HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

Bachelor ARBEIT

Herr
Jasley Jens-Uwe Jangk

Entwicklung und Dokumentation eines Lua-Realtime-Debuggers für die Composer IDE der Anwendung WarehousePilot auf Basis der LuaJ Bibliothek.

Mittweida, 2022

Fakultät Angewandte Computer- und Biowissenschaften

Bachelor ARBEIT

Entwicklung und Dokumentation eines Lua-Realtime-Debuggers für die Composer IDE der Anwendung WarehousePilot auf Basis der LuaJ Bibliothek.

Autor:
Herr

Jasley Jens-Uwe Jangk

Studiengang:
Angewandte Informatik

Seminargruppe:
IF17wS-B

Erstprüfer:
Prof. Dr. rer. nat. habil. Thomas Haenselmann

Zweitprüfer:
Dipl.-Inform. Klaus Schapp

Einreichung:
Mittweida, 25.07.2022

Verteidigung/Bewertung:
Mittweida, 2022

Faculty Applied Computer Sciences &
Biosciences

BACHELOR THESIS

Development and documentation of a Lua-realtime-debugger for the IDE named Composer of the application WarehousePilot based on the LuaJ library.

author:
Mr.

Jasley Jens-Uwe Jangk

course of studies:
Applied Computer Science

seminar group:
IF17wS-B

first examiner:
Prof. Dr. rer. nat. habil. Thomas Haenselmann

second examiner:
Dipl.-Inform. Klaus Schapp

submission:
Mittweida, 25.07.2022

defence/ evaluation:
Mittweida, 2022

Bibliografische Beschreibung:

Nachname, Vorname: Jangk, Jasley Jens-Uwe

Entwicklung und Dokumentation eines Lua-Realtime-Debuggers für die Composer IDE der Anwendung WarehousePilot auf Basis der LuaJ Bibliothek.

51 Seiten, Hochschule Mittweida, Fakultät Angewandte Computer- und Biowissenschaften, Bachelorarbeit, 2022

Referat:

Ziel dieser Bachelorarbeit ist es ein Konzept für Lua Debugger zu entwickeln, welcher mit der Luaj Implementierung kompatibel ist. Darüber hinaus wird betrachtet wie ein Debugger aufgebaut ist und funktioniert.

Abstract:

The goal of this bachelor thesis is to develop a concept for Lua Debugger, which is compatible with the Luaj implementation. In addition, it is explained how a debugger is structured and how it functions.

Inhalt

Inhalt **I**

Abkürzungsverzeichnis ..... **III**

1 Einleitung **5**

1.1 Motivation *5*

1.2 Problemstellung *5*

1.3 Zielsetzung *6*

2 Das MFCS **7**

2.1 Was ist das MFCS *7*

2.2 Aufbau des MFCS *8*

2.3 Die Composer IDE *8*

3 Debugger **10**

3.1 Was ist ein Debugger *10*

3.2 Arten von Debuggern *10*

3.3 Aufbau und Funktion eines Debuggers *11*

4 Lua **15**

4.1 Was ist Lua *15*

4.2 Der Aufbau von Lua *16*

4.2.1 Ausführung von Lua Code und die virtuelle Maschine 16

4.2.2 Die wichtigste Datenstruktur in Lua der Table 16

4.2.3 Die Umgebung von Lua 17

4.2.4 Die Debug Library 17

4.3 Was ist Luaj *18*

5 Der eigene Debugger für LuaJ **19**

5.1 Ziel und Funktion des Debuggers *19*

5.2 Architektur und Planung des Debuggers *19*

5.2.1 Erstes Konzept 19

5.2.2 Zweites Konzept 20

5.2.3 Architektur und Aufbau 21

5.3 Implementierung des Debuggers 24

5.3.1 Herangehensweise der Implementierung 24

5.3.2 Herausforderungen bei der Implementierung 25

6 Die Remote Verbindung 26

6.1 Das Steuerungsprotokoll 26

6.2 Datenschutz und Sicherheit 27

7 Fazit und Ausblick 29

Literatur 30

Anlagen, Teil 1 Lua XXXII

Anlagen, Teil 2 Java XXXV

Selbstständigkeitserklärung 45

Abkürzungsverzeichnis

MFCS	Material Flow Control System
UUID	Universally Unique Identifier (Universeller Einzigartiger Bezeichner)
JSON	Java Script Objekt Notaion
RPC	Remote Procedure Call
VM	Virtuelle Maschine
CPU	Centeral Procesing Unit
API	Application Programming Interface

1 Einleitung

1.1 Motivation

Jeder Entwickler braucht es spätestens, wenn das geschriebene Programm nicht funktioniert und man sich auf die Fehlersuche begibt, um den Bug zu finden, kommt der Debugger zum Einsatz. Ein Softwaretool, um Fehler in Programmen zu finden, dass in der heutige moderne Softwarewelt nicht mehr wegzudenken ist. Der Debugger hat seinen Namen vom englischen Wort „Bug“, für Käfer oder Schädling, denn daraus geht die Legende hervor, dass eine Motte einen sehr alten Computer, den „Harvard Mark I“ störte, der noch mit Relais arbeitete. Nach dem Entfernen des Käfers, also dem Debugging, englisch für entwanzen, lief der Computer wieder reibungslos. Somit verbreitete sich dann der Begriff des Debuggings, für das Finden von Fehlern im Programmcode und so war auch später der Debugger geboren. [1]

Debugger spielen in der Entwicklung von Software eine sehr wichtige Rolle, zum einen können sie dabei helfen Fehler zu finden und diese dann auch zu verstehen. Auch kann ein Debugger genutzt werden, um Programmcode nachzuverfolgen, um somit ein besseres Verständnis von einem vorliegenden Programm zu erhalten. Weiter kann ein Debugger auch genutzt werden, um zu prüfen ob ein Programm korrekt funktioniert und die Abläufe so sind, wie diese definiert wurden. Sie stellen somit ein unerlässliches Werkzeug in der Entwicklung von Software dar und besitzen weit mehr Einsatzmöglichkeiten als nur das Finden von Fehlern im Programmcode.

1.2 Problemstellung

Ein Debugger ist für die effiziente Entwicklung von Software ein wertvolles und fast unersetzbares Werkzeug. Deswegen ist es notwendig dem Entwickler ein entsprechendes Tool zur Verfügung zu stellen. Da jedoch durch die geringe Bekanntheit des Luaj Frameworks, einer Reimplementierung von Lua 5.2 in Java, keine bereits vorgefertigten Lösungen für das Debuggen existieren und dazu noch der Debugger in ein bestehendes Softwaresystem intrigiert werden muss, ist die einzige Möglichkeit die Eigenentwicklung eines Debuggers der kompatibel mit Luaj ist.

1.3 Zielsetzung

Ziel der Arbeit ist es einen Debugger für die Programmiersprache Lua zu entwickeln, welcher kompatibel zum Framework Luaj ist. Der Debugger soll dabei als Remotedebugger ausgeführt sein, also über das Netzwerk steuerbar sein. Dafür soll der Debugger, eine einfache abgesicherte Schnittstelle über das Netzwerk zur Verfügung stellen. Der Debugger soll dabei über die wichtigsten Grundfunktionen verfügen, wie das Setzen von Breakpoints, das darauffolgende Anhalten im Programm und den Programmcode Zeile für Zeile auszuführen. Weiter soll der Debugger alle derzeitig aktuellen global und lokal Variablen zur Verfügung stellen.

Des Weiteren ist der Debugger noch in das MFCS (siehe Kapitel 2) einzubinden, um in dieser Anwendung als Remotedebugger für das hier eingesetzte Luaj zu fungieren. Die Steuerung des Debuggers soll dabei über die Composer IDE erfolgen, eine interne Entwicklungsumgebung der iFD GmbH. Weiter soll der Debugger an die Bedürfnisse der Entwickler angepasst werden.

2 Das MFCS

Die Abkürzung MFCS steht für Material Flow Control System übersetzt ins deutsche Materialfluss Kontrollsystem oder auch kurz als Anwendung „Warehouse Pilot“ genannt. Dies ist eine Entwicklung der iFD GmbH deren sitzt sich in Chemnitz befindet. Die iFD GmbH ist seit 1990 ein Softwareunternehmen, dass sich auf die Entwicklung von Software für den Lager und Intralogistik Bereich spezialisiert hat. [2]

2.1 Was ist das MFCS

Das MFCS ist ein Softwaresystem, dass in automatisierten Lagern oder Logistik Umfeld eingesetzt wird. Dabei übernimmt die Software die Logik der Transporte von Waren im Lager. So können beispielsweise Paletten von einem Lagerstandpunkt zu einem Abholplatz vollautomatisch mittels des MFCS transportiert werden und dabei werden auch optimale Routing Strategien durch das Lager berücksichtigt. Das System stellt somit das Gehirn eines Lagers dar, da es alle nötigen Warentarnsporte innerhalb eines Lagers verwaltet. Diese Systeme sind meist sehr komplex, da sie eine Vielzahl unterschiedlichster Schnittstellen zu verschiedensten Hardwaresystemen implementieren müssen. [2]

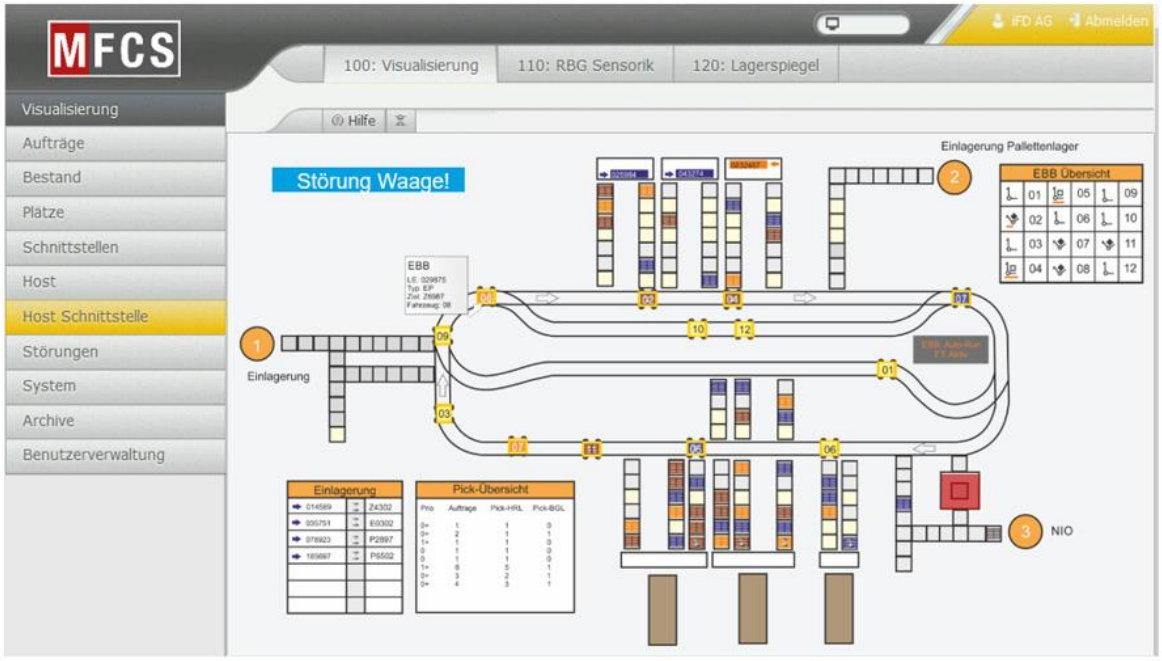

Abbildung 2.2-1: MFCS Oberfläche mit visualisierter Fördertechnikanlage

2.2 Aufbau des MFCS

Das MFCS stellt ein fertiges Softwaresystem dar und wie ein Baukastensystem aufgebaut, dass in einzelne funktionale Module gegliedert ist. So lässt sich das System für unterschiedliche Einsatzzwecke konfigurieren, indem benötigte Funktionen, also Module, hinzufügt oder entfernt werden. Dies ermöglicht ein hohes Maß an Individualität. Da nicht jede Fördertechnikanlage gleich ist, benötigt diese möglicherweise nicht alle Schnittstellen und Funktionen. So lässt sich das System einerseits verschlanken und einfacher konfigurieren.

Die Konfiguration des MFCS erfolgt dabei einerseits über eine sogenannten Projekt Datei, in dieser sind die Fördertechnik, also die physische Anlage im Lager, sowie viele andere konfigurierbare Elemente hinterlegt. Andererseits ist es möglich und meist notwendig das MFCS im Code anzupassen, indem neue spezifische Module nur für diesen Einsatzzweck hinzugefügt werden. Diese implementieren meist komplexere Logiken oder spezielle Abläufe für das Projekt, beziehungsweise individuelle Anforderungen des Kunden.

Weiter ist das MFCS in der Lage, kleine Logiken zu verarbeiten, die direkt in der Projekt Datei als Skript hinterlegt werden. Diese Logiken sind dabei in der Programmiersprache Lua verfasst und werden vom MFCS System ereignisbasiert ausgeführt.

2.3 Die Composer IDE

Der Composer ist eine integrierte Entwicklungsumgebung speziell für das MFCS. Die Anwendung wurde ebenfalls durch die iFD GmbH entwickelt und ist ein internes Entwicklungstool, um die Konfiguration des MFCS beziehungsweise der Projektdatei mit grafischen Mitteln zu vereinfachen. Im Composer lässt sich daher der Aufbau von Fördertechnikanlagen modellieren und abbilden, die im übrigen als Graph aufgefasst werden können. Weiter ist es auch möglich, einzelne Komponenten der Anlage gleich mit Logiken zur Verarbeitung von Daten oder Events zu versehen. Der Composer beinhaltet daher auch einen Lua Editor, um Skripte für die einzelnen Logiken zu verfassen und zu bearbeiten. Daher soll dieser auch die Steuerung des Remote Debuggers übernehmen sowie alle Daten anzeigen, die vom Remote Debugger gesendet werden. Dafür wurde der vorhandene Lua Editor um einige Funktionen und grafische Mittel erweitert.

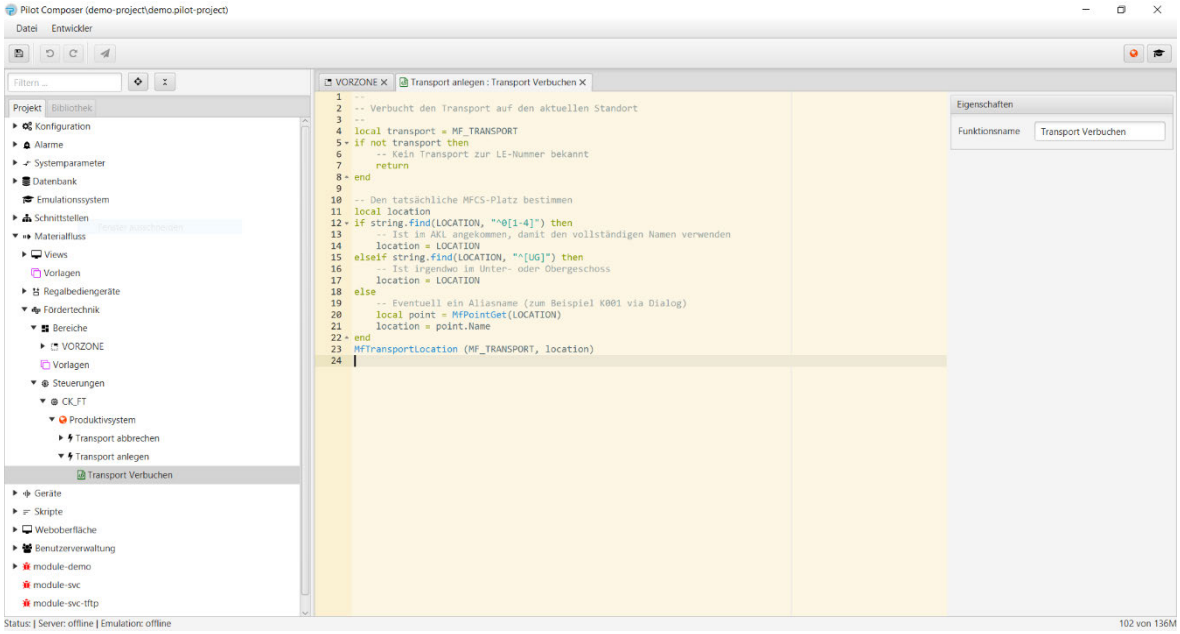

Abbildung 2.3-1: Composer IDE mit Lua Editor

3 Debugger

Debugger sind in der heutigen Softwarewelt teilweise zu hochkomplexen Anwendungen gewachsen, mit denen sich viel mehr machen lässt, als nur Zeile für Zeile Programmcode auszuführen. Daher wird dieses Kapitel den Nutzen, die verschiedenen Arten von Debuggern, sowie einen grundlegenden generischen Aufbau eines Debuggers beleuchten.

3.1 Was ist ein Debugger

Ein Debugger ist ein Softwaretool, das entweder als eigenständig Programm zur Verfügung steht oder in einer integrierten Entwicklungsumgebung für eine bestimmte Programmiersprache bereits vorhanden ist. Der Debugger ermöglicht es Programmcode zu Debuggen, also nach Fehlern (Bugs) im Programmcode zu suchen. [1]

Debugger besitzen meist mehrere Funktionen, wobei fast alle Debugger folgende Funktionen beinhalten. Das Setzen von Breakpoints, also das Anhalten des ausgeführten Programms an einer spezifischen Zeile im Code. Nachdem ein Breakpoint getroffen wurde, kann dann meist durch den Code, mittels einer Einzelschrittverarbeitung gegangen werden, die es ermöglicht, den Code des Programms Zeile für Zeile auszuführen. Eine Erweiterung davon ist das Überspringen von Funktionen oder Code-Bestandteilen. Dadurch wird ermöglicht Teile des Codes zu überspringen, ohne dass jeder einzelne Schritt einzeln ausgeführt werden muss. Diese Funktion stellen allerdings nicht alle Debugger bereit. Weiter stellen Debugger eigentlich immer alle Variablen eines Programms zur Laufzeit und somit ihren aktuellen Wert zur Inspektion bereit. Einige Debugger ermöglichen es, diese Variablen auch zur Laufzeit des Programmes zu bearbeiten. [1]

3.2 Arten von Debuggern

Es gibt viele verschiedene Arten von Debuggern, obwohl der Prozess des Debuggens meist gleich oder ähnlich ist, benötigt jede Sprache oder auch Anwendung einen eigenen speziellen Debugger. Daher werden hier nur die wichtigsten und meist genutzten aufgezählt sowie kurz beschrieben.

Hardware beziehungsweise Maschine Level Debugger sind Debugger die auf der untersten Ebene debuggen, also im direkten Kontakt mit der CPU stehen welche den Maschinecode ausführt. Diese Debugger werden beispielsweise bei Mikrocontrollern oder Assembler sprachen genutzt. Auch kann so ein Debugger für kompilierte Sprachen, wie C, genutzt werden um im Zusammenhang mit dem geschriebenen Code und dessen kompilierter Form Fehler zu finden. [1]

Source Level Debugger hingegen sind auf der Ebene der Programmiersprache und lassen den Benutzer glauben, dass hier der eigentliche Programmcode ausgeführt wird, so wie dieser definiert, also geschrieben, wurde. Dabei können diese Debugger entweder über einen unterliegenden Maschine Level Debugger verfügen, wobei dann ein Zusammenhang zwischen Source Level und kompilierter Form des Codes hergestellt wird. Eine andere Möglichkeit bieten Sprachen, welche in einer virtuellen Maschine, wie beispielsweise Java, ausgeführt werden. Diese Sprachen besitzen meist eine Debug API, welche auf die VM oder Sprachbestandteile zugreifen, dazu zählt auch Lua. [1]

Wobei sich Source Level Debugger nochmals leicht unterteilen lassen, in Top und Low Level. Ein Low Level Debugger nutzt die oben beschriebenen Mittel. Wohingegen ein Top Level Debugger, in Bezug auf Lua, auf die Sprache mittels der API aufgesetzt werden kann. Dieser greift somit nicht in die virtuelle Maschine aktiv ein, sondern nutzt Mittel, die von der Sprache bereitgestellt werden. Der in dieser Arbeit beschriebene Debugger stellt somit ein Top Level Debugger dar.

Die letzte hier vorgestellte Art von Debuggern ist der Betriebssystem Debugger. Diese sind speziell für das Debuggen von Betriebssystemen und deren Komponenten entworfen. Meist werden sie zum Debuggen von Hardware beziehungsweise, Gerätetreibern einsetzt. Aber auch um interne Sachen zum Beispiel am Kernel des Betriebssystems zu debuggen, können solche Debugger eingesetzt werden. Meistens sind Betriebssystems Debugger als Remote Debugger ausgeführt. [1]

Der Remote Debugger stellt einen Sonderfall dar, da dieser keine Art eines Debuggers darstellt, sondern eine spezielle Ausführung dessen ist. Somit können alle weiter oben genannten Debugger, ebenfalls als Remote Debugger ausgeführt werden. Die Besonderheit dieser Debugger besteht darin, dass diese aus der Entfernung, also Remote, genutzt werden können. Das heißt, entweder über ein Netzwerk von einer anderen Maschine aus sich steuern lassen, oder sich an eine Anwendung andocken können, die einen Debugger integriert und eine API zur Steuerung dessen bereitstellt. [1]

3.3 Aufbau und Funktion eines Debuggers

Debugger gibt es in vielen verschieden Arten (siehe 3.2) und Formen. Für die hier folgende Beschreibung der Architektur eines allgemeinen Debuggers mit grafischer Oberfläche, wird weder eine spezielle Art eines Debuggers noch eine bestimmte Sprache oder Betriebssystem angenommen. Denn grundlegend lässt sich festhalten, dass viele Debugger nach dem folgenden Schema konstruiert sind. Es besteht allerdings ein grundlegender Unterschied, für zu Maschinencode kompilierte Sprachen wie C und Interpretersprachen, wie Java die in einer virtuellen Maschine ausgeführt werden, in der Art und Weise wie der Debugger mit dem zu debuggenden Programm interagiert. Kompilierte Sprachen nutzen meist eine API des Betriebssystems zur Ablaufsteuerung des Programms. Während Sprachen, die in einer virtuellen Maschine ausgeführt werden, eine Debugger-API bereitstellen, diese Konzepte

der Debug-API sind allerdings nicht allgemein gültig, sondern es kommt auf die Art des Debuggers und der Programmiersprache an. [1]

Der Aufbau eines Debuggers wird vereinfacht als Zwiebelschalenmodell aufgefasst (siehe Abbildung 3.3-1). Ein Debugger lässt sich dabei in drei grundlegende Bestandteile aufteilen. Einmal die grafische Oberfläche, die Infos vom Debugger Kernel präsentiert, der Kernel bildet dabei den zweiten Teil und einer Debug-API, wobei diese nicht direkt Bestandteil des Debuggers ist, sondern von einer Sprache oder dem System bereitgestellt wird. [1]

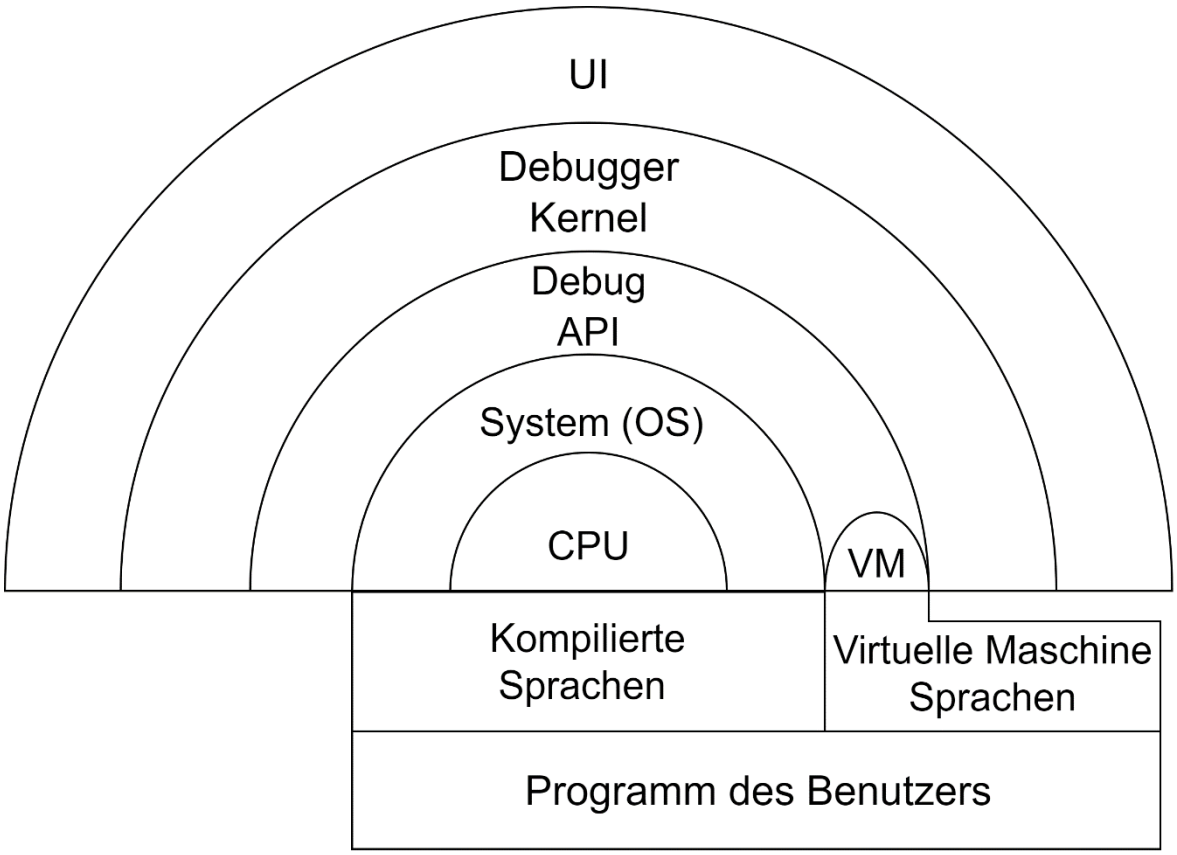

Abbildung 3.3-1: Vereinfachte Typische Debugger Architektur [1]

Mittels der grafischen Oberfläche wird der User informiert und kann den Debugger steuern. Dem User werden dabei Informationen zum Zustand der Ausführung des Programms angezeigt, sowie die Werte von Variablen und wie diese sich verändern. Ebenfalls wird auch der Quellcode angezeigt und wo sich das Programm in der Ausführung gerade befindet. Darüber hinaus kann eine Ansicht enthalten sein, die den Code in Assembly anzeigt, um so ein besseres Verständnis des kompilierten Maschinecodes zu erhalten. Weiter stellt der Debugger eine Stack View zur Verfügung in dieser wird angezeigt welche Funktionsaufrufe, in welcher Reihenfolge, zu diesem Punkt im Programm geführt haben. [1]

Die Steuerung des Debuggers in der grafischen Oberfläche kann über verschiedene Mittel erfolgen. So ist es im Debugger möglich Breakpoints zu setzen, also ein Punkt im Programm meist eine bestimmte Zeile, wo der Debugger dann anhält und die Ausführung des Programms vorläufig stoppt. Dabei wird dann der User darüber benachrichtigt, dass ein

Breakpoint erreicht wurde und es stehen weitere Mittel der Steuerung zur Verfügung, wie die Einzelschritt-Verarbeitung, um die Anweisungen des Programmcodes einzeln auszuführen. Sollten keine weiteren Interaktionen erfolgen, kann das Programm auch weiter fortgesetzt werden, indem die Ausführung wieder normal fortläuft. [1]

Der Debugger Kernel fasst mehrere einzelne Bestandteile ineinander und bildet Quasi das Gehirn eines Debuggers. Der Kernel übernimmt die Benachrichtigungen der grafischen Oberfläche sowie das Bereitstellten aller Informationen, die diese benötigt. Die Steuerung des Debuggers wird auch vom Kernel verwaltet. Dieser nimmt die Befehle der Oberfläche entgegen und übergibt diese der Prozesssteuerung, die für dir Steuerung des Programmprozesses zuständig ist. Weiter besitzt der Debugger Kernel Datenstrukturen, um Variablen abzuspeichern und zu verwalten, die dem Kernel bereitgestellt werden. Darüber hinaus besitzt der Kernel alle weiteren Funktionen, die für eine ordentliche Repräsentation aller Daten, des Sourcecodes, der Assembly Ansicht, der Stack View und der Bezugsherstellung zum Maschinecode benötigt werden. [1]

Das Herzstück eines jeden Debuggers bildet der Debugger Main Loop, ein Bestandteil des Debugger Kernels. Dieser übernimmt die Aufgabe der eigentlichen Steuerung des Debuggers sowie die Bereitstellung aller Daten, an die jeweiligen APIs des Debuggers. Der Loop wird dabei so gut wie immer für jede ausgeführte Instruktion des Programmecodes aufgerufen, um entweder Konditionen zu prüfen, wie das Erreichen eines Breakpoints oder bereitstellen der Daten für den Kernel. Der Loop ist daher eher kompakt gehalten und ruft dann die entsprechenden Funktionen im Kernel auf, die die Weiterverarbeitung übernehmen. [1]

Schlussbemerkung zu Debugger Architekturen

Die Vorangegangene Beschreibung der Architektur, lässt sich zwar auf viele Debugger anwenden, der letztendliche Aufbau kann dennoch davon abweichen. Besonders Moderne Debugger sind vielmehr zu abstrakten Anwendungen gewachsen, bei der sich die einzelnen Bestandteile, die auf die Debug-API aufbauen, also der Debugger Kernel, nur geringfügig für verschiedene Sprachen unterscheiden. Die einzelnen Ebenen sind so abstrakt konstruiert, dass eine Interkompatibilität mit mehreren Sprachen möglich wird, ohne jedes Mal einen komplett neuen Debugger entwickeln zu müssen. Jedoch ist die Anpassung an die jeweilige zu Debuggende Sprache unabdingbar. Da diese spezialisiert über die jeweilige Debug-API angesprochen werden muss. Grundsätzlich lässt sich festhalten, dass heutige Debugger dennoch nach diesem Schema aufgebaut sind, jedoch abstrakter.

Daher können heutige Debugger auch in einer zweigeteilten Architektur vereinfacht aufgefasst werden. Der Teil der speziell für die Sprache zugeschnitten ist und eine API zum ansprechen des generalisten Standard Teils des Debuggers bereitstellt. Eine Bespiel dafür ist der GDB GNU Debugger. Dieser implementiert einen Teil, der die gesamte Debug Verwaltung übernimmt, während ein anderer Teil sich um die Ausführung und Steuerung des Programms kümmert, also vereinfacht es das Ansprechen der jeweiligen Debug-API. Damit wird es möglich den Debugger relativ einfach auf andere Sprachen zu portieren. Zu bemerken ist jedoch, dass das Schema des GNU GDB hier sehr vereinfacht dargestellt wurde. [3]

4 Lua

Dieses Kapitel dient dazu, die Skriptsprache Lua zu beleuchten sowie auf deren Besonderheiten einzugehen und die Sprache in ihren Grundzügen zu erklären. Die hier dargelegten Informationen zu Lua beziehen sich auf die Version 5.0 aufwärts beziehungsweise explizit zur Version 5.2.

4.1 Was ist Lua

Lua ist portugiesisch und steht für Mond, dies zeigt sich auch im Logo der Sprache die einen Mond in seiner Umlaufbahn zeigt. Lua ist eine schnelle, leichtgewichtige Multiparadigmensprache, die sich vor allem durch ihre Einfachheit und geringe Größe auszeichnet. Die Sprache besteht nur aus 22 Key Wörtern und ist somit einfach zu erlernen. Lua wurde von Roberto Ierusalimschy, Luiz Henrique de Figueiredo, und Waldemar Celes 1993 am Department of Computer Science der Pontifical Catholic University of Rio de Janeiro in Brasilien entwickelt. Dabei wurde die Sprache erst als in-house Entwicklungstool im Labor Umfeld erschaffen. Dann aber schnell in der Industrie als eingebettete Sprache bekannt und erfreut sich auch heute noch großer Beliebtheit. [4]

Lua wurde schon direkt als Sprache entworfen, um in andere Sprachen eingebettet zu werden. Das zeigt sich vor allem auch in der geringen Größe der Sprache so findet „*Eine vollständige Distribution (Quellcode, Manual sowie einige Binärdatein für einige Plattformen) findet auf einer Floppydisk ausreichend Platz*“ [4] (ca. 1,3 Mb) Eine für Windows Kompiliertes Lua 5.2 (Interpreter, Libraries und Compiler) benötigt knapp weniger als 1Mb Speicher. Da Lua komplett in Standard ANSI (ISO) C geschrieben wurde, lässt es sich auch für nahezu jede Plattform kompilieren und ist daher sehr universell auf verschiedenster Hardware von leistungsstarken Großrechnern bis Mikrocontroller einsetzbar. [5]

Ein besonders Auszeichnungsmerkmal von Lua ist seine Möglichkeit der Erweiterbarkeit, so lässt sich die Sprache mittels Anpassung des Quellcodes um weitere Funktionalitäten ergänzen. Darüber hinaus besitzt Lua in ihrer Sprache einfache Mittel, mit denen sich sehr komplexe Konstrukte bauen lassen. So ist es auch möglich, obwohl Lua keine objektorientierte Sprache ist, Objekte mittels der zur Verfügung stehenden Table in Lua zu implementieren. [4] [5]

Leider ist Lua jedoch nicht frei von Fallstricken. So ist Lua keine typenfeste Sprache, was unter Umständen besonders bei der Einbettung der Sprache in andere zu wirklichen Problemen führen kann. Es kann dazu kommen, dass beim übertragen Werte den falschen Datentyp bekommen. Als Beispiel ist es möglich, dass im Lua ein numerischer String als wert

vorliegt der jedoch beim Übergeben dieses in eine andere Sprache als integer wert aufgefasst wird, was jedoch falsch wäre. [2]

Auch kann es sein, dass die Einfachheit von Lua, auch wenn es paradox klingt, den Programmierer überfordern kann, da einfache Funktionen fehlen, die in anderen Programmiersprachen als Standard angesehen werden, diese müssen erst selbst implementiert werden.

4.2 Der Aufbau von Lua

4.2.1 Ausführung von Lua Code und die virtuelle Maschine

Der grundlegende Aufbau von Lua bildet eine virtuelle Maschine, die wie eine CPU aufgefasst werden kann. Denn die virtuelle Maschine von Lua ist so aufgebaut, dass für Variablen Register deklariert werden, wo die Werte zur einfachen Manipulation abgelegt werden. Daher wird die virtuelle Maschine für Lua auch als Register basierte VM bezeichnet. Den Code, welchen wiederum die VM ausführt, sind Opcodes, die wie Maschinen Befehle aufgefasst werden können. Denn Lua Code wird zuerst zu Opcodes mittels eines Compilers kompiliert und dann von einem Interpreter, der die virtuelle Maschine darstellt, ausgeführt. Dies macht Lua zu einer Interpretierten Sprache, die vorher zu Bytecode für eine schnellere, initiale Ausführung und Platzersparnis kompiliert werden kann. [5]

4.2.2 Die wichtigste Datenstruktur in Lua der Table

Der Table stellt in Lua die einzige Datenstruktur für strukturierte Daten dar und ist als assoziative Arrays implementiert. Ein assoziatives Array kann dabei mit jeglichen Werten für den Schlüssel versehen werden (außer *nil* in Lua), auch können alle möglichen Werte als Wert gespeichert werden. Der Table ist somit äquivalent zu Listen in anderen Programmiersprachen anzusehen. [5] [4]

Weil es in Lua nur den Table gibt, übernimmt dieser auch gleichzeitig die Aufgabe von Arrays. Diese sind Tables, die nur Integer-Werte als Schlüssel besitzen. Eine Besonderheit ist, dass das Lua nicht wie andere Sprachen mit 0 im Array anfängt, sondern bei 1. [5] [4]

Da sich im Table verschiedenste Daten speichern lassen, wie auch andere Table und Pointer zu Funktionen, was wiederum sehr Nützlich sein kann, denn so lassen sich ganze Libraries in Tables Laden, was im Lua genauso gemacht wird. Dadurch, dass der Table eine sehr große Flexibilität bietet, macht ihn dies zu einer mächtigen Datenstruktur in Lua. [4]

4.2.3 Die Umgebung von Lua

Die Umgebung für ein Lua Skript stellt ein Lua Scope dar, also ein abgeschlossenen Code Block. Dies kann eine Funktion, eine Schleife, ein komplettes Skript oder ein selbst definierter *do end* Block sein. Innerhalb eines Scopes können lokale Variablen definiert werden, die nur für diesen Scope gültig sind. [4]

Weiter gibt es in Lua auch globale Variablen. Diese sind für ein Lua Skript definiert und über Scopes hinaus gültig sowie zugreifbar. Alle globalen Variablen werden dabei in einem Table abgelegt, wobei nicht nur Variablen, sondern auch alle zugreifbaren Libraries in diesem Table gespeichert werden. So lässt sich beispielsweise einstellen, auf welche externen Programm Teile zugegriffen werden kann. Durch die Verwendung eines Tables für die globalen Variablen macht es dies sehr einfach, diese zu bearbeiten und im Überblick zu behalten. Ein weiter Vorteil der Verwendung des Tables ist, dass Lua Scopes auch mit eigenen globalen Variablen ausgeführt werden können, falls diese erst speziell dafür initialisiert werden, ohne dass Veränderungen an den globalen Variablen des Skriptes vorgenommen werden. [4]

4.2.4 Die Debug Library

Die Debug Library in Lua stellt einem alle nötigen Funktionen zur Verfügung, um Lua selbst debuggen zu können. Die Library ist nicht sehr groß, aber dafür sehr handlich. Allerdings ist kein fertig benutzbarer Debugger oder ähnliches enthalten, sondern es muss mit den vorhandenen Mitteln ein eigenes Debug Tool entwickelt werden. Das hat den Vorteil, dass die Tools nach blieben mit den benötigten Funktionen gefüllt werden und unnötige Funktionen können so weglassen werden. Somit ist es möglich, sehr individuelle zugeschnitten Lösungen zu entwickeln. Der offensichtliche Nachteil, der sich daraus ergibt, ist, dass die Tools erst geplant und entwickelt werden müssen.

Für den Debugger stellt daher die Library die wichtigste Schnittstelle zu Lua dar und deshalb werden folgend die wichtigsten Funktionen, die der Debugger benötigt genauer erklärt. Allerdings nur für was die Funktionen nützlich sind und was diese ermöglichen. Wie diese zu verwenden sind, ist dem Lua 5.2 Reference Manual [6] zu entnehmen. Zu erwähnen ist noch, dass die Methoden, die von der Library bereits gestellt werden, meist eine weitere Logik benötigen, um sinnvoll eingesetzt zu werden.

Ein wichtiges Konzept der Debug Library das in vielen Funktionen der Library Anwendung findet ist die Verwendung eines sogenannten Levels. Da in Lua Funktionen auch ineinander geschachtelt werden können oder Funktionen andere immer weiter aufrufen können, ist das Level eine einfache Möglichkeit eine bestimmte Ausführungsebene zu adressieren. Dabei ist das Level 0 immer die derzeitige Funktion, die gerade ausgeführt wird. Level 1 ist dann die Funktion, die diese aufgerufen hat und immer so weiter. Wobei sich jedes Level auch als ein Abschnitt eines Lua Scopes auffassen lässt [4]

Mittels der Funktion *getinfo()* lässt sich ein Table mit nützlichen Informationen zum Skript selbst oder Funktionen abrufen. Dieser enthält den Namen des Skriptes, die derzeitige Codezeile des Aufrufs sowie für Funktionen, ab und bis welcher Zeile diese definiert wurden. Weiter enthält der Table für Funktionen noch, wie viele Parameter dieser übergeben werden können. Eine weitere nützliche Funktion der Methode, ist, sich alle ausführbaren Codezeilen in einem Table wiedergeben zu lassen. Dabei ist jedoch zu beachten, dass nur Codezeilen gezählt werden, die zum derzeitigen Scope, also Level, gehören. Also Zeilen innerhalb von definierten Funktionen fallen somit nicht darunter und werden nicht mitgezählt. Dafür aber die letzte Zeile der Definition der Funktion, also das *end* Statement am Schluss. [6]

Mit der Funktion *getlocal()* lassen sich alle lokale Variablen eines bestimmten Levels abrufen. Unter die lokalen Variablen fallen hierbei auf Funktionsparameter und temporäre Variablen. [6]

Die Funktion *sethook()* kann wiederum genutzt werden, um event basiert eine Funktion aufrufen zu lassen. Events, die dabei zur Verfügung stehen, sind das Aufrufen einer Funktion, das Zurückkehren von einer Funktion oder jedes Mal, wenn Lua eine Zeile Code ausgeführt hat und in eine neue Zeile springt. Besonders nützlich ist das Event für jede Zeile, da so einfach eine Funktion für die Einzelschritt-Verarbeitung von Lua Code geschaffen werden kann. [6]

Eine weitere Funktion, die noch nützlich ist zum Verändern von Variablen ist *setlocal()*. Mittels dieser Funktion lassen sich die Werte lokaler Variable auf dem angegebenen Level ändern. [6]

Für das Abrufen und Verändern von Upvalues, die auch lokale Variablen, aber nur in einem anderen Scope, darstellen, sind ebenfalls Funktionen vorhanden. Die Funktionen `getupvalue()` und `setupvalue()` lassen sich analog zu den für lokalen Variablen verstehen. Ihre Benutzung unterscheidet sich jedoch. [6]

4.3 Was ist Luaj

Luaj ist eine Reimplementierung von Lua komplett in Java. Dies macht es einfacher Lua in Java als eingebettete Sprache einzubinden und so auch Funktionen von Java im Lua aufzurufen und umgekehrt. Luaj implementiert dabei den Versionsstand von Lua 5.2 nahezu vollständig bis auf kleine Ausnahmen. Die Lua API in Luaj selbst ist dabei bis auf wenige Bestandteile komplett vorhanden und identisch in der Funktion zum normalen Lua. Unterschiede bestehen hauptsächlich in der API zu Java, die äquivalent zur C-API im originalen Lua zu betrachten ist, jedoch an einigen Stellen erhebliche Unterschiede im Umfang aufweist, die Funktionalität bleibt für die vorhandenen Funktionen dieselbe. [7]

5 Der eigene Debugger für LuaJ

Dieses Kapitel wird den Weg vom Konzept zum Luaj Remote Debugger behandeln. Dafür wird die Architektur sowie die Implementierung des Debuggers erklärt. Dieses Kapitel wird nur über die einzelnen Komponenten des Remotedebuggers handeln und nicht über die UI oder externe Steuerung dessen. Dies ist nicht Bestandteil dieser Arbeit. Erklärt werden Konzept Aufbau und Implementierung des Debuggers für das Luaj Framework.

5.1 Ziel und Funktion des Debuggers

Ziel ist es, einen Debugger kompatibel zum Luaj Framework zu entwickeln. Dieser erfüllt folgende Funktionen als Mindestanforderung:

- Das Setzten von Breakpoints
- Das Inspizieren von lokalen und globalen Variablen innerhalb eines Skriptes
- Die Einzelschritt-Verarbeitung von Code
- Das normale weiter Ausführen eines Skriptes, nachdem es angehalten wurde
- Weiter soll der Debugger über eine Netzwerk-Schnittstelle verfügen, über diese der Debugger angesteuert werden kann.

5.2 Architektur und Planung des Debuggers

5.2.1 Erstes Konzept

Da Lua wie im Kapitel 4.1 beschrieben eine absichtlich kleingehaltene Sprache ist, spiegelt sich das auch im Sourcecode von Lua wieder, der klein und kompakt ist und dazu auch noch gut lesbar. So war eine der ersten Anlaufstellen für die Konzeption des Debuggers, Sourcecode von Lua und Luaj zu lesen und zu verstehen. Einerseits mussten Unterschiede zwischen Lua und der Reimplementierung vom Lua in Java also dem Luaj gefunden werden. Dies ist daher sehr wichtig, um Unterschiede in den zu verwendbaren Funktionen zu finden. Anderseits musste die genaue Funktionsweise der Funktionen nachvollzogen werden. Damit wird es erst möglich, dass ein ordentliches funktionierendes Konzept für den Debugger entwickelt werden kann, da dieser tief in die Sprache eingreift.

Das erste Konzept war, den Debugger komplett, wie es im Lua möglich ist, über die externe API, auch C-API genannt, zu implementieren. [8] Auf Luaj angewendet bedeutet dies, den Debugger komplett im Java zu implementieren und die API zu verwenden, die Luaj bereitstellt. Jedoch stellt dies nach Analyse der API und des Sourcecodes von Luaj ein Problem dar, denn Luaj unterscheidet sich hierbei zum originalen Lua im Umfang der API, so dass eine externe Nutzung der API aufgrund von fehlenden Funktionen nicht möglich ist. Konkret

fehlen Luaj dabei effektive Funktionen in der Java-seitigen Debug API, um einen Debugger effizient zu implementieren. Denn Luaj stellt zwar auch alle Debug-Funktionen bereit, die das originale Lua besitzt, aber dies nur innerhalb der Sprache selbst. [7]

Ein Zugriff auf diese Funktionen von externer Seite wäre nur möglich, indem der Ablauf geändert wird und wie die Skripte ausgeführt werden. Das heißt, es müsste aktiv in die Ausführung der Skripte eingegriffen werden, was beuteten würde, dass die implementierte Logik im Luaj für die Ausführung dieser, also die virtuelle Maschine angepasst werden müsste. Dies wäre nur mit einem hohen Aufwand in der Entwicklung und Anpassung des Luaj selbst möglich gewesen. Deshalb ist dieses Konzept für die Entwicklung des Luaj Debuggers für den vorgesehen Einsatzweck ungeeignet.

5.2.2 Zweites Konzept

Eine andere Möglichkeit, den Debugger zu implementieren, ist es den eigentlichen Debugger im Lua selbst zu programmieren. Das hat den Vorteil, dass einerseits nur auf Funktionen zugegriffen werden muss, die von der Lua Debug Library bereitgestellt werden und somit in der Sprache und auch in anderen Lua Implementierungen vorhanden sind. [8] [6]

Anderseits ist dies dahingehend ein Nachteil, dass eine geteilte Architektur geschaffen werden muss, um den Debugger effektiv zu verwalten und zu steuern. Daraus resultiert eine höhere Komplexität der Architektur, was mit einem erhöhten Aufwand in der Implementierung einhergeht, aber dennoch weniger als wie im ersten Konzept.

Das zweite Konzept sieht daher vor, die Architektur des Debuggers aufzuteilen in einen Java-seiteigen Steuerungsteil und Lua-seitigen Debugger Teil. Somit werden die Vorteile aus beiden Welten vereint. Der Lua-Teil des Debuggers, ist daher nur für das Abfragen von Daten, sowie für die Ausführungs-Steuerung und Prüfung der Breakpoints zuständig. Dieser Teil lässt sich auch als eine API für den Steuerungsteil auffassen. Da jedoch nur Lua für die Funktionen des Debuggers verwendet werden kann, ist eine effiziente und clevere Nutzung der Debug Library von Lua unerlässlich. Daher war es nötig, ein sehr gutes Verständnis für diese Library zu bekommen. Aufgrund der schon vorangegangen Recherchen zum ersten Konzept des Debuggers konnte das Vorwissen daraus gut genutzt werden, um weitere Teile des Sourcecodes der Debug Library einfacher und vollumfänglich zu verstehen. Somit lassen sich auch alle Funktionen, die der Debugger benötigt, über die Debug Library des Lua implementieren.

Der Java-seitige Teil des Debuggers ist dabei für die Steuerung zuständig. Dieser soll verschiedene Instanzen von Debuggern verwalten sowie alle Variablen und Breakpoints speichern, um diese manipulieren zu können. Außerdem muss eine Schnittstelle vorhanden sein, um den Debugger effektiv über das Netzwerk zu steuern.

5.2.3 Architektur und Aufbau

Der Debugger wird, nach dem im Kapitel 5.2.2 beschriebenen Konzept aufgebaut. Für die Verwaltung und Steuerung des Debuggers wird ein in Java geschriebener Teil benötigt, der in der Lage sein soll, mehrere Instanzen von Debuggern gleichzeitig zu verwalten, um auch in mehreren Skripten simultan debuggen zu können. Dafür muss es eine Klasse geben, die alle aktiven Debugger Instanzen vorhält und verwaltet. Dieser genannte *DebuggerController* verfügt daher über die Funktionen, eine neue Debugger Instanzen anzulegen, zu löschen und aus einer gespeicherten Liste von aktiven Debugger Instanzen eine zu finden.

Eine Debugger Instanz entspricht dabei einem Skript mit einem dazugehörigen Debugger, der all die Information zum Zustand der Ausführung des derzeit zu debuggend Skriptes speichert. Jede Instanz wird dabei über eine UUID des Debuggers und Skriptes eindeutig identifiziert.

Die einzelnen Debugger Instanzen benötigen wiederum eine Klasse für die Verwaltung und Speicherung von Variablen des Lua-seiteigen Debuggers. Die Klasse ist daher maßgeblich der externe Kopf des Lua-seitigen Debuggers und verwaltet den Zustand des Debuggers auf Lua Seite, zu jedem Zeitpunkt. Diese Klasse *DebuggerMain* muss daher wissen, welches Skript zur derzeitigen Instanz gehört. Dafür werden die Skripte mittels UUID referenziert. Alle Breakpoints für das jeweilige Skript werden ebenfalls in einer Liste gespeichert und abrufbar gehalten. Zusätzlich muss der Debugger speichern, ob gerade die Einzelschritt-Verarbeitung für ein Skript aktiv ist, damit der Lua seitige Teil des Debuggers auch bei der nächsten Code Zeile wieder stoppt. Weiter muss die Klasse alle lokalen und globalen Variablen des Skriptes speichern, sobald diese vom Lua-seitigen Teil geliefert werden. Um im Java gespeichert zu werden, müssen diese allerdings erst von Lua Datentypen in Java Datentypen umgewandelt werden. Dafür ist ein Parser notwendig, der die von Lua gelieferte Table Struktur in eine Java Liste mit Key-Value-Paaren umwandelt. Der Parser und andere Funktionen, die öfters Anwendung finden, werden dabei in einer *LuaUtils* Klasse zusammengefasst.

Für die Schnittstelle vom Lua- zum Java-Teil des Debuggers sind mehrere Handler Klassen zu erstellen. Die Handler übernehmen dabei die Aufgabe der Abfrage von Daten aus dem Java-Teil und das Übergeben von Daten aus dem Lua-Teil in die Java-Seite des Debuggers. Folgende Handler Klassen werden daher benötigt die über das Lua aufrufbar sein müssen.

GlobalValueParser und *LocalValueParser*: Diese beiden Handler übernehmen jeweils das Parsen und Abspeichern der Variablen aus Lua. Dabei wird in lokale und globale Variablen unterschieden.

GetBreakPoints: Dieser Handler ist dafür zuständig, die auf der Java Seite gespeicherten Breakpoints für den Lua Debugger abzufragen und in Lua zu übergeben. Für das Übergeben der Breakpoints ist ebenfalls ein Parser notwendig, um die Java-Liste in einen Lua Table umzuwandeln.

Der *CodeStepperHandler* übernimmt bei der Einzelschritt-Verarbeitung von Lua-Code die Benachrichtigung des Remote Clients und ebenso wenn ein Breakpoint erreicht wird.

Der Handler *CodeStepperActive* ist für das Abfragen zuständig, ob für dieses Skript die Einzelschritt-Verarbeitung derzeit aktiv ist. Dafür wird der in der Debugger Instanz gespeichert wert ins Lua übergeben.

Der Lua-Teil des Debuggers besteht wiederum nur aus einem einzigen Skript, das vollständig in Lua geschrieben ist. Dieses Skript muss alle Funktionen besitzen, die der Debugger benötigt, um die Ausführung des Lua Skriptes zu steuern und Variablen abzufragen. Der Debugger benötigt dabei eine Funktion, um zu prüfen ob ein Breakpoint erreicht wurde. Dafür werden erst die auf der Java-Seite gespeicherten Breakpoints mittels Handler abfragt und geprüft, ob der Debugger einen Breakpoint erreicht hat. Weiter sind Funktionen notwendig um die globale und lokale Variablen der derzeitigen Lua Umgebung abzufragen und ins Java, mittels der Handler Klassen, zu übergeben.

Das Herzstück des Debuggers bildet der Haupt Loop *mainDBGloop*. Dieser wird für jede neue Zeile aufgerufen, in die Lua springt. Dieser ruft die einzelnen Funktionen in einer bestimmten Reihenfolge weiter auf um so einen ordnungsgemäßen Ablauf zu gewährleisten. So wird als erstes im Loop die Prüfung der Breakpoints aufgerufen, beziehungsweise geprüft ob der Codestepper aktiv ist. Sollte kein Breakpoint getroffen worden sein oder der Codestepper nicht aktiv sein, wird die nächste Zeile Lua-Code ausgeführt und der Loop wird erneut aufgerufen. Gibt aber eine der beiden Funktionen ein positives Ergebnis zurück, wird also ein Breakpoint getroffen oder der Stepper ist aktiv, werden als erstes die globalen Variablen und dann die lokalen Variablen abgefragt sowie in den Java-Teil übergeben. Anschließend wird er *CodeStepper* aufgerufen, der den *CodeStepperHandler* in Java aufruft und den Remote-Client benachrichtigt. Dieser wartet dann auf eine Antwort des Remote-Clients, wie weiter verfahren werden soll. Nach der Antwort des Remote-Clients beginnt der Prozess erneut.

Für die Remoteverbindung kommt das Framework jsonrpc4j (siehe Kapitel 6.1) zum Einsatz. Dieses benötigt eine Server Klasse, die einen Server bereitstellt, der für eingehende Verbindungen geöffnet ist. Weiter werden dann durch das Konzept des Remote Procedure Calls Methoden benötigt, die vom Client aufgerufen werden können. Diese werden in einer Handelsklasse gebündelt und beim Start des Servers an diesen übergeben. Bei einer neuen Anfrage durch den Client sucht der Server die passende Methode und schickt dem Client eine entsprechende Antwort.

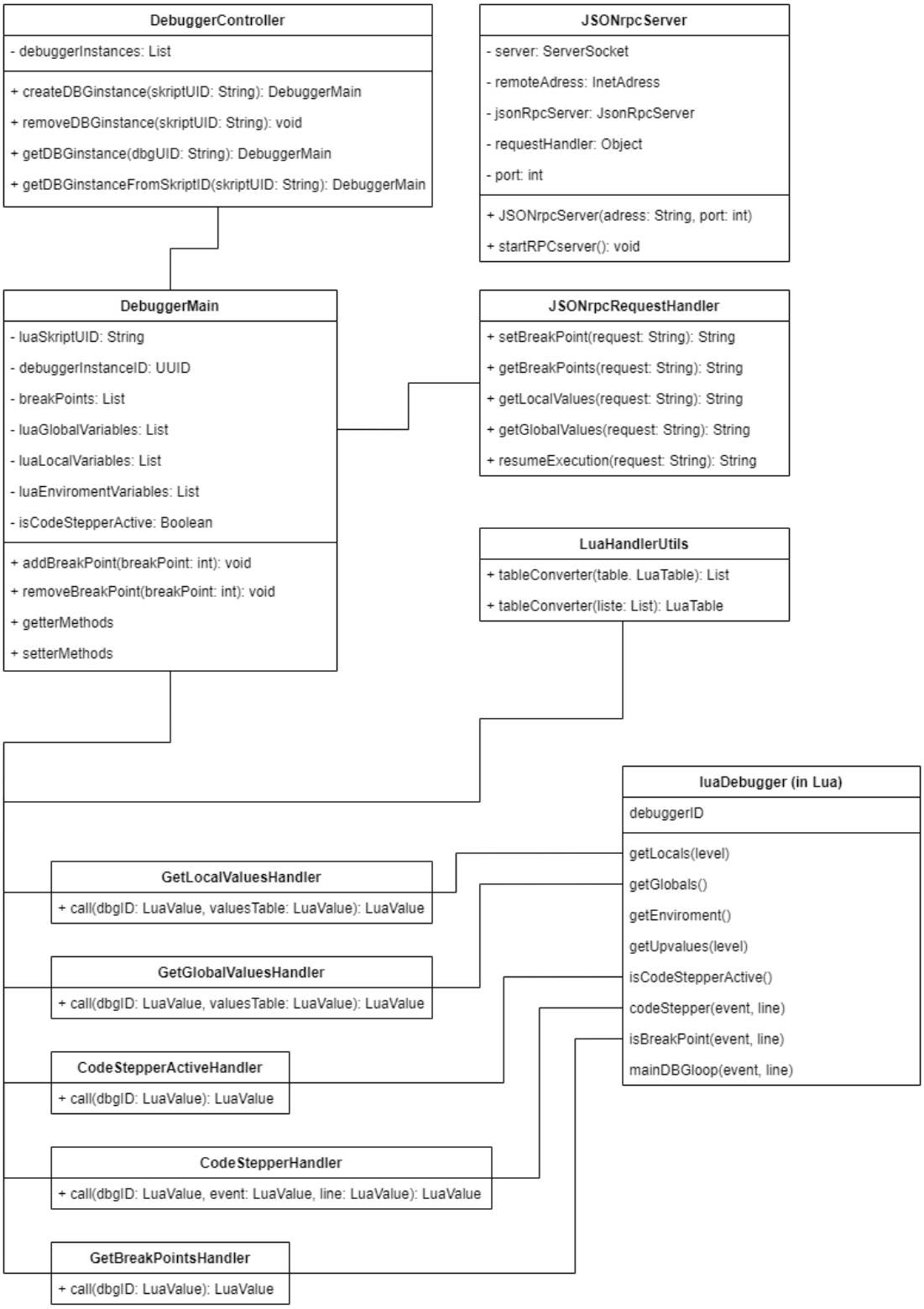

Abbildung 5.2.3-1: Vereinfachtes Klassendiagramm der Debugger Architektur

5.3 Implementierung des Debuggers

5.3.1 Herangehensweise der Implementierung

Nach der Recherche, Findung des Konzeptes und groben Entwurf der Architektur des Debuggers ist es soweit, diesen in funktionsfähige Software zu überführen. Wichtig ist dabei schon von Anfang an die geteilte Architektur zu berücksichtigen. Den Anfang bildet daher der Java-seitige Teil der Architektur, um eine Grundstruktur aufzubauen, auf dem dann der Lua-Teil des Debuggers aufgebaut werden kann. Als erstes ist der Java-seitige Kopf des Debuggers zu implementieren. Der *DebuggerMain* mit all den Funktionen die notwendig sind, um die jeweiligen Objekte zu bearbeiten (siehe Analgen, Teil 2 Java). Als nächstes ist die Verwaltung der einzelnen Instanzen des Debuggers zu implementieren. Diese Klasse wird auch mit den Funktionen gefüllt Instanzen für einen neuen Debugger anzulegen und auch zu löschen, sowie aus einer Liste von Instanzen zu finden. Dies ist wichtig, um von der Debugger ID der Debugger Instanzen auf das dazugehörige Objekt der *DebuggerMain* Klasse zu kommen.

Nachdem die Grundstruktur geschaffen wurde, muss eine Umgebung zum Entwickeln der Lua-Seite des Debuggers aufgebaut werden. Dafür ist ein Lua Skript anzulegen in welchem der Debugger entwickelt werden kann. Dazu ist eine Luaj Umgebung zu initialisieren welche das Skript lädt und ausführt.

Die Entwicklung des Skriptes des Lua Debuggers ist etwas komplizierter als der Java-Teil. Da einerseits kein Debugger zur Verfügung steht, um implementierte Funktionen zu testen, jedoch die Korrektheit dieser gewährleistet sein muss. Es kann daher nur auf primitive Mittel wie *print-Statements* zurückgegriffen werden. Die Entwicklung beinhaltet deshalb, zum besseren Verständnis, ab diesem Punkt immer noch das teilweise lesen von Sourcecode des Luaj.

Für das Lua Debugger Skript sind erstmals alle Funktionen zu implementieren die den eigentlichen Debugger Loop bilden. Also die Methoden, die Variablen lokal als auch global aus der derzeitigen Skript Umgebung abrufen. Eine Methode die prüft ob eine bestimmte Zeile Code erreicht wurde, also ob ein Breakpoint getroffen wurde.

Nachdem die einzelnen Debugger Funktionen in Lua implementiert wurden, ist es notwendig die Java Handler aufzubauen. Die jeweils die Informationen zwischen dem Lua- und Java-Teil austauschen. Handler für die Übergabe der Variablen und ihrer dazugehörigen Parser sind zu implementieren. Nach Abschluss der Implementierung der Handler, können diese im Lua-Teil der Debugger Funktionen eingebunden werden und es kann erstmals getestet werden, ob eine Steuerung des Debuggers über den Java teil erfolgen kann. Gleichzeitig kann aber auch so die Korrektheit der Parser überprüft werden.

Zum Abschluss des Lua-seitigen Teils ist der Debugger Loop zu implementieren (siehe Analgen, Teil 1 Lua). Dieser bildet das Herz des Lua-seitigen Debuggers. Der Aufruf des Loops erfolgt dabei über die *sethook* Funktion der Lua Debug Library. Der Hook wird dabei vor ein Skript gestellt und als *linehook* konfiguriert, also für jede Zeile, damit wird der Loop für jede Zeile Code ausgeführt die aufgerufen wird.

Der letzte zu implementierende Teil ist die Remote Verbindung. Als erstes wird dafür die *JSONrpcRequestHandler* Handler Klasse aufgebaut, welche die Methoden enthält, die per Remote Procedure Call vom Client aufgerufen werden können. Diese sind unter anderem Methoden zum Abrufen der Variablen einer gegeben Debugger Instanz und das Setzten und Entfernen von Breakpoints einer Debugger Instanz.

Weiter ist dann die Server Klasse *JSONrpcServer* welche den JSON Remote Procedure Call Server initialisiert und zu implementieren, um Anfragen vom Client zu verarbeiten. Dabei wird bei der Erstellung des Servers die *JSONrpcRequestHandler* Klasse am Server hinterlegt.

5.3.2 Herausforderungen bei der Implementierung

Debugger Loop

Eine besondere Herausforderung stellte die Implementierung des Debugger Loops (siehe Analgen, Teil 1 Lua) im Lua dar. Da die richtige Reihenfolge gefunden werden musste, die einerseits am performantesten ist und anderseits die Funktionen für das Testen auf Breakpoints und Abfragen von Variablen am sinnvollsten durchläuft. Hier musste mit Trail-and-Error und guter Überlegung der richtige Ablauf erst gefunden werden.

Listen Parser

Eine weitere Herausforderung stellte der Parser für das Parsen der Variablen Table vom Lua ins Java dar. Da Tables auch ineinander geschachtelt werden können und es so sein kann, dass ein Table auf sich selbst verweist. Würde das einfache Durchlaufen des Tables mittels einer Schleife, in einer Endlosschleife enden. Daher ist es wichtig sich gegen diesen Fall abzusichern, um ein ordnungsgemäßes Parsen zu gewährleisten. Um dieses Problem zu lösen wird dafür beim Parsen eine Liste angelegt, in der alle schon durchlaufenen Lua Tables mittels ihrer Referenz (Hash-Header-Wert) gespeichert werden, um zu prüfen ob ein Table schon durchlaufen wurde. Sollte ein Table schon einmal durchlaufen worden sein, kann hier auch gleich die Referenz auf den Table eingefügt werden, anstatt den Table komplett nochmals in die Liste einzufügen, da dieser schon vorhanden ist.

6 Die Remote Verbindung

Diese Kapitel behandelt das eingesetzte Protokoll für die Steuerung und Kommunikation des Debuggers mit der Client Anwendung, also dem Composer, ohne dabei auf direkte spezifische Eigenschaften der Kommunikation einzugehen. Weiter werden die verschiedenen, zu beachtenden Sicherheitsaspekte und deren Lösungs-Strategien beleuchtet.

6.1 Das Steuerungsprotokoll

Für die Kommunikation mit dem Debugger über das Netzwerk ist es notwendig, auf einige wichtige Punkte zu achten, die genau für diesen Anwendungsfall benötigt werden. So ist es unabdingbar, dass es sich um eine verlässliche Kommunikationsmethode handelt, bei der keine Daten auf dem Kommutationsweg verloren gehen können. Weiter ist darauf zu achten, dass die Möglichkeit der beidseitigen Kommunikation besteht, um einerseits Befehle an den Debugger zu senden und andererseits Daten vom Debugger zu erhalten. Hinzu kommt noch, dass die Kommunikation nicht zu langsam, also zu zeitverzögert stattfinden darf, um einen reibungslosen Ablauf des Systems zu gewährleisten. Dieses Kriterium ist allerdings zum Teil zu vernachlässigen, da heutige Kommunikationssysteme keinen größeren Zeitverzögerungen unterworfen sind. Allerdings nicht zu vernachlässigen ist, dass die Verarbeitung einer Nachricht auch Zeit in Anspruch nimmt. Dies teilt sich dabei in zwei Bestandteile auf. Einmal der Teil des Debuggers oder Clients, der den Inhalt der Nachricht interpretiert und der Zweite Teil, der die Kommunikation abwickelt. Bei der Abwicklung der Kommunikation ist daher darauf zu achten, dass ein System eingesetzt wird, das Anfragen mit guter Performanz bearbeiten kann.

Weiter muss die Methode der Kommunikation imstande sein verschiedenste Daten zu senden und zu empfangen. Diese können unter anderem Steuerungsbefehle für den Debugger sein, aber auch strukturierte Daten, wie zum Beispiel Listen mit Variablen-Namen und deren dazugehörigen Wert. Unter der Berücksichtigung der genannten Kriterien fiel die Wahl daher auf das Protokoll JSON-RPC (JavaScript Object Notation Remote Procedure Call). [9] Dieses Protokoll ermöglicht einen sehr flexiblen Einsatz durch die Verwendung des Datenaustauschformates JSON. Somit lassen sich Daten in nahezu jeglicher Strukturierung und Form übertragen. Hinzu kommt die einfache Handhabung durch die Verwendung von Remote Procedure Calls, also das Aufrufen von Routinen aus der Ferne, in diesem Fall über das Netzwerk. Dadurch lässt sich die Steuerung des Debuggers sehr gut in einzelne Methoden implementieren, die der Client dann aufrufen kann, um den Debugger zu verwalten. Die Wahl fiel dabei auf das Framework jsonrpc4j also JSON-RPC für Java, einer Implementierung des JSON-RPC Standards in der Version 2.0 komplett in Java. [9] [10]

6.2 Datenschutz und Sicherheit

Debugger stellen einen wichtigen aber vor allem auch kritischen Punkt in einer Programmiersprache dar. Denn mittels eines Debuggers ist es möglich, direkten Eingriff in die Ausführung von Programmcode zu nehmen. Speziell für Luaj bedeutet dies, dass durch die Verwendung der Debug Library die komplette virtuelle Maschine für Angriffe vulnerabel ist. [11] Dies bedeutet das Code manipuliert werden kann. Dadurch ist es möglich Schadcode nachzuladen, der die Ausführung des eigentlichen Programms erheblich beeinträchtigen kann oder aber genutzt werden kann, um andere Teile der Infrastruktur zu gefährden oder auch zu beschädigen. [12] Zum Beispiel kann dies einen Angreifer ermöglichen Zugriff auch auf nicht IT-Infrastruktur zu erlangen, die aber vom System verwaltet wird. Dies bedeutet, um hier nochmal den Logistik Bezug der Anwendung hervorzuheben, dass ein Angreifer Zugriff auf Fördertechnik-Anlagen bekommen kann, bei der auch menschliche Interaktionen eine Rolle spielen können. Weiterhin könnten dadurch Menschenleben gefährdet werden falls bestimmte Sicherheitsmechanismen außer Kraft gesetzt werden. Aufgrund dessen muss sichergestellt sein, dass der Debugger nicht von dritten unbefugt missbraucht werden kann.

Es ist daher sehr wichtig, dass Debugger besonders vor unbefugten Zugriffen oder Manipulationen durch Dritte geschützt werden, da Debugger einen kritischen Teil einer Softwareinfrastruktur darstellen. Besonders unter dem Aspekt des Remote Debuggers ist es sehr wichtig, die IT-Sicherheits-Schutzziele zu berücksichtigen.

So muss auf Vertraulichkeit besonderen Wert gelegt werden, um keine Daten mit unbefugten Dritten zu teilen, da der komplette Datenverkehr des Remote-Debuggers über das Netzwerk versendet wird. Um Vertraulichkeit zu erreichen, wird für diesen Luaj Remote-Debugger daher eine Verschlüsselung eingesetzt. Die verwendete Verschlüsslung ist dabei TLS *Transport Layer Security*, da diese sich einfach in das verwendete Framework *jsonrpc4j* für die Kommunikation einbinden lässt und einen nach derzeitigem Stand sicheren und verlässlichen Datenaustausch ermöglicht. Der Sicherheitsaspekt der Datenintegrität ist dabei ebenfalls durch die Verschlüsselung abgedeckt, da es so Dritten nahezu unmöglich gemacht wird, Daten während der Kommunikation zu manipulieren. [9] [12]

Ein weiterer wichtiger Aspekt ist die Authentizität, da der Remote-Debugger, auch wie der Name schon sagt, Remote eingesetzt werden soll, also auch nicht direkt vor Ort. Ist es notwendig zu prüfen, ob der Client der den Debugger Steuerungsbefehle übermittelt, auch dazu berechtigt ist, dies zu tun. Deshalb muss ein Mechanismus auf beiden Seiten der Anwendung vorhanden sein, um zu prüfen, ob die Gegenseite mit der gerade eine Kommunikation aufbaut werden soll, auch dazu berechtigt ist dies zu tun, beziehungsweise die Anwendung ist die sie vorgibt zu sein. Dafür kommt ein Token System, auch als Security-Token bezeichnet, zum Einsatz. Dieser wird dabei in der Client Anwendung, also dem Composer, der den Debugger steuert, hinterlegt und dort sicher gespeichert. Mittels dieses Tokens ist es dann möglich sich beim Debugger zu authentifizieren und so zu gewährleisten, dass sich nur die Composer Anwendung mit dem Remote-Debugger verbinden kann. Ein

wichtiger Punkt, der hierbei nicht außer Acht gelassen werden darf, ist, dass der Zugriff auf den Debugger mittels dieses Tokens reguliert wird. Das heißt also, sollte der Token von unbefugten Dritten dupliziert werden, ist der Remote-Debugger damit in der Gefahr kompromittiert zu werden, denn so wird es möglich, dass auch Dritte Zugriff auf den Remote-Debugger erhalten. [12] Daher hängt die Sicherheit des Debuggers auch an der korrekten und vor allem sicheren Speicherung des Tokens in der Client Anwendung ab. Dieser Kompromiss wird allerdings wissentlich und willentlich eingegangen, da eine Authentifizierung, mittels Login-Name und Passwort, einen zu großen Overhead bedeuten würde und somit der praktische Nutzen der Anwendung, im Hinblick auf einfache und schnelle Nutzung, sinken würde.

Für den produktiven Einsatz im MFCS System ist der Debugger allerdings nicht gedacht, sondern nur im Zeitraum der Entwicklung und Anpassung der Anwendung, für den jeweiligen Kunden des Systems. Daher wird der Debugger im MFCS System in einem eigenen Modul untergebracht, was es ermöglicht, beim ausliefern der Software dieses Modul zu entfernen und somit keine Gefahr vom Debugger, im produktiven Einsatz der Software, mehr ausgeht. Trotzdem sind die Sicherheitsaspekte zu beachten, da auch lange Entwicklungszeiten beim Kunden vor Ort, für solche Systeme nicht unüblich sind.

7 Fazit und Ausblick

Der implementierte Remote-Debugger für Luaj, erfüllt die gestellten Anforderungen wie das Setzten von Breakpoints, der Ausführung von Lua Code Schritt für Schritt und das Inspizieren von Variablen. Erreicht wurde auch, dass der Debugger mit dem Luaj kompatibel ist und mittels des Frameworks realisiert wurde.

Weitere Verbesserungen, die den Debugger hinsichtlich seiner Benutzung aufwerten würden, wären das Abfragen von Upvales aus anderen Lua Scopes. Dies wurde allerdings in der derzeitigen Version des Debuggers nicht realisiert, da hierfür kein Bedarf bestand. Weitere Funktionen die für das effizientere Debuggen noch nützlich wären, zu implementieren, sind, dass abschnittsweise Ausführen von Lua Code, also komplette Funktionen auszuführen, genauer einen Lua Scope Block. Diese Funktion ist auch aus anderen Debuggern bekannt und wird meist als Step Over bezeichnet. [1]. Eine ähnliche Variante der vorangegangenen Funktion, die ebenfalls nützlich wäre, ist das fertig Ausführen eines Lua Scopes oder Funktion, also das Springen zum *end* Statement. Der praktische Vorteil liegt darin, dass bei der Fehlersuche so Funktionen oder Code-Bestandteile übersprungen werden können. Was wiederum die Fehlersuche verschnellern kann, da somit nicht jede Zeile einzeln durchgeklickt werden muss.

Eine Erweiterung der Breakpoints ist ebenfalls möglich. So können noch Breakpoints implementiert werden, die nur einmalig getroffen werden. Wobei hier speziell für das MFCS in zwei unterschiedlichen Arten von einmaligen Breakpoints zu unterscheiden ist. Einmal Breakpoints, die innerhalb eines Lua Skriptes einmalig stoppen würden, dies ist zum Beispiel für Schleifen sehr nützlich. Weitere mögliche Breakpoints sind die, die nur bei der ersten Ausführung des Skriptes anhalten, denn im MCFS werden Skripte teilweise periodisch immer wieder aufgerufen. Dies kann von Vorteil sein, wenn zum Beispiel nur überprüft werden soll, ob Werte vom Java ins Lua richtig übergeben wurden. Besonders die zweite Art von Breakpoints, würde die Arbeit für den Entwickler sehr vereinfachen und sollte daher noch ergänzt werden.

Eine letzte Verbesserung wäre noch die Aufruf-Historie von Lua Funktionen anzuzeigen, auch als *traceback* bekannt. [1] Dies kann unteranderem nützlich sein, um zu verstehen, wie und von wo eine Funktion aufgerufen wurde. Da im MFCS die Skripte meist relativ klein sind und somit nicht viele einzelne Funktionen enthalten, ist dies ein Feature mit niedriger Priorität deren Nutzen klar abzuwägen ist.

Literatur

[1] J. B. Rosenberg, How Debuggers Work: algorithms, data structure, and architecture, John Wiley & Sons, Inc., 1996.

[2] "iFD GmbH," iFD GmbH, [Online]. Available: https://www.ifd-gmbh.com/. [Accessed 04 07 2022].

[3] G. W. Amy Brown, The Architecture of Open Source Applications, Volume II, 2012.

[4] R. Ierusalimschy, Programmieren in Lua, 3. ed., München: Open Source Press, 2013.

[5] W. C. L. H. d. F. Roberto Ierusalimschy, "The Implementation of Lua 5.0," 2005. [Online]. Available: https://www.lua.org/doc/jucs05.pdf. [Accessed 27 04 2022].

[6] W. C. L. H. d. F. Roberto Ierusalimschy, "Lua 5.2 Reference manual (als Buch)," 2011-2012. [Online]. Available: https://www.dcc.ufrj.br/~fabiom/comp220142/lua52-refman.pdf.

[7] I. F. James Roseborough, "Sourcecode von Luaj," [Online]. Available: https://github.com/luaj/luaj.

[8] R. Ierusalimschy, "Sourcecode von Lua," [Online]. Available: https://github.com/lua/lua.

[9] "JSON-RPC," [Online]. Available: https://www.jsonrpc.org/. [Accessed 7 06 2022].

[10] Google, "Sourcecode jsonrpc4j," [Online]. Available: https://github.com/briandilley/jsonrpc4j. [Accessed 07 06 2022].

[11] J. Roseborough, "Roseborough.com," [Online]. Available: http://www.luaj.org/luaj.html / http://luaj.org/luaj/3.0/api/index.html. [Accessed 21 05 2022].

[12] P. D. C. Eckert, IT-Sicherheit Konzepte – Verfahren – Protokolle, Auflage 8 ed., München: Oldenbourg Wissenschaftsverlag GmbH, 2013.

Anlagen, Teil 1 Lua

Lua Debugger Skript (luaDebugger.lua) Komplett

```
local ID = ...

-- function for getting all local variables in script or specific scope
-- the parameter that should be passed is the parm for the stack level
local function getLocals(stkLevel)
    local counter = 1
    local level = stkLevel
    local variableHolder = {};

    while true do
      local name, value = debug.getlocal(level, counter)

      if not name then break end

      variableHolder[name] = value
      counter = counter + 1
    end

    LocalValueParser(ID,variableHolder)
end


-- function for getting the global variables
local function getGlobals()
    GlobalValueParser(ID,_G)
end


-- function for getting the upvalues on a specific value
local function getUpvalues(level)
    local variableHolder = {};
    local level = (level or 1) + 1

    local funcClosure = debug.getinfo(level, "f").func
    for i = 1, math.huge do
      local name, value = debug.getupvalue(funcClosure, i)
      if not n then break end
      table.insert(variableHolder, {name,value})
    end

end
```

```
-- function for getting the environment
local function getENV()
    ENVvalueParser(ID,_ENV)
end


-- ask the java debugger instance if the code stepper is active
local function isCodeStepperActive()
    return CodeStepperActive(ID,nil)
end


-- handles codestepper an notifys the client ofer the java side
local function codeStepper(event, line)
    CodeStepperHandler(ID, line, event)

    print("CDline: ",line)
end


-- checks if a breakpoint was hit
local function isBreakPoints(event,line)

    local breakPoints = GetBreakPoints(ID)

    for key, var in pairs(breakPoints) do
       if(line == var) then
         local s = debug.getinfo(1)
         print("sucess: ", line, s)
         -- if the is a breakpoint at this line we return true
         return true
       end
    end
    -- if the is no break point we return false
    return false

end
```

```
-- the main debugger loop will called for every single line of code that
was executed

local function mainDGBLoop(event, line)

    --if we hit a breakpoint or the stepper is active then we retrieve
values
    if  isBreakPoints(event,line) or isCodeStepperActive() then

      -- retrieve the Global Values
      getGlobals()
      -- retrieve ENV
      getENV()
      -- retrieve the local Values
      getLocals(1)     -- Level noch korrekt uebergeben

      codeStepper(event,line)
    end


end

--hook function for skripts
debug.sethook(mainDGBLoop,"l")
```

Anlagen, Teil 2 Java

Java Debugger Teilauszug aus dem Sourcecode

Java *DebuggerMain* Klasse

```
package luaj_debugger_dev_standalone.debugger;

import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

/**
 * Class for the Debugger brain this class manages
the whole debugger and saves all its corresponding values
 * @author JJA
 *
 */
public class DebuggerMain {

	/**
	 * Name and skriptUID for correctly idenfinfy the skript
	 */
	private String luaScriptUID;
	private String luaScriptName;

	private final UUID debuggerInstaceID;

	/**
	 * All the Breakpoints for this specific skript.
	 */
	private ArrayList<Integer> breakPoints;

	/*----------- LUA Debugger Specific Values------------*/

	private List luaGlobalVariables;
	private List luaLocalVariables;
	private List luaEnviroment;

	private List globalValuesFromInit;

	private boolean codeStepperActive = false;
```

```
	public DebuggerMain() {
		// dem Debugger eine uniqe id Zuweisen
		debuggerInstaceID = UUID.randomUUID();

		breakPoints = new ArrayList<Integer>();

	}

	public DebuggerMain(String luaScriptID) {
		// dem Debugger eine uniqe id Zuweisen
		debuggerInstaceID = UUID.randomUUID();
		luaScriptUID = luaScriptID;

		breakPoints = new ArrayList<Integer>();
	}

	/**
	 * Adds an breakpoint to the existing Breakpoints
	 */
	public void addBreakPoint(int breakPoint) {

		breakPoints.add(breakPoint);
	}

	/**
	 * Removes a Breakpoint from the list of Breakpoints
	 */
	public void removeBreakPoint(int breakPointToRemove) {

		if (breakPoints.contains(breakPointToRemove)) {
			breakPoints.remove(breakPoints.indexOf(break-
PointToRemove));
		}
	}

	/**
	 * @return the luaScriptUID
	 */
	public String getLuaScriptUID() {
		return luaScriptUID;
	}

	/**
	 * @param luaScriptUID the luaScriptUID to set
	 */
	public void setLuaScriptUID(String luaScriptUID) {
		this.luaScriptUID = luaScriptUID;
	}
```

```
/**
 * @return the luaScriptName
 */
public String getLuaScriptName() {
      return luaScriptName;
}

/**
 * @param luaScriptName the luaScriptName to set
 */
public void setLuaScriptName(String luaScriptName) {
      this.luaScriptName = luaScriptName;
}

/**
 * @return the breakPoints
 */
public ArrayList<Integer> getBreakPoints() {
      return breakPoints;
}

/**
 * @param breakPoints the breakPoints to set
 */
public void setBreakPoints(ArrayList<Integer> breakPoints) {
      this.breakPoints = breakPoints;
}

/**
 * @return the luaGlobalVariables
 */
public List getLuaGlobalVariables() {
      return luaGlobalVariables;
}

/**
 * @param luaGlobalVariables the luaGlobalVariables to set
 */
public void setLuaGlobalVariables(List luaGlobalVariables) {
      this.luaGlobalVariables = luaGlobalVariables;
}

/**
 * @return the luaLocalVariables
 */
public List getLuaLocalVariables() {
      return luaLocalVariables;
```

```
	}

	/**
	 * @param luaLocalVariables the luaLocalVariables to set
	 */
	public void setLuaLocalVariables(List luaLocalVariables) {
		this.luaLocalVariables = luaLocalVariables;
	}

	/**
	 * @return the luaEnviroment
	 */
	public List getLuaEnviroment() {
		return luaEnviroment;
	}

	/**
	 * @param luaEnviroment the luaEnviroment to set
	 */
	public void setLuaEnviroment(List luaEnviroment) {
		this.luaEnviroment = luaEnviroment;
	}

	/**
	 * @return the globalValuesFromInit
	 */
	public List getGlobalValuesFromInit() {
		return globalValuesFromInit;
	}

	/**
	 * @param globalValuesFromInit the globalValuesFromInit to set
	 */
	public void setGlobalValuesFromInit(List glob-
alValuesFromInit) {
		this.globalValuesFromInit = globalValuesFromInit;
	}

	/**
	 * @return the codeStepperActive
	 */
	public boolean isCodeStepperActive() {
		return codeStepperActive;
	}

	/**
	 * @param codeStepperActive the codeStepperActive to set
	 */
```

```
	public void setCodeStepperActive(boolean codeStepperActive) {
		this.codeStepperActive = codeStepperActive;
	}

	/**
	 * @return the debuggerInstaceID
	 */
	public UUID getDebuggerInstaceID() {
		return debuggerInstaceID;
	}

}
```

Java *DebuggerController* Klasse

```
package luaj_debugger_dev_standalone.debugger;

import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

/**
 * This Class manages the different Debugger instances and is also used
 * for the creation of new instances.
 *
 * @author JJA
 *
 */
public class DebuggerController {

	private static List<DebuggerMain> debuggerInstances;

	public DebuggerController() {
		debuggerInstances = new ArrayList<DebuggerMain>();
	}

	/**
	 * Creates a new Debugger instance and saves the created instance
	 * but only creates a new instance if there is not an existing one
	 * for the this specific script.
	 */
	public DebuggerMain createDBGinstance(String luaScriptUID) {
		if (getDBGinstanceFromSkriptID(luaScriptUID) == null) {
			DebuggerMain dgb = new DebuggerMain();
			debuggerInstances.add(dgb);
			return dgb;
		}
		return null;
	}

	/**
```

```
       * Gets the debugger instance via the ID
       *
       * @param ID of the debugger instance to be found
       * @return DebuggerInsatce
       */
      public static DebuggerMain getDBGinstanceFromID(String ID) {
            for (DebuggerMain debuggerInsta : debuggerInstances) {
                  UUID dgbID = debuggerInsta.getDebuggerInstaceID();
                  if(dgbID.toString().equals(ID)) {
                        return debuggerInsta;
                  }
            }

            return null;
      }

      /**
       * Gets the debugger instance via the lua ScriptID
       * which is unique for each lua script.
       *
       * @param luaSkriptID
       * @return
       */
      public static DebuggerMain getDBGinstanceFromSkriptID(String luaS-
criptID) {
            for (DebuggerMain debuggerInsta : debuggerInstances) {
                  String skriptID = debuggerInsta.getLuaScriptUID();
                  if(skriptID.toString().equals(luaScriptID)) {
                        return debuggerInsta;
                  }
            }

            return null;
      }

      /**
       * Removes an Debugger instance from the list
       *
       * @param dgbInstance
       */
      public static void removeDBGinstanceFromList(DebuggerMain dgbIn-
stance) {
            debuggerInstances.remove(dgbInstance);
      }

      /**
       * Removes an Debugger instance from the list via the ID
       * of the Debugger
       *
       * @param ID
       */
      public static void removeDBGinstanceFromList(String ID) {
```

```
            DebuggerMain dgbInst = getDBGinstanceFromID(ID);
            debuggerInstances.remove(dgbInst);
      }


}
```

Java *getBreakpointsHandler* Klasse

```
/**
 *
 */
package luaj_debugger_dev_standalone.debugger;

import java.util.ArrayList;

import org.luaj.vm2.LuaTable;
import org.luaj.vm2.LuaValue;
import org.luaj.vm2.lib.OneArgFunction;

/**
 * Handler for getting the breakpoints from Debugger instance.
 *
 * @author JJA
 *
 */
public class GetBreakPointsHandler extends OneArgFunction {

      @Override
      public LuaValue call(LuaValue arg) {

            String dgbID = arg.toString();

            DebuggerMain   dgbinst   =   DebuggerController.getDBGin-
stanceFromID(dgbID);
            ArrayList<Integer>   breakPoints   =   dgbinst.getBreak-
Points();

            LuaTable breakPointsLuaTable = new LuaTable();
            int counter = 0;
            for (Integer integer : breakPoints) {
                  breakPointsLuaTable.insert(counter,   LuaValue.val-
ueOf(integer));
                  counter++;
            }


            return breakPointsLuaTable;
      }
```

```
}
```

Rekursiver Lua Table zu Java List Parser

```
/**
      * Method for converting a LuaTable to a List this method is
for converting name,
      * value tables that store the variables out of the LuaVM
      * This method is so designed that tables that reference itself
in the same table
      * so that they create a loop are not run over by the method
again
      * Every table is only iterated once and then stored as a
reference if the table
      * occurrs multiple times
      *
      * format in the list that is returned is that every key value
pair
      * is stored in an String Array of len 2 [name] [value] and
than stored into the list
      *
      * @param table To convert vom LuaTable to Java List
      * @return
      */
     public static List tableConverterSave(LuaTable table) {
          ArrayList tableRegister = new ArrayList();

          class RekTableConverter{

               private  List  tableConverterInner(LuaTable  table,
ArrayList register) {
                    List variableList = new ArrayList();

                    Varargs fistPair = table.next(LuaValue.NIL);
                    String name = fistPair.tojstring(1);
                    String value = fistPair.tojstring(2);
                    LuaValue key = fistPair.arg1();

                    // hinzufuegen der ersten werte zur Liste
                    Variable
List.add(new String[] { name, value });

                    while (!key.equals(LuaValue.NIL)) {
                         Varargs pairs = table.next(key);
                         key = pairs.arg1();
                         name = pairs.tojstring(1);
                         value = pairs.tojstring(2);
```

```
                            if (key.equals(LuaValue.NIL)) {
                                 break;
                            }

                            if(pairs.arg(2).istable()) {

                                 if(register.contains(value)) {
                                      //  falls  der  table  schon
durchlaufen wurde speichern wir nur die referenz
                                      // (Den eigendlichen Table in
die Liste ab)
                                      variableList.add(value);

                                 } else {
                                      // falls table nicht im re-
gister fuegen wir es hinzu
                                      register.add(value);

                                      // und rufen den table parser
rekursiv fuer den darunter liegen table auf
                                      variableList.add(tableCon-
verterInner(pairs.arg(2).checktable(), register));
                                 }
                            }

                            //Werte die kein Table sind in die Liste
abspeichern
                            variableList.add(new  String[]  {  name,
value });
                       }
                       return variableList;
                 }
            }

            RekTableConverter one = new RekTableConverter();
            return one.tableConverterInner(table, tableRegister);
      }
```

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Annaberg-Buchholz, den 25.07.2022

[redacted]

Jasley Jens-Uwe Jangk