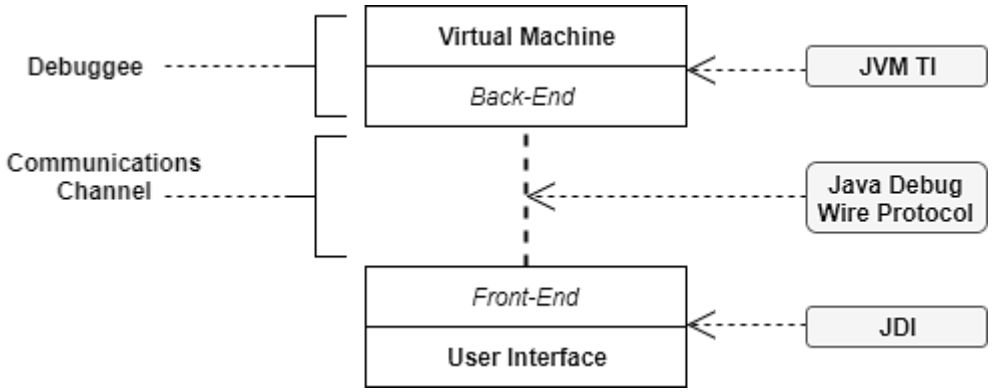

Figure 7. Overview of the JPDA, adapted from Oracle Documentation [34]

The JDWP protocol specifies the format of communication between two Virtual Machines; the communication is bilateral, in that either can act as client or server depending upon the specifics of the implementation. In Oracle's definitions, they identify a front-end process as the debugger and the back-end as the target process of the debuggee; the Wire Protocol defines the specific format of the exchange between these processes. The JDB is, in fact, a specific implementation of the JPDA that utilises this same communication protocol, across an OS socket, for delivery and retrieval of data. It is this same architecture – implemented in the front-end of our application via the JDI – that was utilised throughout the design of our script-based debugger solution.

The JDI was selected as the library most applicable to the original specification, implementing a front-end communications channel standard to debugger applications, intending to

fulfil the original requirement of utilising unedited source code of the target VM by placing the onus upon the debugger itself to control communications without editing the source code.

5.2 Core Solution Layout

Having chosen a JPDA implementation most suitable for communications and control of the targeted Java virtual machine (the debuggee) by the script-based debugger program, a core layout for the development of the debugger was developed as a template to work from throughout the project. A basic communications diagram (Figure 8) formed the foundation for development with basic structure indicated; throughout development, this diagram was constantly iterated upon and developed to provide the framework used to implement the debugger.

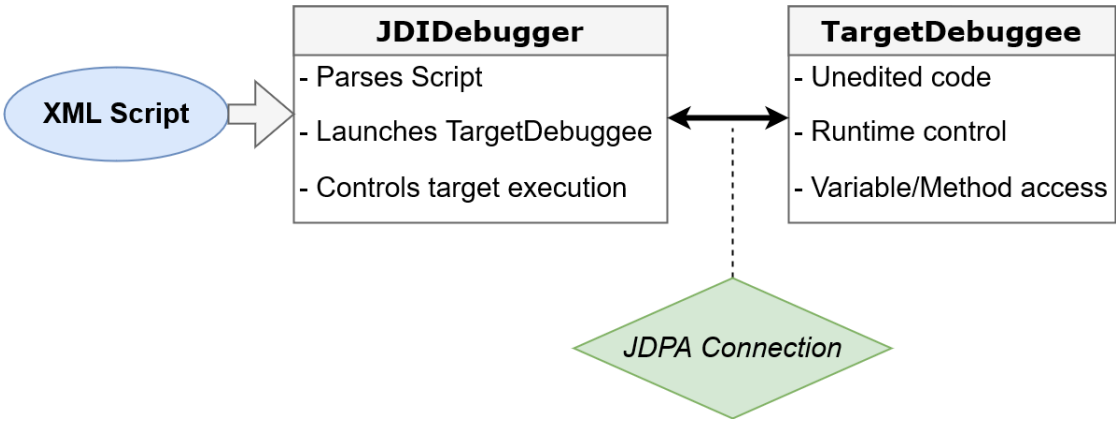

Figure 8. Basic outline diagram for the Scripted Debugger

Though crude, the essentials of the diagram were used to make several core design decisions. The debugger itself, implementing the JDI interfaces, launches the connection with the target debuggee and retains control over its execution. The XML script itself would contain the instruction set utilised by the Debugger to limit the feedback from debuggee to programmer. It was of importance that the script be human-readable and easy to interpret by the programmer. Throughout early development, it became clear quickly that the most efficient way to utilise the debugger would be for a programmer to simply develop the script manually, as oppose to working through a UI, to be able to quickly make any changes as each debug session finished and reported back. To fulfil this requirement, it became important that the script be easy both to parse for the user and to edit to add new functionality to existing sets of instructions.

With these developments in mind, a simplified class diagram was created to decompose essential elements of the functionality into portions for development to begin. The specifics of the JDI implementation were separated into two sections, shown in Figure 9:

- a communications package (initially just the *JVMComms* class), responsible for implementing the connection to the target Virtual Machine and controlling the target's execution while dispatching sets of instructions to it
- a parser package which would contain code relating to reading in the XML file and translating the contents into JDI commands to be executed on the target. Finally, separated by the script and report files, the User Interface would be implemented to allow essential user functionality over the execution

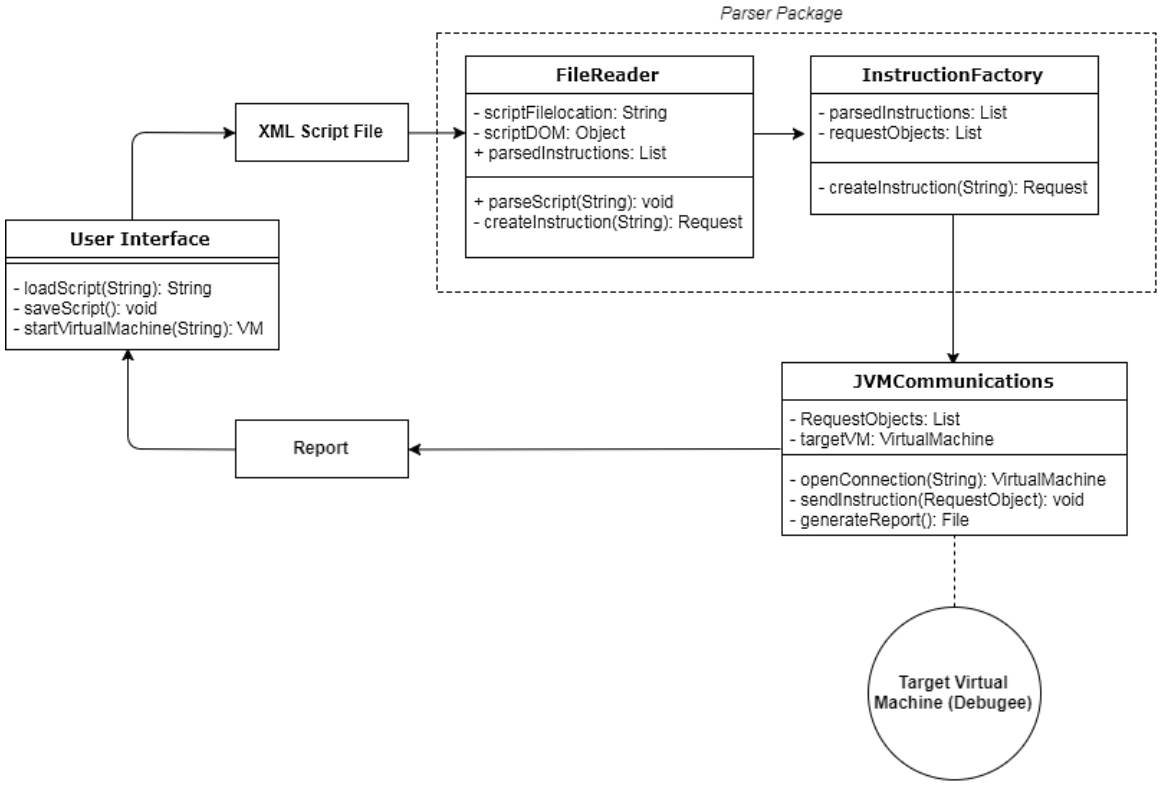

Figure 9. Early diagram to illustrate class layout of Debugger program

Later developments, primarily related to the intricate structure of the JDI itself and the requirements of working within the confines of this structure, would develop this class structure significantly. Several unexpected challenges arose during the implementation of this solution which proved pivotal to the final solution. Issues related particularly to the structure and challenges of the debugger's internal storage solution (see Chapter 9), which had to be functional within the scope of the existing JDI objects that were being utilised by the communications channel.

***A note on the JDI Documentation**: Throughout the following chapters, significant references will be made to many of the objects that exist within the JDI structure. To avoid overwhelming the text with individual references to each object in the documentation, each object will be italicised when first mentioned and can be found under the 'All Classes' sidebar in the main JDI index, found in reference [34].*

6 JDI System Design

6.1 The structure of the Java Debug Interface

Utilising the Java Debug Interface (JDI) as the framework around which the debugger is constructed at once provides large amounts of useful structure and potential options to the program, including a core framework for the potential functionality of the debug options themselves, while also providing limitations and the particular challenges of working within an existing structure. The JDI is structured as an interconnected 'web' of interfaces, typically multi-layered and interconnected with other layered structures. An example of this structure is shown in Figure 10: the *Event* interface has only a single superclass of its own, that of the *Mirror* interface; however, the Event class' own varied subclasses link to other superclasses, all interfaces without concrete implementations, dependent upon the subclass' specific implementations.

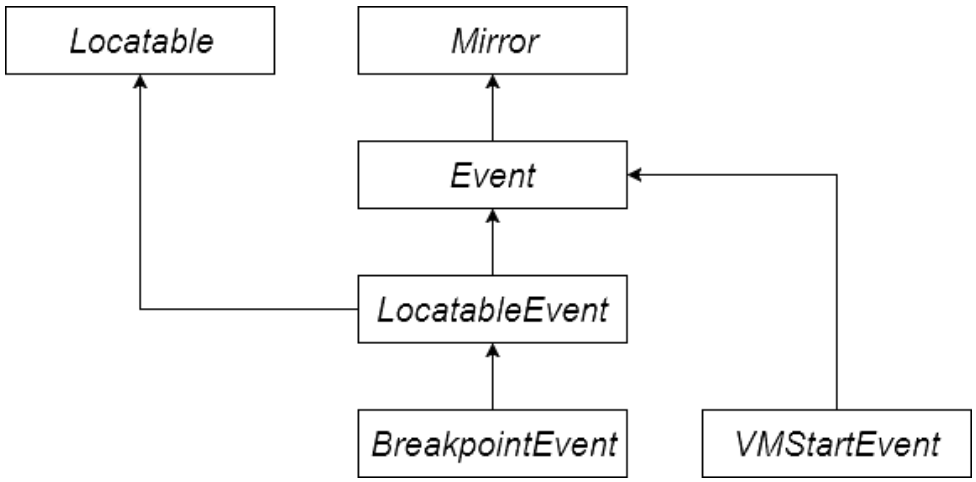

Figure 10. Typical web-like structure of JDI objects

Navigating the JDI to extract functionality can be a challenge unto itself but there are many advantages offered by the JDI structure. One such advantage is a natural solution to the problem of timing, a core challenge when attempting to smoothly ensure two virtual machines interact; the target virtual machine has to be commanded by the debugger, which itself has to be able to respond appropriately in time before the target program continues onwards in execution and runs the risk of altering the values that the debugger needs to read before the debugger is able to extract the correct value. This control of the targeted virtual machine is key to correctly extracting or setting the values correctly in any debug implementation, which rely entirely on the interrogation of targeted fields at a specific point of execution; if those fields

are altered or removed from memory before the debugger can access them, the problems are quickly evident and the debugger becomes unreliable. The JDI provides an innate solution to this problem via use of its *EventRequest* system.

6.2 The Event-Response system within the JDI

Syncing both virtual machines together to allow correct communication between each is something of a challenge, one fixed inherently by the nature of the JDI's implementation. The JDI execution is structured as a strict Request-Event, or 'call and response', system of communication between both virtual machines. The front-end debugger program creates *EventRequest* objects via the *EventRequestManager* – a single instance of which is contained within each *VirtualMachine* that is connected – which dispatches these requests to the debuggee and awaits the requested operation to occur within that virtual machine. Upon the satisfaction of one of these Requests, the target machine generates a corresponding *Event*, stores it alongside any simultaneous Events in an *EventSet*, and places it upon the queue (the *EventQueue* object for each VirtualMachine object). This structure is illustrated in Figure 11.

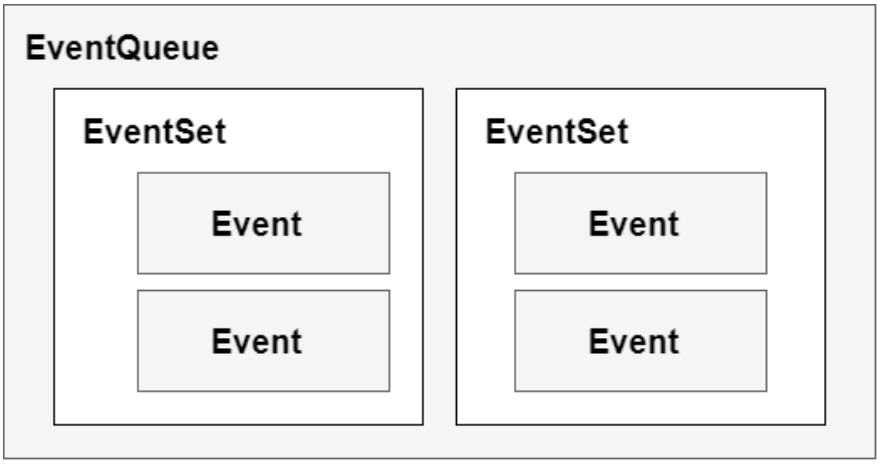

Figure 11. Packaging of Event objects within the JDI

The full list of possible EventRequests within the manager largely defines the functionality of the debugger itself, and includes a suite of potential options related to common debugger options as examined in Chapter 2.2:

- notification of a given class being loaded in the target Virtual Machine
- setting breakpoints within a given class
- watchpoints on a given Variable field, to respond to either access or modification of that field's value
- request notification if an Exception occurs in the debuggee

An important distinction is to separate out the internal Events within both virtual machines, such as those used in typical UI programming, with the JDI-implemented Events from the EventQueue. These events are only generated in response to a JDI Request being satisfied and are designed to be responded to in a separate virtual machine to the one that generates them. In the case of a front-end JDI implementation, the debugger will respond to these events as they are sent from the target; typically, this response will be implemented as either a request for specific information, such as the values of a named field, or the generation of additional EventRequests (for instance, the setting of a given Breakpoint only after the Class had been loaded and initialised within the target program).

The key to implementing this system, as a series of linked Request-Event-Request objects, is to ensure that each request is made only after the location to set each Request is loaded within the target machine. There are several key Events that can be utilised to ensure that the location has been loaded and a new request can be made, by requesting notification and pausing target execution upon the start of a virtual machine or when a class is prepared within the target VM. This system, in which the timing of the requests is required to be precise to ensure the location can be located when the request is created, is demonstrated in Figure 12. As execution continues downwards, each Event leads to a new Request using that loaded location, until finally the variable is extracted during a BreakpointEvent:

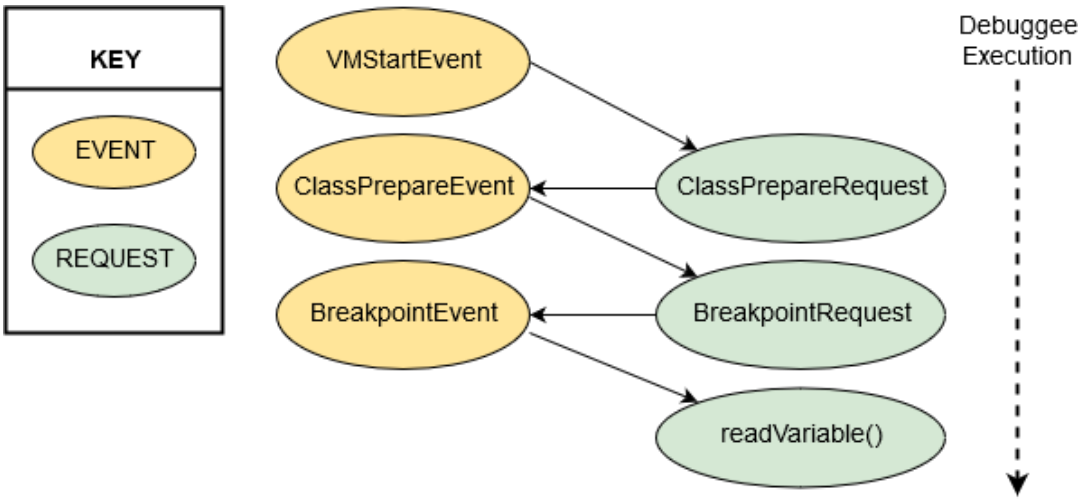

Figure 12. Path of execution through the JDI Request-Event system at runtime

The control over runtime execution of the target virtual machine is tied into the suspend policy, set individually during the EventRequest constructor, which indicates when execution should be paused as the requests are satisfied and the Events are generated. In all instances within our specific implementation, however, the default policy of pausing the execution of all threads in the target whenever a requested alert was generated until manually restarted via the

front-end debugger program was maintained. This allowed the debugger all the time that it required to execute the queued commands for any given event, the number of which could not have been easily ascertained prior to execution due to the nature of the script read.

6.3 Event Handling within the Debugger

Working within the JDI event system requires a program to constantly iterate through the queue of incoming Events from the target machine, checking each as they occur and extracting those events that the program is interested in. In the case of the debugger, those events generated directly as a result of our EventRequests and those generated automatically by the JDI implementation, such as notification of:

- a target virtual machine's successful start and subsequent suspension
- the correct disconnection of the debuggee after execution is complete
- the death event generated if the target machine disengages without warning

A reduced portion of the code from the EventHandler debugger class is included below, which iterates through all incoming events while the connection between both virtual machines remains active. It then despatches each Event correctly to the appropriate method within the RequestHandler class, which contains methods intended to respond to specific Events.

```
public boolean isConnected = true;

public void trackEvents (VirtualMachine vm) {
      while (isConnected) {
               EventQueue eventQueue = vm.eventQueue();
               EventSet eventSet = eventQueue.remove();
               EventIterator iterate=eventSet.eventIterator();

               while (iterate.hasNext()) {
                        handleEvents (iterate.nextEvent(), eventSet, vm);
               }
     }
}

private void handleEvents(Event iEvent, EventSet es, VirtualMachine vm) {
      if (iEvent instanceof VMStartEvent) {
               isConnected = true;
               RequestHandler.createClassRequests (vm, iEvent);
               es.resume();
      }
      if (iEvent instanceof ClassPrepareEvent) {
               RequstHandler.setBreakpoints(vm, iEvent);
               es.resume();
      }
      if (iEvent instanceof BreakpointEvent) {
               RequstHandler.checkVariables(iEvent, vm);
               es.resume();
      }
      if (iteratedEvent instanceof VMDisconnectEvent) {
               isConnected = false;
               es.resume();     }
}
```

The *trackEvents* loop continues for as long as the connection remains active, indicated through via *isConnected* global Boolean value; throughout the program, any event that disconnects the target virtual machine or responds to the same disconnection will set the value to false and disable the constantly iterating loop. This loop is responsible for checking the EventQueue, unpacking individual events and then calling the *handleEvents* method to despatch the Event to the correct response. This code covers several of the major events in order, from the initial *VMStartEvent*, through the *ClassPrepareEvent*, which creates the associated breakpoints within each class as it is loaded, and onto the Breakpoints themselves. Finally, it responds to any notification of the disconnection of the target by ensuring the "isConnected" Boolean is disabled.

While Chapter 9 will discuss the internal storage solutions in greater detail, some elements of the JDI allow us to utilise EventRequest objects themselves to assist with dealing a very specific problem: as the debugger is designed to function from a script, which is read in at the start of the debugger's execution, much of the information contained within the file is linked together but will be utilised at different points in the execution. For instance, a breakpoint will be set only after a class has been loaded and a ClassPrepareEvent triggered in response to a request; the program, however, must be able to tell which breakpoints from the scripted instructions pertain to which class. Similarly, the variables to be read out at a given breakpoint are only required to be known after that breakpoint has been reached but are specific to that breakpoint itself; in the case of variables, the problem is magnified as the same variable is often repeated across multiple breakpoints within the same class. This problem is unavoidable, as the locations to set breakpoints are not locatable until the class which contains them has been prepared by the target; as a demonstration, a snippet of code used to set the breakpoints is included below:

```
public void setBreakpoints (VirtualMachine vm, Event event) {
   ClassPrepareRequest request = event.request();
   ClassType classLocation = event.referenceType();
   String name = request.getProperty("classname");
   EventRequestManager rm = vm.eventRequestManager();

   while (requestedBreaks.iterator().hasNext()) {
      Map<String,ArrayList<String>> map = iterator.next();

      if (map.containsKey(name)) {
           int location = map.get(name);
           EventRequest breakpoint = null;
           List<Location> loc =
                     classLocation.locationsOfLine(location);
           Location lineloc = loc.get(0);
           breakpoint = rm.createBreakpointRequest(lineloc);
           breakpoint.enable();
   }
}
```

Here, the debugger is reacting to a previously requested ClassPrepareEvent being satisfied within the target virtual machine. It has been iterated through the EventHandler class and despatched accordingly, so the debugger can be assured as to the exact nature of the Event.

This example highlights two of the features the JDI objects offer to assist with the previously discussed problems. The first is a direct link between a generated JDI Event object and the EventRequest that caused it to be generated; within each event is included a reference to the request that can be extracted via the `Event.request()` method. The second advantage offered by the JDI is the ability, within each EventRequest, to store some data in the form of a key/value Map [35]. This functionality is built into the EventRequest object itself – set and accessed via the use of included methods – and is utilised by the debugger to store related information such as the class to which a breakpoint is to attached. In the above example, the debug program is able to extract the name of the class from the request, store it within the "*name*" variable, and use an iterator to compare with the global List of requested Breakpoints ('*requestedBreaks'*) for any matching keys. When matches are found, the line number for the breakpoint is extracted from the List and the matching Location object is found within the class. A similar approach is used when a BreakpointEvent is reached: the request that generated the Event is extracted and previously stored information related to which variables must be dealt with at that specific point is extracted from it, embedded within the request as a Map of a key value and a List of the requested variables.

7 JDI Communications

7.1 JDI Connectors

A core necessity for a functioning debug implementation is naturally the ability to connect to the debuggee. Any implementation requires that the front-end debugger be able to connect to and send/receive information from the back-end target, constantly and consistently, throughout the execution of both programs. The JPDA implements this connection via the Wire Protocol, a "command and response" packet-based communications protocol based upon the Transmission Control Protocol (TCP) [36] used in internet communications. Within the JDI, the wire protocol connection is abstracted into a *Connector*; a unique interface with several subinterfaces that specify the nature of the connection. There are two varieties of communication for the connection to utilise – each a two-way transmission between a client and a "server" – abstracted to a *Transport* object; each of the Connector objects within a *VirtualMachineManager* encapsulates a single Transport, communicating between debugger and debuggee across a single connection. This connection can either be Socket-based – which can function either locally on the same machine or across different machines via the internet – or can function using a shared memory location across both virtual machines, each reading Wire Protocol data packets from this apportioned memory.

Through the selection of a specific Connector implementation, the JDI allows for either debugger or debuggee to function as the server – defined as the program which initiates the connection – while the client program is the virtual machine which accepts the connection. The Connectors are available as three options:

- a *LaunchingConnector*, which locates a compiled Java program and launches it from the .class file
- a *AttachingConnector*, which connects to an already running virtual machine and takes control over its execution
- a *ListeningConnector*, which instead listens for an incoming connection from a target machine while still functioning as the server that implements the connection

From an early stage of development, the scripted debugger was designed to implement the LaunchingConnector. This offers a few significant advantages. Firstly, due to the nature of the scripted instructions, which require significant internal setup before the connection is opened, the LaunchingConnector allows the debugger to only begin the connection and target execution after all setup is complete at runtime; specifically, after the script has been read in and the

instructions parsed correctly, to allow the debugger to be immediately ready to configure the target according to those instructions. Secondly, one of the original specifications mandated that no editing of the target code be required for the debugger to function; this ruled out the ListeningConnector as a useful option, while the LaunchingConnector allows the debugger to function as a server with relative simplicity. The simplicity of implementing the LaunchingConnector as an option was also useful, with clearly defined initial arguments which included the option to immediately suspend the target virtual machine at launch. Crucially, the LaunchingConnector also allows us to implement communications without having to consider the actual nature of the connection itself; the LaunchingConnector does not offer a Socket or Shared Memory option externally, instead automatically functioning as one of these two options based on the currently executing Operating System.

7.2 Implementing the LaunchingConnector

With the LaunchingConnector selected as the most applicable approach, implementing the connection into the program was the first developmental priority. The first milestone for implementation was a program that could successfully launch a targeted program to be debugged, connect and retain control over its execution, and give notification of any disconnection from the target machine when it occurred. An initial LaunchingConnector was implemented using the default arguments provided by the JDI, obtained from the connector itself as a Map using the `defaultArguments()` method (illustrated in the code below). The default arguments themselves are well documented by Oracle and allow the debugger programs some control over the nature of the connection and the control it can exert upon the target virtual machine after it has been launched.

```
private VirtualMachine connectVM() {

 LaunchingConnector vmConn = vmManager.defaultConnector();
   Map<String,Connector.Argument> argumentmap =
                vmConn.defaultArguments();
   if (argumentmap.containsKey("main")) {
        Connector.Argument location = argumentmap.get("main");
        location.setValue(
               "-classpath E:\\ITNP099\\Test ProgName");   }
   else {
        diagnostic("no Main argument found");
   }
 if (argumentmap.containsKey("suspend) {
        Connector.Argument pauseVM =
                 argumentmap.get("suspend");
        pauseVM.setValue("true");
 }
 else {
        diagnostic("no Suspend argument found");
 }
 return vmConn.launch(argumentmap);     }
```

While the whole table of accepted Arguments for the Connector is reproduced below (Figure 13), there are two that are most applicable to the launcher for the scripted debugger itself: the mandatory *'main'* argument, which specifies the file location and name of the main .class file to be launched, and the *'suspend'* argument, which determines if the target is launched correctly while executing the main method or is paused immediately upon initialisation. These arguments can be extracted as a Map<String,Connector.Argument> via the name, as seen above, then overridden using the `setValue(String`. Most of these arguments were left as default values (Figure 13) while the 'main' argument was overridden to point towards the correct target program. The 'suspend' policy was also tested and set manually by the debugger to ensure functionality, as it is of particular importance to the debugger's correct execution; the default value was thus overridden to ensure it remained true.

name	required?	default value	description
`home`	no	current `java.home` property value	Location of the Java 2 Runtime Environment used to invoke the Target VM.
`options`	no	""	Options, in addition to the standard debugging options, with which to invoke the VM.
`main`	yes	""	The debugged application's main class and command line arguments.
`suspend`	no	true	True if the target VM is to be suspended immediately before the main class is loaded; false otherwise.
`quote`	yes	"\""	The character used to combine space-delimited text on the command line.
`vmexec`	yes	"java"	The VM launcher executable. This can be changed to javaw or to java_g for debugging, if that launcher is available.

Figure 13. Table of Arguments accepted by the LaunchingConnector object[1]

This leads to a relatively simple implementation of the Connector itself; the debugger creates the LaunchingConnector, imports the default argument Map for that Connector and overrides those arguments which pertain to the specifics of its requirements. The sample code above was taken from the *JVMComms* class and demonstrates the method responsible for

[1] http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/conninv.html

actually launching the connection. After the method is executed, during which the target machine is launched and its execution suspended, the JDI generates a specific Event automatically: the VMStartEvent, indicating the successful launch of a targeted virtual machine. This method then returns the VirtualMachine reference, containing the newly opened connection as an object, which the debugger sends to the *EventHandler* class to begin iterating through the EventQueue for that VirtualMachine.

8 Script Design and Use

8.1 Creating the Script

The decision was made early on to create the script using XML as a base. Many of the advantages were immediately evident to this approach: XML's widespread adoption by the industry means that many potential users would be familiar with the format, either directly from previous XML documents or through a derivative such as HTML. Alongside this, the Java language itself contains several integrated options for the parsing of XML files without having to implement an entirely new parser for a custom script. Other advantages are innate to XML, such as the relative human-readable nature of the mark-up [37] and the ability to restrict the script to match the functionality of the parser through the use of a Document Type Definition (DTD) file alongside the script itself. Using XML as a base allowed the script design process to progress quickly and was easy to expand as the functionality of the actual debugger was fleshed out, with additional elements or attributes to enable the script to define the new options.

Initially, the script was separated into four primary elements that would be required:

- the root element, simply a 'script' tag, not utilised specifically in the parser to encode any instructions
- a core 'class' element, which acts as the first element that the parser utilises and which contains all additional instructions, nested within the class to be loaded
- 'break' elements, linked to the parent 'class' node, which specify the line number to pause execution at
- a 'variable' element, containing the specifics of the information to be retrieved at each breakpoint

As functionality expanded, these four elements remained the core of the script, with only an additional 'watch' element being added to enable the specification of watchpoints on a given field; this option is nested at the same level of the script as the existing break element enclosed within a class tag. Other options within the debugger were added to existing elements, as either optional or required attributes, such as the option to set the value of a specified variable as oppose to simply reading the value at during suspended execution. Keeping the elements restricted to only five options and expanding options through the attributes allowed the script to remain easy to work with for the user; initial designs for the user interface included a suite of options designed to create the script itself but this approach

quickly proved time consuming in both implementation and in usage for the programmer. For even a moderately sized script, the UI proved clunky and difficult to use efficiently; individually creating each element within the interface and writing it to a script file. As a result, it became increasingly important to maintain readability for the user to work directly with the script file and to provide appropriate documentation to enable them to learn it easily.

To this end, the script was created with a documentation file to flesh out the details in an easily referenced format. A sample script template to work from and a strict separate DTD file to ensure the script complied with the parser were also included. The DTD is included below:

```
<!ELEMENT script (class+) EMPTY #REQUIRED>

<!ELEMENT class (break|watch)+>
<!ATTLIST class name CDATA #REQUIRED>

<!ELEMENT break (variable+)>
<!ATTLIST break line CDATA #REQUIRED>

<!ELEMENT watch (variable+)>
<!ATTLIST watch variable CDATA #REQUIRED>
<!ATTLIST watch action (access|modify) #REQUIRED>

<!ELEMENT variable>
<!ATTLIST variable name CDATA #REQUIRED>
<!ATTLIST variable type (int|String|boolean|Float|Double|char) #REQUIRED>
<!ATTLIST variable action (GET|SET) "GET">
<!ATTLIST variable value CDATA #IMPLIED>
```

As defined above, the final script is relatively simplistic in implementation. The empty root script element is required, as the parser is designed to skip this element to locate the classes to be requested. This element requires a minimum of one ‘class’ tag to be included within it. This child element must include at least one of either a breakpoint or a watchpoint, and requires a mandatory ‘name’ attribute to define the class itself.

At the next nested level of the script, the options for a ‘break’ element at a specified line of code or a watchpoint on a given class variable or field are required to be included. Both elements require at least one variable element within it to act upon at runtime when the point is reached. The break element, which is simple to define within the script, requires only a line number to give the targeted location. The watch element requires two attributes to locate it: the variable to be observed – by name – and a given action to be watched for on that variable, which can only be either Access (anytime the value is checked by the target) or Modification (when the value is altered). Finally, at the lowest level, the variables to be located at each break are defined by a mandatory name and type, which can currently only be one of the five specified types the debugger can support. Finally, the script defines the action that should be

taken upon that variable – either GET or SET the value – with a default value of GET if neither is specified within a user script to read out the value rather than setting it at runtime. Alongside this option, the script allows a value for the variable to be specified that will be used if the option to set the value is selected.

8.2 Parsing the Script

The Java libraries for parsing an XML file offer a variety of different options:

- the DOM (Document Object Model) [38] parser, which reads in the entire file and creates a structured tree like model of the elements and nodes within it
- the SAX parser, which reads the file in portions based on event triggers within the program itself
- several alternative options largely derived from the first two, such as the JDOM and JSAX parsers [39]

The SAX parser reads the file linearly based on internal triggers and is designed primarily for documents that are not deeply nested [40]; for the script, where knowledge of the file structure is required, such as which class each breakpoint is nested within, to correctly parse it was not a suitable choice. As such, the debugger utilises the simpler DOM parser, constructing a model of the scripted instructions in order to understand the exact structure of the instructions to convert these to appropriate instructions for the JDI.

The DOM parser allows the script to function without any duplication of data within, able to easily check the parent class for a breakpoint or to work through all of the children of a given 'class' node to link each with the correct ClassPrepareRequest. Initial code simply creates the Document model: locating the file to read from and preparing a new instance of a DocumentBuilderFactory, then using the factory in turn to create the DocumentBuilder and then the Document itself, utilising the File previously located. Finally, the root element (the empty "script" element) is extracted from the file to enable the debugger to begin moving through the document model.

With the object model loaded in and fully constructed, the debugger can begin to parse through the instructions. One initial problem became very apparent during implementation of the script parser was the nature of the JDI EventRequest constructors; specifically, many of the EventRequest objects require a valid *Location* to be created, extracted from the target virtual machine. This requires that they be created only after the class in question has been prepared by the target. It was thus important that all the information contained within the scripted

instruction set be extracted and stored until the program required them, at runtime, to translate them into JDI-compatible requests. These relationships are illustrated below in Figure 14, showing the potential links between nodes of the script and the problem of variable duplications across these instructions:

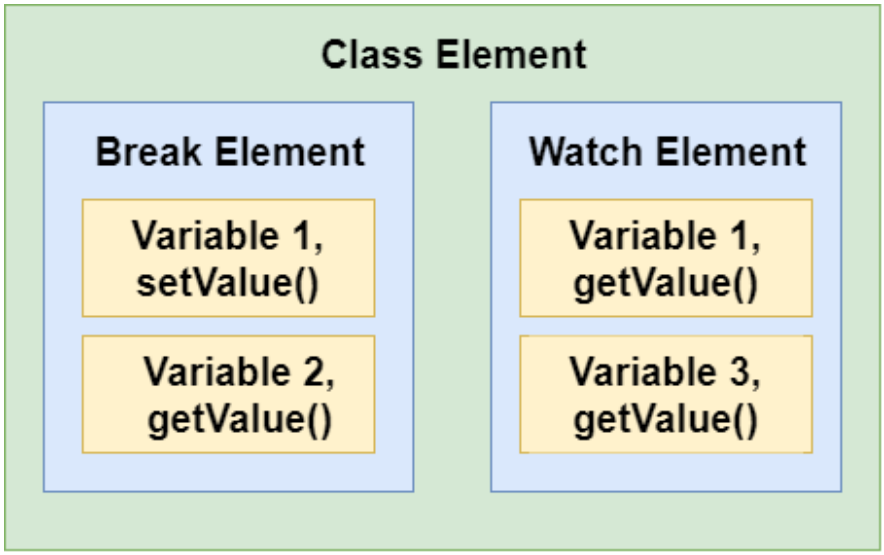

Figure 14. Nested nature of the script structure

While the exact storage solution utilised within the debugger will be discussed in detail in Chapter 9, this problem also required a specific implementation of the parser. The solution was a series of nested loops, iterating through the script elements to maintain the links between nested nodes; specifically, the parser iterates through the class objects individually, loading the class name, then iterates sequentially through any associated break or watch nodes to link them to the current 'class'. This pattern is repeated in a third nested loop, iterating over each associated 'variable' elements as the breakpoint is parsed and storing them such that they are identified by link to the break or watch element within which they were nested in the script. Once all nested elements have been parsed, the parser then moves on to the next class element and repeats the algorithm for any nested elements within it. This allows the debugger to retain all relational information from the DOM for each element, such that the information is not duplicated in the script itself to maintain simplicity and readability, by loading all the child nodes for each element within the existing loop for the parent:

```
while (classnode != null) {
   if (classnode instanceof Element) {
       String className = classElement.getAttribute("name");
       Node breakNode = classElement.getFirstChild();

       while (breakNode != null) {
         if (breakNode instanceof Element) {
             // Extract Break Information and store internally
```

```
                Node varNode = breakNode.getFirstChild();

                while (varNode != null) {
                    if (varNode instanceof Element) {
                        // Extract Variable details and store
                        varNode = varNode.getNextSibling();
                            }
                    else {
                        varNode = varNode.getNextSibling();
                    }
                }
                breakNode = breakNode.getNextSibling();
            }
            else {
                breakNode = breakNode.getNextSibling();
            }
        }
        classnode = classnode.getNextSibling();
    }
    else {
         classnode = classnode.getNextSibling();
    }
}
```

The final parser, included above with the dense code specific to storing the instructions internally in memory removed, functions as a series of nested while loops and if statements, linked directly to the nested nature of the script itself. Each 'level' of the parser code begins with a top-level loop, which continues iterating until the next extracted node from the script at that level is null. As an example, the first parsed level of the script is the "class" element. The corresponding while loop will continue iterating until no new class elements are available in the DOM at that level of the tree. This while loops contains an embedded "if" statement, which queries if the extracted node is an instance of an actual Element; this avoids any unwanted text nodes such as spacing characters or new lines included for human readability. This definition allows the user to insert whatever additional notes or personal notations into the script for their own use, while ensuring that anything not a strict XML element will be ignored by the debugger at runtime. If the extracted node is not an instance of an Element, the program iterates immediately to the next. Otherwise, the debugger extracts and stores pertinent details from the element and transitions to the next 'level' loop. In the case of the class level, this loop iterates through all children of the class Element, discarding text with a new 'if' statement to search for allowable 'break' and 'watch' Elements. Finally, the variables are read from the last level of the DOM, again by filtering the nodes and reading only those nodes that are children of the currently iterated breakpoint.

9 Internal Storage at Runtime

9.1 Working with the Script

While the creation of a debugger and working within the JDI caused many challenges already discussed, it was the unique approach of attempting to create a debugger that functioned through a scripted set of instructions that provided the biggest challenge of development. While the general parsing of the script itself proved to be an initial problem, the wealth of information extracted from the script at this stage required that it be stored and made available at multiple points of execution. The nature of the JDI Event system often required nested instructions to be executed only after specific events have occurred within the debuggee; alongside the need to maintain the links between nested instructions, it was also necessary to store significant amounts of additional information for each request to use.

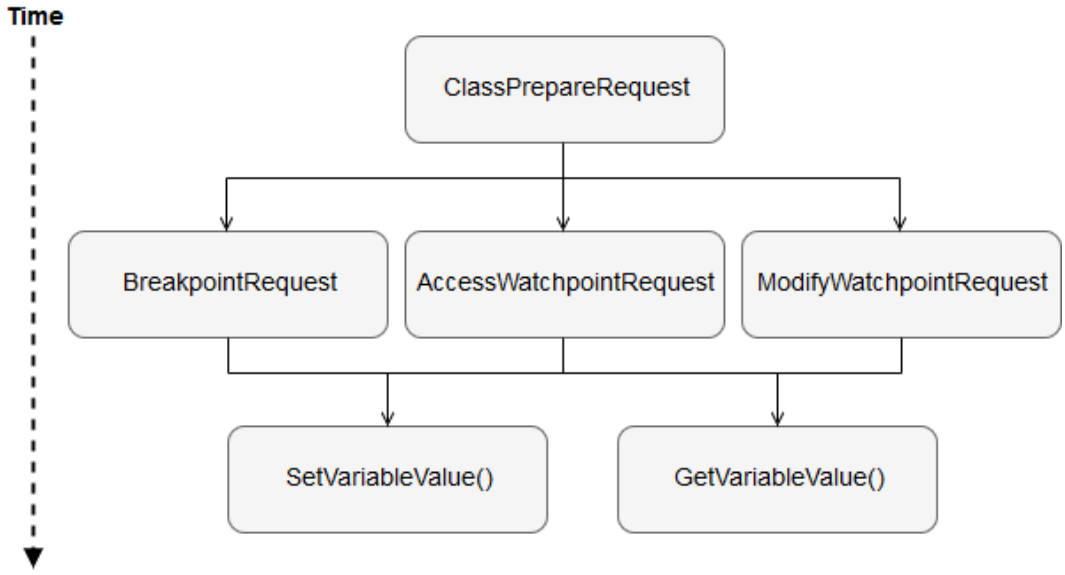

Figure 15. General flow of instruction execution within the Debugger

Both the structure of the script and the debugger's execution function as three 'levels' of instructions, visualised in Figure 15. The nested structure of the script largely follows the same pattern as the execution of the instructions: a *ClassEvent* leads to associated Breakpoints and Watchpoints being set within the target class, which in turn leads to the debugger acting upon the values of internal variables. As a result, instructions must be processed twice: initially read in from the script and stored in internal memory, then translated from this stored memory into JDI EventRequest objects. This requires a specific design to allow the debugger to transition between these different sets of stored instructions without errors or loss of data, particularly in regards to the relationships between individual instructions.

Two possible solutions were considered at this phase. The first was to utilise the DOM created by the parser to move between nodes at each Event, locating Elements relevant to each break in execution and analysing the structure surrounding each to locate the information when each Request is created. With this method, the debugger would constantly return to the *ScriptReader* class to request information on the instruction being, extracting parent and child nodes, to allow the debugger to be designed to move through the DOM tree to link the parsed instructions together at each point of suspension. While useful, this approach was discarded in favour of importing the instructions internally at an earlier stage before discarding the object model entirely. One advantage of this approach over using the DOM is the ability to retain information internally after each instruction to utilise in generating the final report.

9.2 Storing the Instructions

Much of the ScriptReader class within the debugger is devoted to this storage solution. To this end, a network of storage solutions is used internally within the debugger to contain the parsed instruction sets; a networked combination of ArrayLists [41] and HashMaps [42]. The HashMaps were utilised when the debugger required a searchable key value, such as situations where it was required to easily match an object to a specific occurrence. ArrayLists provided a flexible data structure to contain lists of Objects to iterate over, such as holding all requested *VariableRequests* after parsing or to contain HashMap objects used to map requested breakpoints to their supplementary information. The primary advantage of the ArrayList over Arrays for storage is the variable nature of the script itself; it is difficult to know beforehand how many instructions will be included, preventing a simple initialisation of a fixed size Array. The ArrayLists offer flexibility, varying according to the specific requirements of the script without requiring any workarounds such as empty 'null' values that an Array might contain, to slow down searches. Additionally, they offer several convenience options such as the innate `iterator()` method. The final structure of the storage is contained within the ScriptReader class and later accessed by the RequestHandler to translate those instructions into JDI objects:

Name	Object	Holds
classList	ArrayList<String>	String value: class name
reqBreaks	ArrayList<Map>	Map<String"class",ArrayList<Strings>
reqVariables	ArrayList<VariableRequest>	Constructed VariableRequest objects

Figure 16. Global internal storage solutions for Debugger

```
public ArrayList<String> classList = new ArrayList<String>();
public ArrayList<Map<String,ArrayList<String>>> reqBreaks =
            new ArrayList<Map<String,ArrayList<String>>>();
public ArrayList<VariableRequest> reqVariables =
            new ArrayList<VariableRequest>();
```

The specifics of the storage begin at the 'class' level, which is identified simply by the name of a given Class as a String. The debugger utilises the '*classList*' to globally store all parsed class nodes for iteration, as a simple List of String values. The challenges of efficient storage, however, become more complicated as it transitions to the second level of the parser. For simple iteration, a single ArrayList (*'requestedBreaks'*) holds both 'break' and 'watch' elements; each individual element is stored as a HashMap, with a key value containing the name of the parent class mapped to an ArrayList, holding additional information. This allows the debugger to iterate over each stored Map at a *ClassPrepareEvent* trigger, to locate those instances where the key value matches the current ClassPrepareEvent before extracting the rest of the information.

A second ArrayList is stored within the Map, containing all additional information pertaining to that Element. At index 0 within this List, the type of the node is stored, either a breakpoint or watchpoint. The specific position allows the debugger to easily extract this information and create the correct request, which also indicates the ongoing structure of the List: for a Breakpoint, this information is simply a line location within the target class. For a Watchpoint, the List contains a location (the name of a Field to be observed) and the action to be watched for (Access or Modification). This information is all extracted at runtime and translated into a JDI-compliant EventRequest, with additional information stored within the Request object as a property:

```
String name = ClassPrepareRequest.getProperty("classname");
while (requestedBreaks.iterator().hasNext()) {
    Map<String,ArrayList>> map = reqBreaks.iterator().next();

    if (map.containsKey(name)) {
           ArrayList<String> breakInfo = map.get(name);
           String breakType = breakInfo.get(0);
           EventRequest bp = null;

           if (breakType.equals("break")) {
               location = breakInfo.get(1);
               List<Location> loc =
                         class.locationsOfLine(location));
               // create Breakpoint request
           }

           else if (breakType.equals("watch")) {
               location = breakInfo.get(1);
               String action = breakInfo.get(2);
```

```
                Field field = classLocation.fieldByName(location);

                if (action.equals("modify")) {
                    // create modify watchpoint
                }
                else if (action.equals("access")) {
                    // create access watchpoint
                 }
            }
            bp.enable();
        }

       List<VariableRequest> variables = new ArrayList();
       breakpoint.putProperty(new String("variables"), variables);

        while (requestedVariables.iterator().hasNext()) {
            VariableRequest var = reqVariables.iterator().next();
            String cName = var.getClassName();
            String breakLoc = var.getLocation();

            if(cName.equals(name)&&breakLoc.equals(location)){
                variables.add(var);    }
        }
    }
```

While the code here is somewhat dense, the actual functionality is relatively simple. After a ClassPrepareEvent has triggered and the original request extracted, the class' name is extracted from the Request's property as a String. Utilising the global *'requestedBreaks'* List, the debugger iterates over all stored break-level instructions; each Map is inspected to locate matching keys to the current ClassPrepareEvent. If they are the same, the ArrayList stored within the Map is extracted and the first entry – Index 0 – is read. This String, containing the Element's 'type' from the script, is then used to correctly create either a breakpoint or a watchpoint. The location of the breakpoint is located from the *ClassType* using the `locationsOfLine(int)` method, which returns a List of Location objects matching the line number. When the breakpoint is created, the Location is taken from the List at index 0.

Following this level of instruction, the global *'requestedVariables'* List is iterated through to match existing VariableRequests to the specific break; initially, a new List is created to hold those variables local to the Breakpoint or Watchpoint request and stored within their innate property Map. The debugger fetches the stored class and location values from each iterated VariableRequest object (Appendix 1) via its public methods and comparing them to those values already held in the iterating loops. If both class and location match, that object is added to the localised List held within a BreakpointRequests property to link the instructions together.

9.3 The VariableRequest object

The storage of the parsed script related to variables to read or set required a particular solution. A variable element, extracted from the script, must be identified by a large amount of information; the same variable, by name and type, might be requested at multiple different breaks in execution. As a result the variable element is uniquely identified by:

- by the name of the class that contains given break
- by either a breakpoint's line value or a watchpoint's observed Field by name
- by the variable itself, through both name and declared type

Alongside these identifiers, the options embedded within the script file require that a variable be stored alongside the action to be taken at execution (either fetching the value of the variable from the target machine or setting it to a given value at runtime) and the value of the variable either to be set in the latter case.

This breadth of information required a more robust solution than the previous combinations of Maps and Lists. The JDI has no innate Variable object; unlike a BreakpointRequest object, there is no native support to request a variable be read from the target. This required that all functionality following the Event that suspended target execution be implemented manually in the debugger. Initially, this mandated the creation of an internal object to store each variable request parsed from the script – a VariableRequest (Appendix 1) – to be stored in a global List and extracted at the appropriate point. Each object stores a single instance of a variable instruction and all information required to identify it, a Boolean value indicating if the value is to be set at runtime or fetched, and the variable's value. This value is either set at script read or is set after the breakpoint to store the extracted value; to facilitate this, the VariableRequest also includes two overloaded constructors. One is used for a 'get variable' request that sets the location values only, while the second includes a given Value for a 'set' action. Publically accessible methods are included for fetching the stored values, checking the Boolean value and storing a new value are.

These objects are used after a breakpoint is triggered in a similar fashion to previous functionality: the request is extracted from the triggered Event, before the stored List of VariableRequests is taken from the EventRequest property. A simple iterator loops through the extracted List and the name, type and value variables are extracted from each object. This information is used to act upon the variable itself. The reduced code for this is included below and contains several features of interest. First, that the JDI treats instance variables separately from those found temporarily on the stack [43]; each type must be extracted from the debugee

differently. The debugger initially searches the class for a Field, by name (using an extracted JDI object, *ReferenceType*) which indicates if the sought variable belongs to either the class or to a method. If the variable was found as a Field, the variable is either fetched from the *ClassType* (a subinterface of ReferenceType) via `getValue()` or set via the `setValue()` method. If not, the JDI mirror *StackFrame* is extracted from the Event's *ThreadReference* and used to search for a *LocalVariable* on the stack by name.

```
public void checkVariables (Event event, VirtualMachine vm) {

    EventRequest request = event.request();
    List<VariableRequest> variables =
                 request.getProperty("variables");

    while (variables.iterator().hasNext()) {
        VariableRequest var = variables.iterator().next();
        String name = var.getVarName();
        String type = var.getVarType();
        Value varVal = null;
        String value = var.getValue();

        ReferenceType ref = event.location().declaringType();
        Field field = ref.fieldByName(varname);

        if (field != null) {
             if (var.getAction()) {
                     ClassType classType = (ClassType) ref;
                     varVal = setValue(type,value,vm);
                     classType.setValue(field, varval);
             }
        else {
             varVal = ref.getValue(field);
        }
     else {
        ThreadReference thread = event.thread();
        StackFrame sFrame = thread.frame(0);
        LocalVariable local = sFrame.visibleVariableByName(name);

        if (var.getAction()) {
                 varVal = setValue(type,value,vm);
                 stackFrame.setValue(localVar, varval);
        }
        else {
                 varVal = stackFrame.getValue(localVar);   }
        }
     if (varVal == null){
             value = "null. Variable not yet initialised.";
     }
     else {
             value = parseValue(varVal,type);
     }
     var.setValue(value);
     report("Value of " + type + " " + name + " is: " + value);
    }
}
```

10 Evaluation

10.1 Critical Assessment:

In general, implementation of the debugger went relatively smoothly. While the JDI documentation was dense and the interface-based structure often difficult to follow to a concrete implementation, it provided a wealth of options and working functionality that could be added into the debugger. Much of this is wrapped up in the various EventRequests, the full list of which is located in the documentation for the *EventRequestManager*. Many of these were successfully implemented into both the script file and the debugger to expand the debugger's functionality; only the ExceptionRequest was implemented into the debugger and left out of the script, by design. It is believed at this point that the project offers a strong set of core functions and would be genuinely useful in executing simple white box tests in the background of general program execution. The strengths of the current program include:

- Simple to utilise and unobtrusive to normal execution of the target program.
- Current iteration provides consistent and targeted results, as original specified.
- Scripts themselves kept simple and easy to work with for the user, with a straightforward learning curve.
- Script system offers flexibility in executing tests; they are easy to edit, simple to run repeatedly with modifications to either the code or the script itself for repeat testing, and modular in design to allow separate tests to be combined or separated easily.
- Execution is efficient, with automation of debugging events and little user interaction with the debugger itself during runtime.

Early attempts to work with the JDI EventRequest objects included several attempts to transition directly from the script parser into the JDI-compliant requests. It became clear during the implementation of this system, following several attempts to work with null Location values and force the objects to construct without an extracted target location, that the JDI structure itself did not allow this workaround and that the EventRequest objects could only be constructed after runtime had begun and the correct class loaded. This caused significant problems with the original design of the system, forcing the creation of a *RequestHandler* class to manage the translation of scripted instructions into JDI instructions at the appropriate time. As a result of this turn of events, no work was performed at the design and early implementation phases towards the system of internal storage used to facilitate this transition; when this system became necessary, significant delays were incurred late into the project itself in the design and successful implementation of this workaround.

As a result of this delay, several unutilised options exist within the JDI EventRequest structure and remain unimplemented largely due to time constraints as the project came to a conclusion; optional functionality could have been added to request notification of entry and exit into a specified method and to allow a variety of Step requests to skip portions of the target execution. While neither was deemed critical to core debugger functionality, the Skip command in particular was originally intended to be implemented as a common feature of industry-recognised debuggers and, given more time, would have expanded functionality significantly for the script and linked debugger. The general structure of the program has proven relatively robust and additional functionality would have been easy to implement into the existing Script/Request/Event program structure; as it stands currently, however, the gaps in JDI functionality remain a notable limitation of the software.

Efforts were also made throughout the project to maintain readability and usability of the script structure, which proved a relatively successful endeavour throughout personal testing and into the user testing and feedback stage. However, initial objectives included a far more fleshed out user interface than is present in the currently implemented program. Despite efforts towards providing functionality related to script design and modification within the UI itself, it proved difficult to implement an efficient solution for the management of the script files within the program. Currently, the script is created and edited separately in a text editor of the user's choice; this method is functionally fine but at odds with the intention of creating a singular script-based program. Early designs for the user interface included a dedicated 'script' tab, which would have displayed the script file, checked it for any syntactic errors and allowed a save/load operation. This proved more difficult to implement successfully than originally assumed and planned for, however, requiring that these plans be abandoned in the later stages of implementation in favour of encouraging the user to work simply with the raw text file. This failing is particularly noteworthy as it severely restricts the usefulness of the program's interface, due largely to a simple lapse in scheduling and a severe underestimation of the time it would take to implement such a system.

Finally, one of the original core objectives remains only semi-realised. The project was intended to allow the debugger to run automatically and function as a piece of test software that could be left to run without user interaction, ideally in isolation and without attendance. Currently, the debugging functionality was successfully implemented in this fashion – specifically, that the programmer does not need to manually restart execution after a breakpoint or cycle through individual lines of code via "Step Over" commands akin to a traditional IDE debugger – and the debugger's virtual machine executes correctly without interaction with the programmer after the script itself is created. There is currently no way to

allow the target virtual machine to ignore user interaction, however; if execution requires events triggered from interface objects such as buttons to progress the function, the user is still required to manually input this. While this does allow the programmer to focus entirely on the target program and let the debugger execute entirely as a background process, it does not yet allow them to entirely leave the debugger to iterate over a large piece of software while they carry on with other tasks. As a result, this particular objective cannot be deemed to have been met in the current iteration.

10.2 User Assessment:

A sample set of six potential users were asked to test the program towards the final stage, after implementation of the user interface and all functionality. Of note, I was personally acquainted with those who participated in the testing phase, which may have influenced their feedback on the project. All participants have had their individual identities removed from this report. Participants had a background in computer science, either professionally or academically, and has programmed at least a few times in the past; this was deemed in all cases to be enough to be familiar with the essentials of software development and basic debugging. Two participants work professionally in software development, though only one professionally uses the Java language.

Each participant was given a sample program as a compiled Java class file – a heavily edited version of a basic "HelloWorld" containing two classes – and the completed debugger program as an executable JAR file with all accompanying documentation. They were also given a feedback and testing sheet (Appendix B) containing three tasks to complete, essential instructions to complete them with and a quick questionnaire to complete afterwards. Finally, all participants were given a consent form to sign to participate and an explanation of the requirements of the test.

All users were able to complete the three tasks. Only one participant had any trouble, while working on Task 3, who felt that while the Watchpoint option was clearly explained the documentation was not sufficiently clear on how to work with additional classes. This was later amended in response to this feedback, by adding a second class to the template script file. One other user initially reported several errors, which appeared to be related to a problem with the script file not reading in correctly; this problem could not be reproduced and was theorised to be a problem with Operating System permissions on the file itself. The user was able to complete the tasks correctly on another machine but the errors were recorded for future

reference and research. Following the tests themselves, users were asked four questions to gather feedback on their time with the debugger:

1. Did you find the script file and structure easy to work with? If not, why?

Users generally responded positively to this question, indicating that they found the script to be simple to use and that the documentation included covered all that they required. As mentioned, one participant felt that the class element was not sufficiently explained. A second felt that the script documentation was confusingly worded and unnecessarily dense; efforts were made following this feedback to reduce the length of this document to only essential guidance.

2. Was the UI clearly laid out? Did it provide all the functionality you required?

The consensus across all replies was that the UI was sufficient and cleanly organised. No-one reported any errors with the offered functionality in execution or any confusion or problems while using it. Several users did note the barebones nature of the current design, however; one user praised it as being perfect for the program's requirements, being easy to set up and leave to execute, while two others noted that they felt it fell short of all the options they would have liked. Both mentioned disliking the need to open up another program to edit or preview the script file and one participant criticised the lack of feedback on the actual script itself from the parser. Another requested the ability to save the generated report to a file for later use, without having to use the workaround of copying the text into another document.

3. Were any errors in operation encountered during the completion of the tasks?

Other than the discussed error related to permissions on the script file itself, most users reported only one problem: that the program seemed to have several problems with repeat executions without refreshing the debugger program itself. Specifically, that running a debug test and then running another without closing the program caused problems with duplicated requests in the report, reports that several variables could not be located in the target machine and, eventually, lead to an exception in the debugger. This bug was not encountered prior to this user test but was easily reproduced afterwards. It has been marked as a priority fix but, at this stage, remains an issue.

4. Would you use this program to help debug your code?

Most users reported that they would use the program to assist them, though several mentioned only in tandem with existing debuggers as they felt it was not yet robust enough to be used in isolation. One user with a professional background was particularly enthusiastic, however, at its potential to aid in the testing phase; he indicated that he would use the program again to aid with this and highlighted the unobtrusive nature of it as a particular attraction. One participant indicated that in its current state, he could not see its applicability over existing debuggers and would likely not use it; he did, however, mention a caveat that better handling of the report file in the future would likely change this for repeat testing purposes and indicated his general satisfaction with its current performance.

Generally, the feedback was positive: 4 out of the 6 participants indicated that they would use the program to help debug code in the future, with a fifth individual indicating that they would use it with some additions. While generally content with this feedback, which was fair and balanced, several points were noted to influence future development and improvement of the software. In particular, the feedback from two users who work in a professional development environment was considered; both were generally positive of the project and offered significant suggestions to improve its potential.

11 Conclusion

11.1 Summary

At conclusion of the project, the scripted debugger is a functioning piece of testing software with a reasonable suite of functionality derived from more developed IDE programs. It offers a simplistic but fully functioning user interface, which includes:

- a file browser to locate the necessary supplementary files from the OS directories
- textual display of the file locations, which can also be manually filled in by the user
- a fully editable and interactive report area, with a scrollbar that activates only if needed
- a formatted final report that reports: a log of events from the debuggee's execution, results from all requested results from the script and the linked break event, notification of the final disconnection of the target machine, and any execution that occurred within the target alongside a note of their status (caught or uncaught) and source location

The script itself offers options that support all debugger functionality, alongside a Document Text Definition file and supporting documentation to assist the user in creating and editing these files. It functions smoothly alongside the debugger program itself, which correctly parses the script and stores all included instructions in the internal memory, before launching the specified target program in a separate, connected virtual machine automatically and beginning to execute the instructions upon it. The debugger executes these instructions autonomously, without any user input or editing of the source code being required, while allowing the debuggee to execute as normal in a separate window for the programmer to interact with. Internally, the debugger halts all target execution at the appropriate point to setup further instruction requests or to execute user requests for information before automatically resuming the target machine, adding all requested information to the final report for the programmer to compare with expected results. After the execution of the target is completed, the debugger displays the results in the UI and allows the user to repeat the test again, either with the same parameters after an identified error it corrected (for use in regression testing) or with slightly different instructions via the editing of the script file itself.

The debugger and the script file together offer a range of possible functionality, heavily influenced by the review of comparable software in section 2.2. In its current iteration, the scripted debugger offers the options to:

- react to the loading of any given class, specified by name, by pausing execution
- set a breakpoint by line number in a class to pause execution and act on specified variables
- set a watchpoint on a given instance variable in a class, by variable name, and specify if it reacts to either internal read access to that field or upon modification of the variable's value
- at each requested pause, specify the exact variables to have their values acted on by name and type; the debugger will only report on these specified values
- set an action for each of the above variable requests: values can either be extracted and reported at this point in the target's execution or be internally overridden with a given value
- report, without specifically being requested, on any Exceptions thrown internally thrown by the target machine: these are reported as either being caught or going uncaught by the target, alongside the location that it was thrown from and the location that caught it (if it was caught at all)

The class design is clearly structured and defined, allowing for easily implemented additions to the functionality in future development. From the script, through the script parser and the event handler, to the final RequestHandler class that executes the instructions themselves at runtime, each stage of execution can be added to with relative ease to expand functionality in the future.

11.2 Future Work

Much of the potential future work involves expanding the core options of the debugger itself to include more of the EventRequest options available in the JDI. These include a variety of "Step" commands to skip parts of the target code entirely at the programmer's request this would enable the debugger to step into ActionPerformed methods and run entirely autonomously without requiring a user to be present at all for the tests, or to bypass large sections of code and test only a small portion of the program in isolation. Similarly, JDI options to trigger breaks in execution on entry or exit to a given Method are planned in tandem with the Step commands, to allow the programmer more freedom in how they locate their variable requests within their code.

While these are the two specific EventRequest objects intended to be implemented, additional functionality derived from the IDE debuggers in section 2.2 include specifying several variations of a "hitcount" for a given watchpoint or breakpoint; this would allow a user to

specify when the breakpoint was triggered using a given *n* numerical value. The count would cause the break to trigger either:

a. report only once after *n* occurrences of that break and not again
b. begin reporting only after *n* previous occurrences of the break continually
c. report only the first *n* occurrences of the event, then stop

This functionality is partly supported by the JDI structure, with each EventRequest including an `addCountFilter()` method to assist us in implementing options a and b. Option c is not enabled by the JDI and would require this functionality to be coded directly into the debugger and store a counter as a property in the object manually. Along similar lines, options to include a conditional trigger on watchpoints could be included; this option would only report the watchpoint back to the programmer if the value of the targeted field did not match a given expected value.

In tandem with additional options in the debugger and script themselves, the user interface implementation has several features that were originally intended. The first is a full script manager tab within the UI, enabling the user to load a script and visualise it; this tab would include option to add skeleton elements to the document, to directly edit the script in a UI-embedded editor, and to check the script for any errors in syntax before the program is launched. This would also include the option to save the edited script file, either in a new document to create alternate tests or to overwrite the loaded one with the updated instructions.

Finally, the top priority for future development is to implement a fix for the bug discovered during the user testing phase, in section 6.2, which causes errors when the debugger is run repeatedly without refreshing the program itself and an eventual Exception within the debugger itself to trigger. As of this point, the exact cause of this bug is not known; at least in part, errors are being caused by the internal storage solution not being cleared between sessions, causing the scripted instructions to simply be added to the storage each time and the instructions to duplicate.

References

[1] L. Vogel, *Eclispe IDE,* 3rd Edition. Hamburg: Vogella, 2013.

[2] P. Jiantao. (1999, April). *Software Testing [Online]*. Available: https://users.ece.cmu.edu/~koopman/des_s99/sw_testing/ [Accessed: 2- Sept- 2017]

[3] Oracle Corporation. (2016). *Java Platform Debugger Architecture [Online]*. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/ [Accessed: 12- June- 2017]

[4] M. Kölling, J. Rosenberg. *BlueJ Homepage [Online]*. Available: https://www.bluej.org/

[5] Eclipse Foundation (2017). *Eclipse IDE for Java Developers [Online]*. Available: https://eclipse.org/downloads/packages/eclipse-ide-java-developers/oxygenr

[6] M. Rouse. (2016, October). *Definition: debugging [Online]*. Available: searchsoftwarequality.techtarget.com/definition/debugging [Accessed: 27- August- 2017]

[7] H. Vliet, *Software Engineering: Principles & Practices*. Sussex:Wiley Publishing, 2000.

[8] B. Littlewood, *Software Reliability: Achievement and Assessment*. London: Blackwell Publications, 1987.

[9] Techopedia.com. (2017) *Debugging [Online]*. Available: https://www.techopedia.com/definition/16373/debugging [Accessed: 27- August- 2017]

[10] ISTQB. (2016) *What is Software Testing? [Online]*. Available: http://istqbexamcertification.com/what-is-software-testing/ [Accessed: 23- August-2017]

[11] P.K.J Mohaptra, *Software Engineering (A Lifecycle Approach)*. Kharagpur: New Age International, 2010.

[12] University of Minnesota, *Debugging and Testing [Online]*. Available: https://www.d.umn.edu/~gshute/softeng/testing.html [Accessed: 16- July- 2017]

[13] SoftwareTestingFundamentals.com. (2017). *White Box Testing Fundamentals [Online]*. Available: http://softwaretestingfundamentals.com/white-box-testing/ [Accessed: 25- August- 2017]

[14] C. Kamer, J. Falk, H.Q. Nguyen, *Testing Computer Software,* 2nd Edition. New York: Wiley Publishing, 1999.

[15] C. Harel. (2013, Dec). *The Definitive List – 7 Java Debuggers You Should Know [Online]* Available: https://dzone.com/articles/definitive-list-7-java [Accessed: 24- June- 2017]

[16] GitHub. (2016).*Top IDE Index [Online]*. Available: http://pypl.github.io/IDE.html [Accessed: 5- August- 2017]

[17] Oracle Corporation. (2017). *Netbeans IDE Features [Online]*. Available: https://netbeans.org/features/index.html [Accessed: 1- July- 2017]

[18] JetBrains. (2017, August). *Discover IntelliJ IDEA [Online]*. Available: www.jetbrains.com/help/idea/discover-intellij-idea.html [Accessed: 1- July- 2017]

[19] Oracle Corporation. (2016). *Jdb – The Java Debugger [Online]*. Available: docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb [Accessed: 1- July- 2017]

[20] Chronon Systems. (2012). *Chronon, a DVR for Java. [Online]*. Available: http://chrononsystems.com/ [Accessed: 3- July- 2017]

[21] N. Neeluru. (2010, June). *Debugging Java Programs using JDB [Online]*. Available: www.packtpub.com/books/content/debugging-java-programs-using-jdb [Accessed: 6- July- 2017]

[22] S. Oaks, H. Wong, *Java Threads*, 2nd Edition: O'Reilly Publishing, 1999.

[23] T. Lindholm, F. Yellin, *The Java Virtual Machine Specification*. Boston: Addison-Wesley Professional, 1997.

[24] JetBrains. (2017, August). *Creating Field Watchpoints[Online]*. Available: www.jetbrains.com/help/idea/creating-field-watchpoints.html [Accessed: 24- July- 2017]

[25] JetBrains. (2017, August). *Debugging[Online]*. Available: www.jetbrains.com/help/idea/debugging.html#d196890e37 [Accessed: 24- July- 2017]

[26] Oracle Corporation. (2017). *Using the Visual Debugger in NetBeans IDE [Online]*. Available: netbeans.org/kb/docs/java/debug-visual.html [Accessed: 25- August- 2017]

[27] Oracle Corporation. (2017). *NetBeans IDE Features: Debugger and Profiler [Online]*. Available: netbeans.org/features/java/debugger.html [Accessed: 25- August- 2017]

[28] Chronon Systems. (2012). *What is Chronon? [Online]*. Available: http://chrononsystems.com/what-is-chronon/technology [Accessed:28- August- 2017]

[29] Chronon Systems. (2012). *Chronon Time Travelling Deugger [Online]*. Available: http://chrononsystems.com/products/chronon-time-travelling-debugger [Accessed:28- August- 2017]

[30] JSwat (2013). *JSwat: a Java Debugger [Online]*. Available: https://github.com/nlfiedler/jswat [Accessed: 6- August- 2017]

[31] Oracle Corporation. (2016). *Java Debug Wire Protocol [Online]*. Available: docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdwp-spec.html [Accessed: 5- June- 2017]

[32] Oracle Corporation. (2016). *JVM Tool interface [Online]*. Available: docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html [Accessed: 6- June- 2017]

[33] Oracle Corporation. (2016). *JDI All Classes [Online]*. Available: https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/allclasses-frame.html [Accessed: 3- June- 2017]

[34] Oracle Corporation. (2016). *JPDA Structure Overview [Online]*. Available: docs.oracle.com/javase/8/docs/technotes/guides/jpda/architecture.html [Accessed: 3- June- 2017]

[35] Oracle Corporation. (2016). *Interface Map<K,V> [Online]*. Available: https://docs.oracle.com/javase/7/docs/api/java/util/Map.html [Accessed: 13- July- 2017]

[36] Cisco.com. (2005, August 10). *TCP/IP Overview [Online]*. Available: www.cisco.com/c/en/us/support/docs/ip/routing-information-protocol-rip/13769-5.html [Accessed: 6- September- 2017]

[37] L. Vogel. (2017, July 25). *Java and XML – Tutorial [Online]*. Available: http://www.vogella.com/tutorials/JavaXML/article.html [Accessed: 21- July- 2017]

[38] TutorialsPoint.com (2017). *Java DOM Parser Overview [Online]*. Available: https://www.tutorialspoint.com/java_xml/java_dom_parser.htm [Accessed: 13-July-2017]

[39] TutorialsPoint.com (2017). *Java XML Parsers [Online]*. Available: www.tutorialspoint.com/java_xml/java_xml_parsers.htm [Accessed: 11-July-2017]

[40] TutorialsPoint.com (2017). *Java SAX Parser Overview [Online]*. Available: https://www.tutorialspoint.com/java_xml/java_sax_parser.htm [Accessed: 13-July-2017]

[41] Oracle Corporation. (2016). *ArrayList<E> [Online]*. Available: docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html [Accessed: 1- August- 2017]

[42] Oracle Corporation. (2016). *HashMap<K,V> [Online]*. Available: docs.oracle.com/javase/7/docs/api/java/util/HashMap.html [Accessed: 28- July- 2017]

[43] W. Adams. (2011, December 17). *Examining Variables in JDI [Online]*. Available: wayne-adams.blogspot.co.uk/2011/12/examining-variables-in-jdi.html [Accessed: 15- August- 2017]

Appendix 1: the VariableRequest Object

```
public class VariableRequest {

	private String className;
	private String bpLocation;
	private String name;
	private String type;
	private String value;
	private boolean set = false;

	// constructor for "get" type variable requests
	public VariableRequest (String vClass, String loc, String
vName, String vType) {

		className = vClass;
		bpLocation = loc;
		name = vName;
		type = vType;
	}

	// constructor for "set" type variable requests
	public VariableRequest (String vClass, String loc, String
vName, String vType, String vValue) {
		className = vClass;
		bpLocation = loc;
		name = vName;
		type = vType;
		value = vValue;
		set = true;
	}

	public String getClassName() {
		return className;
	}

	public String getLocation() {
		return bpLocation;
	}

	public String getVarName() {
		return name;
	}

	public String getVarType() {
		return type;
	}

	public void setValue(String val) {
		value = val;
	}

	public String getValue() {
		return value;
	}

	public boolean isSet() {
		return set;
	}
}
```

Appendix 2 – User Test Sheet

This user test presents you with a number of tasks to perform using the provided diagnostic program, designed to allow the debugging of Java applications without editing the original code. You will use the script file itself to schedule debugging tasks at runtime and request that it reports back on variable values at defined breakpoints. For each task, please read the description and then attempt to complete it. You may use any help facility the Debugger's documentation or the Java documentation provides. When you have completed the task – or if you find you cannot complete it – please answer the questions that follow the task description.

Task 1: Set a Breakpoint within HelloWorld

Create a script from the sample template provided that specifies a breakpoint within the provided HelloWorld sample program at line 12 and requests the value of the Integer variable "test2" to be reported back. Activate the target program to be debugged via the debugger software and execute the script to receive a report of the execution, which should include the value of test2 at the time of the breakpoint.

Feedback

1. Were you able to complete the task? Yes / No
2. If not, please describe how far you got and any error messages that were displayed.
3. On a scale of 1 to 5 where 1 is very difficult and 5 is very easy, how did you find the task?
4. Please indicate any problems with the interface or documentation that hindered your attempts to complete the given task.

Task 2: Set two Breakpoints within the program's execution

Open the script from Task 1 and edit it to include another breakpoint at line 16. This script should request a second report of integer "test2" at this stage and also report back on the value of the String variable "test1" at this breakpoint. Activate the target program via the debugger software and execute the script to receive a report of the execution, which should include the values of the variables at the time of the breakpoint and again at the time of the second breakpoint.

Feedback

1. Were you able to complete the task? Yes / No
2. If not, please describe how far you got and any error messages that were displayed.

3. On a scale of 1 to 5 where 1 is very difficult and 5 is very easy, how did you find the task?
4. Please indicate any problems with the interface or documentation that hindered your attempts to complete the given task.

Task 3: Add a Watchpoint to the HelloWorld script

Via the script created during Task 2, add an instruction that requests a Watchpoint on the String variable "Test3", which will activate when the program modifies this variable. This variable is located inside a second class within the program, named "Supplementary". Execute the script file on the target program, "HelloWorld2" and receive a report on the execution.

Feedback

1. Were you able to complete the task? Yes/No
2. If not, please describe how far you got and any error messages that were displayed.
3. On a scale of 1 to 5 where 1 is very difficult and 5 is very easy, how did you find the task?
4. Please indicate any problems with the interface or documentation that hindered your attempts to complete the given task.

Final Questionnaire:

- ✓ **Did you find the script file and structure itself easy to work with? If not, why?**

- ✓ **Was the UI clearly laid out? Did it provide the functionality you required?**

- ✓ **Were there any errors in operation you encountered during the completion of the tasks?**

- ✓ **Would you use this program to help debug your code? Please explain and critique here.**

Thank you for your time and feedback on this project. Your help is incredibly appreciated and your feedback will be treated with respect and in accordance with all Data Protection laws; you will not be individually identified in the final report and final data will be anonymised before submission.

Appendix 3 – Quick Installation guide

For future development the current program consists of six compiled .class files: JVMComms, which comprises the main class, and the following files: debugGUI, EventHandler, Request-Handler, ScriptReader, and VariableRequest. All files must be present to execute the program or an executable JAR compiled with contains them.

All files have a corresponding .java source file under the same name, which will be required for future editing of the functionality. This program **must** be compiled on a correct Java Development Kit runtime environment, as it makes heavy use of the *tools.jar* Java library file throughout.

To ensure correct functionality if executing the program, the debugger must be able to be directed from the user interface to a correctly defined Script file: this file can be located anywhere on the OS, but must include a reference to the included DTD file, *!xmldefine.txt,* to avoid potential errors with the script read at launch. See the included script template file, *!template.txt,* for a sample structure.

Files required for correct execution:

- JVMComms.class
- debugGUI.class
- EventHandler.class
- RequestHandler.class
- ScriptReader.class
- VariableRequest.class
- !xmldefine.txt
- userscript.txt