Bachelorarbeit: Software-Debugging mithilfe von Code Cities oder: Software-Debugging with Code Cities

Bachelorarbeit

Lennart Kipka

Erster Gutachter: Prof. Dr. rer. nat.Rainer Koschke
Zweiter Gutachter: Dr. René Weller

13.10.2020

Fachbereich Mathematik / Informatik
Studiengang Informatik

Erklärung

Ich versichere, die Bachelorarbeit ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Bremen, den 13.10.2020

. . . .

(Lennart Kipka)

In dieser Arbeit wird das generische Maskulin verwendet, um die Lesbarkeit zu steigern. Es sind aber immer alle Geschlechter gemeint.

INHALTSVERZEICHNIS

1 Einleitung **1**

1.1 Zielsetzung . . . 2

1.2 Aufbau der Arbeit . . . 2

2 Grundlagen **3**

2.1 Debugging . . . 3

2.1.1 Debugger . . . 4

2.1.2 Debugging Prozess . . . 5

2.2 Software Maintenance . . . 6

2.2.1 Perfective Maintenance . . . 7

2.2.2 Adaptive Maintenance . . . 7

2.2.3 Preventive Maintenance . . . 8

2.2.4 Corrective Maintenance . . . 8

2.3 Software Reengineering . . . 8

2.3.1 Reverse Engineering . . . 8

2.3.2 Forward Engineering . . . 9

2.3.3 Dynamische Softwareanalysen . . . 9

2.4 Softwarevisualisierung . . . 9

2.4.1 Code Cities . . . 10

2.4.1.1 SEE . . . 11

2.4.1.2 Dynamische Aufrufgraphen in SEE . . . 12

2.5 Game Engine . . . 14

2.5.1 Unity 3D Engine . . . 14

2.6 Eclipse IDE . . . 15

2.6.1 Debugging in Eclipse . . . 15

3 Entwurf **17**

3.1 Anforderungen . . . 17

3.1.1 Aufzeichnung von ausgeführtem Java Code . . . 17

3.1.2 Dateiformat zur Speicherung der Aufzeichnung . . . 17

3.1.3 Visualisierung der Aufzeichnung in SEE . . . 18

3.2 Ausgeführten Code aufzeichnen . . . 18

3.2.1 Java Debug Interface 18
3.2.2 Entwurf eines JDI Java Loggers 20
3.3 Log Dateiformat 21
3.4 Visualiserung in SEE 23
3.4.1 Daten 23
3.4.2 Visualisierung 24
3.4.3 Steuerung 25

4 Implementation 27

4.1 ExecutedLoCLogger 27
4.1.1 Vorbereitende Methoden 27
4.1.1.1 Klassen des Programms 27
4.1.1.2 Verbindung und Start der JVM 27
4.1.2 Requests und Eventhandler 28
4.1.2.1 Requests 28
4.1.2.2 Events 28
4.1.3 Log-Methode 29
4.2 SEE-Debugger 31
4.2.1 Daten 31
4.2.2 Visualisierung 32
4.2.3 Steuerung 35
4.2.3.1 Automatischer Modus 35
4.2.3.2 Manueller Modus 36

5 Evaluation 37

5.1 Planung 37
5.1.1 Evaluation 37
5.1.2 Fragebogen 38
5.1.3 Einschränkungen 38
5.2 Durchführung 40
5.3 Ergebnisse 41
5.3.1 Auswertung der Messwerte 41
5.3.2 System Usability Scale 43
5.3.3 Fragen zum Systemvergleich 45

6 Fazit und Ausblick 47

6.1 Fazit 47
6.2 Ausblick 47

6.2.1 Weitere Studien 47
6.2.2 Überarbeitung und Erweiterung 48
6.2.3 Virtual und Augmented Reality 48
6.2.4 Andere Programmiersprachen 49
6.2.5 Live-Debugging 49

Abbildungsverzeichnis **51**

Listings **53**

Literaturverzeichnis **58**

KAPITEL 1

Einleitung

Der Trend ist eindeutig: Softwaresysteme werden immer größer und komplexer. Das hat viele verschiedene Gründe. Zum einen werden die Anforderungen der Benutzer immer anspruchsvoller und und zum anderen ermöglichen die stetigen technologischen Fortschritte bei Soft- und Hardware den Entwicklern größere und bessere Software zu bauen. Besonders gut lässt sich dieser Trend bei der Entwicklung der gängigen Betriebssysteme beobachten. In Abbilung 1.1 ist das Wachstum des Betriebssystem Microsoft Windows und dem Linux Kernel zu sehen. Während das 1993 veröffentliche Windows NT 3.0 nur etwa vier bis fünf millionen Zeilen Code umfasste, ist das 19 Jahre später erschienene Windows 8 mit 80 Millionen Zeilen zwanzig mal so groß . Ein ähnliches Wachstum lässt sich beim Linux Kernel verzeichnen. 2003 umfasste die Version 2.6.0 5.2 Millionen Zeilen Code und die 2011 veröffentlichte Version 3.6 15.9 Millionen Zeilen [37].

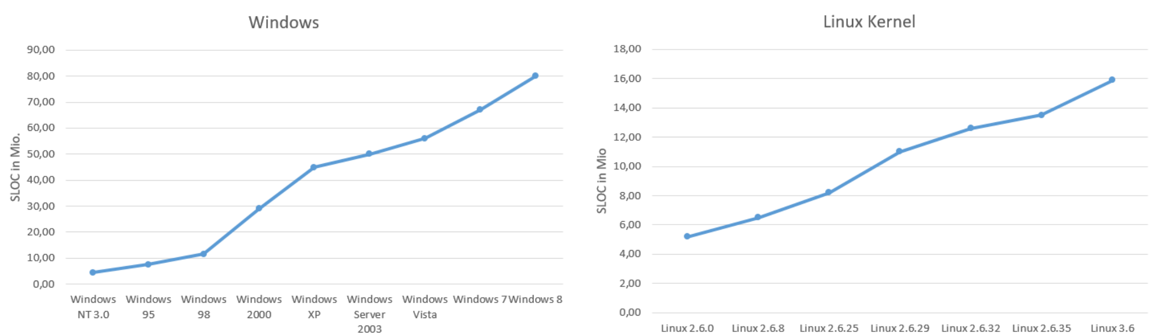


Abbildung 1.1: Entwicklung der SLOC von Microsoft Windows und vom Linux Kernel

Dieser Trend hat aber nicht nur positiven Seiten. In einer Untersuchung aus dem Jahr 2014 von Tashtoush et al. konnte gezeigt werden, dass es eine direkte Verbindung zwischen der Anzahl von Lines of Code eines Softwaresystems und seiner Komplexität gibt [46]. Also wird mit eriner wachsenden Anzahl Lines of Code und der damit steigenden Komplexität das Verständnis von Softwaresystemen erschwert. Weiter konnte V. Nguyen in einem kontrollierten Experiment 2010 zeigen, dass bis zu 50 Prozent der benötigten Zeit zur Softwarewartung alleine für das Verständnis des Programms benötigt werden [36]. Daraus ergibt sich eine Korrelation zwischen der Komplexität eines Softwaresystems und den Wartungskosten. Diese Korrelation konnte auch von Edward E. Ogheneovo in einer Veröffentlichung von 2014 gezeigt werden [37].

Zusätzlich beansprucht die Wartung 60 bis 80 Prozent des gesamten Aufwands bei der Softwareentwicklung [5]. Somit ist das Verstehen von Softwaresystemen ein großer Teil des Entwicklungs- und Wartungsprozesses und daher maßgeblich auch für die hohen Kosten mitverantwortlich.

Aus diesem Grund gibt es das Forschungsgebiet der Softwarevisualiserung. Auf diesem Gebiet

wird an Methoden geforscht, mit denen Softwaresysteme und Informationen auf verschiedene Art und Weise visualisiert werden können. Das Ziel dieser Visualisierungen ist es, das Verständnis von Software und deren statische und dynamische Analyse dieser zu erleichtern, um den steigenden Wartungskosten entgegenzuwirken.
Interessant ist hier die Visualiserungsmethode von R. Wettel et al., die eine Stadmetapher nutzen, um Software zu visualisieren. In dieser Metapher stellen die Gebäude der Stadt die Klassen des Softwareprogramms dar und die Pakete die Stadtdistrikte [51]. Diese Visualsierung kann allerdings nicht nur zum besseren Verständnis der reinen Softwarearchitektur verwendet werden, sondern auch so erweitert werden, dass es möglich ist, Programme statisch und dynamisch zu analysieren. Torben Groß hat 2020 eine solche Softwarestadt verwendet, um die Aufrufgraphen eines Programmablaufes zu visualisieren [22]. Vor diesem Hintergrund kommt die Frage auf, ob sich auch ein Softwarewartungsprozess, wie zum Beispiel das Software Debuggen, mithilfe einer solchen Softwarestadt visualisieren lässt.

1.1 Zielsetzung

Auf Grundlage dieser Fragestellung ist das Ziel dieser Arbeit, mithilfe einer Softwarestadt einen Debug-Prozess zu visualisieren. Dazu wird eine bestehende Software, die solche Softwarestädte bauen kann, um ein Tool erweitert. Dieses soll einen vorher ausgeführten und aufgezeichneten Quellcode in der Stadt anzeigen. Zusätzlich wird ein kleines Programm entwickelt, mit dem die ausgeführten Code Zeilen eines Java Programms zur Laufzeit aufgezeichnet werden können. Diese Aufzeichnungen sollen dann mit dem Visualisierungstool wie ein Video in der Softwarestadt abgespielt werden. Mithilfe einer Softwarestadt soll es so möglich sein, Fehler im Quellcode genauso wie mit den üblichen Verfahren zu finden. Abschließend soll in einer Evaluation überprüft werden, wie das Tool im Vergleich zu einem herkömmlichen Debug Programm beim Erkennen und Reparieren von Fehlern abschneidet und ob sich durch die Visualisierung Vorteile ergeben.

1.2 Aufbau der Arbeit

In Kapitel 2 werden die Grundlagen für diese Arbeit gelegt. Dazu werden wichtige Themenbereiche und Technologien erläutert und es wird auf relevante Arbeiten eingegangen. Kapitel 3 beschäftigt sich mit dem Konzept für die zu entwickelnden Programme. Hier werden Designentscheidungen getroffen und Programmentwürfe gemacht. Anschließend wird in Kapitel 4 die Implementation der in Kapitel 3 entworfenen Software erläutert. In Kapitel 5 wird die abschließende Evaluation der Arbeit geplant, durchgeführt und ausgewertet. Zuletzt wird in Kaptiel 6 ein Fazit zu dieser Arbeit und den Ergebnissen gezogen und im Ausblick werden zukünftige Möglichkeiten und Erweiterungen dieser Arbeit behandelt.

KAPITEL 2
Grundlagen

2.1 Debugging

Debugging ist einer der wichtigsten Aktivitäten eines Softwareentwicklers. Das Ziel dieser Aktivität ist es, wie der Name andeuten lässt, sogenannte Bugs in einer Software zu finden und diese zu reparieren beziehungsweise „zu entfernen“ oder im Englischen „to debug Software“. In der Softwareentwicklung ist ein Bug ein Fehler oder Mangel, der ein ungewünschtes oder fehlerhaftes Verhalten eines Programms verursacht. Der Begriff des Bugs stammt ursprünglich aus dem Ingenieurwesen und wurde im Laufe der Zeit auch in der Softwareentwicklung populär. Im Ingenieurwesen hat ein Bug die Bedeutung eines Fehlers im System. Seinen Ursprung hat er in der Vorstellung, dass ein Käfer (Bug) sich in eine Maschine verirrt und ein Fehlverhalten verursacht. Der Begriff wird schon seit dem 19. Jahrhundert verwendet, wie ein Zitat von Thomas Edison aus einem Brief vom 03.03.1878 zeigt:

> You were partly correct, I did find a „Bug“ in my apparatus, but it was not in the telephone proper. It was of the genus „callbellum“. The insect appears to find conitions for its existence in all calll apparatus of Telephones.[19]

Nach Andreas Zeller impliziert das Wort Bug fälschlicherweise im Zusammenhang von Software, dass sich die Fehler unabhängig vom Entwickler in die Software einschleichen. In Wirklichkeit werden Bugs aber direkt durch Fehler der Entwickler erzeugt. Deshalb hat Zeller drei neue Begriffe für Softwarefehler definiert[54]:

Defect/Defekt: Fehlerhafter Programmcode

Infection/Infektion: Fehlerhafter Programmzustand

Failure/Fehler: Ein sichtbares Fehlverhalten des Programms

Im weiteren Verlauf dieser Arbeit werden die Begriffe von Zeller für Bugs verwendet.
Eine Software wird in ihrem gesamten Lebenszyklus immer wieder gedebugged. Da Software kontinuierlich weiterentwickelt wird, entstehen immer wieder neue Defekte, die repariert werden müssen. Deshalb ist das Debuggen verantwortlich für einen großen Teil des zeitlichen Aufwandes und damit der Kosten in der Softwareentwicklung. In einer Studie von LaToza et al. haben Entwickler angegeben, dass sie fast die Hälfte ihrer Zeit damit verbringen, Defekte zu korrigieren [29]. Dies spiegelt auch die Aussage von B. Hailpern und P. Santhanam, dass 50 bis 75 Prozent der gesamten Entwicklungskosten durch das Debuggen, Testen und Verifizieren verursacht werden, wider [24].
Im nun Folgenden werden die nötigen Grundlagen des Debuggings erläutert. Zuerst werden die gängigsten Techniken vorgestellt und es wird der grobe Ablauf eines Debugprozesses nach A. Zeller gezeigt. Am Ende des Kapitels erfolgt die Einorndnung der Arbeit in diesen Kontext.

2.1.1 Debugger

Es gibt verschiedene Werkzeuge und Verfahren, die Softwareentwickler beim Korrigieren von Defekten unterstützen. In diesem Abschnitt der Arbeit werden einige bewährte Verfahren und Werkzeuge vorgestellt, aber auch der Back In Time Debugger, der nur selten Verwendung findet, wird erläutert.

Print Debugging Das erste Verfahren ist das Print Debugging, auch Tracing oder „Printf()"-Debugging (nach der C-Methode) genannt. Das Print Debugging ist einfach zu verwenden und sehr bekannt. Die Entwickler bauen an geeigneten Stellen im Programmcode neue Befehle ein, die Informationen zur Laufzeit des Programms in einer Konsole ausgeben oder in eine neue Datei schreiben. So lässt sich der Verlauf eines Programms verfolgen oder der Wert eines Feldes oder einer Variable zur Laufzeit einfach auslesen. In einer Studie von M. Perscheid et al. in 2014 war das Print Debugging das zweitmeist verwendete Verfahren zum Debuggen und wird von circa 75 Prozent der Teilnehmer regelmäßig verwendet. Außerdem war das Verfahren allen Teilnehmern der Studie bekannt [45]. Ein sehr ähnliches Verfahren ist das Post Mortem Debugging. Dabei wird bei dem Absturz eines Programms eine Ausgabe erzeugt, mit dessen Hilfe der Absturz nachvollzogen werden kann.

Step-Through Debugger Das laut der Studie von M. Perscheid et al. am häufigsten genutzte Hilfsmittel, mit regelmäßiger Verwendung durch circa 80 Prozent der Teilnehmer, ist der Step-Through oder Interactive Debugger[45]. Beim interactive Debugger handelt es sich um ein selbstständiges Werkzeug, das in den meisten Fällen bereits in eine Entwicklungsumgebung eingebaut ist. Mit diesem Werkzeug lässt sich ein Programm Schritt für Schritt ausführen. Dazu müssen im Programm eine oder mehrere Code Zeilen als sogenannter Breakpoint markiert werden. Erreicht das Programm dann beim Ausführen einen dieser Breakpoints, wird es angehalten und der Entwickler kann schrittweise weiter durch das Programm gehen, allerdings nur vorwärts. Dabei kann er in jedem Schritt den Zustand genau inspizieren, indem das Werkzeug ihm alle konkreten Werte der Variablen und Felder, die in diesem Schritt verfügbar sind, anzeigt. Zusätzlich können die Werte verändert und so der Verlauf des Programms beim Debuggen aktiv beeinflusst werden. Abbildung 2.7 auf Seite 15 zeigt die Oberfläche des Step-Through Debuggers, die in der Eclipse Entwicklungsumgebung integriert ist.

Asssertions Das letzte weit verbreitete Verfahren sind Assertions[45]. Eine Assertion, zu Deutsch Behauptung, ist beim Programmieren eine Überprüfung einer Bedingung. In so einer Bedingung kann getestet werden, ob sich ein Programm so verhält, wie es soll. Dazu vergleicht man mit einer Assert-Methode das tatsächliche Ergebnis einer Funktion, der in der Methode bestimmte Startwerte gegeben werden, mit dem erwarteten Ergebnis. Man geht grundsätzlich davon aus, dass die Bedingung immer erfüllt ist. Entspricht das tatsächliche Ergebnis, wie angenommen, dem Erwarteten, gibt die Assert-Methode nichts aus und der Entwickler weiß, dass der Code sich in diesem Fall richtig verhält. Andernfalls reagiert die Assert-Methode und gibt, je nach Implementierung, verschiedene Ausgaben zurück, die dem Entwickler helfen den Fehler, der das falsche Verhalten verursacht hat, zu entdecken.

Back in Time Dem normalen Step-Through Debugger fehlt es an einer entscheidenden Funktion. Wird ein Breakpoint in der Ausführung des Programms getroffen, gibt es nicht die Möglichkeit, rückwärts durch den ausgeführten Code zu schreiten. Bei vielen Fehlern liegt der

Defekt bereits vorher in der Ausführung des Programms[54] und dieser lässt sich daher mit dem normalen Step-Through Debuggern nur schwer finden. Aus diesem Grund werden die Back in Time Debugger entwickelt. Diese erlauben es dem Entwickler, den ausgeführten Code, wie bei einem Step-Through Debugger, auch rückwärts Schritt für Schritt zu untersuchen und so Defekte zu finden, die nicht direkt einen Fehler verursachen. Ein Back in Time Debugger hat allerdings einen großen Nachteil. Um rückwärts durch den ausgeführten Code zu gehen, muss jeder Zustand des Programms von Start bis Ende aufgezeichnet werden, sodass diese zu einem späteren Zeitpunkt wiederhergestellt werden können. Das führt zu erheblichen Einbußen bei der Performanz des aufgezeichneten Programms. Programme laufen, während sie aufgezeichnet werden, zehn bis hundert Mal langsamer als normalerweise[8]. Nach der Studie von Perscheid et al. haben fast die Hälfte der Entwickler noch nie von einem Back in Time Debugger gehört und nur weniger als 5 Prozent nutzen dieses Werkzeug regelmäßig[45].

2.1.2 Debugging Prozess

Nach Andreas Zeller lässt sich der Debug-Prozess in sieben Schritte unterteilen:

- 1. Track the problem in the Database.
- 2. Reproduce the failure.
- 3. Automate and simplify the test case.
- 4. Find possible infection origins.
- 5. Focus on the most likely origins.
- 6. Isolate the infection chain.
- 7. Correct the defect.

Nimmt man nun von allen Schritten den ersten Buchstaben, kommt das Wort „TRAFFIC“heraus. Mit den „TRAFFIC“ Schritten lässt sich nach A. Zeller immer beim Debuggen vorgehen[54]. Im nun Folgenden werden alle Schritte kurz erklärt.

Track the problem in the Database. In diesem Schritt wird das Problem aufgezeichnet und dokumentiert. Dies dient dazu, dass der Fehler bekannt wird und in der Zukunft auf die Aufzeichnungen zurückgegriffen werden kann, um beim Lösen neuer Probleme zu helfen.

Reproduce the failure. Ziel dieses Schrittes ist es, durch gezielte manuelle Eingaben den Fehler bewusst auszulösen.

Automate and simplify the test case. Mit dem im zweiten Schritt gefundenen Wissen wird das Auslösen des Fehlers automatisiert. Dazu wird eine Testfuntkion erstellt, die mit möglichst wenigen Eingaben das Programm in den Zustand versetzt, in dem der Fehler auftritt.

Find possible infection origins. An dieser Stelle werden alle möglichen Ursprünge zusammengetragen, die den Infekt, der zum Fehler führt, ausgelöst haben könnten. Dazu schaut man sich die Zustände des Programms an. Ab einem gewissen Zeitpunkt ist der Zustand sichtbar infiziert. Hat man diesen Zustand gefunden, kann man davon ausgehen, dass zwischen dem letzten eindeutig korrekten Programmzustand und dem „Infizierten“ der Defekt aufgetreten sein muss. Man kann seine Fehlersuche dann auf alle Funktionen einschränken, die zwischen diesen beiden Zuständen ausgeführt wurden.

Focus on the most likely origins. Hier werden nun die im vierten Schritt gefundenen möglichen Ursprünge genauer überprüft und auf alle Funktionen beschränkt, die tatsächlich eine Auswirkung auf den infizierten Zustand haben.

Isolate the infection chain. In diesem allerletzten Schritt der Fehlersuche wird mit den im vorherigen Schritt gefundenen Funktionen die Fehlerkette von der Infektion bis zurück zum Defekt, dem tatsächlichen Ursprung, hergestellt. Nun ist der gesamte Verlauf vom Defekt zum Fehler bekannt und kann repariert werden.

Correct the defect. Im letzten Schritt wird basierend auf den Annahmen der vorherigen Schritten der Defekt korrigiert. Nun muss nur noch mit dem automatisierten Testfall getestet werden, ob der Defekt und damit auch der Fehler tatsächlich beseitigt wurden.

Besonders aufwändig im gesamten Prozess sind die Schritte vier bis sechs, die sich mit dem Verstehen des Codes beschäftigen[54]. Deshalb sind dies auch die Schritte, bei denen vor allem die vorher erläuterten Debugging-Werkzeuge und -Methoden verwendet werden. 2002 haben Entwickler vom RTI geschätzt, dass durch Verbesserungen an der Test- und Debuginfrastruktur alleine in den USA jährlich bis zu 22 Billionen US-Dollar eingespart werden können[47]. In dieser Arbeit soll ein neues Debugging Werkzeug implementiert und getestet werden, das durch Visualisierungen Verbesserungen gegenüber den üblichen Werkzeugen mitbringen soll.

2.2 Software Maintenance

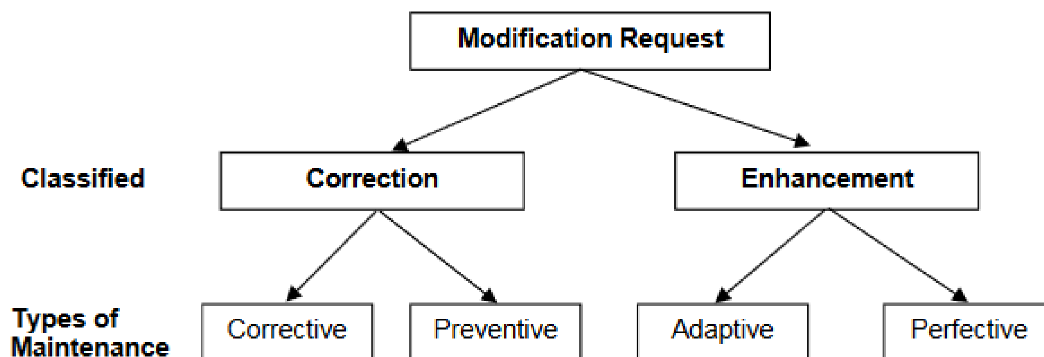


Abbildung 2.1: Modification Request nach ISO/IEC 14764 [1]

Software Maintenance oder auch Softwarewartung genannt, ist ein weiterer wichtiger Bereich der Softwareentwicklung. Für diese Arbeit ist die Softwarewartung relevant, da das Debuggen nicht nur bei der ersten Entwicklung einer Software eine Rolle spielt, sondern auch in der Wartung. Eine Definition für die Softwarewartung gibt der Standard ISO/IEC/IEEE 24765:2017 von 2017[2]:

> Software maintenance is the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment.

Software wird entwickelt, um Probleme aus der realen Welt mithilfe von Computern zu lösen. Die echte Welt befindet sich im stetigen Wandel und dadurch verändern sich bestehende Probleme oder es tauchen neue Probleme auf. Das führt dazu, dass Software mit dem Laufe der Zeit veraltet und ihren Mehrwert verliert. Um dies zu verhindern, gibt es die Softwarewartung, deren klares Ziel es ist, Software so lange wie möglich weiterzuentwickeln und zu verbessern.
E. B. Swanson et al. haben 1978 ursprünglich drei Kategorien der Softwarewartung benannt: Corrective Maintenance, Adaptive Maintenance und Perfective Maintenance [31]. Mit dem neuen Standard ISO/IEC 14764 wurden die Kategorien aktualisiert und um eine weitere ergänzt: Preventive Maintenance [1].
In Abbildung 2.1 ist eine grobe Übersicht einer Modification Request zu sehen. Eine Modification Request beschreibt eine gewünschte Veränderung während der Wartung an einem Softwareprodukt. Diese Request kann sowohl von den Benutzern der Software, als auch den Entwicklern gestellt werden.
Darüber hinaus kann man erkennen, dass im International Standard die vier Wartungskategorien weiter in zwei Klassifizierungen aufgeteilt werden. Bei der Correction Maintenance handelt es sich um Wartungsarbeiten, die ausgeführt werden, um existierende Fehler in der gewarteten Software zu reparieren oder zu verhindern. Bei der Enhancement Maintenance wird die Software erweitert, um neuen Anforderungen, die nach der initialen Auslieferung des Produkts entstehen, gerecht zu werden. Die vier spezifischen Kategorien werden im nun Folgenden genauer erläutert.

2.2.1 Perfective Maintenance

Unter Perfective Maintenance versteht man im Allgemeinen die Erweiterung der Anforderungen einer Software nach ihrer Auslieferung. Mit steigender Erfahrung bei der Verwendung von Software wünschen sich Benutzer neue Funktionen und Anpassungen, um den Nutzen dieser weiter zu steigern [23]. Änderungen in der Perfective Maintenance umfassen Verbesserungen für die Benutzer durch das Anpassen der Bedienung, dem Hinzufügen von neuen Funktionen und der Steigerung der Performanz und Wartbarkeit [1]. Eine klassiche Modification Request im Sinne der Perfective Maintenance ist das nachträgliche Einfügen einer Funktion, die in den initialen Anforderungen vergessen wurde, in ein bestehendes System. Im gesamten Wartungsprozess sind Aufgaben dieser Kategorie für 50 Prozent des Aufwandes verantwortlich [25].

2.2.2 Adaptive Maintenance

Die Adaptive Maintenance umfasst alle Wartungsarbeiten, die sicherstellen, dass eine Software in einer sich veränderden Umgebung weiter funktioniert. Diese Veränderungen können sowohl durch Hardware-, als auch durch Softwarewechsel verursacht werden [1]. Gängige

Beispiele aus der Praxis sind der Wechsel auf ein neues Betriebssystem oder die Ausstattung von Arbeitsplätzen mit neuen Computern. In diesen Fällen muss weiter gewährleistet werden, dass die Software einwandfrei funktioniert, und dafür sorgt die Adaptive Maintenance. Die Funktionen einer Software werden im Rahmen dieser Wartungsarbeit normalerweise nicht erweitert. Insgesamt sind 25 Prozent der anfallenden Wartungsarbeiten in dieser Kategorie [25].

2.2.3 Preventive Maintenance

In der Preventive Maintenance geht es darum, potentielle Fehlerquellen zu beheben, bevor diese Probleme verursachen. Eine weiteres Teilgebiet ist die Verbesserung der Wartbarkeit [23]. In der Praxis besteht präventive Wartung hauptsächlich aus Code Optimierungen und Restrukturierungen. Diese Kategorie ist die kleinste und verwendet nur etwa 5 Prozent des gesamten Wartungsaufwandes [25].

2.2.4 Corrective Maintenance

Zuletzt gibt es noch die Corrective Maintenance. Bei dieser Art der Wartung ist das Ziel, nach der Auslieferung einer Software entdeckte Fehler zu korrigieren [1]. Diese Fehler umfassen alles, was zu einem ungewollten oder falschen Verhalten der Software führt. Sie werden mithilfe des Debuggings indentifiziert und repariert, wie in Kapitel 2.1 beschrieben. Der Aufwand für die Corrective Maintenance beträgt 20 Prozent des Gesamtaufwandes der Wartung und ist damit nicht unbedeutend [25]. An dieser Stelle im Warutungsprozess kann ein guter visualisierter Debugging-Prozess die Softwarewartung erleichtern und damit Zeit und Kosten sparen.

2.3 Software Reengineering

Das Software Reengineering ist ein Teilgebiet der Softwareentwicklung, das in direktem Zusammenhang mit der Softwarewartung steht. Die Ziele des Software Reengineerings beinhalten unter anderem die Verbesserung der allgemeinen Wartbarkeit, Steigerung der Wartungsproduktivität und eine bessere Änderbarkeit [9, 34]. Im Deutschen lässt sich das Reeingeering auch als Renovierung von Software ausdrücken. Nach Chikofsky und Cross II ist das Reengineering die Analyse und Veränderung eines Softwaresystems, um es in einer neuen verbesserten Form zu implementieren [12]. Generell beinhaltet das Reengineering Aktivitäten des Reverse Engineering, um eine verständlichere abstraktere Vorstellung des Systems zu erhalten, und darauf aufbauend des Forward Engineering, um das System zu renovieren[12]. Die Begriffe Reverse Engineering und Forward Engineering werden nun erklärt. Anschließend wird in Kapitel 2.3.3 noch die dynamische Analyse erläutert, da sich diese Arbeit auch in diesen Bereich einordnen lässt.

2.3.1 Reverse Engineering

Baumöl et al. haben 1996 basierend auf den drei gängisten Definitionen für Reverse Engineering von Chikofsky und Cross[12], McClure[34] und Yu[53] folgende Definition aufgeschrieben[9]:

> Identifizierung der einzelnen Komponenten eines Software-Dokuments und ihrer Beziehungen durch eine methodische Analyse sowie eine sich gegebenenfalls an-

> schließende Darstellung dieser Informationen in Form von Software-Dokumenten eines abstrakteren Niveaus.[9]

Beim Reverse Engineering wird also ein in einer beliebigen Form existierendes Softwaresystem genommen und analysiert, um die Komponenten, die das Gesamtsystem ausmachen, zu identifzieren und ihre Zusammenhänge zu erkennen. Danach wird das in der Analyse erlangte Wissen über das System in einer abstrakteren Form festgehalten. Beim Reverse Engineering handelt es sich um einen rein beobachtenden Prozess, der keine Änderung am Status quo der Software vornimmt[12]. Gentuzt wird Reverse Engineering hauptsächlich, um das Verständnis eines Softwaresystems zu verbessern, Komponenten zu identfizieren, extrahieren und wiederzuverwenden oder um das Design der Software wiederzuerlangen[9].

2.3.2 Forward Engineering

Das Forward Engineering beschreibt den Prozess, aus abstrakten Dokumenten, wie Softwarearchitekturen, Anforderungen, Klassendiagrammen und Ähnlichem, konkrete Software zu implementieren. Genauer gesagt wird in diesem Prozess aus abstrakten Beschreibungen eines Softwaresytems entsprechender Programmcode und damit ein lauffähiges Programm.

2.3.3 Dynamische Softwareanalysen

Bei einer dynamischen Softwareanalyse werden die Eigenschaften einer Software zur Laufzeit oder mithilfe von zur Laufzeit gesammelten Informationen analysiert[6, 14]. Die dynamische Analyse ist im Stande fehlerhaftes Verhalten aufzudecken und den Entwicklern Informationen zum tatsächlichen Verhalten ihrer Software zu liefern[6]. Die Vorteile der dynamischen Analyse sind die Präzision im Bezug auf das tatsächliche Verhalten der Software, und dass die zu analysierenden Daten durch gezieltes Steuern des Programms auf das Relevante beschränkt werden können. Aber die dynamische Analyse hat auch Einschränkungen. Unter anderem sind die Daten unvollständig, da nur die tatsächlich ausgeführten Funktionen analysiert werden, nicht alle möglichen. Zusätzlich ist die Skalierbarkeit einer dynamischen Analyse beschränkt, da teilweise enorme Mengen an Daten verarbeitet und gespeichert werden müssen, was wiederum zu Leistungseinbußen und Speicherkapazitätsproblemen führen kann[14].
Die dynamische Analyse entstand als Methode zum Debuggen und Testen von Software. Alle in Kapitel 2.1.1 vorgestellten Methoden und Werkzeuge verwenden eine Form der dynamischen Analyse. Mit immer zunehmender Komplexität und Größe von Software wurde die dynamische Analyse als Werkzeug zum Verständnis von Software populärer und Gegenstand eigener Forschungen[14].

2.4 Softwarevisualisierung

Software ist immateriell. Deshalb ist es schwer für den Menschen, sich die Funktionsweisen und Zusammenhänge innerhalb einer Software vorzustellen. Dies führt wiederum dazu, dass das Verstehen von Softwaresystemen einen wesentlichen Anteil der Softwareentwicklung ausmacht [13, 36]. Die Softwarevisualisierung versucht dieses Problem zu lösen. Nach Stephan Diehl lässt sich die Softwarevisualisierung definieren als:

> „the visualization of artifacts related to sofware and its developement process“[16].

Es geht also darum, visuelle Representationen für verschiedene Gebiete der Softwareentwicklung zu finden. Dazu werden Wege und grafische Darstellungen entwickelt, um die Struktur, das Verhalten, die Evolution und den Programmcode selbst zu visualisieren[16, 7].

Struktur Die Struktur einer Software beschreibt ihre statischen Komponenten und die Beziehungen innerhalb des Sytems. Diese Informationen können aus den Dateien der Software gewonnen werden, ohne dass das Programm aktiv laufen muss. Die Struktur umfasst das Datenmodell, die statischen Aufrufgraphen der Funktionen und die Unterteilung des Systems in beispielsweise Teilsysteme und Module[16].

Verhalten Beim Verhalten wird das Ausführen eines Programms visualisiert. Das Ausführen eines Programms lässt sich in eine Abfolge von sogenannten Programmzuständen vereinfachen. Ein Programmzustand ist eine Verbindung aus ausgeführtem Quellcode und konkreten Daten, die zu diesem Zeitpunkt vom Programm verwendet werden[16]. Mithilfe dieser beiden Informationen und der Visualisierung lässt sich dann das Verhalten der Software nachvollziehen.

Evolution Die Evolution einer Software beschreibt die Veränderung des Programmcodes, der Struktur und der Zusammenhänge im Verlaufe der Entwicklung.

Programmcode Auch der Quellcode selbst kann visualisert werden. Ein Beispiel dafür ist das Code Highlighting[7]. Dabei werden Befehle und Schlüsselwörter einer Programmiersprache in besonderen Farben markiert. Auch die Formatierung von Quellcode ist eine Form der Programmcodevisualisierung.

Das Ziel der Softwarevisualisierung ist es, das Verständnis und die Analyse von Software zu erleichtern und damit den Softwareentwicklungsprozess zu fördern[7, 16, 44]. Im Kontext dieser Arbeit ist die Metapher einer Stadt zur Visualsierung von Software wichtig, da diese als Grundlage für das visuelle Debugging-Werkzeug genutzt wird. Im folgenden Kapitel 2.4.1 wird diese Metapher erklärt und in Kapitel 2.3 wird die Software vorgestellt, die als Grundlage verwendet wird.

2.4.1 Code Cities

Code Cities oder Software-Städte verwenden die Metapher einer Stadt, um Software zu visualisieren. R. Wettel und M. Lanza waren unter den ersten Entwicklern, die diese Idee umgesetzt haben. In dieser Stadtmetapher repräsentieren die Gebäude die Klassen einer Software und die Position beziehungsweise der Stadtdistrikt, in dem sie stehen, repräsentiert das Paket, zu dem die Klassen gehören. Sie haben sich aus mehreren Gründen für diese Metapher entschieden. Zum einen ist eine Stadt dem Menschen ein bekanntes räumliches Konzept, in dem er sich mit klaren Orientierungspunkten wie Straßen und Gebäuden zurechtfinden kann. Zum anderen wird eine Stadt mit zunehmender Größe komplexer, genauso wie eine Software. So kann eine Visualisierung durch eine Stadt der Komplexität von Software gerecht werden, ohne diese zu sehr zu vereinfachen. Zuletzt bilden die Klassen und Pakete einer objektorientieren Software die grundlegenden Orientierungspunkte für einen Entwickler[50, 51]. Dies deckt sich mit den Gebäuden und Distrikten einer Stadt als Orientierungspunkte.
Abbildung 2.2 zeigt die Softwarestadt von ArgoUML v.0.24, einem UML-Werkzeug, das mit

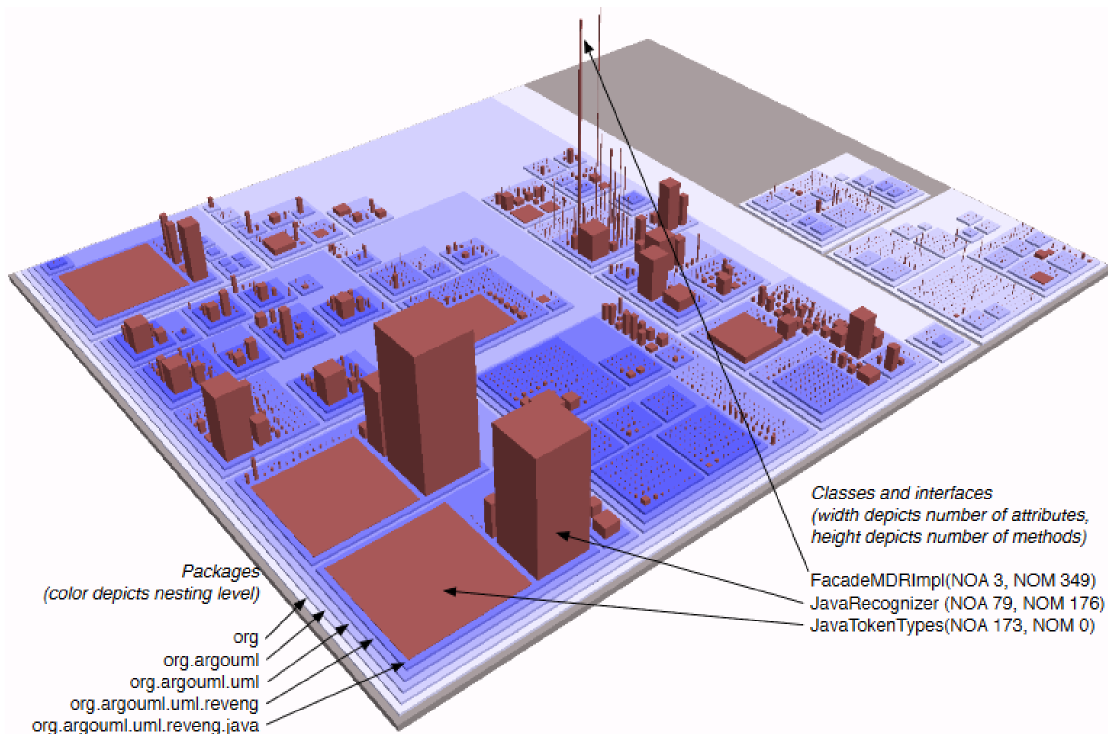


Abbildung 2.2: Ein Überblick der CodeCity von ArgoUML v.0.24[51].

der Software CodeCity von R. Wettel und M. Lanza generiert wurde. CodeCity ist ein Visualisierungswerkzeug, das unabhängig von der Programmiersprache, aus Software interaktive Städte erzeugen kann[51].

2.4.1.1 SEE

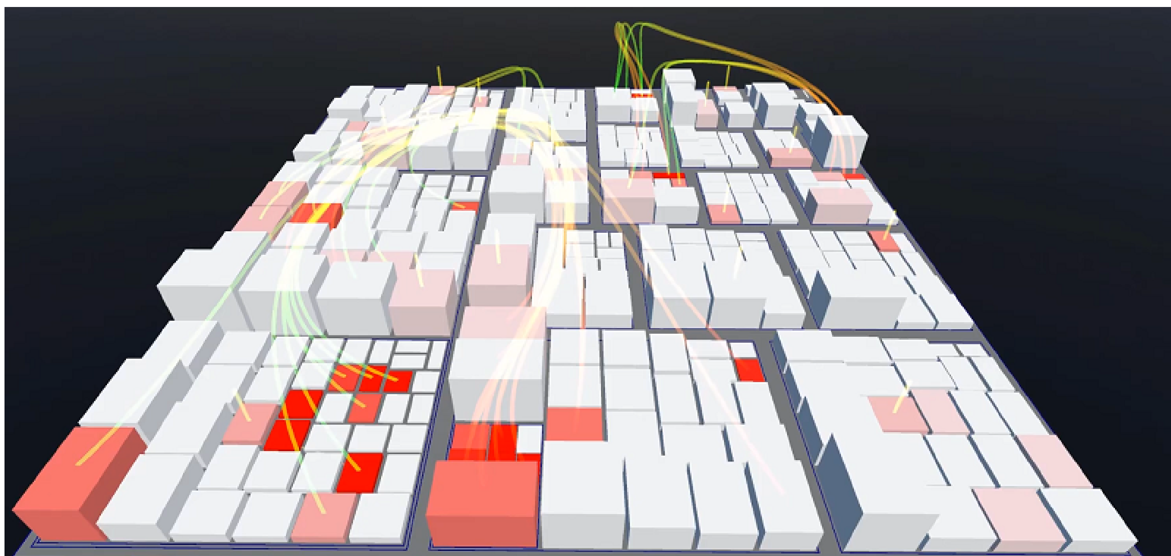


Abbildung 2.3: Softwarestadt des Linux Net Subsystems erzeugt mit SEE[28].

SEE ist ein Softwarevisualisierungswerkzeug, ähnlich wie CodeCity, das Software als Stadt visualisieren kann. Es wird von der Arbeitsgruppe Softwaretechnik der Univerität Bremen

entwickelt und als Grundlage für das visuelle Debugging-Werkzeug, das in dieser Arbeit entwickelt wird, genutzt. In SEE können die Gebäude verschiedene Softwareartefakte, wie zum Beispiel Klassen, Dateien oder Methoden, widerspiegeln. Mithilfe der Dimensionen und der Farbe eines Gebäudes lassen sich dann die verschiedenen Softwaremetriken visualisieren[4]. Beispielsweise könnte ein Gebäude eine Klasse darstellen, indem die Breite des Gebäudes die Anzahl der Methoden zeigt, die Länge des Gebäudes die Anzahl der Attribute und die Höhe die Anzahl der Zeilen Code in der Klasse insgesamt.
In Abbildung 2.3 sieht man eine von SEE erzeugte Stadt des Linux Net Subsystems. Zusätzlich zu den Dimensionen der Gebäude, die hier verschiedene Softwaremetriken darstellen, zeigt die Einfärbung eines Gebäudes, wie viel Code der Klasse, die das Gebäude repräsentiert, aus anderen Klassen kopiert wurde. Die Graphen, die die Gebäude verbinden, zeigen woher Code kopiert wurde.
SEE umfasst zum Zeitpunkt dieser Arbeit drei Funktionen:

- Die Visualisierung von statischen Informationen einer Software.
- Die Visualisierung von mehreren sequentiellen Versionen einer Software und ihren statischen Informationen. So lässt sich die Entwicklungshistorie einer Software zeigen.
- Die Visualisierung des Laufzeitverhaltens einer Software beziehungsweise der dynamischen Aufrufgraphen zur Laufzeit, nachdem die Software ausgeführt wurde.

Im folgenden Kapitel 2.4.1.2 wird die letzte Funktion bezüglich ihrer Entwicklung und der Funktion genauer vorgestellt.

2.4.1.2 Dynamische Aufrufgraphen in SEE

Die Funktion der dynamischen Aufrufgraphen in SEE wurden im Rahmen einer Bachelorthesis von Torben Groß entwickelt. Das Ziel der Arbeit war es, innerhalb von SEE das Laufzeitverhalten von Software unabhängig von Programmiersprachen mit Aufrufgraphen zwischen den Methoden zu visualisieren. Die Arbeit lässt sich in zwei Bereiche unterteilen. In dem einen Teil geht es um die Beschaffung und Speicherung in einem geeigneten Dateiformat von Laufzeitdaten. Der andere Bereich beschäftigt sich mit der Visualisierung dieser Daten und der Implementierung in SEE[22].
Bei der Beschaffung der Daten hat Torben Groß beispielhaft Instrumentierungen von Java und C++ Code entwickelt. Bei Instrumentierungen wird zusätzlicher Code in den eigentlichen Programmcode eingefügt, um Laufzeitdaten und Diagnosedaten zu sammeln. Zur Instrumentierung des Java Codes wurde ein Java Agent verwendet. Ein Java Agent kann auf bereits kompilierten Javacode zugreifen und diesen anpassen[15]. Diesen Agent hat Torben Groß so implementiert, dass er, wenn eine Methode betreten wird, den Methodennamen und die Klasse der Methode aufzeichnet, und auch wenn die Methode wieder verlassen wird, zeichnet der Agent dies auf. Dazu fügt der Agent einfach entsprechenden Javacode zu Beginn und am Ende aller Methoden des Programms ein. In C++ hat Groß ein Makro geschrieben, dass er manuell in alle Funktionen eines zu analysierenden Programms eingefügt hat.
Um die durch die Instrumentierungen erhaltenen Laufzeitinformationen speichern und weiterverwenden zu können, hat Groß das DYN-Format entwickelt. Dies ist in Abbildung 2.4 zu sehen. Die Zahlen auf der linken Seite (unterhalb von „Level“) zeigen die Tiefe des Aufrufstapels, in der sich der Aufruf befindet, und der rechte Text (unter „Linkage.Name“) ist der Name der Methode.

```
"Level" "Linkage.Name"
"0" "int main(int,char **)"
"1" "void dir_a::file_a_01::function_a_01(void)"
"2" "void dir_a_a::file_a_a_01::function_a_a_01(void)"
"3" "void dir_a_b::file_a_b_01::function_a_b_01(void)"
"4" "void dir_a_b::file_a_b_02::function_a_b_02(void)"
"3" "void dir_a_b::file_a_b_02::function_a_b_02(void)"
"2" "void dir_a_b::file_a_b_01::function_a_b_01(void)"
"3" "void dir_a_b::file_a_b_02::function_a_b_02(void)"
"2" "void dir_a_b::file_a_b_02::function_a_b_02(void)"
"1" "void dir_b::file_b_01::function_b_01(void)"
"2" "void dir_b::file_b_02::function_b_02(void)"
"1" "void dir_c::file_c_01::function_c_01(void)"
```

Abbildung 2.4: Aufrufgraphen im DYN-Format. Erzeugt mit einer der beiden Insturmentierungen[22].

Mithilfe eines Parsers, den Groß selbst entwickelt hat, können DYN-Dateien in SEE eingelesen werden und dann visualisiert werden. Bei der Visualsierung können die Methodenaufrufe Schritt für Schritt, ähnlich wie bei einem Step-Through Debugger, angezeigt werden. Diese bestehen aus Kugeln, die zwischen den Gebäuden beziehungsweise den Methoden, die durch die Gebäude repräsentiert werden, fliegen. Durch die Flugrichtung und einen Farbverlauf, der sich an der Viridis-Farbpalette orientiert[42], lässt sich erkennen, welche Methode welche aufruft. Zusätzlich werden die Gebäude aktiver Methoden eingefärbt und mit einer Animation zyklisch vergrößert und verkleinert, sodass sich diese leicht erkennen lassen[22]. In Abbildung 2.5 ist diese Visualisierung zu sehen.

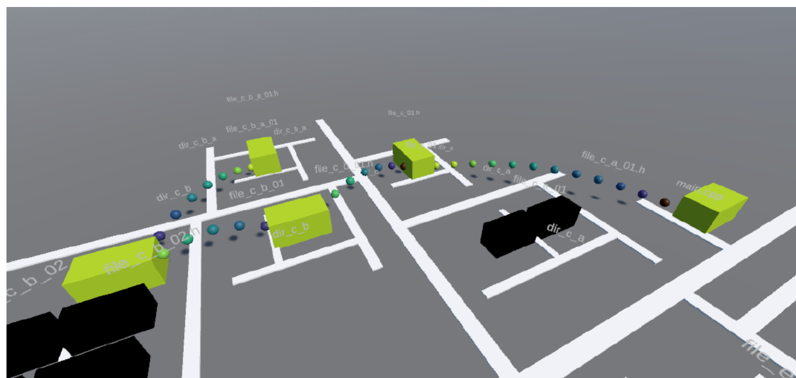


Abbildung 2.5: Visualisierung von Aufrufgraphen in SEE[22].

Abschließend wurde in einer Evaluation getestet, ob Benutzer einer größeren Nutzen aus der Anwendung am Desktop oder in der Virtual Reality ziehen. Dazu haben sich die Probanden das Laufzeitverhalten von zwei verschiedenen Programme am Desktop und in der Virtual Reality angeschaut und die Frage beantwortet, welche Funktion im jeweiligen System am häufigsten aufgerufen wurde. Bewertet wurde ein System daran, wie schnell ein Proband die Frage beantwort hat und ob die Antwort korrekt war. Die Ergebnisse lassen vermuten, dass Benutzer in Virtual Reality besser mit größeren Software-Städten zurechtkommen[22].
Die von Torben Groß implementierte Visualisierung von Aufrufgraphen wird in dieser Arbeit in leicht abgeänderter Form verwendet, um in dem visualisierten Debug-Prozess den Aufrufstapel sichtbar zu machen.

2.5 Game Engine

Eine Game Engine oder Spiel-Engine ist eine Sammlung von Software-Modulen, die zum Entwickeln von Computerspielen verwendet werden. Diese Module umfassen beispielsweise Funktionen zum Rendern von 3D-Grafiken und 2D-Grafiken, Verarbeiten von Eingaben durch Benutzer und Sound-Engines, Animationssysteme und Physik-Engines, mit denen sich das Verhalten von Objekten nach den Regeln der Physik in Spielen leicht implementieren lässt. Der Sinn von Spiel-Engines ist es, Entwickler bei der Computerspielentwicklung optimal zu unterstützen[21, 30].
Game Engines eignen sich aber nicht nur zum Entwickeln von Computerspielen, sondern allgemein zum Entwickeln von graphischen Computeranwendungen. Das in dieser Arbeit verwendete Softwarevisualisierungstool SEE wird mit der Game Engine Unity 3D entwickelt. Diese wird im nächsten Kapitel 2.5.1 vorgestellt.

2.5.1 Unity 3D Engine

Die Unity 3D Engine ist eine Spiel-Engine, die von dem Unternehmen Unity Technologies entwickelt wird. Die Entwicklungsumgebung unterstützt zur Zeit die Betriebssysteme Windows, Linux und macOS und ist in der Lage graphische Anwendungen wie Spiele, Animationen oder Filme für über 25 verschiedene Plattformen zu entwickeln. Zu diesen Plattformen gehören natürlich die gängigen Computer Betriebssysteme Windows, Linux und macOS, aber auch Spielekonsolen, wie die Playstation 4 und die Xbox One, sowie mobile Geräte mit iOS oder Android Betriebssystem[49]. Abbildung 2.6 zeigt die Oberfläche der Unity Engine.

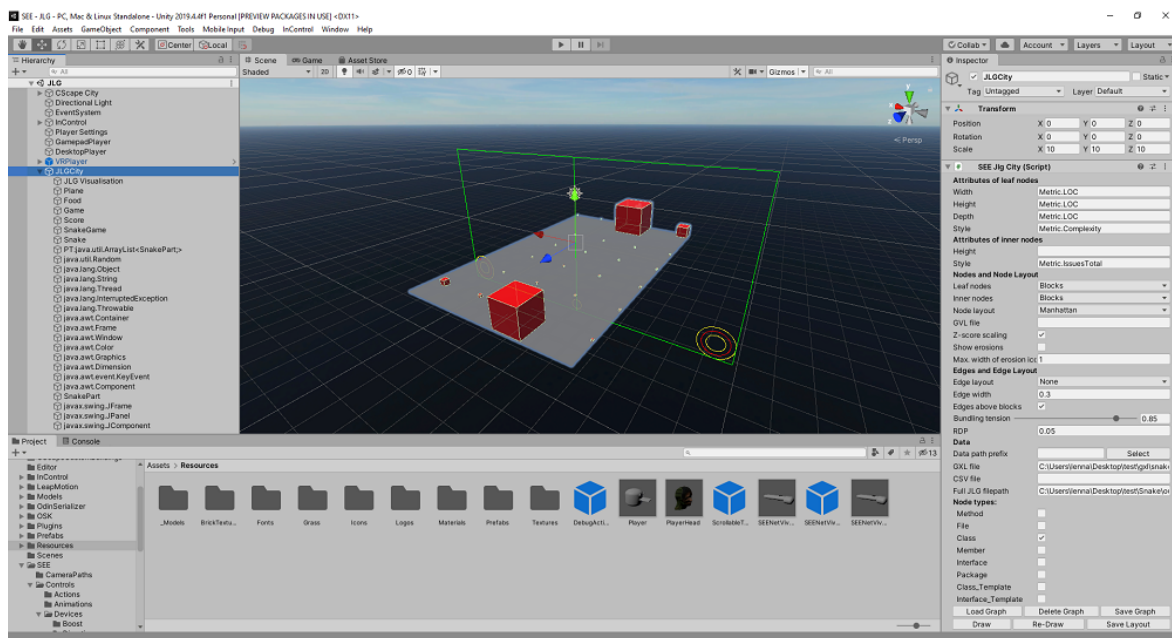


Abbildung 2.6: SEE in der Unity Engine.

In der Mitte der Oberfläche ist eine Szene zu sehen. Mithilfe dieser Szene lässt sich die Welt der Anwendung gestalten. Objekte in der Welt können hier ausgewählt und frei manipuliert werden, was Position, Skalierung und Rotation angeht. Auf der linken Seite sieht man die Hierachie aller Objekte, in Unity „GameObjects“ genannt, die sich in der Welt befinden. Diese Objekte können sowohl sichtbar, als auch unsichtbar sein. An die GameObjects können sogenannte Komponenten angebracht werden. Diese Komponenten beeinflussen zum einen

das Aussehen der GameObjects, und zum anderen das Verhalten. Zu den visuellen Komponenten gehören unter anderem Materialien, 3D und 2D Formen und Effekte. Komponenten wie physikalische Eigenschaften, Klänge und Skripte verleihen den GameObjects verschiedene Verhalten. Besonders sind dabei die Skripte, die von Entwicklern selbst in der Programmiersprache C# implementiert werden. Mithilfe dieser kann ein Entwickler Objekten gewünschte Funktionen oder Verhalten verleihen. Die Komponenten eines Objekts lassen sich mit dem Inspektor betrachten und bearbeiten. Der Inspektor eines Objekts öffnet sich, indem man das Objekt in der Hierachie auf der linken Seite oder in der Szene in der Mitte anklickt und ist auf der rechten Seite in Abbildung 2.6 zu sehen. Zuletzt können die Dateien eines Projekts unten im Engine eigenen Datei-Explorer gesehen werden.

2.6 Eclipse IDE

Zuletzt wird die Eclipse Entwicklungsumgebung vorgestellt. Im Rahmen der Evaluation wird die eingebaute Java Debugging-Funktion von Eclipse als Vergleich zum Werkzeug, das in dieser Arbeit entsteht, verwendet. Warum die Wahl auf die Eclipse IDE gefallen ist, wird in Kapitel 5.1 erklärt.
Eclipse wird seit 2004 von der Eclipse Foundation entwickelt. Vorher wurde Eclipse von Entwicklern bei IBM gebaut. Eclipse wurde ursprünglich als Entwicklungsumgebung für die Programmiersprache Java entwickelt, unterstüzt mittlerweile aber auch die Sprachen C, C++, Javascript, PHP und mehr[18].

2.6.1 Debugging in Eclipse

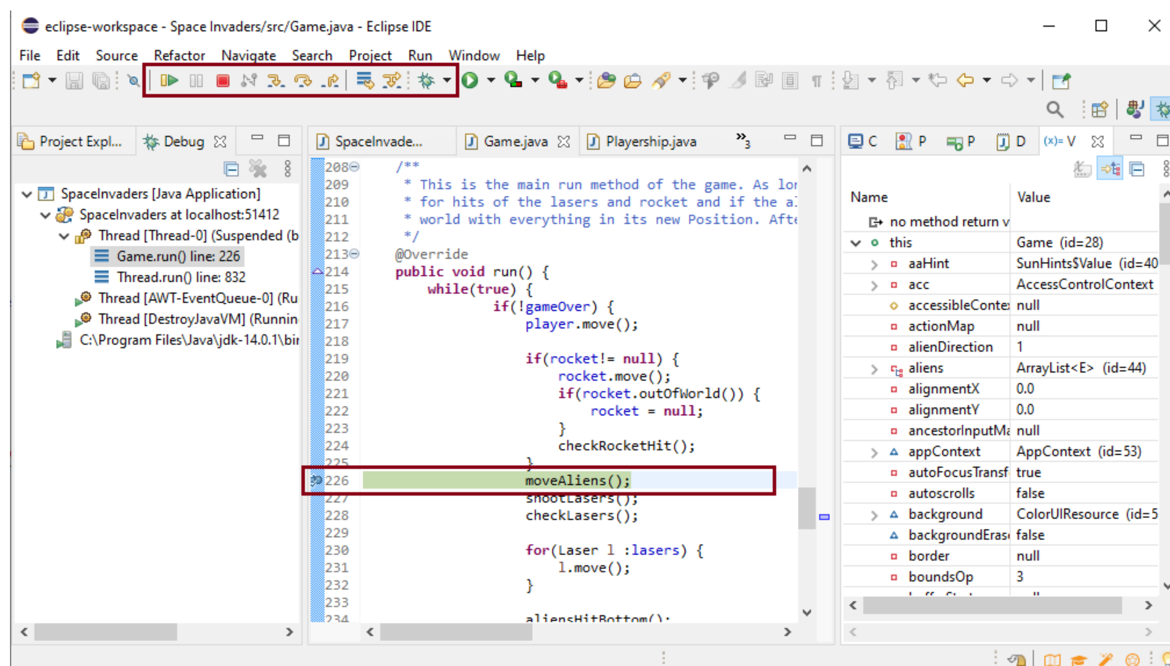


Abbildung 2.7: Oberfläche von Eclipse beim Debuggen eines Java Programms.

Die Eclipse IDE liefert einen integrierten Java Debugger. Dieser entspricht dem Typ eines Step-Through Debuggers, der in Kapitel 2.1.1 genauer vorgestellt wurde. Die Oberfläche des Debugging-Werkzeuges in Eclipse ist in Abbildung 2.7 zu sehen.

In der Mitte der Abbildung sieht man den Java Code des Programms. Hier wurde in Zeile 226 ein Breakpoint gesetzt (rot markiert). Ein Breakpoint lässt sich an dem blauen Kreis neben der Code Zeile erkennen. Im roten Rechteck oben sind die Steuerelemente des Debuggers zu sehen. Mit dem „Käfer“ rechts lässt sich das Programm im Debugmodus starten. Dann kann man das Programm mithilfe der Knöpfe links davon zur Laufzeit pausieren, abspielen und mit den Pfeilen Schritt für Schritt vorwärts durchlaufen. Auf der linken Seite sieht man die aktiven Threads des Programms und auf der rechten Seite sind die Werte der Variablen des Programms zu sehen.

KAPITEL 3

Entwurf

3.1 Anforderungen

Im Rahmen dieser Arbeit werden ein Programm, ein Dateiformat und ein Werkzeug innerhalb von SEE entwickelt: ein Programm zum Aufzeichnen von ausgeführten Java Statements, ein Dateiformat, in dem die Aufzeichnungen gespeichert werden können und ein Werkzeug in SEE, genannt SEE-Debugger, mit dem sich die Aufzeichnungen innerhalb eines visualisierten Debugprozesses in der Softwarestadt anzeigen lassen. Der SEE-Debugger kombiniert die Funktionen eines Step-Through-Debuggers und eines Back in Time Debuggers. Für jedes dieser drei werden nun eigene Anforderungen definiert.

3.1.1 Aufzeichnung von ausgeführtem Java Code

Dieses Programm wird basierend auf vier verschiedenen Anforderungen entworfen, die hier erklärt werden. Die wichtigste Anforderung ist die Vollständigkeit der aufgezeichneten Informationen. Das Programm muss in der Lage sein, alle Laufzeitdaten zu sammeln, die in der Visualisierung gezeigt werden sollen. Welche Informationen das sind, wird in Kapitel 3.1.3 in den Anforderungen an die darzustellenden Informationen erläutert. Eine weitere Anforderung an das Programm ist eine gute Dokumentation. Diese ermöglicht ein einfaches Verständnis der Funktionsweise und begünstigt die nächste Anforderung. Das Programm soll einfach erweiterbar sein, um bei Bedarf leicht weitere Informationen aufzeichnen zu können. Zuletzt soll die Anwendung die Performanz des Zielprogramms so wenig wie möglich beeinflussen. Wie bereits in Kapitel 2.1.1 erklärt, laufen Programme, die aufgezeichnet werden zehn bis hundert mal langsamer[8]. Diese Anforderung ist allerdings weniger wichtig als die Vollständigkeit der Daten, da es in dieser Arbeit primär um die Visualisierung geht.

3.1.2 Dateiformat zur Speicherung der Aufzeichnung

An das Dateiformat wird nur eine Anforderung gestellt. Dafür ist es umso wichtiger, dass diese Anforderung optimal erfüllt ist. Das Dateiformat muss so entwickelt werden, dass der Speicherbedarf der Datei so klein wie möglich wird. Software führt, je nach Art, viele tausende Statements pro Sekunde aus. Versucht man alle ausgeführten Statements mit zusätzlichen Informationen, wie Variablen und ihren Werten, unkomprimitiert in einer Datei zu speichern, gelangt man in kürzester Zeit in den hohen Megabyte- oder sogar in den Gigabyte-Bereich.

3.1.3 Visualisierung der Aufzeichnung in SEE

Die Anforderungen an die Visualisierung in SEE lassen sich nochmal in drei Kategorien aufteilen: den Umfang der darzustellenden Informationen, die Visualisierung selbst und zuletzt die Steuerung beziehungsweise Funktionen dieser. Die Anforderungen werden nun aufgeteilt nach den Kategorien vorgestellt.

Darzustellende Informationen Mithilfe dieser Visualiserung soll es möglich sein, eine Software zu debuggen. Dazu müssen jederzeit Laufzeitinformationen ersichtlich sein, mit denen sich der aktuelle Zustand der Software nachvollziehen lässt. Diese Laufzeitinformationen umfassen die aktuell ausgeführte Zeile Code, die lokalen Variablen, die in dieser Code Zeile relevant sind, die Werte, die in den Feldern des Objekts dieser Zeile gespeichert sind, und die Rückgabewerte der Methoden. Außerdem ist auch der Aufrufgraph wichtig, um den Zustand einer Software nachzuvollziehen. Alle diese Informationen müssen in der Visualisierung dargestellt sein und damit auch vom Aufzeichnungsprogramm festgehalten werden.

Visualisierung Die Visualisierung soll leicht verständlich und übersichtlich gestaltet sein. Außerdem soll es keine Inkonsistenzen geben. Der ausgeführte Quellcode mit den Werten für die lokalen Variablen, Feldern und Rückgabewerten soll immer gut sichtbar und auch lesbar sein. Der Aufrufgraph des Programms muss sowohl anhand der Stadtvisualisierung, als auch am Code erkennbar sein.

Steuerung Die Laufzeitinformationen sollen sich auf zwei Arten ansehen lassen. Die Erste nutzt die Metapher eines Videos. Dabei lässt sich das Laufzeitverhalten eines Programms in der Softwarestadt abspielen. Dieses „Video“ soll genau wie ein Film, den man sich auf Videokasette oder DVD anschaut, gesteuert werden können. Dazu sollen Funktionen implementiert werden, die das Video schneller und langsamer abspielen lassen. Weiter soll es in beide Richtungen, also vorwärts und rückwärts, abgespielt werden können. Diese Art soll sich besonders für das Verständnis der Zusammenhänge zwischen Klassen und Methoden, sowie dem allgemeinen Verständnis der Software eignen.
Zum Debuggen soll noch eine feinfühligere Steuerung implementiert werden, die es dem Entwickler erlaubt, sich Statement für Statement in beide Richtungen durch den ausgeführten Code zu arbeiten. Dazu müssen Funktionen implementiert werden, mit denen sich das „Video“ anhalten und dann schrittweise durch den Code gehen lässt. Da die Aufzeichnungen sehr groß werden können, soll es auch in diesem Werkzeug eine Breakpoint-Funktion geben, mit der man direkt zur gewünschten Code Zeile in der Ausführung springen kann. Als letztes soll es möglich sein, die Visualsierung zum Startpunkt zurückzusetzen.

In diesem Kapitel werden Lösungen für diese Anforderungen entworfen und dabei wird zusätzlich erläutert, inwiefern die Anforderungen erfüllt werden. Die hier entwickelten Entwürfe werden dann in Kapitel 4 implementiert.

3.2 Ausgeführten Code aufzeichnen

3.2.1 Java Debug Interface

Zum Aufzeichnen von Java Code wird das Java Debug Interface verwendet. Das Java Debug Interface, kurz JDI, wird von der Oracle Corporation, die auch für die Entwicklung der

Programmiersprache Java verantwortlich ist, entwickelt[38, 39]. Das JDI ist eine Java Progammierschnittstelle, die Funktionen enthält, mit denen sich Java-Debugger und ähnliche Programme entwickeln lassen. In diesem Kapitel wird diese Programmierschnittstelle beschrieben.
Um ein Java Programm während seiner Laufzeit beobachten zu können, braucht man Zugriff auf dessen Java Virtual Machine. Eine Java Virtual Machine, kurz JVM, ist die Laufzeitumgebung eines Java Programms. Sinn dieser Virtual Machine ist es, Java Programme unabhängig vom Betriebssystem des Computers ausführen lassen zu können. Jedes Programm läuft in seiner eigenen JVM. JDI enthält **Connector** Klassen, die Funktionen enthalten, mit denen es möglich ist, sich mit aktiven JVMs zu verbinden oder ein Programm in einer neuen JVM zu starten und sich dann mit dieser zu verbinden. Hat man sich mithilfe des JDI mit einer Java Virtual Machine verbunden, hat man Zugriff auf den **EventQueue**. Jede Virtual Machine besitzt im Java Debug Interface einen **EventQueue** und einen **EventRequestManager**. Mit dem **EventRequestManager** lassen sich Requests für alle möglichen Ereignisse in der Virtual Machine erstellen. Erfüllt ein Ereigniss in der VM eine Request, wird ein entsprechendes Event auf dem **EventQueue** abgelegt. Die Events auf dem **EventQueue** lassen sich dann einzeln genau betrachten. Je nach Eventtyp enthalten diese verschiedene Laufzeitinformationen, die zum Debugging verwendet werden können[38]. Die für diese Arbeit relevanten Events sind:

ClassPrepareEvent: Dieses Event wird ausgelöst, sobald ein Objekt in der JVM erzeugt wird. Es enthält Informationen darüber, von welcher Klasse ein Objekt erzeugt wird.

MethodEntryEvent: Das **MethodEntryEvent** ist das Ereigniss, wenn ein ausgeführtes Statement in der JVM eine neue Methode betritt. In diesem Event sind Informationen darüber enthalten, welche Methode betreten wurde und der **StackFrame** selbst. Aus dem **StackFrame** lässt sich der aktuelle Zustand des Programms auslesen. Man kann aus dem Zustand die lokalen Variablen und Informationen darüber, in welcher Code Zeile man sich befindet, auslesen.

MethodExitEvent: Das **MethodExitEvent** ist das Gegenstück zum **MethodEntryEvent**. Es wird ausgelöst, wenn ein Statement eine Methode verlässt. Es enthält dieselben Informationen wie das **MethodEntryEvent** und zusätzlich noch die Information, welchen Wert die verlassene Methode zurückgibt.

StepEvent: Dieses Event wird ausgelöst, wenn ein Statement in der JVM ausgeführt wurde. Dieses Event enthält auch den **StackFrame**, aus dem sich Informationen darüber finden lassen, welches Statement in diesem Event ausgeführt wurde und wie der Zustand des Programms zu diesem Zeitpunkt aussieht.

ModificationWatchpointEvent: Das **ModificationWatchpointEvent** wird ausgelöst, wenn der gespeicherte Wert in einem Feld eines Objekts verändert wird. Auch dieses Event enthält wieder den **StackFrame** und die Information, welcher neue Wert in dem Feld gespeichert wird.

Aus diesen Events ist es möglich, alle nötigen Laufzeitinformationen aus der JVM zu extrahieren, die für die Visualisierung benötigt werden. Zu jedem dieser Events existiert eine entsprechende Request. Bis auf die **StepRequest** können von jeder Request mehrere im **EventRequestManager** existieren. Jeder Request können Filter hinzugefügt werden, sodass sie nur bei Events bestimmter Klassen ausgelöst werden[38].
Nun wird mit diesem Interface ein Programm entworfen, das in der Lage ist, ausgeführte Java Statements aufzuzeichnen.

3.2.2 Entwurf eines JDI Java Loggers

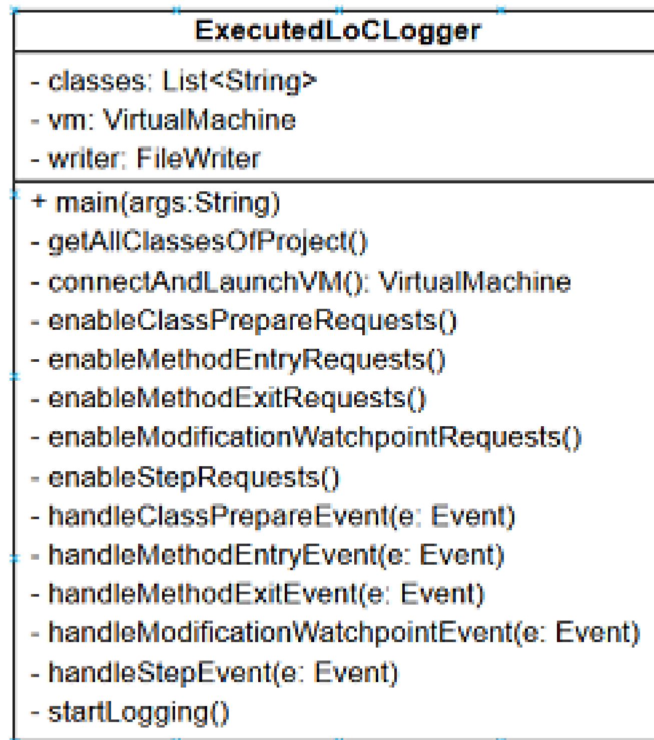


Abbildung 3.1: UML-Diagramm des ExecutedLoCLoggers.

In diesem Kapitel wird ein Logger geplant, der ausgeführten Java Code aufzeichnet und am Ende eine Datei erzeugt, die die Aufzeichnungen enthält. Dazu werden alle benötigten Methoden entworfen und am Ende in einem UML-Diagramm zusammengefasst. Zuerst soll das Programm kompakt und leicht auszuführen sein. Deshalb wird alles innerhalb einer Datei, also einer Klasse, implementiert. Diese Klasse und auch das Programm heißen **ExecutedLoCLogger**, wobei das LoC für Lines of Code steht. Damit das Programm ausgeführt werden kann, benötigt es zuerst eine **main()**-Methode. Als nächstes müssen mit einer Methode alle Klassen des Projekts gesammelt werden. Dies ist wichtig, um später die Filter der Requests setzen zu können, sodass nur Events von Objekten relevanter Klassen auf dem **EventQueue** landen. Das Ergebnis der Methode wird in einer Liste von Strings in der Klasse gespeichert. Dann muss eine Verbindung zur Ziel Java Virtual Machine aufgebaut werden. Da das komplette Laufzeitverhalten vom Start eines Programms bis zum Ende aufgezeichnet werden soll, wird in dieser Methode die Virtual Machine zusammen mit dem Zielprogramm gestartet. Ergebnis der Methode ist eine Referenz auf die gestartete Virtual Machine in Form eines **VirtualMachine** Objekts. Die **VirtualMachine** Klasse stammt aus dem JDI. Diese Referenz wird in einem Feld der Klasse **ExecutedLoCLogger** gespeichert.
Ist eine Verbindung zur Ziel Virtual Machine hergestellt, müssen Requests für die zu beobachtenden Events erstellt werden. In diesem Fall reichen die in Kapitel 3.2.1 vorgestellten Events. Für jedes dieser Events wird eine Methode erstellt, die entsprechende Requests in der Virtual Machine hinzufügt. Um die von den Requests erzeugten Events zu verarbeiten,, werden weitere Methoden benötigt. Dies sind die sogenannten **HandleEvent()**-Methoden.
In einer Hauptmethode werden nun die vorher erstellten Methoden verwendet, um den eigentlichen Log-Prozess durchzuführen. Diese Methode wird **startLogging()** genannt und wird von der **main()**-Methode aufgerufen. Innerhalb dieser Methode wird noch ein sogenannter **Writer** erstellt und in der Klasse gespeichert. Dieser **Writer** wird verwendet, um die Laufzei-

tinformationen aus den Events in eine Datei zu schreiben. Das Ergebnis dieses Entwurfs ist in Abbildung 3.1 zu sehen. Die Abbildung zeigt das UML-Diagramm des **ExecutedLoCLogger** ohne Getter- und Setter-Methoden und Hilfsmethoden.
Durch die Kombination der **MethodeEntry**- und **MethodExitEvents** und der **StepEvents**, ist es möglich, sowohl jedes ausgeführte Java Statement aufzuzeichnen, als auch den Aufrufgraphen nachzuvollziehen, da man genau weiß, wann welche Methode betreten und wann welche verlassen wird. Zusätzlich hat jedes dieser Events Zugriff auf den **StackFrame**, mit dem sich die Werte der lokalen Variablen und Rückgabewerte von Methoden aufzeichnen lassen. Durch die **ModificationWatchpointEvents** lassen sich zusätzlich Veränderungen in den Feldern eines Objekts aufzeichnen. Somit sind die Anforderungen an die Daten aus Kapitel 3.1.3 erfüllt. Weiter ist das Java Debug Interface von Oracle vollständig dokumentiert[38]. Auch die Erweiterbarkeit ist gegeben. JDI unterstützt noch weitere Eventtypen, die einfach mit einer Request-Methode und einer Handle-Methode in den Logger eingefügt werden können. Die Performanz hängt allerdings hauptsächlich von der Implementierung der JDI Funktionen ab. Dennoch wird in Kapitel 4.1 beschrieben, wie an verschiedenen Stellen der Implementierung die Auswirkung des Loggers auf die Performanz der Zielsoftware verringert wird.

3.3 Log Dateiformat

An das Dateiformat, in dem die Aufzeichnungen gespeichert werden sollen, gibt es eine Anforderung. Der Speicherbedarf der Datei soll möglichst gering sein. Dies wird mit zwei verschiedenen Methoden erreicht: einer Lookup-Tabelle und einem Schlüssel, mit dem die Art einer Informationen identifiziert werden kann. Das Dateiformat wird JLG-Format genannt, was für „Java-Log“ steht.
In einer Lookup-Tabelle werden lange Texte, Wörter oder ähnliches, die in einer Datei immer wieder vorkommen, einem kurzen eindeutigen Ausdruck zugeordnet. Dann kann in der Datei anstelle des langen Textes der Ausdruck verwendet werden, um Speicher zu sparen. Im Nachhinein kann dann mit der Lookup-Tabelle der Ausdruck wieder durch den ursprünglichen Text ersetzt werden. Im Rahmen des Logs lohnt es sich, die Methodennamen und die Namen der Felder der Klassen in einer Lookup-Tabelle zu speichern, da diese meist lang sind und sich häufig wiederholen. In Listing 3.1 sieht man ein Beispiel einer Lookup-Tabelle für Methodennamen und Feldnamen einer JLG-Datei. Die Methodennamen werden durch die Kombination aus einem „-“ und einer Zahl abgekürzt und die Feld-Namen aus einer Kombination von einer „#“ und einer Zahl. Wofür die Zeichen vor den Zahlen und das „*“ vor der Tabelle stehen, wird jetzt erläutert.

Listing 3.1: Beispiel einer Lookup-Tabelle in einer JLG-Datei.

```
*-0=Main.main(java.lang.String[]);-1=Main.options();-2=UnterOrdner.
    ↪ UnterOrdnerTest.<init>();-3=UnterOrdner.UnterOrdnerTest.printTest()
    ↪ ;-4=CountConsonants.<init>();-5=CountConsonants.countConsonants(java.
    ↪ lang.String);-6=Main.afterfunction();#0=consonants;
```

In der Aufzeichnung werden mehrere Arten von Daten gespeichert. Um diese voneinander unterscheiden zu können, werden Identifzierer festgelegt. Diese stehen in der Aufzeichnung am Anfang einer Zeile und lassen erkennen, welche Informationen in dieser Zeile gespeichert sind. Es wird für jedes der Events und die Daten, die dazugehören, ein Identifikator festgelegt:

- „-...>...“. Der - indiziert ein neues ausgeführtes Statement. Nach dem „-“ kommt ein

Wert aus der Lookup-Tabelle, der die Methode, in der die Zeile steht, repräsentiert. Nach dem „>“ kommt dann die Zahl der Zeile.

  - Betritt ein Statement eine Methode, wird anstelle von „-“ „-/“ verwendet.
  - Verlässt ein Statement eine Methode, wird anstelle von „-“ „/-“ verwendet.

- „=>“. Die Kombination der Zeichen „=>“ am Anfang einer Zeile zeigen den Rückgabewert des letzten Statements, das eine Methode verlassen hat.

- „#“. Ist dieses Zeichen am Anfang einer Zeile, steht in dieser eine Veränderung eines Feldwertes.

- „$“. Nach diesem Zeichen kommen die Pfade zu allen Klassendateien des Zielprogramms.

- „*“. Dieses Zeichen markiert den Start der Lookup-Tabellen, die immer am Ende der Log-Datei stehen.

- Ist kein Zeichen an der ersten Stelle einer Zeile, so steht in dieser Zeile der Wert einer lokalen Variable, die zum letzten ausgeführten Statement gehört. Für diese Information wurde kein Zeichen als Identifzierer gewählt, da es die meisten Zeilen der Log-Datei belegt und man so weiteren Speicher sparen kann.

Listing 3.2: Ausschnitt einer JLG-Datei.

```
-4>20
-/5>14
y=250
x=250
-5>15
y=250
x=250
#6=250
-5>16
y=250
x=250
#7=250
-5>17
y=250
x=250
/-5>17
=><void value>
-4>20
```

In Listing 3.2 sieht man einen Ausschnitt aus einer JLG-Datei mit den verschiedenen Identfizierern. Mit Hilfe der in diesem Kapitel vorgestellten Methoden konnte der Speicherbedarf einer JLG-Datei ohne Informationsverlust wesentlich reduziert werden.

3.4 Visualiserung in SEE

3.4.1 Daten

Für die Visualisierung müssen die in einer JLG-Datei gespeicherten Laufzeitdaten in die SEE Anwendung beziehungsweise Unity geladen werden. Dafür muss ein Parser entwickelt werden. Ein Parser ist in der Softwarentwicklung ein Programm, das eine Eingabe in einen Zustand verarbeitet, der von einem anderen Programm genutzt werden kann. Der JLG-Parser benötigt grundsätzlich zwei Funktionen: eine, mit der die JLG-Datei eingelesen wird, und eine, die diese verarbeitet. Damit die Daten in Unity weiter verwendet werden können, müssen sie in Datenklassen gespeichert werden. Datenklassen bestehen aus Feldern, in denen Daten gespeichert werden können, und den dazugehörigen Getter- und Setter-Methoden.
Für den JLG-Parser werden zwei Datenklassen entwickelt: eine, die ein einzelnes Java Statement speichert, und eine, die eine ganze Log-Datei repräsentiert. Ein Java Statement besteht aus den Lokalisationsdaten der Code Zeile, also Methodenname und Nummer der Zeile, lokalen Variablen und Informationen darüber, ob die Zeile einen Feldwert eines Objektes verändert und ob die Zeile einen Wert zurückgibt. Weiter kann ein Java Statement eine Methode betreten oder verlassen. Diese Informationen über eine ausgeführte Code Zeile werden in den Feldern einer Java Statement Datenklasse gespeichert.
Die zweite Datenklasse ist die **ParsedJLG** Klasse. In ein Objekt dieser Klasse wird eine ganze Log-Datei vom Parser gespeichert. Die Klasse besteht aus einer Liste, in der alle ausgeführten Statements einer Log-Datei als Objekte der Java Statement Datenklasse gespeichert werden. In zwei weiteren Listen werden die Dateipfade zu allen Klassen des aufgezeichneten Programms und die Lookup-Tabellen gespeichert.
Das UML-Diagramm zu diesem Parser- und Datenklassenentwurf ist in Abbildung 3.2 zu sehen. Der Parser wird in Kapitel 4.2.1 genauer eklärt.

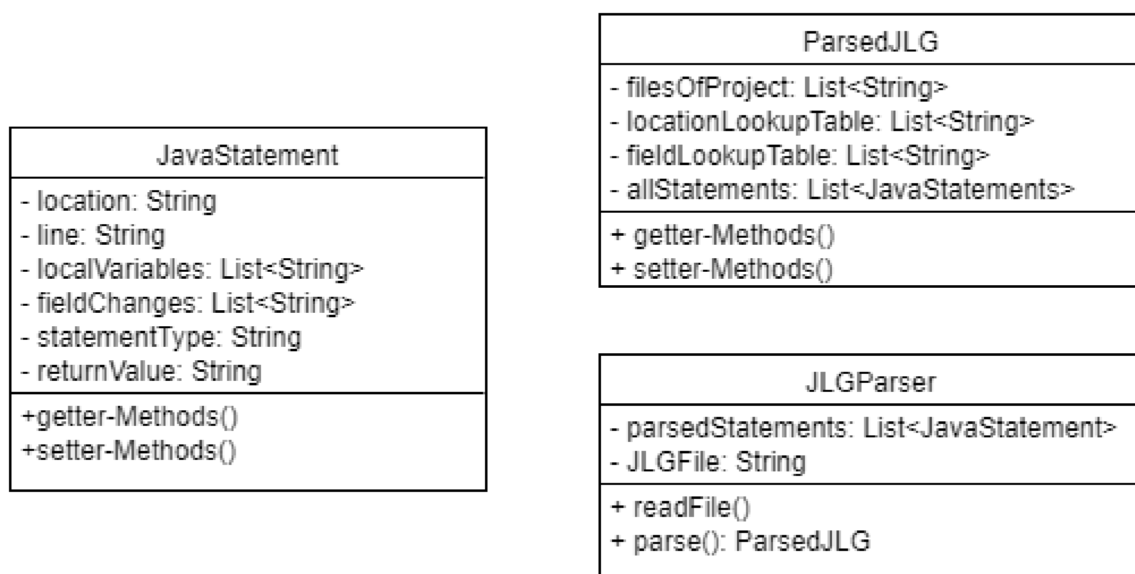


Abbildung 3.2: UML-Diagramm der JLG-Datenklassen und -Parser.

3.4.2 Visualisierung

Der Entwurf der Visualisierung lässt sich wieder in drei Teile aufteilen: die Visualisierung der ausgeführten Code Zeilen, der Laufzeitdaten, also den lokalen Variablen, Veränderungen von Feldwerten und Rückgabewerte und die des Aufrufgraphens.

Ausgeführte Code Zeilen Es soll jederzeit ersichtlich sein, welche Code Zeile zuletzt ausgeführt wurde. Um dies zu erreichen, wird mittig über der Stadt ein großes Textfenster platziert. In diesem Textfenster wird der Code einer Klasse angezeigt. Der Text der letzten ausgeführten Zeile Code wird in einem kräftigen Grünton gefärbt. Die vorherigen Statements verblassen in fünf Schritten zurück zu weiß. So lassen sich jederzeit die letzten fünf ausgeführten Zeilen und ihre Reihenfolge erkennen. Dieser Farbverlauf lässt sich in Abbildung 3.3 erkennen. Hier sieht man, dass die Zeilen 24, 25, 26, 27 und 30 in der Reihenfolge ausgeführt wurden.

```
23.     public void reposition() {
24.             x= r.nextInt(490);
25.             y= r.nextInt(490);
26.             int i = r.nextInt(2);
27.             if(i == 1) {
28.                     c =Color.RED;
29.             }else {
30.                     c = Color.ORANGE;
```

Abbildung 3.3: Ausschnitt eines Textfenster einer Klasse in SEE. An den grün eingefärbten Zeilen erkennt man, welche zuletzt ausgeführt wurden.

Laufzeitdaten Die Laufzeitdaten, die bei der letzten ausgeführten Code Zeile verfügbar waren, werden in einem extra Textfenster gezeigt. Dieses befindet sich direkt neben dem großen Textfenster, in dem der Code zu sehen ist. Hier werden lokale Variablen, Feldveränderungen und Rückgabewerte angezeigt. Abbildung 3.4 zeigt zwei Ausschnitte der Laufzeitdaten des Codes aus Abbildung 3.3.

```
Line 25

Field Changes at this line:
y=323
Last Return: Score.<init>() returned <void value>
```

```
Line 30
Local variables accessible at this line:
i=0
Field Changes at this line:
c=class java.awt.Color (no class loader)
Last Return: Score.<init>() returned <void value>
```

Abbildung 3.4: Laufzeitdaten zu den Code Zeilen 25 und 30 aus Abbildung 3.3. Diese erscheinen in einem weiteren kleinen Textfenster.

Aufrufgraphen Um den Aufrufgraphen in der Stadt zu visualisieren, werden die Funktionen der Arbeit von Torben Groß verwendet[22]. Betritt ein Statement eine neue Methode

in einer neuen Klasse, wird der Aufruf durch Kugeln, die zwischen den beiden Gebäuden fliegen, gezeigt. Diese werden mit der Funktion erzeugt, die Groß dafür implementiert hat. Der Animationsloop, in dem die Farben und Größen der Gebäude verändert werden, wird nicht verwendet. Stattdessen werden alle Klassen, die sich im Aufrufgraphen befinden, hellblau eingefärbt und die Klasse, in der Code ausgeführt wird, dunkelblau. In Abbildung 2.5 ist die Visualisierung von Aufrufgraphen von Torben Groß zu sehen und in Abbildung 3.5 die Verwendung der Aufrufgraphen im SEE-Debugger.

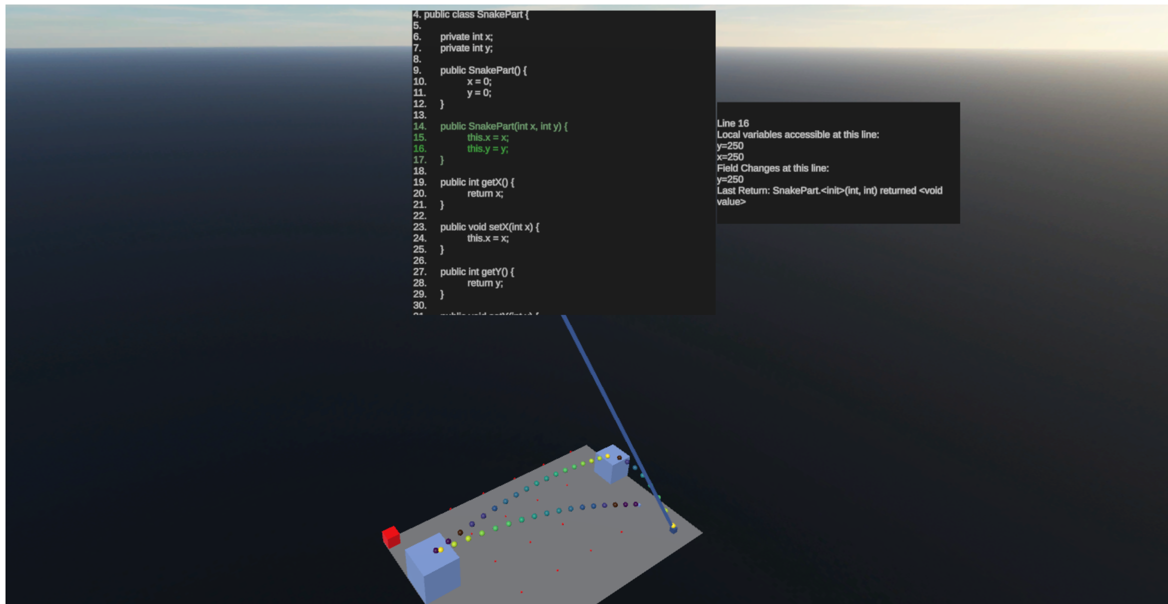


Abbildung 3.5: Ausschnitt eines visualisierten Debug-Prozesses in SEE.

3.4.3 Steuerung

Die Bewegung durch die Spielwelt wird schon von SEE implementiert und basiert auf dem beliebten WASD-Steuerungsmodell, das in sehr vielen Computerspielen verwendet wird. Die Tastenbelegung zur Bewegung ist:

Taste	Effekt
W oder Pfeiltaste oben	Vorwärts bewegen
S oder Pfeiltaste unten	Rückwärts bewegen
A oder Pfeiltaste links	Nach links bewegen
D oder Pfeiltaste rechts	Nach rechts bewegen
E	Nach oben bewegen
Q	Nach unten bewegen
Shift	Schneller bewegen

Tabelle 3.1: Bewegungssteuerung in SEE. Die Bewegungsrichtung ist abhängig von der Blickrichtung.

Die Blickrichtung lässt sich durch das „Gedrückthalten“ der rechten Maustaste und bewegen der Maus steuern.
Die Steuerung der Visualisierung lässt sich nach den Anforderungen in zwei Modi unterteilen:

einen automatischen und einen manuellen. Zwischen den Modi wechselt man mit der Taste „F“. Im automatischen Modus wird die Ausführung des Programms wie ein Video abgespielt. Die Abspielgeschwindigkeit und -richtung sollen angepasst werden können. Daraus ergibt sich für den Modus folgende Steuerung:

Taste	Effekt
1	Abspielgeschwindigkeit halbieren
3	Abspielgeschwindigkeit verdoppeln
R	Abspielrichtung wechseln
F	in den manuellen Modus wechseln

Tabelle 3.2: Steuerung der automatischen Visualisierung.

Im manuellen Modus steuert der Benutzer den Ablauf der Statements in der Visualisierung selbst. Außerdem soll er die Möglichkeit haben, per Tastendruck zu einem vorher gesetzten Breakpoint springen zu können. Der manuelle Modus wird gesteuert mit:

Taste	Effekt
1	Ein Statement zurück
3	Ein Statement vor
B	Zum nächsten Breakpoint springen
N	Zurück zum Anfang springen
F	in den automatischen Modus wechseln

Tabelle 3.3: Steuerung der automatischen Visualisierung.

KAPITEL 4

Implementation

In diesem Kapitel wird die Implementierung des **ExecutedLoCLoggers** und des **SEE-Debuggers** vorgestellt. Dazu wird auf die wichtigsten Methoden beider Programme eingegangen. Der komplette **ExecutedLoCLogger** wird in der Programmiersprache Java entwickelt.

4.1 ExecutedLoCLogger

4.1.1 Vorbereitende Methoden

Zunächst werden zwei Methoden vorgestellt, die noch vor dem Log-Prozess ausgeführt werden. Diese sammeln für den Prozess nötige Informationen und bereiten die Umgebung, in der Java Code aufgezeichnet werden kann, vor.

4.1.1.1 Klassen des Programms

Die Methode **getAllClassesOfProject()** findet alle Java-Klassen, die zum aufzuzeichnenden Programm gehören. Dazu durchsucht eine rekursive Methode den Projektordner nach Dateien mit einer „.java"-Endung. Diese Methode überprüft, beginnend mit dem Projektordner, für jedes Element in diesem, ob es ein Ordner ist. Wenn ein Element ein Ordner ist, wird die Methode auf jedes Element, das wiederum in diesem Ordner enthalten ist, aufgerufen. Findet die Methode eine Datei, die eine „.java"-Endung hat, handelt es sich um eine Java Klasse und der Dateiname wird mit dem vollständigen Dateipfad in einer Liste gespeichert. Diese Liste wird verwendet, um die Filter der Requests auf die Klassen des Projekts zu setzen und um später in SEE den Inhalt der Klassen laden zu können.

4.1.1.2 Verbindung und Start der JVM

Wie in Kaptiel 3.2.1 beschrieben, benötigt man Zugriff auf die Java Virtual Machine des Programms, das man aufzeichnen möchte. Der **ExecutedLoCLogger** erreicht das mit der **connectAndLaunchVM()**-Methode. In der Methode wird ein Objekt der Klasse **LaunchingConnector** aus dem JDI erzeugt. Diesem Objekt übergibt man den Namen der Klasse des Projekts, die die **Main()**-Methode enthält. Dann kann mit der **launch()**-Methode des LaunchingConnectors das Zielprogramm gestartet werden. Die **launch()**-Methode gibt ein **VirtualMachine** Objekt zurück, das die Java Virtual Machine repräsentiert, in der das gestartete Programm läuft. Mit diesem Objekt kann nun auf die JVM zugegriffen werden.

4.1.2 Requests und Eventhandler

4.1.2.1 Requests

Listing 4.1: Erstelllung einer Request im Allgemeinen.

```
1 private void createAndEnableEventRequest(VirtualMachine targetVM){
2 EventRequestManager evm = vm.eventRequestManager();
3 EventRequest er = evm.createEventRequest();
4 er.addClassFilter(classname);
5 er.enable();
6 }
```

Das **VirtualMachine** Objekt besitzt einen **EventRequestManager**, mit dem sich Requests für alle möglichen Ereignisse in der JVM erstellen lassen. Grundsätzlich werden alle Requests, unabhängig vom Eventtypen, auf die gleiche Weise erstellt. Das Listing 4.1 zeigt in Code, wie eine Request erstellt wird. Um diesen Code für ein spezielles Event anzupassen, muss nur das „Event“ an allen Stellen durch den Namen des gewünschten Events, zum Beispiel „MethodEntry“, ersetzt werden.
Zuerst werden die Requests für die **ClassPrepareEvents**, die **MethodEntryEvents** und die **MethodExitEvents** erzeugt. Da es für diese Events keine Begrenzung in der Anzahl im **EventRequestManager** gibt, kann für jede Klasse des Projekts eine Request für jedes der drei Events erstellt werden. Dazu wird in den drei Methoden zur Erstellung der Requests eine Schleife eingebaut, die über die Liste der Klassen des Zielprojekts aus Kapitel 4.1.1.1 iteriert und für jede Klasse in der Liste eine eigene Request mit entsprechendem Filter in den **EventRequestManager** hinzufügt. Durch das Verwenden der Filter werden ausschließlich die selbst implementierten Klassen des Projekts beobachtet. Primitive Java Klassen, wie zum Beispiel **String** oder **Integer**, erzeugen so keine Events, was die Auswirkung auf die Laufzeit des Zielprogramms verringert.
Die Erstellung der **ModificationWatchpointRequests** und **StepRequests** weicht davon leicht ab. Für die Erstellung einer **ModificationWatchpointRequest** benötigt man zusätzlich noch eine Referenz zu dem Feld, das in der Request beobachtet werden soll. Deshalb werden die **ModificationWatchpointRequests** erst in der **handleClassPrepareEvent()**-Methode angefertigt. In dieser Methode hat man Zugriff auf ein Event, in dem ein neues Objekt in der VM erzeugt wurde. Mit dem Event hat man die Referenz auf das Objekt und damit die Referenz auf alle Felder. Dann kann für jedes Feld eine **ModificationWatchpointRequest** gebaut werden.
Von einer **StepRequest** kann es nur eine aktive Request pro Thread in der VM geben. Aus diesem Grund kann nicht für jede Klasse des Projekts eine eigene **StepRequest** existieren, wie zum Beispiel bei den **ClassPrepareRequests**. Deshalb werden in Handlern für die **MethodEntry-** und **MethodExitEvents** dynamisch **StepRequests** für die Klasse des in der VM aktiven Objekts erstellt. Dies wird in Kapitel 4.1.2.2 genauer erläutert.

4.1.2.2 Events

Durch die in Kapitel 4.1.2.1 erstellten Requests landen entsprechende Events auf dem **EventQueue** der VM. Diese Events müssen noch verarbeitet werden. Dazu wird für jedes der fünf Eventtypen eine **handleEvent()**-Methode geschrieben.

handleClassPrepareEvent() Das **ClassPrepareEvent** wird ausgelöst, wenn in der VM ein neues Objekt einer Klasse erzeugt wird. Über das Event kann auf das neu erzeugte Objekt zugegriffen werden. Die **handleClassPrepareEvent()**-Methode erzeugt für jedes Feld des Objekts aus dem Event eine **ModificationWatchpointRequest**.

handleModificationWatchpointEvent() Ein **ModificationWatchpointEvent** wird ausgelöst, wenn der Wert eines durch eine Request beobachteten Feld verändert wird. In dieser Methode wird aus dem Event gelesen, welches Feld zu welchem Wert verändert wird. Diese Information wird dann von der Methode in die Log-Datei geschrieben. Zusätzlich wird hier die **FieldLookup**-Tabelle mit den Namen der Felder gefüllt. In der Ausgabe in die Log-Datei wird dann der Index des Feldnamens in der Tabelle verwendet.

handleMethodEntryEvent() Diese Methode verarbeitet **MethodEntryEvents**. Ein **MethodEntryEvent** beinhaltet ein ausgeführtes Java Statement, dass eine neue Methode betritt. Die Ausführung diese Statements wird in der Log-Datei festgehalten, indem diese Methode den Namen der betretenen Methode, die Zeilennummer des Statements und alle lokalen Variablen an dieser Stelle in Datei schreibt. Für den Namen der Methode wird, wie bei den Feldnamen, der Index genommen, an dem der Name in der **MethodLookup**-Tabelle steht. Existiert der Name noch nicht in der Tabelle, wird er hinzugefügt. Zusätzlich erstellt und aktiviert die Methode eine **StepRequest** für die Klasse der betretenen Methode, sodass auch der ausgeführte Code innerhalb der Methode aufgezeichnet wird. Dazu wird die vorherige **StepRequest** aus dem **EventRequestManager** gelöscht und eine neue wird eingefügt. Außerdem wird der Methodenaufruf auf einen **Stack** abgelegt. Dieser Stack repräsentiert den Aufrufgraphen und ist in der **handleMethodExitEvent()**-Methode wichtig.

handleMethodExitEvent Diese Methode wird aufgerufen, wenn in der VM ein Statement eine Methode verlässt. Zuerst löscht die Methode die **StepRequest** der Methode, die verlassen wurde, und entfernt ihren Aufruf vom Aufrufgraphenstack. Dann wird die Ausführung des Statements mit dem Index der Methode aus der Lookup-Tabelle, der Nummer der Code Zeile und dem Rückgabewert der Methode in der Log-Datei gespeichert. Abschließend wird dann noch eine neue **StepRequest** für die Methode, die oben auf dem Aufrufgraphen liegt, erstellt. Durch das dynamische Erstellen der **StepRequests** innerhalb der **handleMethodEntry()**- und **handleMethodExit()**-Methoden werden nur **StepEvents** der eigenen Klassen des Projekts ausgelöst. Dies reduziert die Anzahl der **StepEvents** drastisch und verringert damit weiter die Auswirkung auf die Laufzeit des aufgezeichneten Programms.

handleStepEvent Zuletzt müssen noch die **StepEvents** bearbeitet werden. Das erledigt diese Methode durch Aufzeichnen der Lokalisierungsdaten eines Statements und der dazugehörigen Laufzeitdaten in einer Log-Datei.

4.1.3 Log-Methode

Die bisher entwickelten Methoden können nun zu einer Methode zusammengefügt werden, die den gesamten Log-Prozess steuert. Diese Methode heißt **startLogging()** und wird von der **main()**-Methode gestartet. Abbildung 4.1 zeigt das dazugehörige Aktivitätsdiagramm. Die Funktionsweise dieser Log-Methode ist simpel. Bevor der eigentliche Log-Prozess beginnt, werden die vorbereitenden Methoden **getAllClassesOfProject()** und **connectAndLaunchVM()** ausgeführt.

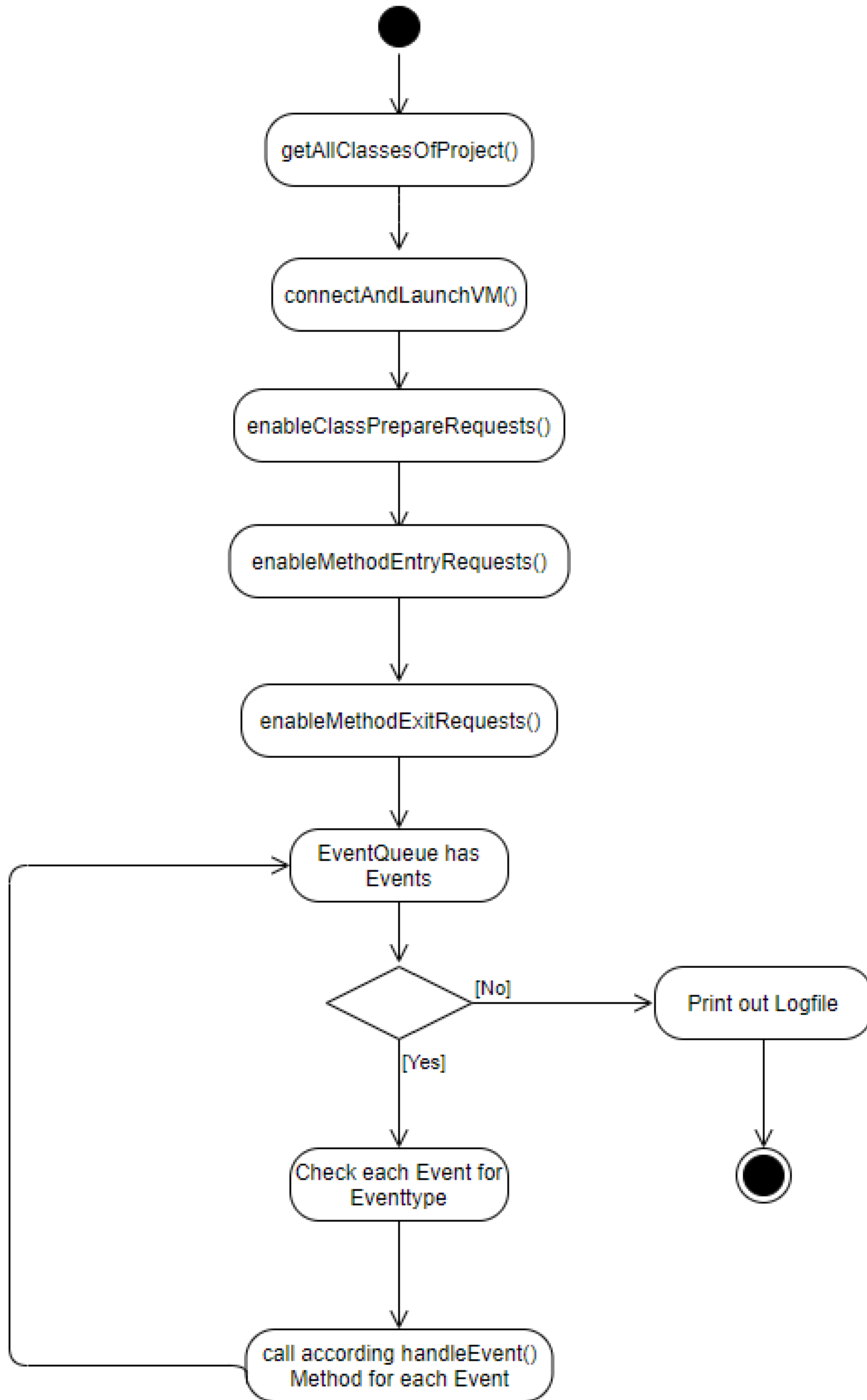


Abbildung 4.1: Vereinfachtes Aktivitätsdiagramm der Logging-Methode des **Executed-LoCLoggers**.

Dann können die **ClassPrepare-**, **MethodEntry-** und **MethodExitRequests** mit dem **EventRequestManager** der VM erstellt werden. Mit den nun existierenden Requests beginnen sich Events im **EventQueue** zu sammeln. An dieser Stelle beginnt der Log-Prozess. Solange der **EventQueue** Events enthält, werden diese von den entsprechenden **handleEvent()**-Methoden verarbeitet. Das laufende Programm versorgt den **EventQueue** mit neuen Events, bis das Programm gestoppt wird. Ist der **EventQueue** leer und das Programm gestoppt, wird die erzeugte Log-Datei im JLG-Format ausgegeben und damit endet das **ExecutedLoCLogger** Programm.
Trotz der in Kapitel 4.1.2.2 erklärten Optimierungen zur Minimierung der zu verarbeitenden Events ist die Aufwirkung der Aufzeichnung auf das Programm spürbar. Wie bereits in Kapitel 2.1.1 erwähnt, laufen aufgezeichnete Programme zehn bis hundert Mal langsamer[8]. Auch der **ExecutedLoCLogger** bewegt sich in diesem Bereich. Die Auswirkung hängt davon ab, wie viele Statements das aufgezeichnete Programm pro Sekunde ausführt. Umso mehr Statements ein Programm pro Sekunde ausführt, desto stärker werden die Auswirkungen.

4.2 SEE-Debugger

Im Folgenden wird die Implementierung des Parsers und der wichtigsten Methoden der Visualisierung und Steuerung des SEE-Debuggers erklärt. Alles, was in diesem Kapitel implementiert wird, wird mit der Programmiersprache C# geschrieben.

4.2.1 Daten

Der erste Schritt zur Visualisierung der Laufzeitdaten in SEE ist das Einlesen der JLG-Datei. Dazu muss ein Parser geschrieben werden, der das Log einliest und daraus verwendbare Datenobjekte erzeugt. Die Datenklassen **JavaStatement** und **ParsedJLG** wurden, genau wie im UML-Diagramm in Abbildung 3.2 beschrieben, implementiert. Die Hauptmethode des Parses ist als Pseudo-Code in Listing 4.2 beschrieben. Die Methode liest die JLG-Datei mit der **ReadLines()**-Methode der C# **File** Klasse ein. Diese Methode ermöglicht es, eine Datei Zeile für Zeile einzulesen, was den Ressourcenverbrauch des Parsers entlastet. Dadurch kann jede Zeile der JLG-Datei durch eine **foreach()**-Schleife einzeln geparsed werden, bis das Ende der Datei erreicht wird. Die Ergebnisse des Parsing-Vorgangs sind eine Liste mit Objekten aller Java Statements in der JLG-Datei, eine Liste mit den Dateipfaden zu allen Java Klassen des Projekts, eine **LocationLookup**-Tabelle und eine **FieldLookup**-Tabelle, die alle in einem **ParsedJLG**-Objekt zusammengefasst werden. Dieses **ParsedJLG**-Objekt ist der Rückgabewert der Methode. Die **Parse...(line)**-Methoden entfernen die Indentifikatoren der Zeilen, sodass nur noch die reine Information übrig bleibt.
Interessant ist dabei vor allem, wie ein **JavaStatement**-Objekt über mehrere Zeilen hinweg zusammengesetzt wird. Beginnt eine Zeile mit einem der Indikatoren „-/“, „-“ oder „/-“, die für ein neues Statement stehen, wird das vorher entstandene **JavaStatement**-Objekt in die Liste, in der alle geparsten Statements gespeichert werden, eingefügt und ein neues Statement erzeugt. Die in den nächsten Zeilen folgenden Laufzeitinformationen, also Methodenname, Zeile des Statements, lokale Variablen, Feldwertveränderungen und Rückgabewerte, werden in diesem neuen **JavaStatement**-Objekt gespeichert. Dieses Objekt wird dann wieder beim nächsten Auftreten der Indikatoren in der **allStatements**-Liste gespeichert und wieder wird ein neues **JavaStatement**-Objekt für das nächste Statement in der Log-Datei erzeugt. Auf diese Weise werden alle in dem aufgezeichneten Programm ausgeführten Statements zu **JavaStatement**-Objekten in der SEE Anwendung.
Das **ParsedJLG**-Objekt, das am Ende dieser Methode entsteht, kann von einem Unity-

Skript verwendet werden, um den in der geparsten JLG-Datei enthaltenen Programmablauf in SEE zu visualisieren. Die Implementation dieses Skripts wird im folgenden Kapitel 4.2.2 vorgestellt.

Listing 4.2: Pseudo-Code des JLGParsers.

```
1  public ParsedJLG Parse(){
2     List allStatements = new List();
3     JavaStatement javaStatement = new JavaStatement();
4     for each line in File.ReadLines(JLG-File)){
5         if (line.StartsWith("-/") or StartsWith("-") or StartsWith("/-")){
6                 allStatements.Add(javaStatement);
7                 javaStatement = new JavaStatement();
8                 javaStatement.SetLocation(ParseLocation(line));
9                 javaStatement.SetLine(ParseLineNumber(line));
10                javaStatement.SetStatementType() depending on the Symbol the
                      ↪ line starts with;
11        } else if (line.StartsWith("#"){
12                javaStatement.GetFieldChanges().Add(ParseFieldChanges(line));
13        }else if (line.StartsWith("=>"){
14                javaStatement.setReturnValue(ParseReturnValue(line));
15        }else if (line.StartsWith("$"){
16                list filesOfProject = buildFilesOfProjectListFromLine(line);
17        }else if (line.StartsWith("*"{
18                list locationLookupTable = buildLocationLookupTableFromLine(
                      ↪ line);
19                list fieldLookupTable = buildFieldLookupTableFromLine(line);
20        }else{
21                javaStatement.getLocalVariables().Add(line);
22        }
23    allStatements.add(javaStatement)\this adds the last created Statement to
          ↪ the list.
24    ParsedJLG parsed = new ParsedJLG(filesOfProject, locationLookupTable,
          ↪ fieldLookupTable, allStatements);
25    return parsed;
26    }
27 }
```

4.2.2 Visualisierung

Die Visualisierung des **ParsedJLG**-Objekts in SEE wird durch das **JLGVisualizer**-Skript umgesetzt. Dieses Skript visualisiert die Statements aus der Statement Liste des **ParsedJLG**-Objekts anhand eines Counters, der je nach Richtung, in der die Visualisierung abgespielt wird, erhöht oder verringert wird. Ein Statement wird durch die **UpdateVisualization()**-Methode visualisiert. Diese Methode passt die SEE Spielwelt so an, dass der Zustand des Programms zum Zeitpunkt des Statements zu erkennen ist. Der sehr vereinfachte Ablauf des Skripts ist im Aktivitätsdiagramm in Abbildung 4.2 gezeigt. In diesem Kapitel wird auf die Implementierung der **UpdateVisualization()**-Methode genauer eingegangen. Die Art und Weise, wann und wie der **statementCounter** angepasst wird, wird im Kapitel 4.2.3 im Rahmen der Steuerung der Visualisierung aufgeklärt.

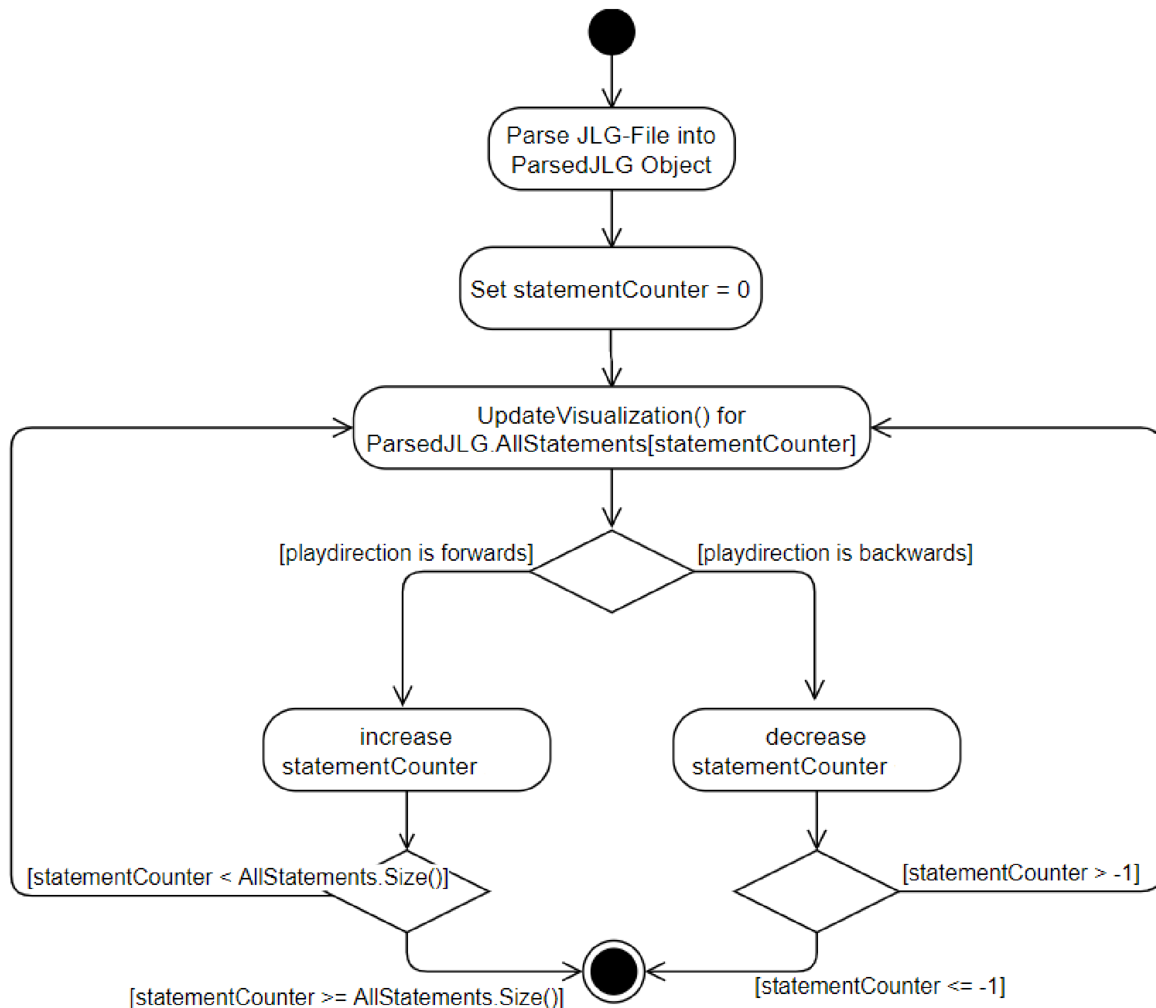


Abbildung 4.2: Vereinfachtes Aktivitätsdiagramm des Visualisierungs Skripts der JLG-Datei für SEE.

Die **UpdateVisualization()**-Methode ist eine Sammlung verschiedener Hilfsmethoden, die die Visualiserung in SEE erschaffen. Diese Methoden werden in den folgenden Absätzen vorgestellt. Die Reihenfolge der Absätze repräsentiert die Reihenfolge, in der die Methoden innerhalb der **UpdateVisualization()**-Methode aufgerufen werden. Zuerst werden aber noch ein paar globale Variablen des Skripts definiert, die für die Methoden relevant sind.

currentGO: Im **currentGO** wird das GameObject des Gebäudes gespeichert, dass die Klasse repräsentiert, aus der das zuletzt visualisierte Statement stammt.

statementCounter: Im **statementCounter** ist der Index des nächsten zu visualisierenden **JavaStatements** gespeichert. Dieser Index ist die Position des Statements, in der Liste des **ParsedJLG**-Objekts mit allen Statements.

playDirection: In dieser **boolean** Variable ist die Richtung gespeichert, in der die Visualisierung abläuft. **True** bedeutet vorwärts und **false** rückwärts.

textWindows: Diese Variable ist ein Stack, in dem alle erzeugten Textfenster der Visualisierung gespeichert sind. Ein Textfenster besteht aus einem großen Text, in dem der

Code einer Klasse steht, und einem kleinen, in dem die Laufzeitdaten stehen. Zu sehen ist ein solches Textfenster in der Abbildung 3.5.

functionCalls: Auch diese Variable ist eine Liste. In dieser Liste werden alle Aufrufgraphen gespeichert, die mit der Funktion von Torben Groß erstellt werden. Zu sehen sind solche Funktionsgraphen in Abbildung 2.5.

Nun können die einzelnen Methoden erklärt werden.

CheckCurrentGO() Diese Methode stellt sicher, dass in der **currentGO**-Variable immer das GameObject des Gebäudes in SEE gespeichert ist, das die Klasse des neu zu visualisierenden Statements repräsentiert. Dazu überprüft sie, ob der Name des gespeicherten GameObjects mit dem der Klasse, aus dem das neue Statement stammt, übereinstimmt. Ist der Name gleich, macht die Methode nichts. Ist der Name nicht gleich, passieren zwei Dinge. Zuerst wird zusätzlich überprüft, ob **playDirection true** ist. Ist **playDirection true**, wird ein Aufrufgraph zwischen dem **currentGo** und dem GameObject, das die Klasse des neuen Statements repräsentiert, erzeugt. Als Zweites färbt die Methode das Gebäude des **currentGO**, das an dieser Stelle noch die Klasse des letzten Statements repräsentiert, in hellblau. Dann wird das GameObject im **currentGO** zu dem Gebäude des neuen Statements gewechselt und dunkelblau gefärbt. So wird die Visualisierung des Aufrufgraphens nach dem Entwurf aus Kapitel 3.4.2 erzeugt.

GenerateScrollableTextWindow() Diese Methode wird aufgerufen, wenn für das **currentGO** noch kein Textfenster existiert. Sie erzeugt mittig über der Softwarestadt ein neues GameObject, das zwei Textfenster enthält, und fügt es zum **textWindows** Stack hinzu. Ein großes Textfenster wird mit dem Code der Klasse des **currentGO** gefüllt. Dafür wird der im **ParsedJLG**-Objekt gespeicherte Dateipfad zur Datei der Klasse verwendet und diese wird dann in das Textfenster geschrieben. Das kleine Textfenster mit den Laufzeitverhalten wird in den **Previous**- und **NextStatement()**-Methoden gefüllt. Zuletzt erzeugt die Methode eine blaue Gerade in der Spielwelt, die die Textfenster mit dem dazugehörigen Gebäude verbindet.

ToggleTextWindow() Diese Methode deaktiviert die GameObjects aller Textfenster im **textWindows** Stack. Nur das Textfenster, das das **currentGO** repräsentiert, bleibt aktiv. So wird immer nur das Textfenster gerendert, das auch gerade verwendet wird. Die deaktivierten Fenster kann man sich während der Visualisierung einfach wieder anzeigen lassen, indem man auf das entsprechende Gebäude klickt. Dann wird die Visualisierung pausiert und das Textfenster des angeklickten Gebäudes erscheint.

UpdateStacks() Die **UpdateStacks()**-Methode aktualiseirt den Anteil der Visualisierung, der für den Aufrufgraphen verantwortlich ist. Das erledigt die Methode basierend auf der Abspielrichtung. Ist die Richtung vorwärts, so überprüft die Methode, ob ein Aufruf beendet wurde. Dazu wird in einer **If**-Bedingung abgefragt, ob das letzte visualisierte Statement vom Typ „exit“ war und ob sich die Klasse dieses Statements von der des aktuellen Statement unterscheidet. Sind beide Bedingungen erfüllt, ist der Aufruf zwischen den beiden Klassen beendet und muss aus der Visualisierung entfernt werden. Dafür wird einfach das entsprechende GameObject aus der **functionCalls** Liste deaktiviert. Wichtig ist, dass das GameObject nur deaktiviert und nicht gelöscht wird, damit der Aufruf beim rückwärts Abspielen durch Reaktivierung des GameObjects wieder visualisiert werden kann.
Ist die Richtung rückwärts, bewirkt die Methode zwei Dinge. Das erste ist die Gegenaktion zu

dem, was die Methode beim vorwärts Abspielen macht. Wird ein Funktionsaufruf rückwärts betreten, also über das **Exit**-Statement, wird das entsprechende GameObject des Aufrufs in der **functionCalls** Liste wieder aktiviert. Wird eine Klasse dann über ein **Entry**-Statement wieder verlassen, ist man in der Visualisierung vor dem entsprechenden Funktionsaufruf. Das GameObject des Funktionsaufrufes kann dann aus der **functionCalls** Liste entfernt und ganz gelöscht werden.

Next- und PreviousStatement() Final wird, je nach Richtung, die **NextStatement()**- oder die **PreviousStatement()**-Methode ausgeführt. Diese Methoden markieren die aktuelle Code Zeile im großen Textfenster und erzeugen den Text des kleinen Fensters, in dem die Laufzeitdaten stehen. Dabei unterscheiden sie sich nur geringfügig.
Zuerst wird die Zeile des Codes markiert. Dazu wird der im GameObject des Textfensters gespeicherte Code Zeile für Zeile in eine Liste gelesen. Der Index der zu visualisierenden Code Zeile in der Liste ist die originale Zahl der Zeile in der Klasse minus eins. Dann wird am Anfang und am Ende diesern Zeile ein **Colortag** hinzugefügt. Die Liste wird dann wieder zu einem **String** zusammengefügt und zurück in das GameObject geladen. Der Text der Code Zeile hat nun die im **Colortag** angegebene Farbe. Hier liegt der Unterschied zwischen den beiden Methoden, darin, dass die **NextStatement()**-Methode die **Colortags** von vorherigen Code Zeilen bearbeitet, sodass wie in Kapitel 3.4.2 die Zeilen Stück für Stück zurück zu weiß verblassen. Die **PreviousStatement()**-Methode färbt immer nur die aktuelle Zeile. So ist es immer leicht zu erkennen, in welche Richtung die Visualisierung abspielt.
Dann werden die Laufzeitdaten, die im **JavaStatement**-Objekt der zu visualisierenden Zeile gespeichert sind, zu einem Text zusammengefasst und in das kleine Textfenster geladen. Abschließend wird der **StatementCounter** um eins erhöht oder verringert.

4.2.3 Steuerung

Die Steuerung der Visuluasierung unterteilt sich in den automatischen und den manuellen Modus.

4.2.3.1 Automatischer Modus

In diesem Modus werden die Statements automatisch wie in einem Video nacheinander visualisiert. Dies wird mit der **Update()**-Methode erreicht. Die **Update()**-Methode eines Skripts wird von Unity einmal in jedem Frame des laufenden Spiels aufgerufen.
Innerhalb der **Update()**-Methode wird dann nach Ablauf eines gesetzten Zeitintervalls die **UpdateVisualization()**-Methode aufgerufen. Dieses Zeitintervall ist standardmäßig eine Sekunde, sodass pro Sekunde ein **JavaStatement** visualisiert wird. Mit den Tasten „3“ und „1“ lässt sich das Zeitintervall halbieren und verdoppeln. Dadurch wird das „Video“ vorgespult beziehungsweise verlangsamt.
Der Richtungswechsel der Visualisierung wird mit der Taste „R“ veranlasst. Im Code wird dann der Wert der **boolean** Variable textbfplayDirection auf den anderen gesetzt. Zusätzlich wird noch der **StatementCounter** um zwei erhöht oder verringert, je nachdem, in welche Richtung die Visualisierung nach dem Drücken der „R“ Taste spielt. Dies lässt sich am leichtesten an einem Beispiel erläutern. Die Visualisierung spielt vorwärts, es wird Statement 50 visualisiert und der **StatementCounter** wird danach auf 51 gesetzt, sodass nach Ablauf des Zeitintervalls Statement 51 visualisiert werden kann. Bevor das Zeitintervall abläuft, wird nun die Taste „R“ gedrückt und die Richtung gewechselt. In der Visualisierung ist Statement 50 zu sehen und der **StatementCounter** steht auf 51, visualisiert werden muss als nächstes

aber Statement 49, da die Visualisierung ja nun rückwärts läuft. Deshalb wird der **StatementCounter** bei diesem Richtungswechsel um zwei verringert.
Mit dem Drücken der „F“ Taste wird der automatische Modus pausiert, in dem innerhalb der **Update()**-Methode nicht mehr **UpdateVisualization()** aufgerufen wird. Die Visualisierung befindet sich nun im manuellen Modus.

4.2.3.2 Manueller Modus

In manuellen Modus können einzelne Statements durch Drücken der Tasten „3“ und „1“ gezeigt werden. Mit der Taste „3“ wird das nächste und mit „1“ das vorherige Statement visualisiert. Dazu setzen die dazugehörigen Methoden den **playDirection** Wert und führen dann einmal die **UpdateVisualization()**-Methode aus. Außerdem wird auch hier, wie beim automatischen Modus, bei einem Richtungswechsel der **StatementCounter** in die entsprechende Richtung um zwei angepasst.
Mit der Taste „B“ kann man zum nächsten Treffer eines gesetzten Breakpoints springen. Die Methode dahinter durchsucht die Liste mit allen **JavaStatements** nach dem nächsten Statement, dass den Kriterien des Breakpoints entspricht und speichert den Index. Dann wird zu diesem Statement gesprungen, in dem die **UpdateVisualization()**-Methode für die vorherigen 300 Statements und dem Breakpoint Statement ausgeführt werden. Ein Breakpoint kann im Inspektor des GameObjekts gesetzt werden, zu dem das Visualisierungsskript als Komponente hinzugefügt wurde.
Weiter lässt sich in diesem Modus die Visualisierung zum Anfang zurücksetzen. Dazu werden einfach alle vom Skript erzeugten GameObjects gelöscht und das **ParsedJLG**-Objekt wird neu geparst. Mit dem Drücken der „F“ Taste gelangt man zurück in den automatischen Modus.

KAPITEL 5
Evaluation

5.1 Planung

5.1.1 Evaluation

Die Evaluation wird mithilfe des DECIDE Usability Framework von Sharp, Rogers und Pierce geplant und durchgeführt[33, 40]. Das DECIDE steht für:

1. **D**etermine the goals.
2. **E**xplore the questions.
3. **C**hoose the evaluation paradigm and techniques.
4. **I**dentify the pracitcal issues.
5. **D**ecide how to deal with the ethical issues.
6. **E**valuate, interpret, and present the data.

Dieses Framework wird oft verwendet, um verschiedene Systeme zu vergleichen. In diesem Kapitel wird beschrieben, wie die Evaluation dieser Arbeit mit dem DECIDE Framework umgesetzt wird.
Ziel dieser Evaluation ist es, den neuen SEE-Debugger hinsichtlich der Effizienz und Effektivität mit einem klassichen Debug-Werkzeug zu vergleichen und die Gebrauchstauglichkeit des SEE-Debuggers zu bewerten. Basierend auf diesen Zielen lassen sich folgende Leitfragen stellen:

- In welchem System finden die Probanden schneller Fehlverhalten und Defekte?
- In welchem System finden die Probanden zuverlässiger Fehlverhalten und Defekte?
- Ist der SEE-Debugger gebrauchstauglich?

Diese Fragen sollen mit dieser Evaluation beantwortet werden.
Zur Beantwortung der ersten beiden Fragen wird ein Labor-Test durchgeführt. In dem Test haben die Probanden zwei Aufgaben, in denen sie zwei verschiedene Programme jeweils mit einem der beiden Werkzeugen debuggen und in jedem der Programme einen Defekt finden sollen. Dabei wird gemessen, wie lange die Probanden benötigen, um das Fehlverhalten und den Defekt, der das Fehlverhalten verursacht, zu finden.
Die Werkzeuge sind der SEE-Debugger und der integrierte Debugger der Eclipse IDE, der bereits in Kapitel 2.6 vorgestellt wurde. Die Eclipse IDE wird verwendet, da es die meistverwendete kostenlose Java IDE ist[10, 20, 26, 27]. Die Programme, die gedebugged werden

sollen, sind zwei in Java programmierte Spiele. Dabei handelt es sich um Snake und Space Invaders. Die Implementation der Spiele ist einfach gehalten und die Architektur der beiden ähnelt sich stark. In beiden Spielen ist jeweils ein Defekt eingebaut worden, der von den Teilnehmern mit den beiden Debuggern gefunden werden soll. Der Defekt in Snake ist so gewählt worden, dass das Fehlverhalten erst später im Programm auftaucht, wohingegen der Defekt bei Space Invaders sofort zu einem Fehlverhalten führt. Die Defekte wurden bewusst so gewählt, um zu testen, ob sich der SEE-Debugger auch dazu eignet, Fehler zu finden, bei denen Defekt und Fehlverhalten nicht sofort aufeinander folgen.
Eine Hälfte der Teilnehmer verwendet den SEE-Debugger für Snake und die Eclipse IDE für Space Invaders, die andere Hälfte verwendet die Eclipse IDE für Snake und den SEE-Debugger für Space Invaders. So entsteht für jedes Spiel ein eigener Datensatz mit Zeiten für beide Werkzeuge. Diese Datensätze werden in Kapitel 5.3 ausgewertet.
Die dritte Frage wird mithilfe eines Fragebogens beantwortet, der in Kapitel 5.1.2 beschrieben wird. Die Einschränkungen, die bei dieser Evaluation beachtet werden müssen, werden in Kapitel 5.1.3 erklärt.

5.1.2 Fragebogen

Der Fragebogen, den die Probanden am Ende der Evaluation ausfüllen sollen, startet mit der Frage „Haben Sie Vorerfahrung mit dem Eclipse Debugger oder ähnlichen Werkzeugen?“. Mit ähnlichen Werkzeugen, sind andere Step-Through Debugger gemeint. Dies wird den Probanden vor dem Fragebogen erklärt. Die Antwort auf diese Frage ist wichtig für die Auswertung der Messergebnisse. Danach ist der Bogen in zwei Teile aufgeteilt. Im ersten Teil sollen die Probanden die Usability des SEE-Debuggers anhand der System Usability Scale, kurz „SUS“, bewerten. Dieser Fragebogen, bestehend aus zehn Fragen, soll die Gebrauchstauglichkeit eines Systems anhand einer quantitativen Bewertung erheben. Er wurde 1986 von John Brooke entwickelt und ist eine in der Nutzerforschung weit verbreitete und anerkannte Methode[11, 33, 41]. Dieser Fragebogen wurde für die Evaluation auf Deutsch übersetzt und ist in Abbildung 5.1 zu sehen.
Der Fragebogen wird ausgewertet, indem die Werte der Antworten zusammengezählt werden. Dazu werden die Werte der Fragen 1, 3, 5, 7 und 9 gezählt, wie sie sind, und die Werte der Fragen 2, 4, 6, 8 und 10 werden erst umgedreht. Eine 1 wird zu einer 10, eine 2 zu einer 9 und so weiter. Ergibt sich am Ende ein Gesamtwert über 68, gilt das getestete System als gebrauchstauglich[33, 43].
Im zweiten Teil des Bogens werden drei Fragen zum Vergleich der beiden Werkzeuge gestellt. Zweck dieser Fragen ist es herauszufinden, welches System den Probanden besser gefallen hat und warum. Dazu werden drei Fragen gestellt:

- Welches System hat Ihnen besser gefallen?
- Warum hat Ihnen das System besser gefallen?
- Haben Sie Verbesserungsvorschläge für den SEE-Debugger?

Basierend auf den Antworten dieser Fragen werden in Kapitel 6 verschiedene Möglichkeiten vorgestellt, um die Funktionen und die Oberfläche des SEE-Debuggers zu überarbeiten.

5.1.3 Einschränkungen

Zur Zeit der Evaluation ist die Welt von der Covid-19 Pandemie betroffen. Aus diesem Grund wird zum Schutz der Teilnehmer die Evaluation nicht physisch stattfinden, also an einem

1. Ich kann mir gut vorstellen, das System regelmäßig zu nutzen.

	1	2	3	4	5	6	7	8	9	10	
Trifft nicht zu	○	○	○	○	○	○	○	○	○	○	Trifft zu

2. Ich empfinde das System als unnötig komplex.

	1	2	3	4	5	6	7	8	9	10	
Trifft nicht zu	○	○	○	○	○	○	○	○	○	○	Trifft zu

3. Ich empfinde das System als einfach zu nutzen.

	1	2	3	4	5	6	7	8	9	10	
Trifft nicht zu	○	○	○	○	○	○	○	○	○	○	Trifft zu

4. Ich denke, dass ich technischen Support brauchen würde, um das System zu nutzen.

	1	2	3	4	5	6	7	8	9	10	
Trifft nicht zu	○	○	○	○	○	○	○	○	○	○	Trifft zu

5. Ich finde, dass die verschiedenen Funktionen des Systems gut integriert sind.

	1	2	3	4	5	6	7	8	9	10	
Trifft nicht zu	○	○	○	○	○	○	○	○	○	○	Trifft zu

6. Ich finde, dass es im System zu viele Inkosistenzen gibt.

	1	2	3	4	5	6	7	8	9	10	
Trifft nicht zu	○	○	○	○	○	○	○	○	○	○	Trifft zu

7. Ich kann mir vorstellen, dass die meisten Leute das System schnell zu beherschen lernen.

	1	2	3	4	5	6	7	8	9	10	
Trifft nicht zu	○	○	○	○	○	○	○	○	○	○	Trifft zu

8. Ich empfinde die Bedienung als sehr umständlich.

	1	2	3	4	5	6	7	8	9	10	
Trifft nicht zu	○	○	○	○	○	○	○	○	○	○	Trifft zu

9. Ich habe mich bei der Nutzung des Systems sehr sicher gefühlt.

	1	2	3	4	5	6	7	8	9	10	
Trifft nicht zu	○	○	○	○	○	○	○	○	○	○	Trifft zu

10. Ich musste eine Menge Dinge lernen, bevor ich mit dem System arbeiten konnte.

	1	2	3	4	5	6	7	8	9	10	
Trifft nicht zu	○	○	○	○	○	○	○	○	○	○	Trifft zu

Abbildung 5.1: System Usability Scale Fragebogen auf deutsch[33, 41].

vorbereiteten System, an dem die Teilnehmer die Tools einfach verwenden können, sondern remote über eine Bildschirmübetragung mit Übergabe der Steuerung an den Teilnehmer. Die Teilnehmer müssen sich die benötigten Programme nicht herunterladen und installieren. Dies vereinfacht den Evaluationsprozess und erhöht die Teilnahmebereitschaft möglicher Probanden.
Für die remote Durchführung der Evaluation wird die Remote-Desktop-Software AnyDesk verwendet. Die Entscheidung ist aus mehreren Gründen auf AnyDesk gefallen. AnyDesk ist eine federleichte Anwendung, die aus einer nur wenige MegaByte großen Programmdatei besteht. Die Anwendung kann ohne Installation verwendet werden. Weiter funktioniert AnyDesk auf allen weit verbreiteten Betriebssystemen. Dazu zählt Windows, macOS, iOS, Android, Linux, FreeBSD und Raspberry Pi. Außerdem bietet AnyDesk bis zu 60 Bilder pro Sekunde und ist für die nicht-kommerzielle Verwendung kostenlos[3]. Andere Optionen sind TeamViewer und Skype. TeamViewer ist zwar auch für die nicht-kommerzielle Nutzung kostenlos, muss aber für die optimale Nutzung installiert werden und ist vielmal größer als AnyDesk[48]. Skype ist vermutlich die bekannteste Software der drei. Allerdings erlaubt die kostenlose Version von Skype nicht die Steuerfreigabe bei einer Bildschirmübertragung. Dafür benötigen beide Teilnehmer eine Skype for Business Lizenz[35].
Die Durchführung über eine Remote-Desktop-Software bringt einen Nachteil mit sich, der unbedingt bei der Auswertung der Ergebnisse der Evaluation beachtet werden muss. Durch Latenz und Einbrüche einer Internetverbindung entstehen unvermeidlich Einbrüche der Verbindungsqualität der Übertragung. Diese Einbrüche wirken sich negativ auf die Effektivität der Nutzung und die gefühlte Usability der beiden Werkzeuge aus. Der SEE-Debuger leidet als dynamisches Visualisierungstool, das von einer hohen Anzahl an Bildern pro Sekunde und schneller Reaktionszeit des Programms auf Eingaben profitiert, besonders an diesen Einbrüchen.

5.2 Durchführung

An der Evaluation haben insgesamt zehn Probanden teilgenommen. Neun Probanden haben eine Ausbildung als Fachinformatiker abgeschlossen. Davon sind fünf Teilnehmer aktive duale Informatik Studenten in fortgeschrittenen Semestern. Die anderen Vier haben ihren Bachelor in Informatik bereits abgeschlossen und arbeiten als Junior-Softwareentwickler. Der zehnte Proband hat einen Bachelor of Science in Physik und studiert zur Zeit der Evaluation Informatik. Alle Teilnehmer haben ausreichend Erfahrung in der Softwareentwicklung und der Programmiersprache Java, um die Defekte zu finden.
Durchgeführt wurde die Evaluation über die AnyDesk Remote-Desktop-Software, wie in Kapitel 5.1.3 beschrieben. Mit dieser Software konnten die Probanden den Computer, auf dem die benötigten Programme installiert war, remote steuern. Für die Unterhaltung mit dem Teilnehmer während des Prozesses wurde die Voice-Over-IP Software Discord verwendet. Discord ist eine weit verbreitete Software des Unternehmens Discord Inc. und kann auch ohne Installation über einen beliebigen Internetbrowser verwendet werden[17]. Die Evaluation wurde mit jedem Teilnehmer einzeln durchgeführt.
Zu Beginn der Evaluation wurde dem Probanden erklärt, worum es in der Evaluation geht und was dabei seine Aufgaben sind. Dann wurde genau erklärt, welche Daten erhoben werden, wofür sie verwendet werden und welche im Rahmen der Arbeit veröffentlicht werden. Nachdem der Proband der Verwendung seiner Daten zugestimmt hatte, wurde mit der ersten Aufgabe gestartet.
Der Ablauf der beiden Aufgaben war derselbe und unterscheidete sich nur im verwendeten Werkzeug und zu dem analysierendem Programm. Eine Aufgabe begann mit einer Einführung

in die Umgebung, mit der diese bewältigt werden soll. Dazu wurden dem Probanden zuerst die Funktionen und die Steuerung des Debug-Werkzeugs, das er verwenden sollte, ausführlich erklärt und dann eine Übersicht über die Funktionen der zu debuggenden Software gegeben. Anschließend wurde dem Probanden das Szenario erklärt, in dem das Fehlverhalten der Software entstanden ist. Dazu wurde gesagt, welche Funktion zuletzt in der Software implementiert wurde und dass durch die Implementierung dieser Funktion irgendwo ein Fehlverhalten entstanden ist. Dies schränkte den Bereich, in dem nach dem Fehler gesucht werden muss, ein. So musste der Proband nicht in der gesamten Software suchen, was die Dauer der Evaluation unnötig verlängert hätte. An dieser Stelle hatte der Teilnehmer die Möglichkeit, Fragen zum Werkzeug, Programm oder zur Aufgabe zu stellen. Waren alle Fragen geklärt, konnte der Teilnehmer damit beginnen, das Fehlverhalten und den Defekt in der Software mit dem gegebenen Werkzeug zu suchen. Die Zeit, die er dafür benötigte, wurde gestoppt. Eine Aufgabe war abgeschlossen, wenn der Proband das Fehlverhalten und den Defekt gefunden hatte, oder wenn der Proband sagte, dass er den Fehler nicht finden kann. Danach wurde derselbe Ablauf für das zweite Werkzeug mit dem zweiten Spiel durchgeführt. Die Teilnehmer wurden, wie in Kapitel 5.1 vorgesehen, in zwei Gruppen aufgeteilt. Die erste Gruppe untersuchte Snake mit dem Eclipse Debugger und Space Invaders mit dem SEE-Debugger. Die zweite Gruppe verwendete für Snake den SEE-Debugger und für Space Invaders den Eclipse Debugger.
Nachdem beide Aufgaben beendet waren, erhielt der Proband den Fragebogen. Dieser wurde anonym auf dem eigenen Gerät ausgefüllt. Bevor oder während der Fragebogen ausgefüllt wurde, konnte der Proband Fragen zu diesem stellen.

5.3 Ergebnisse

5.3.1 Auswertung der Messwerte

Werkzeug	Teilnehmer	Fehlverhalten	Defekt
Eclipse	1	10:02	13:10
	3	1:00	3:10
	5	4:16	6:32
	7	2:37	4:20
	9	7:52	11:02
SEE-Debugger	2	3:25	5:35
	4	7:29	8:45
	6	3:23	4:15
	8	0:50	3:06
	10	2:02	4:37

Tabelle 5.1: Benötigte Zeiten in Minuten, um Auslöser des Fehlverhaltens und Defekt in Space Invaders zu finden.

Die Ergenisse der Zeitmessungen sind in den Tabellen 5.1 und 5.2 zu sehen. Tabelle 5.1 zeigt die Messwerte für Space Invaders und Tabelle 5.2 die Werte für Snake. Sortiert sind die Tabellen nach dem zum Debuggen verwendeten Werkzeug und der Teilnehmerzahl. Die Zeiten sind in Minuten dargestellt. Softort ersichtlich ist, dass es jeder Teilnehmer geschafft hat, beide Aufgaben vollständig zu lösen. Daraus lässt sich schließen, dass beide Werkzeuge geeignet sind, um die Softwarefehler in dieser Evaluation zu analysieren und zu finden.
Um die Effizienz der beiden Systeme zu vergleichen, werden die gemessenen Zeiten betrachtet. Damit die Zeiten besser verglichen werden können, werden sie mit einem Boxplot Diagramm,

Werkzeug	Teilnehmer	Fehlverhalten	Defekt
Eclipse	2	3:56	4:32
	4	4:00	5:32
	6	4:47	6:56
	8	1:55	3:47
	10	2:29	5:24
SEE-Debugger	1	3:42	5:06
	3	2:36	3:31
	5	2:07	3:36
	7	3:35	4:18
	9	5:33	6:54

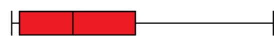
Tabelle 5.2: Benötigte Zeiten in Minuten, um Auslöser des Fehlverhaltens und Defekt in Snake zu finden.

Space Invaders in Eclipse

Space Invaders in SEE

Snake in Eclipse

Snake in SEE



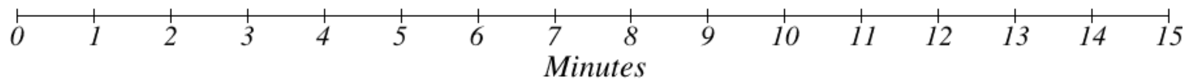


Abbildung 5.2: Boxplot der Messerergebnisse. Die roten Boxen repräsentieren den SEE-Debugger und die blauen Eclipse. Boxplot erstellt mit IMathAS[32].

das in Abbildung 5.2 zu sehen ist, visualisiert.
Aufgrund der geringen Teilnehmerzahl und der in Kapitel 5.1.3 erläuterten Einschränkungen können mit den Daten lediglich Vermutungen und keine statistisch signifikanten Folgerungen aufgestellt werden.
Die beiden Boxplots der Messergebnisse für das Snake Programm unterscheiden sich kaum, dagegen unterscheiden sich die beiden Boxplots für Space Invaders stark. Dies liegt an den Probanden 1 und 9, die vergleichsweise sehr lange zum Finden des Defekts in Eclipse gebraucht haben. Im Gegensatz dazu waren die beiden Probanden beim Debuggen des Snake Programms nur etwas langsamer als die Teilnehmer 3, 5 und 7. Um diese Daten besser auswerten zu können, werden die Ergebnisse der Frage „Haben Sie Vorerfahrung mit dem Eclipse Debugger oder ähnlichen Werkzeugen?„ herangezogen. Diese sind in Tabelle 5.3 zu sehen.

Teilnehmer	Vorerfahrung in Eclipse
1	Nein
2	Ja
3	Ja
4	Ja
5	Ja
6	Nein
7	Ja
8	Ja
9	Nein
10	Ja

Tabelle 5.3: Antworten auf die Frage „Haben Sie Vorerfahrung mit dem Eclipse Debugger oder ähnlichen Werkzeugen?“.

In dieser Tabelle ist zu sehen, dass Teilnehmer 1 und 9 keine Vorerfahrung mit dem Eclipse Debugger haben. Daraus lässt sich die Vermutung aufstellen, dass das SEE-Debugging Tool leichter zu erlernen ist, als der in Eclipse eingebaute Debugger. Dies wird weiter durch die Ergebnisse von Teilnehmer 6 unterstützt. Teilnehmer 6 hat ebenfalls keine Vorerfahrung im Debuggen in Eclipse oder ähnlichen Werkzeugen und ist in der anderen Gruppe der Langsamste beim Debuggen von Snake mit Eclipse gewesen. Konträr dazu war Teilnehmer 6 mit dem SEE-Debugger Zweitschnellster.
Ohne die beiden Ausreißer sind auch die Boxplots für Space Invaders ähnlich. Über die Effizienz der beiden Systeme lässt sich aufgrund der starken Varianz der Messdaten kein fundierter Vergleich anstellen.

5.3.2 System Usability Scale

Die Ergebnisse des System Usability Scale Fragebogens sind in Tabelle 5.4 zu sehen. Die Zahlen bedeuten, wie oft eine Bewertung für das Kriterium abgegeben wurde. Mit diesen Werten kann nun überprüft werden, ob der SEE-Debugger als „gebrauchstauglich“ eingestuft werden kann. Dazu müssen die Zahlen, wie in Kapitel 5.1.2 beschrieben, zusammengerechnet und ausgewertet werden. Zählt man die Werte zusammen erhählt man eine Gesamtpunktzahl von 740. Das entspricht einer Bewertung von 74 bei zehn Teilnehmern. Da ein System mit einer Bewertung über 68 als „gebrauchstauglich“ gilt, ist der SEE-Debugger nach der System Usability Scale „gebrauchstauglich“.

Frage/Bewertung	1	2	3	4	5	6	7	8	9	10
1. Ich kann mir gut vorstellen, das System regelmäßig zu nutzen.	0	1	1	0	1	3	0	3	0	1
2. Ich empfinde das System als unnötig komplex.	3	3	0	2	1	0	1	0	0	0
3. Ich empfinde das System als einfach zu nutzen.	0	0	1	0	0	3	2	2	2	0
4. Ich denke, dass ich technischen Support brauchen würde, um das System zu nutzen.	4	0	3	1	1	1	0	0	0	0
5. Ich finde, dass die verschiedenen Funktionen des Systems gut integriert sind.	0	0	0	0	3	1	2	3	1	0
6. Ich finde, dass es im System zu viele Inkonsistenzen gibt.	4	2	2	1	1	0	0	0	0	0
7. Ich kann mir vorstellen, dass die meisten Leute das System schnell zu beherrschen lernen.	1	0	0	1	1	0	0	2	2	3
8. Ich empfinde die Bedienung als sehr umständlich.	1	5	0	0	1	3	0	0	0	0
9. Ich habe mich bei der Nutzung des Systems sehr sicher gefühlt.	0	0	1	2	1	0	4	1	0	1
10. Ich musste eine Menge Dinge lernen, bevor ich mit dem System arbeiten konnte.	1	1	5	2	1	0	0	0	0	0

Tabelle 5.4: Ergebnisse des System Usability Scale Fragebogens.

5.3.3 Fragen zum Systemvergleich

In den drei abschließenden Fragen des Bogens wurden die Teilnehmer unter anderem gefragt, welches System ihnen besser gefallen hat und warum. Insgesamt bevorzugen in dieser Evaluation mehr Teilnehmer den SEE-Debuger, wie in Abbildung 5.3 zu sehen ist.

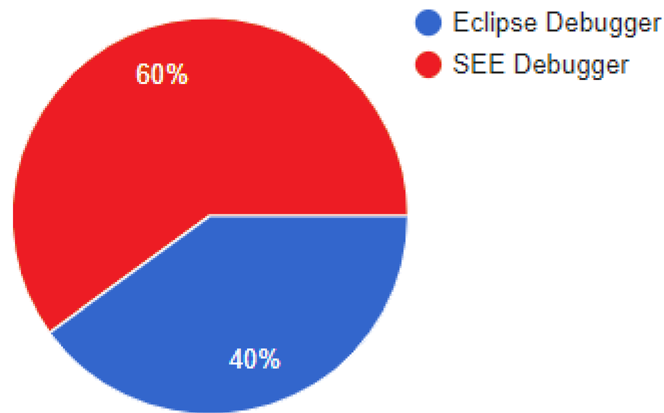


Abbildung 5.3: Ergebnis der Frage „Welches System hat Ihnen besser gefallen?“.

In Tabelle 5.5 sind die Gründe aufgelistet, die die Teilnehmer für ihre Entscheidung angegeben haben. Die meisten Antworten wurden wörtlich übernommen, andere wurden paraphrasiert.

SEE-Debugger	Eclipse IDE Debugger
„Der größte Vorteil ist die Aufzeichnung des Programmablaufs. Die erlaubt es nämlich während des Debuggens nicht nur vor, sondern insbesondere auch zurück zu gehen.“	/
Es gab die Möglichkeit rückwärts im Code zu gehen.	/
„Übersichtlichkeit aufgrund von wenig angezeigten Variablen im Vergleich zu Eclipse. Dort wurden einfach alle Variablen in einer riesigen Liste angezeigt. Großer Vorteil des „rückwärts“ Debuggen.“	/
„Rücklauffunktion und schnelleres Springen im Code.“	/
„Die optische Komponente leitet einen gut durch den SourceCode. Wenn man sich etwas länger mit der Visualisierung durch SEE beschäftigt, kann man dem Modell sicher auch noch leicht ein Fehlverhalten entnehmen. Die praktischste Funktion hierbei war aber der Code mit den Gradienten nach Ausführung und die Möglichkeit sich darin frei entlang der Ausführung zu bewegen. “	/

Einfacher Überblick, einfache Steuerung, Rückwärts abspielen	/
/	Einfache Bedienbargkeit, Vertrautheit mit der Anwendung und bessere Übersicht.
/	„Ich bin an den klassichen Debugger gewöhnt. Im SEE-Debugger dauert es zu lange, zum gewünschten Punkt zu kommen. Ein großer Vorteil des klassichen Debuggers ist das direkte Editieren von Code.“
/	„Der fast unschlagbare Vorteil ist für mich die direkte Integration in die IDE und der damit verbundene geringe „overhead“ (z.B. beim setzen von Breakpoints). Was mir aber in Eclipse sehr gefehlt hat, war die Möglichkeit wieder in die Vergangenheit zurückzugehen.“
/	„Alles in einem Programm.“

Tabelle 5.5: Gründe für die Entscheidung, warum ein System besser gefallen hat.

Diese Daten basieren auf persönlichen Präferenzen der Teilnehmer, was ganz klar daran zu erkennen ist, dass die Gründe der einfacheren Steuerung, des schnelleren Springens im Code und der besseren Übersicht auf beiden Seiten genannt wurden. Deshalb lässt sich mit diesen Daten kein objektiver Entschluss darüber fassen, welches System das bessere ist. Dennoch ist interessant, dass die Funktion des SEE-Debuggers, die es erlaubt, „rückwärts durch den Code zu gehen“, in allen Antworten für den SEE-Debugger erwähnt worden ist und sogar in einem Grund für den in Eclipse integrierten Debugger erwähnt wurde. Daraus lässt sich schließen, dass diese Funktion von Softwareentwicklern sehr begehrt bei Debuggern ist. In der letzten der drei Fragen konnten die Probanden Verbesserungsvorschläge für den SEE-Debuger geben. Diese werden im folgenden Kapitel 6 besprochen.

KAPITEL 6
Fazit und Ausblick

6.1 Fazit

Im Rahmen dieser Arbeit sind zwei Programme und ein Dateiformat entstanden. Das erste Programm ist der **ExecutedLoCLogger**, der in der Lage ist, ein Java Programm zu starten und alle während der Laufzeit ausgeführten Zeilen Code mit den lokalen Variablen, Rückgabewerten und Feldwertveränderungen aufzuzeichnen. Die Ausgabe dieses Programms ist eine Log-Datei im JLG-Format, das in Kapitel 3.3 entworfen wurde. Diese Datei kann vom zweiten Programm, dem SEE-Debugger, einglesen und innerhalb einer Softwarestadt in SEE visualisiert werden. In der Visualisierung kann der ausgeführte Code gelesen, Laufzeitdaten gesehen und der Aufrufstapel des Programms nachvollzogen werden. Der Programmablauf kann in der Visualisierung sowohl vorwärts als auch rückwärts inspiziert werden.
In einer Evaluation wurde der SEE-Debugger mit dem integrierten Debugger der Eclipse IDE verglichen. Aufgrund von geringen Teilnehmerzahlen und negativen Auswirkungen der remote Durchführung der Evaluation aufgrund der weltweiten Corona-Pandemie lässt sich mit den Messwerten nur die Vermutung aufstellen, dass der SEE-Debugger leichter zu erlernen ist, als der Step-Through-Debugger der Eclipse IDE. Weitere Ergebnisse lieferte der Fragebogen der Evaluation. In dem Fragebogen wurde der SEE-Debugger von den Probanden anhand der System Usability Scale bewertet. Der Debugger wurde insgesamt mit 74 Punkten bewertet und gilt damit nach der System Usability Scale als gebrauchstauglich.
In zwei weiteren Fragen wurden die Probanden gefragt, welcher Debugger ihnen besser gefallen hat und warum. Auch wenn die Antworten auf diese Fragen auf persönlichen Präferenzen basieren, konnte eine Gemeinsamkeit bei fast Allen festgestellt werden. Die Funktion des SEE-Debuggers, Codeabläufe sowohl vorwärts, als auch rückwärts zu zeigen, wurde in jeder Antwort, die sich für diesen Debugger aussprach, genannt. Sogar Antworten, in denen sich für den Eclipse Debugger entschieden wurde, erwähnten diese Funktion als großen Vorteil des SEE-Debuggers. Daraus lässt sich schließen, dass die Funktion, ausgeführten Code in beide Richtungen inspizieren zu können, sehr gefragt bei Entwicklern ist.

6.2 Ausblick

In diesem Kapitel werden Erweiterungsmöglichkeiten präsentiert und Vorschläge für das weitere Vorgehen mit den Ergebnissen dieser Arbeit gegeben.

6.2.1 Weitere Studien

Die Evaluation dieser Arbeit hat nur die Vermutung zugelassen, dass das Erlernen der Verwendung des SEE-Debugger leichter ist. Aufgrund großer Varianzen bei Messwerten und der

geringen Teilnehmerzahl, konnte die Effizienz des SEE-Debuggers im Vergleich zum Eclipse Debugger gar nicht bewertet werden. Um die Vermutung zu überprüfen, und mehr über die Performanz des SEE-Debuggers zu erfahren, sollten weitere Studien ausgeführt werden. Bei diesen Studien sollte eine größere Anzahl an Probanden teilnehmen und die Probanden sollten die Systeme direkt verwenden statt remote. Dadurch kann die Evaluation genauere und statistisch signifikantere Messdaten erzeugen. Bei weiteren Studien könnte zusätzlich getestet werden, inwieweit sich das Programm für weitere Aufgaben, wie zum Beispiel dem reinen Programmverständnis, eignet.

6.2.2 Überarbeitung und Erweiterung

In der Evaluation wurden die Probanden gefragt, ob sie Verbesserungsvorschläge für die Visualsierung und die Benutzeröberflache des Debuggers in SEE haben. Die Antworten auf diese Frage werden im Folgenden zusammengefasst.
Eine mögliche Erweiterung des Debuggers wäre die bessere Anbindung an die Entwicklungsumgebung. Dazu könnte eine Funktion implementiert werden, mit der sich der SEE-Debugger direkt aus der Entwicklungsumgebung starten lässt, und Breakpoints, die in dieser gesetzt wurden, übernimmt.
In dieser Arbeit wurde nur eine Funktion implementiert, mit der sich nur ein Breakpoint vor dem Start des Debuggers im Inspektor setzen lässt. Diese Funktion kann so erweitert werden, dass es möglich ist, mehrere Breakpoints zu setzen und diese auch während der Visualisierung zu bearbeiten.
Der Code der Klassen wird in der Visualiserung noch als einfarbiger weißer Text auf dunkelgrauem Hintergrund dargestellt. Durch Syntax-Highlighting könnte die Lesbarkeit des Codes deutlich verbessert werden.
Die Aufrufe zwischen Klassen werden durch eine Animation von fliegenden Kugeln visualisiert. Will man die Details zum Aufruf wissen, muss man die entsprechenden Gebäude anklicken und in den dazugehörigen Textfenstern nach den gesuchten Informationen schauen. Hier könnte man die Visualiserung so erweitern, dass die Details, wie zum Beispiel aus welcher Methode der Aufruf stammt, welche Methode aufgerufen wird und welche Parameter im Aufruf enthalten sind, mit der Animation zusammen gezeigt werden.
Diese Verbesserungen und Erweiterungen könnten dazu führen, dass der SEE-Debugger in der System Usability Scale besser bewertet wird. Die aktuelle Bewertung von 74 ist nur knapp über den 69 Punkten, ab denen ein System als gebrauchstauglich gilt.

6.2.3 Virtual und Augmented Reality

Der SEE-Debugger wurde in dieser Arbeit primär für die 3D-Umgebung am normalen Monitor entwickelt und getestet. SEE ist allerdings auch mit Virtual Reality und Augmented Reality kompatibel. Die Erweiterung des Debuggers für die Virtual Reality und Augmented Reality ist mit den Funktionen der Unity Engine möglich. Interessant wäre hier, besonders der Vergleich der drei verschiedenen Systeme hinsichtlich der Effizienz, da Wettel et al. 2011 in einem Experiment zeigen konnten, dass sich das Verständnis von Software mit Softwarestädten verbessert, wenn diese in Virtual Reality betrachtet werden[52]. Im Rahmen einer weiteren Evaluation könnte dann getestet werden, ob sich die Verwendung von Virtual oder Augmented Reality positiv auf den Debug-Prozess auswirkt.

6.2.4 Andere Programmiersprachen

In dieser Arbeit wurden ausschließlich Java Programme aufgezeichnet und visualisiert. Zusätzlich könnte der Debugger weiterentwickelt werden, um noch weitere Programmiersprachen visualisieren zu können. Dazu müsste das JLG-Format so angepasst werden, dass es die Aufzeichnungen aller Programmiersprachen repräsentieren kann und dementsprechend müsste der Debugger so angepasst werden, dass er das veränderte Format unterstützt.

6.2.5 Live-Debugging

Der SEE-Debugger ist bisher nur in der Lage Programme post mortem, also nachdem sie beendet wurden, anhand der Aufzeichnung zu analysieren. Die meisten Debugger können Programme bereits zur Laufzeit betrachten. Dabei können auch Werte von zum Beispiel lokalen Variablen oder Feldern manipuliert und das Verhalten der Software aktiv beeinflusst werden. Das bringt viele Vorteile. Ein Vorteil ist, dass so gezielt bestimmte Werte für Variablen oder Felder gesetzt und überpüft werden können, ohne dass diese aufwendig mit dem Programm reproduziert werden müssen. Mit dieser Funktion im SEE-Debugger wäre es möglich, gleichzeitig das Verhalten des Programms und das Verhalten des Codes zu beobachten. Um die Funktion in den SEE-Debugger einzubauen, muss ein Programm geschrieben werden, das SEE mit einer aktiven Debug-Schnittstelle kommunizieren lässt. Dann könnten die Daten der Debug-Schnittstelle in SEE visualisert werden und die Debug-Schnittstelle wiederum aus SEE heraus gesteuert werden.

ABBILDUNGSVERZEICHNIS

1.1 Entwicklung der SLOC von Microsoft Windows und vom Linux Kernel 1

2.1 Modification Request nach ISO/IEC 14764 [1] 6
2.2 Ein Überblick der CodeCity von ArgoUML v.0.24[51]. 11
2.3 Softwarestadt des Linux Net Subsystems erzeugt mit SEE[28]. 11
2.4 Aufrufgraphen im DYN-Format. Erzeugt mit einer der beiden Insturmentierungen[22]. 13
2.5 Visualisierung von Aufrufgraphen in SEE[22]. 13
2.6 SEE in der Unity Engine. 14
2.7 Oberfläche von Eclipse beim Debuggen eines Java Programms. 15

3.1 UML-Diagramm des ExecutedLoCLoggers. 20
3.2 UML-Diagramm der JLG-Datenklassen und -Parser. 23
3.3 Ausschnitt eines Textfenster einer Klasse in SEE. An den grün eingefärbten Zeilen erkennt man, welche zuletzt ausgeführt wurden. 24
3.4 Laufzeitdaten zu den Code Zeilen 25 und 30 aus Abbildung 3.3. Diese erscheinen in einem weiteren kleinen Textfenster. 24
3.5 Ausschnitt eines visualisierten Debug-Prozesses in SEE. 25

4.1 Vereinfachtes Aktivitätsdiagramm der Logging-Methode des **ExecutedLoCLoggers**. 30
4.2 Vereinfachtes Aktivitätsdiagramm des Visualisierungs Skripts der JLG-Datei für SEE. 33

5.1 System Usability Scale Fragebogen auf deutsch[33, 41]. 39
5.2 Boxplot der Messerergebnisse. Die roten Boxen repräsentieren den SEE-Debugger und die blauen Eclipse. Boxplot erstellt mit IMathAS[32]. 42
5.3 Ergebnis der Frage „Welches System hat Ihnen besser gefallen?“. 45

LISTINGS

3.1 Beispiel einer Lookup-Tabelle in einer JLG-Datei. 21
3.2 Ausschnitt einer JLG-Datei. 22
4.1 Erstelllung einer Request im Allgemeinen. 28
4.2 Pseudo-Code des JLGParsers. 32

LITERATURVERZEICHNIS

[1] *ISO/IEC/IEEE International Standard for Software Engineering - Software Life Cycle Processes - Maintenance*. ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998), S. 1–58, 2006.

[2] *ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary*. ISO/IEC/IEEE 24765:2017(E), S. 1–541, 2017.

[3] AnyDesk Software GmbH: *AnyDesk*. `https://anydesk.com/de`.

[4] Arbeitsgruppe Softwaretechnik Uni Bremen. `https://github.com/uni-bremen-agst/SEE`.

[5] Arthur, L. J.: *Software Evolution: The Software Maintenance Challenge*. John Wiley & Sons, New York, NY, 1988.

[6] Ball, T.: *The concept of dynamic analysis*. In: *Software Engineering—ESEC/FSE'99*, S. 216–234. Springer, 1999.

[7] Ball, T. und S. G. Eick: *Software visualization in the large*. Computer, 29(4):33–43, 1996.

[8] Barr, E. T. und M. Marron: *Tardis: Affordable Time-Travel Debugging in Managed Runtimes*. OOPSLA '14, S. 67–82, New York, NY, USA, 2014. Association for Computing Machinery.

[9] Baumöl, U., J. Borchers, S. Eicker, K. Hildebrand, R. Jung und F. Lehner: *Einordnung und Terminologie des Software Reengineering*. Informatik-Spektrum, 19(4):191–195, 1996.

[10] Binstock, A. und S. Maple: *The Largest Survey Ever of Java Developers*. `https://blogs.oracle.com/javamagazine/the-largest-survey-ever-of-java-developers`, 2018.

[11] Brooke, J.: *SUS-A quick and dirty usability scale*. 1986.

[12] Chikofsky, E. J. und J. H. Cross: *Reverse engineering and design recovery: a taxonomy*. IEEE Software, 7(1):13–17, 1990.

[13] Corbi, T. A.: *Program understanding: Challenge for the 1990s*. IBM Systems Journal, 28(2):294–306, 1989.

[14] Cornelissen, B., A. Zaidman, A. van Deursen, L. Moonen und R. Koschke: *A Systematic Survey of Program Comprehension through Dynamic Analysis*. IEEE Transactions on Software Engineering, 35(5):684–702, 2009.

[15] Corporation, O.: *Package java.lang.instrument*. `https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html`.

[16] Diehl, S.: *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer Berlin Heidelberg, 2007.

[17] Discord Inc.: *Discord*. https://discord.com/.

[18] Eclipse Foundation: *Eclipse*. https://www.eclipse.org.

[19] Edison, T. A.: *Letter from Thomas A. Edison to William Orton, 03.03.1878*. The Thomas Edison Papers. Smithsonian Institution, Washington, D.C. – National Museum of American History Archives Center. http://edison.rutgers.edu/NamesSearch/SingleDoc.php?DocId=X099AT.

[20] Geer, D.: *Eclipse becomes the dominant Java IDE*. Computer, 38(7):16–18, 2005.

[21] Gregory, J.: *Game Engine Architecture, Third Edition*. CRC Press, 2018.

[22] Gross, T.: *Visualisierung von dynamischen Aufrufgraphen mit VR-basierten Software-Städten*, January 2020.

[23] Grubb, P. und A. A. Takang: *Software Maintenance: Concepts And Practice*. World Scientific Publishing Co. Pte. Ltd., second Aufl., 2003.

[24] Hailpern, B. und P. Santhanam: *Software debugging, testing, and verification*. IBM Systems Journal, 41(1):4–12, 2002.

[25] Hunt, B., B. Turner und K. McRitchie: *Software Maintenance Implications on Cost and Schedule*. In: *2008 IEEE Aerospace Conference*, S. 1–6, 2008.

[26] JRebel by Perforce Software, Inc: *Java IDE Usage Stats*. https://www.jrebel.com/blog/java-ide-usage-stats, 2014.

[27] JRebel by Perforce Software, Inc: *2020 Java Technoology Report*. https://www.jrebel.com/blog/2020-java-technology-report#IDE, 2020.

[28] Koscke, R.: *Animation of the Evolution of Linux Net Subsystem with Tree Map Layout*. https://www.youtube.com/watch?v=w7gDIHzEW0Q, 2020.

[29] LaToza, T. D., G. Venolia und R. DeLine: *Maintaining Mental Models: A Study of Developer Work Habits*. In: *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, S. 492–501, New York, NY, USA, 2006. Association for Computing Machinery.

[30] Lewis, M. und J. Jacobson: *Game engines*. Communications of the ACM, 45(1):27–31, 2002.

[31] Lientz, B. P., E. B. Swanson und G. E. Tompkins: *Characteristics of Application Software Maintenance*. Commun. ACM, 21(6):466–471, Juni 1978.

[32] Lippmann, D.: *IMathAS - An Internet Mathematics Assessment System*. https://www.imathas.com/. Link zum Boxplot Grapher https://www.imathas.com/stattools/boxplot.html.

[33] Malaka, R. und T. Döring: *Mensch-Technik-Interaktion*, 2019. Wintersemester 2019-2020, Kapitel Evaluation.

[34] McClure, C.: *The Three Rs of Software Automation: Re-Engineering, Repository, Reusability*. Prentice-Hall, Inc., USA, 1992.

[35] Microsoft: *Skype*. https://www.skype.com/de/.

[36] Nguyen, V.: *Improved size and effort estimation models for software maintenance*. In: *2010 IEEE International Conference on Software Maintenance*, S. 1–2, 2010.

[37] Ogheneovo, E. E.: *On the Relationship between Software Complexity and Maintenance Costs*. Journal of Computer and Communications, 2:1–16, 2014. http://dx.doi.org/10.4236/jcc.2014.214001.

[38] Oracle Corporation: *Java™ Debug Interface*.

[39] Oracle Corporation: *Oracle Java*. https://www.oracle.com/de/java/.

[40] Preece, J., H. Sharp und Y. Rogers: *Interaction design: beyond human-computer interaction*. John Wiley & Sons, 2002.

[41] Rauer, M.: *Quantitative Usability-Analysen mit der System Usability Scale(SUS)*. https://blog.seibert-media.net/blog/2011/04/11/usablility-analysen-system-usability-scale-sus/.

[42] Rudis, B., N. Ross und S. Garnier: *The viridis color palettes*. https://cran.r-project.org/web/packages/viridis/vignettes/intro-to-viridis.html, 2018.

[43] Sauro, J.: *Measuring usability with the System Usability Scale(SUS)*. https://measuringu.com/sus/, February 2011.

[44] Schumann, H. und W. Müller: *Visualisierung: Grundlagen und allgemeine methoden*. Springer Berlin Heidelberg, 2013.

[45] Siegmund, B., M. Perscheid, M. Taeumel und R. Hirschfeld: *Studying the Advancement in Debugging Practice of Professional Software Developers*. In: *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, S. 269–274, 2014.

[46] Tashtoush, Y., M. Al-Maolegi und B. Arkok: *The Correlation among Software Complexity Metrics with Case Study*. International Journal of Advanced Computer Research, 4(2):414–419, 2014. https://arxiv.org/abs/1408.4523.

[47] Tassey, G.: *The Economic Impacts of Inadequate Infrastructure for Software Testing*. 05 2002.

[48] TeamViewer AG: *TeamViewer*. https://www.teamviewer.com/de/.

[49] Unity Technologies: *Unity*. https://unity.com/de.

[50] Wettel, R. und M. Lanza: *Visualizing Software Systems as Cities*. In: *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, S. 92–99, 2007.

[51] Wettel, R. und M. Lanza: *CodeCity: 3D Visualization of Large-Scale Software*. ICSE Companion '08, S. 921–922, New York, NY, USA, 2008. Association for Computing Machinery.

[52] Wettel, R., M. Lanza und R. Robbes: *Software systems as cities: A controlled experiment*. In: *Proceedings of the 33rd International Conference on Software Engineering*, S. 551–560, 2011.

[53] Yu, D.: *A View on Three R's (3Rs): Reuse, Re-Engineering, and Reverse-Engineering*. SIGSOFT Softw. Eng. Notes, 16(3):69, Juli 1991.

[54] ZELLER, A.: *CHAPTER 1 - How Failures Come to Be*. In: ZELLER, A. (Hrsg.): *Why Programs Fail (Second Edition)*, S. 1 – 23. Morgan Kaufmann, Boston, Second Edition Aufl., 2009.