HAW HAMBURG

BACHELORTHESIS
Johannes Nodop

Dynamische Visualisierung von Objektzuständen zur Verbesserung des Programmverständnisses

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Johannes Nodop

Dynamische Visualisierung von Objektzuständen zur Verbesserung des Programmverständnisses

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Jens von Pilgrim
Zweitgutachter: Prof. Dr. Axel Schmolitzky

Eingereicht am: 29. März 2022

Johannes Nodop

Thema der Arbeit
Dynamische Visualisierung von Objektzuständen zur Verbesserung des Programmverständnisses

Stichworte
Visualisierung, Programmverständnis, Objektzustände, Debugger

Kurzzusammenfassung
Um gute, objektorientierte Software entwerfen und entwickeln zu können, ist ein ausführliches Verständnis von Objekten und deren Strukturen erforderlich. Komplexere Datenstrukturen sind jedoch schwer zu durchschauen, erst recht für Programmieranfänger. Dies kann durch die Entwicklung eines dynamischen Visualisierers erleichtert werden, welcher die Abläufe und Strukturen eines ausgeführten Programmes aufbereitet. Ein im Rahmen dieser Arbeit entwickelter Prototyp soll als Grundlage für eine Einbindung in das 'OPPSEE' Projekt dienen. Dadurch bestehen gewisse Anforderungen an die Arbeit, wie die Teilung der Anwendung in eine Client- und eine Serverkomponente, wobei die Clientkomponente als eine Erweiterung für die Entwicklungsumgebung 'Visual Studio Code' entwickelt werden soll. Mit Hilfe der konzeptionierten und entwickelten Anwendung soll das Programmverständnis von Benutzern verbessert werden.

Johannes Nodop

Title of Thesis
Dynamic visualization of object states to improve program comprehension

Keywords
Visualization, Program comprehension, object states, Debugger

Abstract To be able to design and implement good, object oriented software it requires a thorough understanding of objects and their structures. Complex datastructures are hard to comprehend, especially for beginners. This can be improved by the development of a dynamic visualizer, which processes the inner workings and structures of an executed program. The in the context of this thesis designed prototype is a blueprint for an integration into the 'OPPSEE' project. Therefore some requirements are given. The prototype is to be seperated into a client and a servercomponent, while the clientcomponent is to be designed as an expansion for the development enviroment 'Visual Studio Code'. With the help of the conceptualized and developed application the program comprehension of users should be improved.

Inhaltsverzeichnis

Abbildungsverzeichnis

Glossar

API Application Programming Interface.

HTML HyperText Markup Language.

IFrame InlineFrames.

JDI Java Debug Interface.

JDWP Java Debug Wire Protocol.

JPDA Java Platform Debugger Architecture.

JSON JavaScript Object Notation.

JVMDI Java Virtual Machine Debug Interface.

LIFO Last in First out.

OPPSEE Online Programming Platform for Software Engineering Education.

UML Unified Modeling Language.

VM Virtuelle Maschine.

VSCode Visual Studio Code.

1 Einleitung

Im Laufe der Programmierausbildung ist es wichtig, ein tiefes und umfängliches Wissen über die internen Abläufe von entwickelten Programmen zu erlangen. Komplexere Programme, höhere Objekte und verschachtelte Sammlungen zu verstehen ist dabei nicht trivial. An der HAW-Hamburg wird die Online-Programmier-Plattform 'Online Programming Platform for Software Engineering Education' (OPPSEE) entwickelt, um die Programmierfähigkeiten von Studenten zu verbessern und die Motivation zum Lernen zu erhöhen.
Um die konkreten Abläufe des entwickelten Codes besser zu veranschaulichen und eventuelle Fehler aufzuzeigen, soll im Rahmen dieser Arbeit eine prototypische Anwendung entwickelt werden, welche in erweiterter Form innerhalb von OPPSEE eingesetzt werden kann. Diese soll ähnlich wie ein Debugger arbeiten und bestimmte Objekte beobachten, deren Zustände und ihre Veränderungen jedoch optisch aufbereiten und zusätzlich komplexere Objektstrukturen wie mehrdimensionale Arrays grafisch besser verständlich machen.
Das Ziel dieser Anwendung ist es, das Verständnis von Programmabläufen zu verbessern und gleichzeitig durch die von der Visualisierung eingeführte Form einer leichten 'Gamification' die Motivation und Freude am Lernen zu erhöhen und damit die Ziele von OPPSEE zu unterstützen, die Ausbildung von Software-Ingenieuren*innen weiter zu fördern.
Im Rahmen der Ausarbeitung soll aufgrund der Ähnlichkeit zu anderen, bereits verfügbaren Online-Programmier-Plattformen ein Bezug zu diesen hergestellt werden. Aufgrund der umfassenden Möglichkeiten an Funktionen, wie die Unterstützung verschiedener Programmiersprachen, Datenstrukturen oder Visualisierungsarten, die umgesetzt werden könnten, soll mit dieser Anwendung eine Grundlage für eine spätere Weiterentwicklung geschaffen werden.
Die OPPSEE Plattform wird als eine Webanwendung entwickelt und basiert auf dem 'Visual Studio Code' (VSCode) Editor von Microsoft. Daher soll der Teil der prototypischen Anwendung, welcher gesammelte Daten visualisiert, mit Hilfe der Extension-API ('Application Programming Interface') als eine Erweiterung für VSCode entwickelt werden.
Die Erstellung der zu visualisierenden Daten hingegen soll als unabhängige Anwendung realisiert werden, damit diese Anwendung serverseitig ausgeführt werden kann um Daten von Benutzern zu empfangen. Auf dem Server werden diese validiert und der darin enthaltene Programmcode mit einer festgelegten Konfiguration ausgeführt. Währenddessen werden zu visualisierende Daten erzeugt, welche abschließend an den Benutzer zurückgeschickt werden.
Für die Umsetzung wird sich für die serverseitige Anwendung auf die Ausführung von auf der Programmiersprache 'Java' basierenden Programmen beschränkt.
Zunächst werden kurz die technischen Grundlagen und die eingesetzten Technologien erläutert. Anschließend wird sich im theoretischen Abschnitt mit verwandten Arbeiten beschäftigt. Die eigentliche Anwendung wird dann zuerst theoretisch betrachtet, indem eine Übersicht über die Architektur aufgebaut und anschließend tiefergehend betrachtet wird. Nach dem theoretischen Aspekt wird die Umsetzung mit ihren Besonderheiten und Einschränkungen beschrie-

ben. Nach einer abschließenden Evaluation werden die wichtigsten Ergebnisse zusammengefasst.

2 Java Debug Interface(JDI)

Das 'Java Debug Interface' (JDI) ist eine von der Firma 'Sun' bereitgestellte Java-API. Dieses ermöglicht es Entwicklern innerhalb einer 'Debugger'-Anwendung Informationen über den (laufenden) Zustand einer 'Debuggee'-'Virtuelle Maschine' (VM) zu erhalten. Um debugbar zu sein, muss eine solche VM das 'Java Virtual Machine Debug Interface' (JVMDI) implementieren, welches wie das JDI ein Teil der 'Java Platform Debugger Architecture' (JPDA) ist[3], was die Grundlage für die hier beschriebene Anwendung bildet. Mit Hilfe des JDI lassen sich delegierte VM's vollständig kontrollieren sowie inspizieren. Insbesondere kann die Ausführung einzelner Threads angehalten werden, um so den jeweiligen Zustand und die in ihm enthaltenen, lokalen Variablen auszulesen. Hierfür werden unter anderem `Breakpoints` und `Watchpoints` eingesetzt. Durch das Auslesen der Zustände an solchen Punkten können präzise Informationen über den Programmablauf gewonnen werden, welche in aufbereiteter Form einem Benutzer dabei helfen können, die internen Vorgänge seines Codes besser zu verstehen und so zu einer Verbesserung des Programmverständnisses führen. Zusätzlich können die hierdurch gewonnen Daten dabei helfen eventuelle Fehlerquellen leichter auffinden und korrigieren zu können.
Bei einem Großteil der vom JDI bereitgestellten Klassen handelt es sich um Interfaces. Ausgenommen hiervon sind sämtliche `Exceptions`, die Klasse `Connection`, welche den Kommunikationskanal zwischen Debugger und Debuggee beschreibt sowie die Klasse `Bootstrap`, welche als Zugriffspunkt auf die Standardimplementierung der einzelnen Interfaces des JDI dient.

2.1 Spiegelkonzept

Als grundlegendes Interface fungiert das `Mirror`-Interface, welches, wie in (1) in Abbildung 1 zu sehen, die Basis für die Repräsentation einer jeden Entität des Debuggee innerhalb des Debuggers bildet. Nahezu alle weiteren Interfaces erweitern `Mirror` oder eines der Subinterfaces dieser Klasse. Eine der wenigen Ausnahmen bildet beispielsweise `Locatable`(4), da dies ein Konzept losgelöst von der Debuggee-VM ist, welches selbst innerhalb der Debugger-Komponente erweitert und eingesetzt wird.

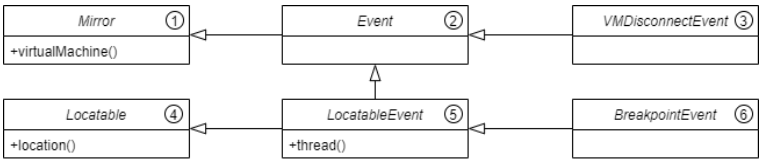

Abbildung 1: UML-Klassendiagramm: Vererbung von Events im JDI

Die Eigenschaften eines 'Mirror' können von der Debugger-VM ausgelesen aber auch manipuliert werden. Ein Mirror dient als Stellvertreter einer Entität der Debuggee-VM für den Debugger, selbst die VM des Debuggee basiert auf dem

Mirror-Interface.
Als einfach verständliches Beispiel können hier die primitiven Datentypen herangezogen werden. Treten innerhalb einer Debuggee-VM primitive Datentypen wie beispielsweise ein Integer auf, so wird dieser Wert innerhalb des Debuggers als `IntegerValue` gespiegelt. Beim `IntegerValue` handelt es sich selbst um ein erbendes Interface, welches auf dem `PrimitiveValue` basiert. Dieses wiederum auf `Value`, welches selbst ein Subinterface des `Mirror` ist. Wie sich anhand des `IntegerValue` schon erkennen lässt, kann die Vererbung innerhalb des JDI sehr komplex werden. So wird `Mirror`(1) vom Interface `Event` beerbt (2), von dem daraufhin `LocatableEvent` (5) erbt. Diese Art von Event erweitert zusätzlich das Interface `Locatable` (4), somit ist die genaue Zuordnung einer Position innerhalb des ausführenden Codes der DebuggeeVM zu einem solchen Event möglich. Zu diesen `LocatableEvents` zählt unter anderem das `BreakpointEvent` (6), welches im Rahmen dieser Anwendung eingesetzt wird. Im Gegensatz hierzu stehen einfache Events, die nicht zu einer festen Position zugeordnet werden können, welche direkt das Interface `Event` erweitern. Das `VMDisconnectEvent` (3) tritt durch die Terminierung der DebuggeeVM auf. Da dies durch verschiedenste Aktionen auftreten kann, gibt es nicht zwangsweise eine genaue Position im Code.
Somit ergibt sich, dass die von `Mirror` bereitgestellte Methode `virtualMachine()`, welche zur Bestimmung der VM eines Mirrors genutzt wird, allen Events bekannt ist. Die Methoden `Location()` und `Thread()` zur Bestimmung der Position oder des Threads können jedoch nur innerhalb der `LocatableEvents` aufgerufen werden.

2.2 Eventkonzept

Ob `Locatable` oder nicht, die Events dienen als Kommunikationsmittel für die Kommunikation zwischen dem JDI implementierenden Debugger und dem JVM-DI konformen Debuggee.
Tritt im Debuggee ein gewisses Ereignis auf, wird dem Debugger unter Nutzung des 'Java Debug Wire Protocol' (JDWP)[4]-Transportprotokolls von diesem in Form eines Events berichtet. Diese Events werden von der `EventQueue` verwaltet und immer in sogenannten `EventSets` gruppiert, selbst dann, wenn diese alleine auftreten.
Obwohl es eine Vielzahl an Events gibt, sind für diese Anwendung nur wenige von Relevanz. Zu diesen zählen das `ClassPrepareEvent`, das `VMDisconnectedEvent`, das `BreakpointEvent` sowie die beiden Erweiterungen des `WatchpointEvent`, namentlich `Access-` und `ModificationWatchpointEvent`.
Einen Teil der Vererbung und Eigenschaften der Events kann aus der vorherigen Abbildung 1 sowie der späteren Abbildung 3 entnommen werden. Die `WatchpointEvents` werden in den angesprochenen Abbildungen nicht aufgeführt, um diese so übersichtlicher zu gestalten.

Das `ClassPrepareEvent` dient als Benachrichtigung darüber, dass eine Klasse in der VM des Debuggee vorbereitet wird. Zu dieser Vorbereitung zählt unter anderem das Anlegen und Initialisieren von statischen Feldern.
Im Gegensatz zu diesem, für den Debugger initialisierendem Event, steht das `VMDisconnectedEvent`, welches beim Verbindungsabbau zwischen Debugger und Debuggee auftritt. Sollte die VM unerwartet abstürzen, steht vor dem `VMDisconnectedEvent` noch ein `VMDeathEvent`. Da dieses Event das Ende des Lebenszykluses der Debuggee-VM beschreibt, würden alle folgenden Abfragen an die Debuggee-VM zu einer `VMDisconnectedExceptions` führen. Innerhalb des Prototypen werden keine Unterscheidungen im Bezug auf die Art der Terminierung vorgenommen, weswegen ein `VMDeathEvent` ignoriert werden kann.

Beim dritten und dem für den Debugger wohl relevantesten Event handelt es sich um das `BreakpointEvent`. Diese Events treten dann auf, wenn ein vorher gesetzter Breakpoint erreicht wird. Die einzelnen Breakpoints können frühestens als Reaktion auf das `ClassPrepareEvent` gesetzt werden und beinhalten eine `Locatable` Position. Beim Auftreten dieses Events stoppt die Ausführung des Debuggee-Threads solange bis dieser einen `Resume()` Befehl empfängt. Während dieses anhaltenden Zustands ist es dem Debugger möglich, auf die Informationen des Threads wie beispielsweise die einzelnen lokalen Variablen und deren Werte, zuzugreifen.
Hierzu wird der aktuelle `Stackframe` vom `CallStack`[1] eines Threads genommen und weiter verarbeitet.

Die letzten für diese Anwendung relevanten Events sind die `WatchpointEvents`, welche in `AccessWatchpointEvent` und `ModificationWatchpointEvent` unterteilt werden.
Ein `WatchpointEvent` unterscheidet sich von einem `BreakpointEvent` dahingehend, dass dieses nicht an einer festen `Location` ausgelöst wird. Stattdessen wird ein Feld innerhalb der Debuggee-VM ausgewählt, welches von einem `Watchpoint` beobachtet werden soll.
Diese Felder müssen zum Zeitpunkt der Einrichtung eines `WatchpointEvents` bereits initialisiert sein, daher kann ein `Watchpoint` nicht als Reaktion auf ein `ClassPrepareRequest` angelegt werden.
Stattdessen können diese beispielsweise als Reaktion auf ein `BreakpointEvent` erzeugt werden. Der Unterschied zwischen den beiden Ausprägungen der `WatchpointEvents` ist deren Auslöser. Ein `AccessWatchpointEvent` tritt dann auf, wenn auf das beobachtete Feld zugegriffen wird. Ein `ModificationWatchpointEvent` hingegen nur, wenn das beobachtete Feld modifiziert wird.
Im Zusammenhang mit den `Watchpoints` stellt die offizielle API des JDI leider nur wenig Informationen bereit und im Gegensatz zu den `Breakpoints` sind auch andere Quellen in Literatur und dem Internet minimal und unzureichend.

[1] Auch `ThreadStack` oder nur `Stack` genannt.

2.3 Frame und Stack

Bei einem Callstack handelt es sich um eine auf dem abstrakten Datentypen Stack basierenden Sammlung einzelner Frames. Dadurch ist der oberste Frame auf diesem Stack immer der neuste, denn ein Frame selbst entsteht immer dann, wenn eine Methode aufgerufen wird und wird auch genau dann auf den Stack gepusht. Ähnlich wie ein `BreakpointEvent` entsteht ein Frame an einer festen Position, daher handelt es sich beim Interface `StackFrame` erneut um eine Erweiterung des `Locatable` Interface. Die Entstehung von Frames und die mit dem Stack verbundenen `push` und `pop` Vorgänge lassen sich durch ein einfaches Beispiel anhand von Algorithmus 1 und Abbildung 2 schnell verstehen.

Algorithm 1 Beispiel Stack & Frame (Pseudocode)

```
1: function a(){
2:     b();
3: }
4: function b(){
5:     console.log("Hello World");
6: }
```

In dieser vereinfachten Darstellung (Abbildung 2) ist der Stack in seinem Ausgangszustand (1) leer. Sobald nun die Funktion `a()` aufgerufen wird, erzeugt der ausführende Thread einen Frame, welcher auf den Stack gepusht wird (2). Im Funktionsrumpf von Funktion `a()` wird die Funktion `b()` aufgerufen (2), so dass nun auch für diese ein Frame erzeugt und zusätzlich auf den Stack gepusht wird (3). Der Stack entspricht daraufhin dem Zustand (4). Werden im Folgenden keine weiteren Funktionen aufgerufen, so werden die Frames nach dem 'Last in First out' (LIFO) Verfahren abgerufen und verarbeitet. Zuerst wird der Frame von Funktion `b()` entfernt, da dieser Frame zuletzt auf den Stack gepusht wurde (5). Anschließend folgt der Frame von Funktion `a()` (6), womit der Stack wieder den Zustand (1) annimmt, den er zu Beginn der Ausführung hatte.

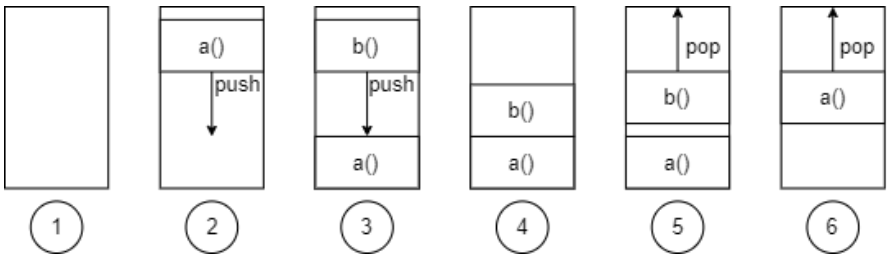

Abbildung 2: Vereinfachte Ansicht eines Callstack

Anders als auf den festen Speicherbereich des 'Stack' unterstützt das JDI keine Zugriffe auf den dynamischen Speicherbereich des 'Heap'.

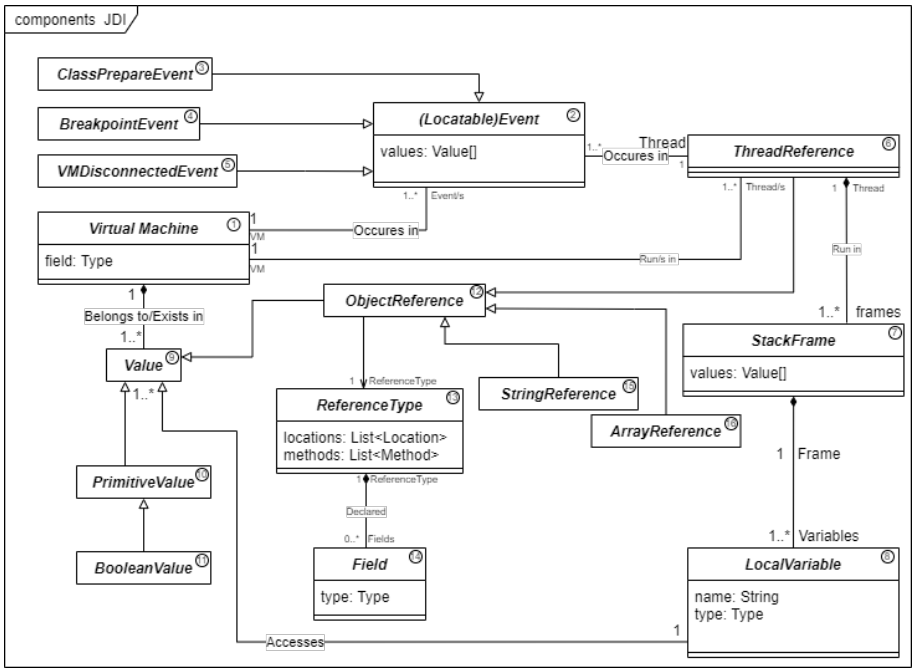

Abbildung 3: UML-Klassendiagramm: Komponenten des JDI

Anhand von Abbildung 3 sollen die folgenden, tiefergehenden Erläuterungen visuell greifbar gemacht werden.
Die einzelnen Frames im Stack (7) eines Threads ermöglichen jeweils den Zugriff auf die lokalen Variablen (8) der aufgerufenen Methode sowie deren derzeitige `Values` (9) in Form von Referenzen.
Innerhalb eines Debuggers wird ein Thread als `ThreadReference` (6) gehandhabt, was wiederum eine Erweiterung des bereits angesprochenen `Mirror` Interfaces und damit eine Spiegelung eines Threads aus der DebuggeeVM ist (1).
Die Values (9) selbst werden unterteilt in einfache `PrimitiveValues` (10) und höhere `ObjectReferences` (12).
Diese beiden Subinterfaces bilden die Grundlage für weitere Unterteilungen. Die `PrimitiveValues` werden von je einem Interface für jeden primitiven Datentyp der Programmiersprache Java erweitert.
Das Interface `BooleanValue` (11) beispielsweise ermöglicht den Zugriff auf eine Instanz eines booleschen Wertes in der Debuggee-VM über die simple Methode `value()`. Das Subinterface `ObjectReference` (12) hingegen bildet ein höherklassiges Objekt der Debuggee-VM(1) ab.
Solch eine Referenz verhält sich genauso wie die aus dem Java-Alltag bekannten Referenzen, im Bezug darauf, dass nur das referenzierte Objekt selbst gespiegelt wird, nicht aber darin enthaltene Felder oder Werte. Über eine `ObjectReference` lassen sich Informationen über den Inhalt eines Objekt abrufen, somit ist es auch möglich, dass eine `ObjectReference` eine oder mehrere

weitere solcher oder anderen Referenzen enthält.
Im späteren Kapitel 'Beispiele' wird ein Beispiel basierend auf einem Autorennen vorgestellt, um das Verständnis der `ObjectReferences` zu erhöhen, wird dieses hier nun aber teilweise vorgegriffen. Eine Klasse `ArrayRennbahn`, welche mehrere Objekte der Klasse `Auto` als ein Array enthält, würde zum Beispiel als eine `ObjectReference Rennbahn` abgebildet werden, welche selbst eine `ArrayReference` (16) enthält, die wiederum aus einer Sammlung von `ObjectReferences` des Typen `Auto` besteht. Diese besagte `ArrayReference` ist eine Erweiterung der herkömmlichen `ObjectReference`, gleiches gilt für andere, höhere Objekte wie zum Beispiel für `StringReferences` (15) oder die schon bekannten `ThreadReferences` (6). Die Werte innerhalb einer `ArrayReference` basieren auf dem grundlegenden Interface `Value` und können dementsprechend auch primitiver Natur sein. Intern werden die Values in einer `ArrayReference` nicht als Array sondern als Liste gespeichert, um so die Konsistenz und Kompatibilität zwischen dem JDI und anderen APIs zu verbessern.
Jede Instanz einer `ObjectReference` verfügt über einen `ReferenceType` (13), welcher den Typen des Objekts innerhalb der Debuggee-VM spiegelt. Je nach Ausprägung des Objekts wird eine Referenz entweder durch
`ClassType` (Klassen), `InterfaceType` (Interfaces) oder `ArrayType` (Arrays) erweitert. Unabhängig von der jeweiligen Erweiterung bietet der `ReferenceType` Zugriff auf typspezifische Informationen wie den `Classloader` sowie die Methoden und Felder eines Typen.
Somit lässt sich zusammenfassend feststellen, dass durch die Nutzung von Breakpoints und einer Kombination von Frames, Values und Referenzen der Zustand einer Debuggee-VM an einer vorher definierten Stelle vollständig ausgelesen und bei Bedarf sogar manipuliert werden kann.

3 Related Work

3.1 JAVAVIS

Schon 2002 wurde von Forschern der Fachhochschule Trier eine Anwendung namens JAVAVIS vorgestellt.[4] Das Ziel von JAVAVIS ähnelt dem dieser Anwendung: Es soll Studenten dabei helfen, die internen Abläufe einer in Java geschriebenen Anwendung (während deren Ausführung) besser zu verstehen. Ebenfalls wie in der hier vorgestellten Anwendung wurde auf das schon damals existierende JDI zurückgegriffen, um an die benötigten Informationen einer laufenden Applikation zu gelangen. Im Gegensatz zu der Darstellung einzelner Datenstrukturen werden in JAVAVIS jedoch zwei Arten von 'Unified Modeling Language' (UML)-Diagrammen dynamisch erzeugt. So ist es dem Nutzer möglich, den Programmcode seiner Debuggee-Anwendung Aufruf für Aufruf oder Zeile für Zeile zu durchlaufen und dabei die Veränderungen anhand eines Objekt- oder Sequenzdiagramms zu beobachten. In der aus der Veröffentlichung entnommenen, beispielhaften Abbildung 4 lässt sich ein solcher Vorgang anhand eines Sequenzdiagramms erkennen. Hier wird im ersten Schritt zunächst eine Liste initialisiert(53), in die daraufhin in zwei weiteren Schritten(56, 57) je ein frisch initialisiertes `ListItem` eingefügt wird. Der statischen Abbildung dieser Arbeit ist es geschuldet, dass die dynamische Generierung des Diagramms nicht besser abgebildet werden kann.

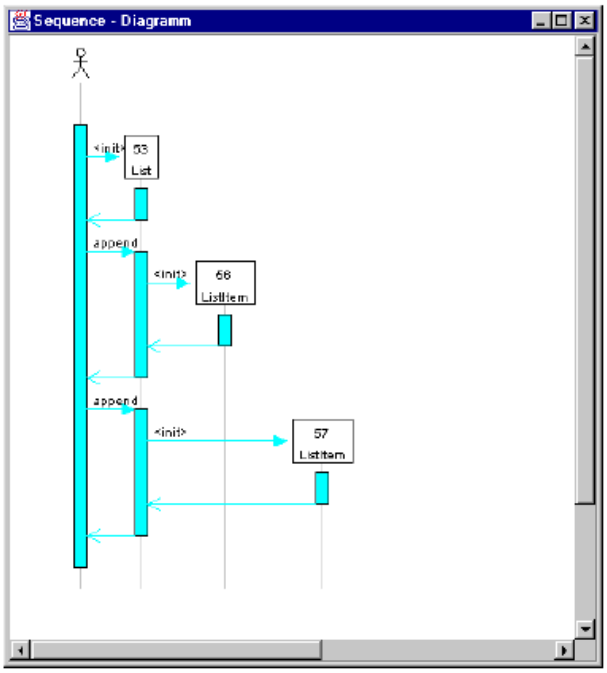

Abbildung 4: Beispiel der JAVAVIS Anwendung (Quelle: [4], Seite 183)

3.2 Coding Games

In den letzten Jahren wurden verschiedene Arten von 'Coding Games' veröffentlicht. Die vermeintlich bekannteste Variante ist das 'Codingame'[2], eine Website welche (unter anderem) Programmieraufgaben verschiedener Größe und Schwierigkeit anbietet und einen Online-Editor zur Verfügung stellt, in dem diese Aufgaben gelöst werden sollen. Hierzu stehen einem Benutzer diverse Programmiersprachen zur Verfügung.
Das besondere Merkmal dieser und anderer Coding Games ist die Gamifizierung. Bei der Gamifizierung handelt es sich um die Idee, aus Spielen bekannte Elemente wie ein Punkte-, Aufgaben- oder Levelsystem, aber auch visuell ansprechende, grafische Aufbereitung auf andere Bereiche zu übertragen, um die Motivation und den Lernerfolg von Lernenden zu verbessern.[1] Im Falle des 'Codingame' werden unter anderem ein Levelsystem, ein Aufgabensystem und ein Auszeichnungssystem geboten, um Langzeitmotivation zu erzeugen. Für den kurzweiligeren Erfolg werden die einzelnen Aufgaben mit Szenarien aus der Popkultur ausgestattet.
Eine der ersten Aufgaben ist eine einfache Maxima-Berechnung, welche hier als Beispiel dienen soll. Das hierfür gewählte Szenario, welches in Abbildung 5 zu sehen ist, ist einfach gehalten:

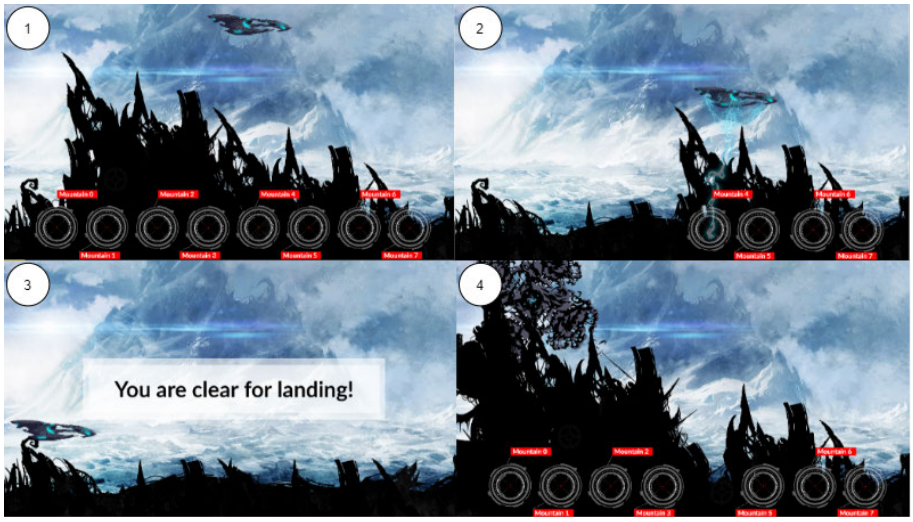

Abbildung 5: Codingame Beispiel

Ein Raumschiff soll auf einem Planeten landen, doch einige Berge sind im Weg.(1) Während des Landeanflugs müssen diese Berge also in absteigender Reihenfolge zerstört werden.(2) Hierzu muss ein Algorithmus entwickelt werden, welcher aus den bereitgestellten Bergen solange den höchsten aussucht und

[2] https://www.codingame.com/start

zerstört, bis das Raumschiff gelandet ist.(3) Sollte dies nicht richtig funktionieren, wird das Raumschiff an einem Berg zerschellen.(4)
Kommt es zu einem Problem im Ablauf des entwickelten Algorithmus, hilft die visuelle Aufbereitung dem Benutzer schnell den Ursprung des Problems zu finden. In der angesprochenen Abbildung ist beispielsweise ersichtlich, dass 'Mountain 4'(4), also der Berg an Index 5, zerstört wurde, obwohl dies nicht der größte war.

Die Gamifizierung ist noch eine relativ neue Erscheinung, weshalb sich noch kein wissenschaftlicher Konsens in Bezug auf deren Effizienz ergeben hat. Experimente wie das 'MCP-Experiment' [1] zeigen bisher jedoch vielversprechende Ergebnisse. Die Lernenden berichten von erhöhter Motivation zu lernen, was sich in der Bewertung ihrer Arbeiten widerspiegelt. Leider fehlen aber noch längerfristige, empirische Beweise um hierzu eine definitive Aussage treffen zu können. Eine potenzielle Fehlersuche ist durch eine grafische Aufbereitung, wie am Beispiel zu sehen, auf jeden Fall einfacher.

Neben dem 'Codingame' gibt es viele Alternativen, deren Ziel ähnlich ist, der Fokus jedoch auf anderen Punkten liegt. Die Seite 'CodeCombat'[3] richtet sich von der Gestaltung und dem Inhalt her eher an eine jüngere Audienz und versucht diese an das Programmieren heranzuführen. Im Endeffekt ist das Ziel aber ein ähnliches wie bei 'Codingame', eine Steigerung von Motivation und Lernerfolg.
Eine Variante ohne eine visuelle Aufbereitung wäre 'Codewars'[4]. Diese Seite unterstützt eine große Anzahl an Programmiersprachen und bietet gamifizierte Inhalte wie ein Fortschrittssystem, verzichtet dabei jedoch auf eine aufwendige, grafische Darstellung.

3.3 Jinsight

Bei 'Jinsight'[2] handelt es sich um eine schon etwas ältere Idee, welche noch auf Java 2 basiert. Wie schon JAVAVIS wurde diese Anwendung im Jahre 2002 vorgestellt und ist damit beinahe zwanzig Jahre alt.
Der Fokus liegt hierbei auf einer Visualisierung des Laufzeitverhaltens einer Anwendung. Dies geschieht mit dem Ziel, die Interaktion zwischen verschiedenen Threads und das Verhalten des `Garbage Collectors` zu veranschaulichen. Gleichzeitig soll 'Jinsight' dabei helfen, eventuelle `Deadlocks` oder `Memory leaks` ausfinding zu machen. Um dies zu ermöglichen werden verschiedene Ansichten angeboten, so zum Beispiel ein Histogramm, welches die Objekte verschiedener Datentypen und deren Verbindungen anzeigen kann. Eine weitere Variante wäre die `Execution View`, welche einen `CallStack` (siehe Frame und Stack) und dabei die Ausführungsdauer einzelner Methoden veranschaulicht, wodurch zusätzlich vermeintlich performance-beeinträchtigende Methoden identifiziert werden können. Jinsight nutzt 'tracing', welches das Aufzeichnen einer

[3] https://codecombat.com/
[4] https://www.codewars.com/

Spur eines ausgeführten Programmes beschreibt. Der Inhalt dieser Spur kann vom Benutzer eingegrenzt werden. Eine Spur kann beispielsweise den Namen sowie Eintritts- und Austrittszeitpunkt einer Funktion beinhalten. Hierzu werden während der Ausführung einer Applikation, über das JDI, die notwendigen Informationen gesammelt und im Nachhinein visualisiert.

4 Entwurf

In den folgenden Unterkapiteln soll der konzeptionelle Entwurf der Anwendung erläutert werden. Dafür wird zunächst auf die einzelnen Komponenten eingegangen und anschließend eine Übersicht der verwendeten Datenstrukturen gegeben. Anschließend wird der Funktionsablauf der beiden Anwendungskomponenten dargestellt und abschließend auf Einschränkungen eingegangen, die getroffen worden sind.

4.1 Komponenten

Die eigentlich Anwendung lässt sich einfach in zwei, voneinander theoretisch sogar unabhängige Komponenten, unterteilen, dessen Kommunikation grob in Abbildung 6 dargestellt ist.
Zuerst wird die 'Debuggerkomponente'(1) verwendet, welche eine Visualisierungsdatei(3) erzeugt. Diese kann von der 'Visualisiererkomponente'(2) eingelesen und verarbeitet werden. Die Kommunikation zwischen den beiden Komponenten findet dabei asynchron über die angesprochene Visualisierungsdatei(3) in 'JavaScript Object Notation' (JSON)-Syntax statt.

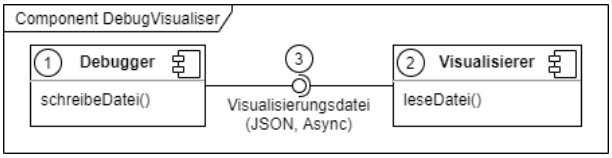

Abbildung 6: UML-Komponentendiagramm: Komponenten & Kommunikation

4.1.1 Debuggerkomponente

Die Debugger-Anwendung basiert auf dem JDI und beinhaltet Funktionalitäten zum Auslesen von Variablen und zur Aufbereitung dieser in eine für die weitere Arbeit verwendbare Form. Im nachfolgenden wird dieser Teil einfach 'Debugger' genannt. Der Debugger liest zu Beginn eine Konfigurationsdatei ein, welche die für seine Ausführung benötigten Informationen enthält. Nach der Ausführung des Codes wird vom Debugger eine neue Datei erstellt, welche die einzelnen Zustände an den jeweiligen, in der Konfigurationsdatei definierten Punkten dokumentiert und eventuelle Testergebnisse beinhaltet.
Wie schon für die Visualisierungsdatei wird auch für die Konfigurationsdatei eine JSON-Syntax verwendet. Eine solche Visualisierungsdatei kann theoretisch auch von anderen Programmen oder einem Nutzer manuell erzeugt werden, was den Debugger vom zweiten Teil, im folgenden 'Visualisierer' genannt, unabhängig macht. Von eben diesem Visualisierer werden die erstellten 'Visualisierungsdateien' eingelesen und verarbeitet. Dementsprechend findet die Kommunikation zwischen den beiden Komponenten auf eine asynchrone Art mit Hilfe von Visualisierungsdateien statt. Beide Komponenten müssen in der Lage sein, diese

Dateien zu verstehen.
Beim Einlesen einer Visualisierungsdatei muss zunächst anhand des Inhaltes der Datei die benötigte Visualisierungsart ermittelt und geladen werden. Technisch gesehen könnte die Art der Visualisierung zwischen den Schritten einer einzelnen Datei wechseln, jedoch würde dies erhöhten Aufwand erfordern und kaum bis keine Vorteile mit sich bringen, weshalb auf die Funktionalität vorerst verzichtet wird. Nach dem Ermitteln der Visualisierungsart wird direkt der erste Schritt geladen und dem Nutzer dargestellt. Sollten sich in der geladenen Datei außerdem Informationen zu ausgeführten Tests befinden, werden diese zusätzlich geladen und neben der Schritt-Visualisierung angezeigt. Der Visualisierer soll es dem Benutzer über eine Benutzeroberfläche ermöglichen, zwischen den einzelnen Schritten der eingelesenen Datei zu wechseln und bei Bedarf eine neue Datei einzulesen.
Da es sich um das **Java**-Debug Interface handelt, wird zur Umsetzung des Debuggers und der Debuggee Testanwendungen die Programmiersprache Java zum Einsatz kommen.
Da, wie in Abbildung 7 zu sehen, für die grundlegende Funktionalität der Debugger-Komponente nur das JDI(1) benötigt wird, gestaltet sich die Architektur dieser Komponente als recht einfach. Um die benötigten JSON-Strukturen zu erstellen wird zusätzlich eine JSON-Bibliothek verwendet. In diesem Fall wurde sich für 'Jackson'(3)[5] entschieden, da es sich hierbei um eine weit verbreitete und gut dokumentierte Bibliothek handelt. Alternativen wie 'Gson' hätten ebenfalls verwendet werden können und ließen sich in der Zukunft austauschen.

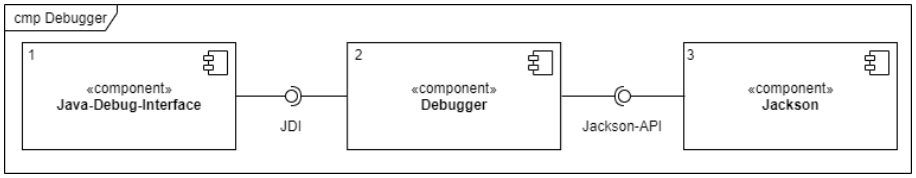

Abbildung 7: UML-Komponentendiagramm:Architektur Debuggerkomponente

4.1.2 Visualisierungskomponente

Die Anforderungen an die 'Visualisierer'-Komponente sehen vor, dass diese als eine Erweiterung für die Entwicklungsumgebung 'Visual Studio Code' entwickelt werden soll. Da die Plattform, die im Rahmen des 'OPPSEE' Projektes entwickelt wird, auf VSCode basiert, liefert der hier entwickelte Prototyp somit einen Beitrag zur Integration der Feedback- und Validierungsmechanismen innerhalb des Projektes.
Aufgrund des Technology-Stack, auf dem VSCode aufbaut, muss eine Erweiterung als eine Node.js-basierte Anwendung realisiert werden. VSCode baut auf dem Anwendungsframework 'Electron' auf, zusammen mit der bereitgestellten Webview-API[6] lassen sich Erweiterungen nahezu exakt wie eine Website gestal-

[5] https://github.com/FasterXML/jackson
[6] https://code.visualstudio.com/api/extension-guides/webview

ten. So kann für die Darstellung innerhalb der Erweiterung das React-Framework[7] verwendet werden. Für die finale Darstellung der bereitgestellten Daten wird keine neue Technologie entwickelt. Stattdessen wird hierfür ein ebenfalls bereits vorhandenes Framework, bestehend aus 'Visualizer-Core'[8] und 'Visualizer-Bundle' von Henning Diedrichs verwendet. Das Framework kann verwendet werden, um JSON-formatierte Daten auf verschiedenste Arten zu visualisieren. Das 'Core'-Paket stellt die grundsätzliche Funktionalität zur Visualisierung bereit. Intern wird eine `VisualizationFactory` verwendet, um die verschiedenen `Visualizer` zu verwalten und für gegebene `VisualizationData` den gewünschten oder besten `Visualizer` auszuwählen. Dieser kann die gegebenen Daten in eine `Visualization` umwandeln, welche wiederum gerendert wird. Das 'Bundle'stellt beispielhafte Visualisierungsarten, teils unter Verwendung weiterer Bibliotheken, zur Verfügung.
Leider ist die Funktionalität zum Einfärben von einzelnen Zellen des `Grid-Visualizer` derzeit nicht vollständig entwickelt. Aus diesem Grund wurde diese Funktion innerhalb des Paketes nachgerüstet, um so die gewünschten Änderungen zwischen den einzelnen, aufgezeichneten Schritten farblich hervorheben zu können.

Im Vergleich zur Architektur der Debuggerkomponente ist die Architektur der 'Visualisierer'-Komponente etwas komplexer, da diese als VSCode-Erweiterung realisiert werden soll.
Wie in Abbildung 8 zu sehen, werden dem Visualisierer (5) sowohl die Extension- als auch die Webview-API von VSCode (6) zur Verfügung gestellt. Um die Benutzeroberfläche innerhalb einer 'Webview' zu kreieren, wird das React-Framework (3) verwendet, welches nicht direkt eine API zur Verfügung stellt, sondern als Grundlage für die Gestaltung der Anwendung verwendet wird. Daten werden mit Hilfe von Dropzone (4) aus Dateien eingelesen und über die 'Dropzone-API'[9] vom Visualisierer verarbeitet. Die eigentliche Darstellung der Zustände baut auf dem Vis-Bundle (2) auf, welches wiederum auf dem Vis-Core (1) aufbaut. Das Bundle stellt hierbei über eine `getVisualisations`-Schnittstelle die Visualisierungsarten zur Verfügung.

[7] https://reactjs.org/
[8] https://github.com/hediet/visualization
[9] https://react-dropzone.js.org/

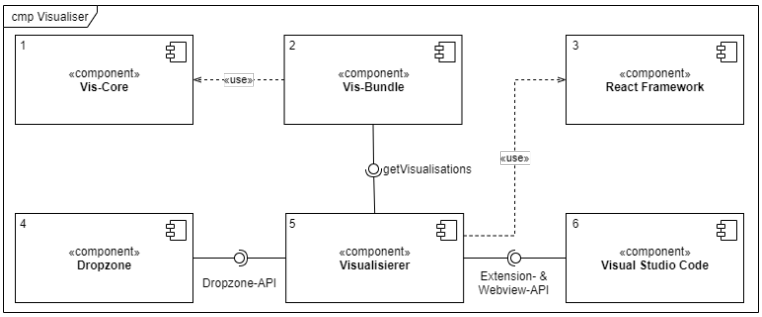

Abbildung 8: UML-Komponentendiagramm:Architektur Visualisierer

4.1.3 Subkomponenten

Innerhalb der beiden Hauptkomponenten kommen weitere, kleinere Subkomponenten zum Einsatz.
Die eigentliche Visualisierer-Komponente basiert auf der hauseigenen VSCode-Extension-API.[10] Diese bietet Entwicklern die Möglichkeit, nahezu alle Aspekte von VSCode zu verändern oder gänzlich neue Funktionen zu entwickeln. Die API ist eines der wichtigsten Bestandteile von VSCode, da sogar viele der Kernfunktionalitäten auf diese zurückgreifen. Aus diesem Grund gibt es eine äußerst hilfreiche und umfangreiche Dokumentation zur Extension-API.
Zusätzlich wird die Extension-API durch die Webview-API erweitert. Hierdurch ist es einer Erweiterung möglich vollständig unabhängige Ansichten zu generieren. Dies wird durch die Unterstützung von 'InlineFrames'(iframes) erreicht. In einem iframe kann ein 'HyperText Markup Language' (HTML)-Dokument gerendert werden, ähnlich wie in einem Webbrowser. Dadurch kann die Visualisierer-Komponente wie eine Website erstellt und anschließend in einer Webview geladen werden. Dementsprechend lässt sich zusätzlich zum HTML-Inhalt auch Javascript-Code ausführen, was die Nutzung von Node.js und Frameworks wie React ermöglicht.
Für die Umsetzung der VSCode-Erweiterung wurde für den Inhalt der Webview auf das React-Framework gesetzt. Dieses vereinfacht die Realisierung von Benutzeroberflächen stark, indem einzelne Daten beobachtet werden und bei Änderungen von deren sogenannten 'States' Funktionen zur Aktualisierung der Oberfläche ausgeführt werden. Außerdem bietet React die Möglichkeit, Typescript zu verwenden und einzelne Komponenten, wie die Dropzone[11], zu definieren und diese in der gesamten Anwendung aufrufen zu können.

Da die im Rahmen dieser Arbeit entstandenen Komponenten 'Debugger' und 'Visualisierer' unabhängig voneinander arbeiten, müssen die vom Debugger erstellten Dateien von der Visualisierungs-Extension wieder eingelesen werden. Hierzu wird die bereits erwähnte React Erweiterung 'Dropzone' verwendet. Die-

[10] https://code.visualstudio.com/api
[11] https://react-dropzone.js.org/

se bietet eine einfache Möglichkeit Dateien per File-Dialog oder ganz simpel per Drag-and-Drop einzulesen und als JavaScript Objekt weiter zu verarbeiten. Somit lassen sich die vom Visualizer benötigten Informationen einfach auslesen und aufbereiten.

4.2 Datenstrukturen

Den beiden bereits erwähnten Dateien (Konfigurations- & Visualisierungsdatei) liegt jeweils ein Muster zugrunde, welche hier weiter erläutert werden soll.

4.2.1 Konfigurationsdatei

Die vom Debugger eingelesene Konfigurationsdatei besteht aus einer Sammlung an JSON-Schlüssel-Wert-Paaren, welche als ein simples Konfigurationsobjekt eingelesen werden:

- mainClass: Name der zu startenden Hauptklasse.(String)
- fileName: Name der zu erstellenden Visualisierungsdatei.(String)
- visType: Gewünschte Visualisierungsart.(String - Default: 'array')
- TestCases: Ist mainClass eine Testdatei?(boolean - Default: 'false')
- Breakpoints: Liste der zu erstellenden Breakpoints.
- Watchpoints: Liste der zu erstellenden Watchpoints.

Bei den Breakpoints handelt es ich um eine eigens erstellte Datenstruktur, welche aus einem Namen, einem Pfad zu einer zu visualisierenden Variable sowie einer Zeilennummer besteht. Die Zeilennummer gibt hierbei an, an welcher Stelle innerhalb der zuvor angegebenen `mainClass` ein Breakpoint gesetzt werden soll. Für Breakpoints, die sich in einer Schleife befinden, wird außerdem die Eigenschaft `isLoop` auf `true` gesetzt. Wenn diese Eigenschaft in einem Breakpoint fehlt, wird sie als `false` initialisiert.
Die Watchpoints sind ebenfalls eine eigens erstellte Datenstruktur. Genau wie für einen Breakpoint muss auch für einen Watchpoint ein Name und ein Pfad zu einer visualisierenden Variable angegeben werden. Da ein Watchpoint ein festgelegtes Feld zu jedem Zeitpunkt beobachtet, wird keine Zeilennummer benötigt. Da jedoch zwischen den `Access-` und `ModificationWatchpoints` unterschieden wird, muss zusätzlich ein `type` in Form von `access` oder `modification` angeben werden. Aufgrund der Ausführung bei jedem Zugriff oder jeder Veränderung ist die Variable `isLoop` überflüssig. Allerdings sind Watchpoints vollkommen optional, weshalb die Liste der Watchpoints im Gegensatz zu den Breakpoints leer sein darf.

4.2.2 Visualisierungsdatei

Die Visualisierungsdatei ist unterteilt in einzelne Schritte, deren Format sich je nach gewünschter Visualisierungsart unterscheidet. Der Aufbau der Datei ist dabei über eine verschachtelte Klassenstruktur definiert, welche in Abbildung 9 aufgezeichnet ist.

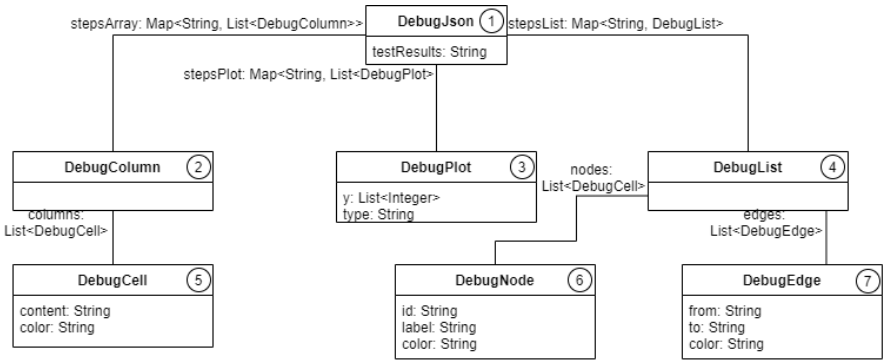

Abbildung 9: UML-Klassendiagramm: Visualisierungsdatei

Auf oberster Ebene wird ein Objekt der Klasse `DebugJson`(1) erzeugt.
Dieses enthält eine Map bestehend aus einzelnen Debug-Schritten, die sich aus jeweils einem String als Schlüssel und einem Objekt als Wert zusammensetzen. Ein solcher Schritt wird vom Debugger für jeden in der Konfigurationsdatei angegebenen Breakpoint erzeugt, sowie für jeden Zugriff auf oder jede Veränderung eines durch einen `AccessWatchpoint` oder `ModificationWatchpoint` beobachteten Feldes. Der Schlüssel ist hierbei der für den jeweiligen Punkt angegebene Name in der Konfigurationsdatei. Sollte es sich um einen Breakpoint innerhalb einer Schleife handeln, zählt eine Zählervariable die Schritte und der Wert dieser Variable wird an den Namen gehängt. Da Watchpoints auf einzelnen Felder liegen, wird auch hierfür eine (separate) Zählervariable verwendet und an den Namen angehängt, um die einzelnen Zugriffe unterscheiden zu können.
Die Informationen, die den Wert bilden, der sich hinter einem Schlüssel versteckt, unterscheiden sich abhängig von der Visualisierungsart. Für eine `Array`-Darstellung wird eine Sammlung aus `DebugColumns`(2) benötigt, wichtig ist hier der Name `columns`, da dieser vom Visualisierer erwartet wird. Eine solche `DebugColumn` besteht aus einzelnen `DebugCells`(5), die wiederum aus den zwei Strings `content` und `color` bestehen. Die Eigenschaft `content` beschreibt hierbei den konkreten Wert, während `color` die Hintergrundfarbe der `debugCell` im Visualisierer definiert und verwendet wird, um etwaige Veränderungen von Objekten zwischen zwei Schritten zu verdeutlichen.
Ein Schritt einer `List`-Darstellung beinhaltet statt den `DebugColumns` eine `DebugList`(4). Diese enthält selbst zwei weitere Sammlungen, `nodes` und `edges`. Die `nodes` bestehen aus `DebugNodes`(6), welche sich aus den Strings `id`, `label` und `color` zusammensetzen. Die `edges` wiederum setzen sich aus `DebugEdges`(7)

zusammen, welche aus den Strings `from`, `to` und `color` bestehen. Das Feld `color` hat die gleiche Funktion wie für eine `Array`-Darstellung. Die `id` wird verwendet, um eine Node zu identifizieren und um als Identifikator für `from` oder `to` einer `Edge` zu fungieren. Das `label` ist das Gegenstück zum `content` und umfasst die eigentliche Information der Node.

Außerdem soll der Visualisierer eine Darstellung von Balken unterschiedlicher Höhe unterstützen. Hierfür wird auf die Graphenvisualisierung des 'Visualizer-Bundles' zurückgegriffen, welche selbst auf der 'Plotly'-Bibliothek[12] aufbaut. Ein Schritt eines solchen `DebugJson` beinhaltet statt `DebugColumns` oder `DebugLists` die `DebugPlots`(3). Im Vergleich zu den vorangegangenen Strukturen gestaltet sich diese einfacher, da sie nur aus einer Sammlung an Integers `y` und dem String `type` besteht. Der `type` gibt hier den Typen des Graphen an, im Falle des gewünschten Balkendiagramms wäre dies 'bar'. Für diese 'bar'-Graphen gibt die Sammlung `y` die Höhe der einzelnen Balken an. Die Komponente des Visualisierers unterstützt durch die Einbindung des 'Visualizer-Bundles' bereits weitere Typen von Graphen, wie Kuchen- oder Kurvendiagramme, welche je nach Typ diverse andere Daten benötigen. Hierfür benötigte Visualisierungsdateien müssen jedoch manuell erzeugt werden, da die Debugger-Komponente zum jetzigen Zeitpunkt nur die Visualisierung von Daten für Balkendiagrammen unterstützt.

[12] https://github.com/plotly/react-plotly.js/

4.3 Funktionsablauf

4.3.1 Debuggerkomponente

Der geplante, genauere Ablauf der Debugger-Anwendung ist in Abbildung 10 in einer etwas vereinfachten Form zu erkennen. Um einen besseren Überblick zu gewähren, wurden die alternativen Abläufe für Debuggee-Anwendungen mit Testfällen nicht eingezeichnet.

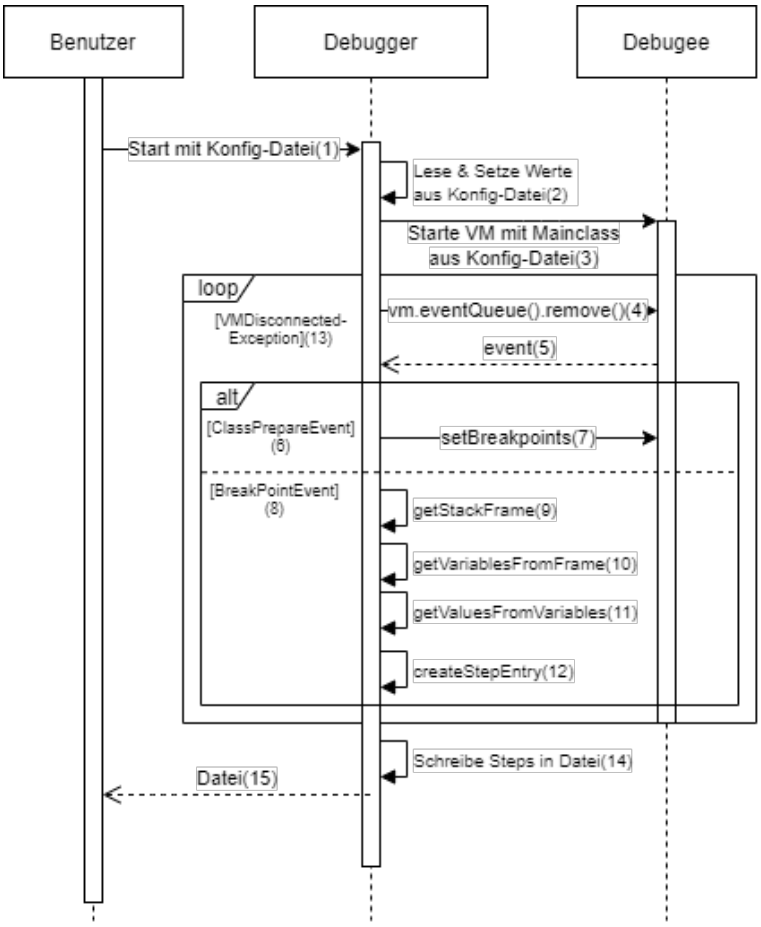

Abbildung 10: UML-Sequenzdiagramm: Ablauf(grob) der Debuggerkomponente

Angestoßen wird der Vorgang durch das Laden einer Konfigurationsdatei durch den Benutzer (1). Der Debugger liest aus dieser seine Konfigurationswerte (2) aus und startet die hinterlegte `Mainclass` in einer von seiner eigenen unabhängigen VM Debuggee (3). Sollte in der Konfigurationsdatei die Variable `testCases` auf `true` gesetzt sein, so wird die eigentliche `mainClass` durch den 'JUnit-platform-console-standalone'-Runner ersetzt, welcher selbst die angege-

bene `mainClass` ausführt.
Daraufhin wartet der Debugger (4) auf Informationen des Debuggee oder auf eine `VMDisconnectedException` (12) welche dann geworfen wird, wenn der Debuggee seine Arbeit einstellt und zwar unabhängig davon, ob dies durch einen eigenen, internen Fehler oder durch erfolgreiche Vollendung der Arbeit auftritt. Währenddessen berichtet der Debuggee dem Debugger von gewissen Events, die innerhalb seiner VM auftreten (5).
Innerhalb der Debuggee-VM treten eine Vielzahl von Events auf, wovon für den Ablauf des Debuggers jedoch nur die Events `ClassPrepareEvent` (6), `BreakpointEvent` (8) und `WatchpointEvent` von Bedeutung sind. Letzteres wurde Aufgrund der Ähnlichkeit zum `BreakpointEvent` und zur Verbesserung der Übersichtlichkeit in diesem Diagramm weggelassen. Der Unterschied zu einem Breakpoint besteht darin, dass ein Watchpoint erst nach dem Initialisieren des zu beobachteten Feldes erstellt werden kann. Somit wird ein Watchpoint nicht als Reaktion auf ein `ClassPrepareEvent` sondern auf ein `BreakpointEvent` eingerichtet, dessen Zeilennummer nach dem Initialisieren des zu beobachtenden Feldes liegen muss. Für die im Rahmen dieser Arbeit entstandene prototypische Anwendung wird die Einrichtung der Watchpoints als Reaktion auf einen Breakpoint mit dem Namen `wpSetup` ausgeführt.
Das `ClassPrepareEvent` (6) informiert den Debugger über die Vorbereitung der `MainClass`, sodass dieser die gewünschten Breakpoints setzen kann, welche im Folgenden die `BreakPointEvents` (8) auslösen. Ist ein Breakpoint erreicht, soll ein Schritt für die Visualisierungsdatei angelegt werden. Hierfür wird zunächst der zugehörige Stackframe vom Stack des Threads genommen, in welchem das `BreakpointEvent` auftritt (9). Aus diesem Frame werden daraufhin die (sichtbaren) Variablen extrahiert (10), aus denen anschließend die laut Konfiguration gewünschten Werte (11) gelesen werden. Diese Werte werden abschließend als Schritt-Eintrag für die Ausgabe formatiert (12).
Der interne Ablauf der Schritte 9-12 unterscheidet sich je nach der genutzten Datenstruktur innerhalb der Debuggee-Anwendung und der gewünschten Visualisierungsart. Unabhängig davon werden die vom Debugger gesammelten Schritte nach dem Abschluss seiner Arbeit (13) in die Visualisierungsdatei geschrieben (14), welche dem Nutzer daraufhin zurückgegeben wird (15).

Anschließend kann die erstellte Visualisierungsdatei in die Visualisiererkomponente geladen werden.
Die Vorgehensweise beim Erstellen eines Schrittes durch ein `WatchpointEvent` unterscheidet sich nicht von der Vorgehensweise bei einem `BreakpointEvent`. Auch hier werden die Schritte (9-12) ausgeführt.

4.3.2 Visualisiererkomponente

Der interne Ablauf der Visualisierer-Komponente ist in Abbildung 11 zu sehen. Im Gegensatz zum Ablauf des Debuggers bietet sich für die Visualisierer-Komponente ein Aktivitätsdiagramm eher an als ein Sequenzdiagramm.

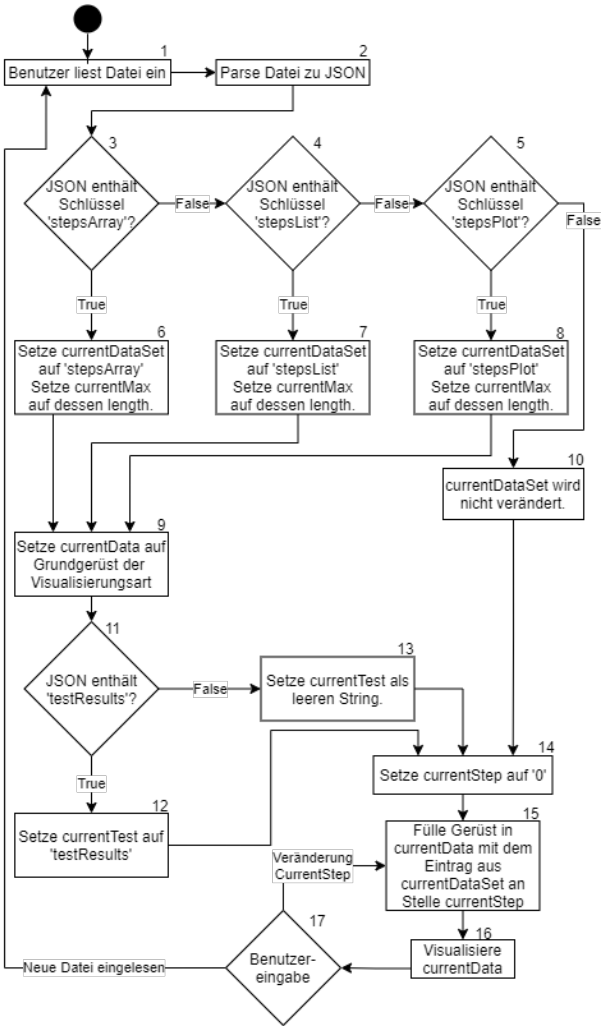

Abbildung 11: Programmablaufplan: Ablauf des Visualisierers

Um den Ablauf zu starten, muss vom Benutzer eine Datei eingelesen werden (1). Der Visualisierer wird die JSON Struktur aus der Datei auslesen (2) und anhand der vorhandenen Schlüssel innerhalb dieser Datenstruktur ermitteln, welche Visualisierungsart vorliegt (3,4,5). Die anhand der vorhandenen Schlüssel ermittelten Daten werden als `currentDataSet`(6,7,8) gespeichert. Dessen Länge wird als `currentMax` festgelegt(6,7,8), sodass eine Obergrenze für den später relevanten `currentStep` existiert. Für die festgestellte Visualisierungsart wird daraufhin ein Grundgerüst in die zu visualisierende Variable `currentData` geladen (9). Sollte es sich zwar um eine valide JSON Datei handeln, jedoch keiner der bekannten Schlüssel vorliegen, so werden keine neuen Daten geladen (10). Befinden

sich in der eingelesenen Struktur neben den Schrittdaten zusätzlich Testdaten `testResults` (11), so werden diese auf die Variable `currentTest` geschrieben (12), andernfalls wird diese Variable auf einen leeren String gesetzt (13). Die Variable `currentStep` wird nun auf 0 gesetzt, was dem ersten Schritt in der geladenen Struktur entspricht (14). Zuletzt wird das in (9) angelegte Gerüst in `currentData` mit den Daten an der Stelle `currentStep` im `currentDataSet` befüllt (15). Das Einlesen und Verarbeiten der Datei ist damit abgeschlossen.
Nun geht die Visualisierer-Komponente in einen sich immer wiederholenden Zustand über. Sie visualisiert die sich derzeitig in `currentData` befindlichen Daten (16) und wartet auf etwaige Veränderungen (17). Über ein Benutzerinterface stehen einem Benutzer nun die folgenden Optionen zur Verfügung:

- Wie zu Beginn kann der Nutzer eine neue Datei einlesen.
- Solange derzeitig nicht der erste Schritt ausgewählt ist, kann immer zum vorherigen Schritt gewechselt werden.
- Solange derzeitig nicht der letzte Schritt ausgewählt ist, kann immer zum nächsten Schritt gewechselt werden.
- Es kann jederzeit zum ersten Schritt gesprungen werden.

Wird der Wert `currentStep` durch eine Benutzereingabe verändert (17), aktualisiert der Visualisierer den Inhalt der `currentData` Variable (15), welche daraufhin neu gerendert wird (16). Sollte eine neue Datei eingelesen werden (17), so beginnt der Ablauf erneut bei (1). Wird ein Schritt ausgewählt, so wird der Schlüsselwert innerhalb des `currentDataSet` aus der eingelesen Datei als Name des Schrittes geladen. Da vom Debugger eine `LinkedHashMap` verwendet wird, ist gewährleistet, dass die Reihenfolge der Schritte immer gleich der zeitlichen Reihenfolge ihrer Erstellung ist.

Für die konkrete Darstellung der einzelnen Daten wird von der Visualisierer-Komponente, wie in der Architektur beschrieben, eine Kombination aus den unter der MIT-Lizenz zur Verfügung stehenden Biblitoheken `Visualizer-Core` und `Visualizer-Bundle`[13] verwendet. Die `Visualizer-Core`-Bibliothek stellt ein Grundgerüst an Infrastruktur zur Verfügung, um Visualisierungen anbieten zu können. Das `Visualizer-Bundle` baut auf der ersten Bibliothek auf und stellt konkrete Visualisierungen, wie die Grid-Visualisierung, zur Verfügung. Auf dem angebotenen Grundgerüst können aber auch eigene Visualisierungen aufgebaut werden, welche in der bereitgestellten React-Komponente registriert und gerendert werden können.

[13] https://github.com/hediet/visualization

Die Visualisierer Komponente des Prototypen nutzt die folgenden, angebotenen Visualisierungen:

- Grid: Gitterdarstellung für ein oder zwei dimensionale Arrays.
- Graph: Graphendarstellung für verkettete Listen.
- Plotly: Diagrammdarstellungen wie Balken- oder Kurvendiagramme.

Die Graph-Visualisierung unterstützt sowohl die Formate 'VisJS' als auch 'Graphviz', wobei letzteres keine Möglichkeit zum Einfärben einzelner Knoten bietet. Die Grid- und Graph-Visualisierungen stellen für ihre Einträge theoretisch eine Möglichkeit bereit, diese einzufärben. Leider funktionierte dies in den verfügbaren Versionen nicht, wodurch die 'Visualizer-Bundle' Bibliothek manuell erweitert werden musste, um die Farbeigenschaft funktionsfähig zu machen. Eine solche Funktionalität steht für die Graphenvisualisierung nicht zur Verfügung.

4.4 Einschränkungen

Aufgrund der festen Anforderungen an die Technologien der Anwendung und um die erfolgreiche Umsetzung gewährleisten zu können, liegen der Anwendung die folgenden Einschränkungen zugrunde.

4.4.1 Java-Einschränkung

Die Visualisierer-Komponente liest eine programmiersprachenunabhängige, JSON-formatierte Visualisierungsdatei ein. Da die Debugger-Komponente auf dem JDI aufbaut, limitiert dies jedoch die automatische Erzeugung von Visualisierungsdateien auf Debuggee-Anwendungen, die auf einer Java-VM basieren. Weiterhin steht das JDI in der hier genutzten Variante nur bis zur Version Java 8 zur Verfügung, für neue Varianten von Java müssten dementsprechend gewisse Anpassungen in der Debugger-Komponente vorgenommen werden.

Im Rahmen des JDI benötigen Watchpoints ein Feld, welches diese beobachten können. Daten primitiver Art, wie beispielsweise ein einfacher `boolean` oder `int`-Wert, bestehen jedoch nur aus einem `PrimitiveValue`, zu dem kein Feld vorliegt. Daher kann ein Watchpoint nicht auf diese einfachen Werte gesetzt werden. Es müssen also die Klassenobjekte wie `Boolean` oder `Integer` genutzt werden oder es müssen vollständige, höhere Objekte beobachtet werden. Diese Einschränkung ist möglicherweise einfach aufzuheben, jedoch ließ sich aufgrund der dürftigen Dokumentation der Watchpoints im JDI keine Lösung finden.

4.4.2 Datenstruktur-Einschränkung

Die Wahl der Datenstruktur innerhalb der Debuggee-Anwendung ist in der prototypischen Anwendung eingeschränkt. Da das JDI bei Referenzen höherer

Objekte zwischen `ObjectReference`, `ArrayReference` und `StringReference` unterscheidet, müssen diese Referenzen unterschiedlich bearbeitet werden. Die Unterscheidung zwischen diesen drei Referenzen kann durch simple Vergleiche erfolgen, wodurch es in der prototypischen Debugger-Anwendung möglich ist, Arrays, Strings und neben primitiven Werten auch höhere Objekte zu untersuchen.
Allerdings unterliegen die höheren Objekte einigen Einschränkungen. So können höhere Objekte, die selbst nur primitive Werte, Arrays oder Strings enthalten, verarbeitet werden, eine iterative Abarbeitung von verschachtelten höheren Objekten wird jedoch nicht unterstützt. In einem solchen Fall muss sich auf ein gewünschtes Objekt festgelegt werden. Um das nachfolgende Beispiel besser nachvollziehen zu können, sei die Klassenstruktur in Abbildung 12 aufgezeichnet. Liegt also ein beispielhaftes `Rennen`(1) Objekt vor, welches eine Sammlung von `Rennbahn`(2) Objekten verwaltet, welche neben einer Länge und einem Namen auch eine Sammlung von `Rennauto`(3) Objekten beinhaltet, kann nicht jede einzelne 'Rennbahn' gleichzeitig visualisiert werden.

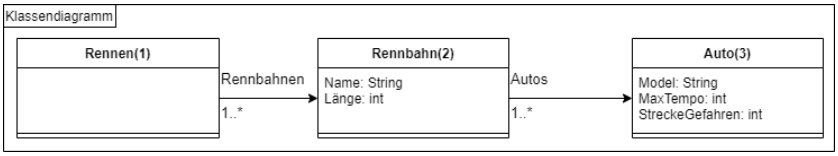

Abbildung 12: UML-Klassendiagramm: Autorennen

Die prototypische Debugger-Anwendung kann in diesem Fall nicht eine komplette Rennbahn aus Rennen visualisieren, aber den Inhalt jedes Autos von einer spezifizierten Rennbahn oder aber auch eine speziell gewünschte Eigenschaft eines oder aller Autos, wie `StreckeGefahren` darstellen. Beispielhafte `variablePath` für diese Fälle wären:

- Alle Autos: rennen.Rennbahnen[0].Autos[*]
- Ein Auto: rennen.Rennbahnen[0].Autos[0]
- Alle `StreckeGefahren`: rennen.Rennbahnen[0].Autos[*].StreckeGefahren
- Ein `StreckeGefahren`: rennen.Rennbahnen[0].Autos[0].StreckeGefahren

Ein '*' steht hierbei für 'alle'. Eine Zahl steht für den Index des gewünschten Objektes. Sollte weder ein '*' noch ein Index angegeben werden, geht der Debugger von 'alle' aus. Daraus folgt, dass 'Alle Autos' auch durch 'rennen.Rennbahnen[0].Autos' ausgegeben werden kann.

4.4.3 Listen- & Map-Einschränkung

Leider bietet das JDI parallel zur `ArrayReference` keine Referenz für Listen oder Maps. Daher müssen für die einzelnen Ausprägungen von
Listen (`ArrayList`, `LinkedList`, `Vector`, ...) und Maps (`HashMap`, `TreeMap`, ...) separate Muster zum Anfertigen von `DebugSteps` erzeugt werden. Daraus resultiert eine weitere Einschränkung des Prototypen: Es können exemplarisch nur Listen der Ausprägung `ArrayList` verarbeitet werden, Maps jeglicher Art werden in der derzeitigen Anwendung gar nicht unterstützt. Diese Einschränkung musste gesetzt werden, damit der Umfang des Prototypen nicht unkontrolliert anwächst.

5 Realisierung

5.1 Datenbeschaffung

Als Grundlage für die Beschaffung der Daten innerhalb der Debugger-Komponente diente 'An Intro to the Java Debug Interface (JDI)' von Baeldung[14]. Die benötigten Informationen wie die `mainClass` werden aus einer wie im Kapitel 'Konfigurationsdatei' beschriebenen Konfigurationsdatei geladen und wie in Abbildung 10 gesetzt.
Nachdem die Debuggee-VM gestartet wurde, werden als Reaktion auf das `ClassPrepareEvent` der `mainClass` die angegebenen Breakpoints gesetzt.
Wird ein Breakpoint erreicht und dadurch ein `BreakpointEvent` ausgelöst, so beginnt der Prozess der Datensammlung. Zuerst wird der Breakpoint deaktiviert und anhand der `isLoop` Eigenschaft geprüft, ob es sich um einen Breakpoint innerhalb einer Schleife handelt. Sollte dies zutreffen, wird der Breakpoint am Ende dieses Prozesses wieder aktiviert und ein Zähler wird inkrementiert. Dessen Stand wird bei der Erstellung eines Visualisierungsschrittes an den Namen des Breakpoints angehängt, um so die Schritte mit gleichem Ursprungsbreakpoint voneinander unterscheiden zu können.
Um nun die eigentlichen Daten zu beschaffen, wird der neuste Frame vom Stack genommen. Die einzelnen Elemente der `variablePath` Eigenschaft werden durch ein Aufteilen des Strings an jedem '.' ermittelt. Dadurch können die sichtbaren Variablen des abgehobenen Frames, welcher in Form einer Map bestehend aus dem Namen der jeweiligen Variable als Schlüssel und deren Wert als Wert vorliegt, nach dem ersten Eintrag durchsucht werden. Sollte dieser einen zusätzlichen Index besitzen (Beispiel: [1]) wird dieser vorerst ignoriert, da das gesuchte Feld nur Anhand des Namens identifiziert werden kann. Erst nachdem das gewünschte Feld gefunden wurde, wird auf einen gewünschten Index geprüft. Ist ein Index vorhanden, muss nun zum ersten Mal zwischen Arrays und Listen unterschieden werden. Wenn es sich bei dem gefundenen Wert der gesuchten Variable um eine `ArrayReference` handelt, kann aus dieser anhand des Index direkt der gewünschte Eintrag extrahiert werden.

Handelt es sich andernfalls um eine Liste, muss aus dieser zunächst das `elementData`-Feld extrahiert werden. Dieser Vorgang ist in Algorithmus 2 zu erkennen.

Algorithm 2 Extrahieren von Listenelement (Javacode)

```
1: ReferenceType refType = listObjRef.referenceType();
2: Field field = refType.fieldByName(ëlementData");
3: ArrayReference arrRef = (ArrayReference)listObjRef.getValue(field);
4: return arrRef.getValue(index);
```

[14] https://www.baeldung.com/java-debug-interface

Hierfür muss die Liste umständlich als `ObjectReference` gehandhabt werden. Von dieser kann der `ReferenceType` (1) genommen werden, welcher Zugriff auf das Feld `elementData` bietet (2). Bei diesem Feld handelt es sich um ein Array (3), über dessen Referenz sich anschließend das eigentlich gesuchte Element entnehmen lässt (4).

Im Vergleich dazu wird zum Extrahieren eines bestimmten Elements einer `ArrayReference` nur die vierte Zeile benötigt.

Anhand des bereits aufgeteilten `variablePath` wird folgendes Verfahren (Algorithmus 3) nun solange iteriert, bis der letzte Eintrag erreicht wurde.

Algorithm 3 VariablePath Iteration (Pseudocode)

```
1: FOR i = 1; i < amountOfVariables; i++
2:     IF value instanceof ArrayReference
3:         ((ArrayReference)value).getValue(variablePath[i])
4:     ELSE IF value.type.name.equals('ArrayList')
5:         getEntryFromList(siehe Algorithmus 2)
6:     ELSE IF value instanceof ObjectReference
7:         getValueFromObjRef(siehe Algorithmus 4)
8:     END IF
9: END FOR
```

Das Verfahren ähnelt dem zur Ermittlung des ersten Schrittes stark. Wenn es sich beim derzeitigen `value` um eine `ArrayReference` (3) handelt, kann der nächste Schritt des `variablePath` einfach extrahiert werden. Ähnliches gilt für eine Liste (5), auch hier muss wie im ersten Schritt das `elementData` Feld extrahiert und durchsucht werden.
Für eine `ObjectReference` (7) muss abweichend davon, wie in Algorithmus 4 beschrieben, der `ReferenceType` bestimmt werden (1). Die Werte `objRef` und `fieldName` sind hierbei Eingabeparameter. Anschließend wird von diesem `ReferenceType` das gesuchte Feld anhand des angegebenen `variablePath` bestimmt (2). Mit Hilfe dieses Feldes kann abschließend aus der ursprünglichen `ObjectReference` der gesuchte Wert extrahiert werden (3).

Algorithm 4 Value aus ObjectReference extrahieren (Javacode)

```
1: ReferenceType refType = objRef.referenceType();
2: Field field = refType.fieldByName(fieldName);
3: return objRef.getValue(field);
```

Im darauf folgenden letzten Schritt werden aus dem zuletzt extrahierten `value`, egal welcher Art, die einzelnen zu visualisierenden Werte entnommen. Handelt es sich beim letzten Wert bereits um ein `PrimitiveValue`, kann dessen

Wert einfach weiterverarbeitet werden. Um diesen einzelnen Wert im nächsten Schritt weiterverwenden zu können, wird er dennoch in eine Liste umgewandelt, um so einen einheitlichen Datentypen gewährleisten zu können. Aus einer `ArrayReference` können die einzelnen Werte genauso einfach wie zuvor extrahiert werden.
Auch für eine Liste unterscheidet sich der Vorgang nur dahingehend, dass alle internen Werte weiterverwendet werden, anstatt nur ein einzelner, interner Wert. Wenn eine vollständige `ObjectReference` visualisiert werden soll, wird diese ähnlich wie ein `PrimitiveValue` einfach weiterverarbeitet und im folgenden Schritt in eine repräsentierbare Struktur umgewandelt.
Nun kann schlussendlich der eigentliche `DebugStep` erzeugt werden. Hierzu werden die einzelnen Einträge der im letzten Schritt entstandenen Liste `Values` durchlaufen und für die Ausgabe formatiert. Die Formatierung der Daten unterscheidet sich je nach Visualisierungsart, auf diese Unterschiede soll hier aber noch nicht weiter eingegangen werden. Interessanter ist hier der finale Unterschied zwischen den verschieden Arten von Daten und deren Handhabung.
Für einen `ArrayReference`-Eintrag in dieser Liste wird über die einzelnen Elemente des Arrays iteriert, jedes Element wird in einen String umgewandelt, welcher eine Zelle innerhalb der gesamten `ArrayReference` darstellt.
`StringReference` und `PrimitiveValues` können hingegen sehr einfach in einen String konvertiert werden und sind damit bereit für die Ausgabe. Bei einer Liste wird erneut ein ähnliches Verfahren wie bei den Arrays angewendet: Über die einzelnen Elemente wird iteriert und deren Inhalt in einen String umgewandelt, welcher später einen Knoten darstellen wird. Von jedem Knoten aus wird eine Verbindung zu seinem Vorgänger erzeugt, wenn dieser vorhanden ist.
Aufgrund der fehlenden Felderfunktionalität (wie beschrieben in 'Einschränkungen)' für `PrimitiveValues` muss eine weitere Unterscheidung für `ObjectReferences` vorgenommen werden. Wenn eine solche Referenz vollständig ausgegeben werden soll, kann deren Inhalt vollständig in einen Ausgabestring umgewandelt werden. Wenn aber der letzte Eintrag des `variablePath` auf ein `PrimitiveValue` verwiesen hat, muss dessen Wert speziell aus der `ObjectReference` gelesen und verarbeitet werden. Dies beruht auf der in Einschränkungen beschriebenen Eigenschaft, dass primitive Datentypen nicht als Felder vorliegen.
Wenn bereits mindestens ein Schritt erzeugt wurde, wird in allen weiteren Fällen der zuletzt erzeugte `debugStep` mitgegeben, um durch Vergleiche der alten und neuen Werte festzustellen, ob diese sich verändert haben. Ist dies der Fall, so wird dies an den einzelne Zellen einer `ArrayReference` oder den einzelnen Knoten innerhalb einer Liste durch eine rote Färbung kenntlich gemacht.

5.2 Datenverarbeitung

Bevor die im vorherigen Schritt erhaltenen Daten visualisiert werden können, müssen diese aufbereitet werden. Die Verarbeitung erfolgt direkt nach der Beschaffung der Daten, sodass diese in aufbereiteter Form in eine Visualisierungsdatei geschrieben werden können. Unabhängig von der Form der Daten wer-

den die einzelnen Schritte als eine Schlüssel-Wert Struktur abgespeichert. Als Schlüssel fungiert jeweils der Name eines Break- oder Watchpoints, der Wert beinhaltet die zu visualisierenden Daten und unterscheidet sich je nach Visualisierungsart. Eine Arraystruktur wird immer als Sammlung einzelner Spalten (`columns`) gespeichert. Innerhalb einer solchen Spalte befinden sich einzelne Zellen, bestehend aus dem eigentlichen Inhalt (`content`), sowie einem Attribut `color`, welches die Farbe festlegt. Über diese Farbe wird eine Veränderung im Vergleich zum vorherigen Schritt verdeutlicht. Beide Felder werden als Strings gespeichert.
Hinter dem Schlüssel einer Listenstruktur befinden sich zwei Sammlungen:
Zuerst die einzelnen Knoten `nodes`, jeweils bestehend aus `id`, `label` und `color`. Das `label` ist das Gegenstück zum `content` und beinhaltet die Daten. Die `id` wird verwendet um Start- und Endpunkte der Kanten festzulegen. Genau wie für die Arraystruktur dient `color` auch hier zur Definition der Farbe. Bei der zweiten Sammlung handelt es sich um `edges`, den Verbindungen zwischen den `nodes`. Diese bestehen jeweils aus der `id` des Ursprungsknotens und der `id` des Zielknotens, respektive `from` und `to`, sowie dem Attribut `color`.
Der Inhalt einer Graphenstruktur unterscheidet sich je nach gewünschtem Graph, welcher mit dem Attribut `type` festgelegt wird. Das in diesem Prototypen beispielhaft verwendete Balkendiagramm wird mit dem `type` 'bar' erzeugt. Für ein solches Diagramm wird zusätzlich eine Sammlung `y` benötigt, welche die Höhe der einzelnen Balken enthält. Dementsprechend erzeugt `y = [1,2,3]` drei Balken, wobei der dritte dreimal so hoch ist wie der erste.
Wurde in der eingelesenen Konfigurationsdatei angegeben, dass Tests ausgeführt werden sollen, so wird zusätzlich ein Element `testResults` in die Visualisierungsdatei geschrieben. Dieses enthält die Anzahl an fehlgeschlagenen Tests, die Anzahl an erfolgreichen Tests und - wenn vorhanden - die Namen der fehlgeschlagenen Tests.

Die Visualisierungs-Komponente ist als Erweiterung für VSCode[15] auf ein Aktivierungsevent angewiesen. Diese Events werden innerhalb des 'Extension-Manifests' festgelegt. Für die prototypische Anwendung ist dies auf einen auszuführenden Befehl innerhalb der VSCode-Anwendung begrenzt, kann im Nachhinein aber auch auf Änderungen am Dateisystem, wie einen Dateiupload, die Vollendung des Startvorgangs oder das Laden eines neuen Fensters innerhalb der Anwendung erweitert werden. Als Inhalt der Erweiterung wird eine React-Webanwendung verwendet, die vollständig in eine VSCode-Webview[16] geladen werden kann.
Innerhalb dieser Anwendung können durch die Nutzung der 'Dropzone'-Bibliothek[17] Dateien hochgeladen und eingelesen werden. Eine so geladene Datei wird in ein `Javascript Object` umgewandelt, sodass deren Inhalt, wie im Kapitel Funktionsablauf beschrieben, geladen und angezeigt werden kann. React bietet sogenannte `Effect Hooks`, mit dieser noch recht neuen Entwicklung

[15] https://code.visualstudio.com/api
[16] https://code.visualstudio.com/api/extension-guides/webview
[17] https://react-dropzone.js.org/

lassen sich einzelne Variablen innerhalb einer Anwendung beobachten. Bei einer Veränderung dieser Variablen wird ein `Effekt` ausgeführt, was einer Funktion entspricht. Solch ein `Hook` ähnelt daher stark einem `Modification Watchpoint` aus der Debugger-Komponente.
Unter Verwendung dieser `Hooks` werden sämtliche Funktionalitäten dieser Komponente umgesetzt. Auf das Laden einer Datei hin wird deren Inhalt extrahiert und als `Dataset` geladen. Durch die Veränderung des `Dataset` wird der derzeitige Schritt auf den initialen gesetzt, wodurch wiederum die Aktualisierung des aktuell angezeigten Datenschrittes angestoßen wird. Eine eventuelle Veränderung der Testinformationen wird als Teil der `Dataset` Aktualisierung durchgeführt.

Wie im Kapitel Visualisiererkomponente beschrieben, werden die darzustellenden Daten dem Benutzer über eine Benutzeroberfläche präsentiert. Diese ist innerhalb dieses Prototypen minimal gehalten und in Abbildung 13 zu sehen. An oberster Stelle (1) sind die Schaltflächen zur Veränderung des aktuellen Schrittes eingebaut. Direkt darunter (2) wird der Name des derzeitigen Schrittes angegeben, in diesem funktionslosen Beispiel einfach nur 'zwei'. Die eigentliche Visualisierung befindet sich im Zentrum der Abbildung, diese ist unterteilt in die Index-Überschriften (3), sowie die visualisierten Daten (4). In diesem Beispiel wird ein zweidimensionales Array, bestehend aus einer Spalten und vier Reihen visualisiert. Zwischen dem hier nicht zu sehenden Schritt 'eins' und dem gerenderten Schritt 'zwei' wurde der Wert des untersten Feldes (5) verändert, daher ist diese Zelle rot gefärbt. Da sich die anderen Werte nicht verändert haben, sind alle anderen Zellen grün eingefärbt. Am unteren Rand der Abbildung befindet sich der 'Teststreifen', dieser beinhaltet die Ergebnisse der ausgeführten JUnit-Tests. In den hier ausgeführten Beispieltests schlägt der Test 'provokeFailure' beabsichtigt fehl, sodass sowohl die Anzahl der fehlgeschlagenen, die Anzahl der erfolgreichen sowie der Name des fehlgeschlagenen Tests angezeigt werden.

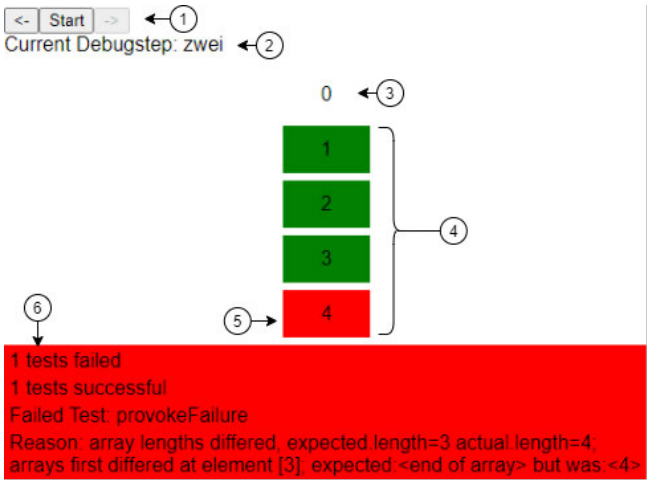

Abbildung 13: Benutzeroberfläche Visualisierer

6 Validierung

6.1 Beispiele

Um die Umsetzung der einzelnen Visualisierungsarten zu testen, wurden verschiedene Beispiele eingesetzt, welche nun zusätzlich zur Erläuterung der verschiedenen Arten genutzt werden sollen.

6.1.1 Autorennen

Als erstes Beispiel für die zu Beginn implementierte Visualisierungsart der Arrays wurde das bereits angesprochene Autorennen genutzt, welches aus einer Praktikumsaufgabe der Lehrveranstaltung `Programmieren 1` an der HAW Hamburg stammt. Hierbei handelt es sich um eine Rennstrecke fester Länge, auf der eine beliebige Anzahl Rennautos platziert werden. Diese Rennautos besitzen zur Identifizierung jeweils einen Namen und einen Typen als String, sowie je einen Integer für ihre maximale Geschwindigkeit und die bereits zurückgelegte Distanz (siehe Algorithmus 5).

Algorithm 5 Beispiel Rennauto (Auszug Javacode)

```
1: public Rennauto(String name, String type, int accMax){
2:     this.name = name;
3:     this.type = type;
4:     this.accMax = accMax;
5:     this.distanceTraveled = 0;
6: }
```

Mit der Methode `simuliereZeitabschnitt` der Rennstrecke werden alle Rennautos auf einer Strecke fahren gelassen. Um dies zu simulieren, wird eine zufällige, ganze Zahl kleiner oder gleich der maximalen Geschwindigkeit zur zurückgelegten Strecke addiert. Zur Darstellung über die Array-Visualisierungsart werden die einzelnen Rennautos als Array auf der Rennstrecke realisiert. Aus diesem Array können dann entweder die gesamten Informationen (Abbildung 14) oder nur eine spezifische Eigenschaft, wie die zurückgelegte Distanz (Abbildung 15) von einem oder von sämtlichen Autos, extrahiert, in eine Visualisierungsdatei geschrieben und dargestellt werden.

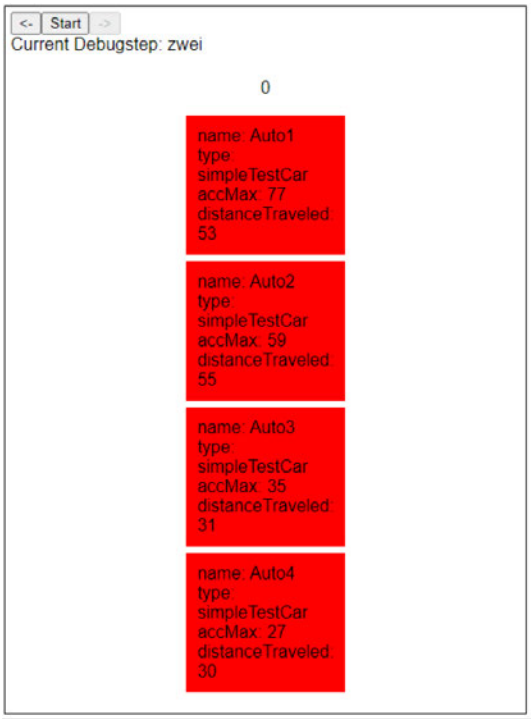

Abbildung 14: Array-Visualisierung Autos

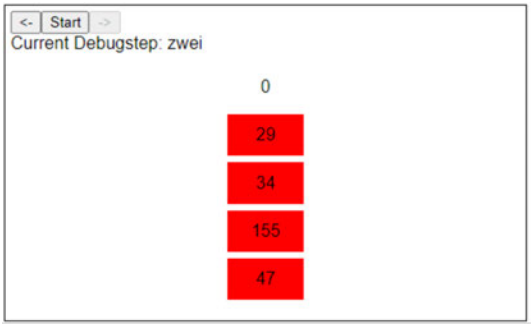

Abbildung 15: Array-Visualisierung Distanz

In den in dieser Beispielabbildung gezeigten Schritten haben sich die Werte aller Autos im Vergleich zum vorherigen Schritt verändert, was durch die rote Hintergrundfarbe verdeutlicht wird. Bei den zwei beginnenden Darstellungen (Abbildung 14 und 15) handelt es sich um Visualisierungen eindimensionaler Arrays. Die dritte Variante in Abbildung 16 zeigt ebenfalls Daten eines Autorennens, jedoch handelt es sich hierbei um eine alternative Realisierung. Statt

eines Arrays wurde hier eine Liste verwendet, um die einzelnen Autos zu gruppieren.

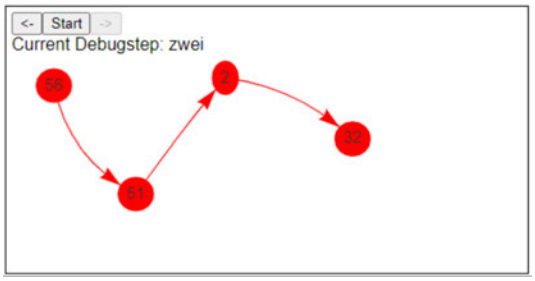

Abbildung 16: Listen-Visualisierung Distanz

Diese Formen der Visualisierung tragen für dieses Beispiel leider nur minimal zur Verbesserung des Programmverständnisses bei. Wünschenswert wäre eine umfangreichere visuelle Aufbereitung, welche die einzelnen Zellen der Arraydarstellung (Abbildung 14) in ihrem Verhältnis zur Länge der Rennstrecke darstellt. So könnten die Zellen je nach zurückgelegter Strecke weiter rechts in der Visualisierung positioniert werden und möglicherweise sogar durch die Darstellung der Länge der Rennstrecke in Form einer Ziellinie unterstützt werden.
Die Umsetzung einer solchen Visualisierung ist innerhalb der genutzten Visualisierungsbibliothek jedoch nur sehr schwer umzusetzen.

Der gewünschte Effekt der Verbesserung des Programmverständnisses wird durch die nachfolgenden Beispiele (Kapitel 6.1.2 und 6.1.3) besser erreicht. Dennoch bietet das Autorennen als initiales Darstellungsbeispiel der Array- und Listenvisualisierung sowie zur Erklärung dieser Konzepte eine gute Grundlage.
Zusätzlich zeigt die Darstellung dieses Beispiels ein weiteres Problem auf. Im Rahmen des `Vis-Bundles` werden für die einzelnen Spalten einer Arrayvisualisierung Spaltennummern projeziert. Dies lässt sich weder deaktivieren noch manipulieren, um so zum Beispiel eine andere Hintergrundfarbe festzulegen, welche eine bessere Abgrenzung vom Rest der Anwendung ermöglichen würde. Außerdem kann diese Funktionalität nicht für die Zeilen hinzugefügt werden, welches jedoch für das Verständnis von multidimensionalen Arrays von Vorteil wäre. Im Gegensatz zur nachgerüsteten Farbfunktionalität lässt sich dies leider auch nicht ohne weiteres einbauen, weswegen im Umfang dieser Arbeit auf eine Anpassung verzichtet wurde.

6.1.2 Bildbearbeitung

Als Beispiel für eine Visualisierung mehrdimensionaler Arrays dient eine einfache Bildbearbeitungsanwendung, welche wie schon das Autorennen aus einer Praktikumsaufgabe der Lehrveranstaltung `Programmieren 1` stammt. In diese Bildbearbeitung wird vom Benutzer ein beliebiges Bild geladen, welches dar-

aufhin in eine Graustufenabbildung umgewandelt wird. Diese besteht aus einzelnen Bildpunkten, welche die Helligkeit mit Werten von 0 bis 255[18] angeben. Die Bildpunkte des geladenen Bildes werden als zweidimensionales Array verarbeitet, eine Dimension für die Reihen und eine für die Spalten. Da selbst die Darstellung von einstelligen Zahlen mehr als einen Bildpunkt beansprucht, ist die Visualisierung des Debuggers deutlich größer und eignet sich daher nur bedingt für große Bilder. Im folgenden Beispiel wird eine kleine, quadratische Ansammlung an Farbpunkten visualisiert, um die ausgegebenen Diagramme des Visualisierers nicht zu groß werden zu lassen. Das verwendete Beispielbild ist umgewandelt in eine Graustufenvariante in Abbildung 17 zu sehen.

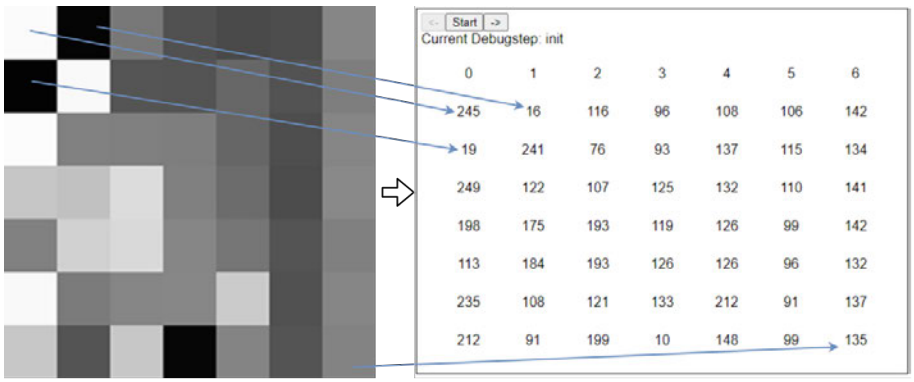

Abbildung 17: Beispielbild der Bildbearbeitung

Normalerweise erfolgt die Umwandlung eines Bildes in seine Graustufenvariante erst im Rahmen der Ausführung der Anwendung. Das Bild wird während der Ausführung der Bildbearbeitung stetig in seiner derzeitigen bildlichen Repräsentation auf einer Leinwand angezeigt, die eigentlichen Werte der Bildpunkte bleiben dem Benutzer dabei verborgen.
Hier kommt nun der Debugger zum Einsatz. Dieser kann, wie in der vorangegangenen Abbildung 17 zu erkennen, die konkreten Zahlenwerte des gesamten Bildes abgreifen und darstellen. Die Zahlenwerte der 7x7 Bildpunkte des eingelesenen Bildes werden als zweidimensionales Array aufbereitet, in der Abbildung werden einige dieser Punkte mit ihren Zahlen verbunden um dies zu verdeutlichen.
Somit ist es möglich, die genauen Veränderungen durch eine Operation nachzuprüfen, Fehler besser zu erkennen und somit ein besseres Programmverständnis zu erlangen.
Die in Abbildung 17 gezeigten Bildpunkte sind im Schritt `init` (1) der Abbildung 18 in ihre Graustufenwerte überführt worden. Im zweiten Schritt `darker` (2) soll durch die Operation 'abdunkeln' der Wert jedes einzelnen Bildpunktes um 25 verringert werden. Dies kann auf einer Leinwand grob eingeschätzt werden, aber nur durch die exakten Werte, welche vom Debugger gesammelt werden, lässt sich dies genau überprüfen.

[18] 0 = schwarz, 255 = weiß

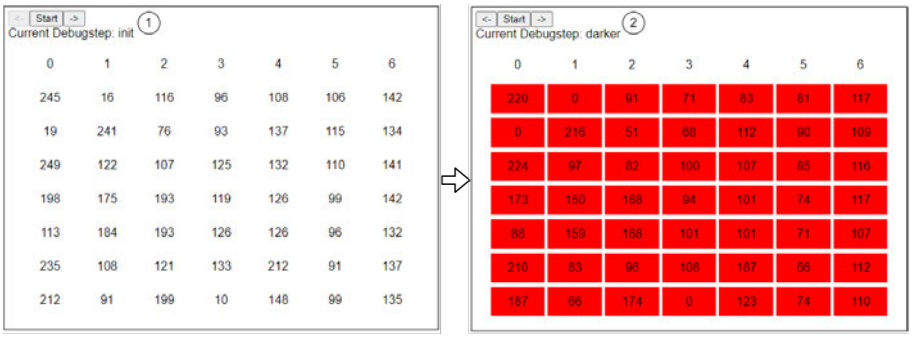

Abbildung 18: Beispiel: Bildbearbeitung abdunkeln

Eine weitere Operation 'weichzeichen' soll für einen gegebenen Bildpunkt den Mittelwert der acht benachbarten Bildpunkte berechnen und als neuen Wert speichern. Die fehlerhafte Implementierung dieser Funktion kann in Abbildung 19 in den beiden Schritten `gauss` (1) und `gauss2` (2)gesehen werden. Initial wurden hierbei die Graustufenwerte aus dem Schritt `init`(1) der Abbildung 18 genutzt. Diese wurden zweifach weichgezeichnet. Trotz eventueller Schreibzugriffe wurden die Werte der in `gauss2` (2) nicht rot hinterlegten Zellen nicht verändert, wodurch diese vom Visualisierer nicht gesondert gekennzeichnet werden.

Abbildung 19: Visualisierung: Array Weichzeichner

Diese Visualisierung zeigt schnell und unmissverständlich, dass die Implementierung des Weichzeichners für jene Bildpunkte nicht korrekt arbeitet, die über mindestens einen Nachbarn verfügen, der einen negativen Index aufweisen würde.

Der Code, welcher verwendet wird, um einen Bildpunkt in Spalte `x` und Reihe `y` weich zu zeichnen, ist im Algorithmus 6 beschrieben, etwaige, geklammerte Zahlen beziehen sich auf die Zeilennummern im Algorithmus.
Der Eingabeparameter `bilddaten` (1) beinhaltet hierbei die Bildpunkte des geladenen Bildes. Die außerhalb dieser Methode definierten Variablen `hoehe` (5), sowie `breite` (7), enthalten die Maße dieses Bildes. In den ersten Zeilen (2-7) werden Variablen initialisiert. Die Variablen `pixel` (2) und `amount` (3) werden verwendet, um den Wert der einzelnen Graustufen der Nachbarn, sowie die Anzahl der beobachteten Felder, zu speichern. Die vier Variablen `rowStart` (4), `rowFinish` (5), `colStart` (6) sowie `colFinish` (7) dienen zur Berechnung der Indizes der zu beachtenden Nachbarn. Erst nach dieser Initialisierung wird die eigentliche Berechnung durchgeführt.
Der Fehler in der hier durchgeführten Berechnung besteht in der Prüfung der Ungleichheit der Werte von 0 und `x` beziehungsweise `y` (10). Eigentlich sollten diese jeweils mit `curRow` oder `curCol` verglichen werden, anstatt mit 0.
Die Werte der einzelnen, benachbarten Felder werden innerhalb der beiden `for`-Schleifen (8,9) aufaddiert.
Abschließend wird der gesammelte Graustufenwert in `pixel` durch die Anzahl der Zellen, auf die zugegriffen wurde (`amount`), geteilt solange mindestens auf eine Zelle erfolgreich zugegriffen wurde. Der Wert, welcher danach von der Variable `pixel` repräsentiert wird, wird als Ergebnis für die weichgezeichnete Zelle zurückgegeben (16).

Algorithm 6 Beispiel Weichzeichnung (Auszug Javacode)

```
 1: private short weichzeichnen(short[][] bilddaten, int x, int y) {
 2:     short pixel = 0;
 3:     short amount = 0;
 4:     int rowStart = Math.max(y - 1, 0);
 5:     int rowFinish = Math.min(y + 1, hoehe - 1);
 6:     int colStart = Math.max(x - 1, 0);
 7:     int colFinish = Math.min(x + 1, breite - 1);
 8:     for (int curRow = rowStart; curRow <= rowFinish; curRow++) {
 9:         for (int curCol = colStart; curCol <= colFinish; curCol++) {
10:             if (y != 0 && x != 0) {
11:                 pixel += bilddaten[curRow][curCol];
12:                 amount += 1;
13:             }
14:         }
15:     }
16:     return (amount > 0 ? pixel /= amount : pixel);
17: }
```

Durch die visuelle Aufbereitung konnte der Fehler einfach erkannt und dessen Quelle schnell ermittelt werden. Die Ergebnisse der Ausführung der korrigierten Variante des Algorithmus sind in Abbildung 20 zu sehen.

Abbildung 20: Visualisierung: Array Weichzeichner

6.1.3 Maxima-Berechnung

Im Kapitel Coding Games wurde das Beispiel der Maxima-Berechnung und dessen aufwendige, visuelle Aufbereitung erwähnt. Dieses Beispiel wurde im Laufe dieser Arbeit innerhalb des Prototypen nachgebaut, jedoch ohne die hübsche Oberfläche und die Gamifizierung. Die Berechnung der jeweiligen Maxima und deren Indizes ist durch eine simple Schleife gelöst, der Wert des ermittelten Balkens wird daraufhin auf 0 gesetzt. Das Verfahren zur Ermittelung des korrekten, nächsten Index ähnelt einer Lösung für das Codingame, dort wird jedoch nur der Index an den Server übermittelt und dieser übernimmt die Zerstörung des Berges. Dargestellt werden die noch vorhanden 'Berge' als Balken in einem Balkendiagramm, wie in Abbildung 21 zu sehen ist.

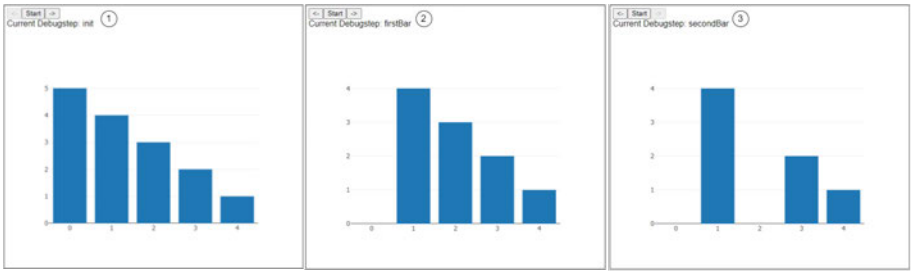

Abbildung 21: Visualisierung: Balkendiagramm

Zunächst wurden die Balken korrekt initialisiert, wie im ersten Schritt zu sehen. Daraufhin wurde korrekt die Höhe des ersten Balkens (Index 0) auf 0 gesetzt, jedoch wurde im Folgenden zweiten Schritt die Höhe des inkorrekten Balkens mit Index 2 auf 0 gesetzt. Durch die Visualisierung der einzelnen Balken kann

dieser Fehler, wie schon im vorherigen Beispiel, einfach erkannt und verstanden werden. Gleichzeitig wurde bei der Ausführung dieses Beispiels mit Tests gearbeitet. Durch das Reduzieren des falschen Balkens schlägt einer der Tests fehl, was dem Benutzer zusätzlich angezeigt wird und weiter zum Verständnis des Programmablaufes und der Fehlerquelle beiträgt. Eine beispielhafte Fehlermeldung ist in der Abbildung 22 zu sehen.

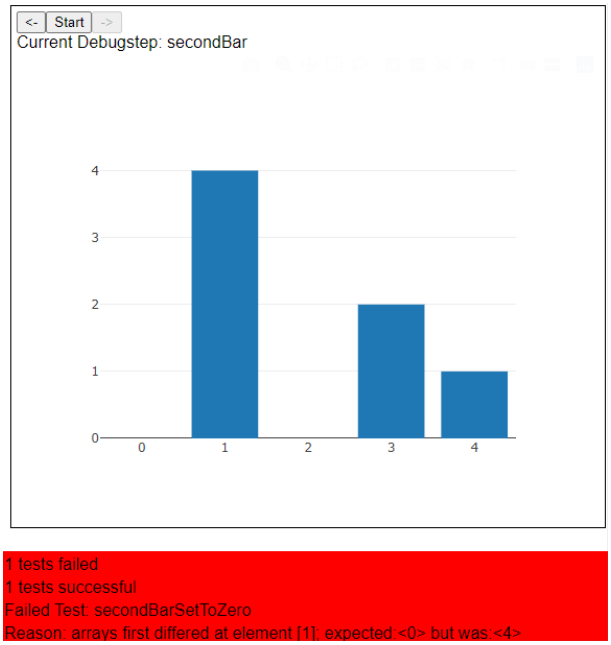

Abbildung 22: Visualisierung: Balkendiagramm mit fehlgeschlagenem Test

6.2 Diskussion

Um das Verständnis von internen Programmabläufen während der Programmierausbildung, aber auch danach, zu verbessern, wurde eine prototypische Anwendung erschaffen, welche die internen Zustände von ausgeführten Programmen beobachten, auslesen und visualisieren kann. Wie durch die vorangegangenen Beispiele gezeigt, wurde das ursprünglich gesetzte Ziel der 'Verbesserung des Programmverständnisses' durch verschiedene 'dynamische Visualisierungen' erreicht. Durch die Verwendung verschiedener Visualisierungsarten können diverse Szenarien abgebildet werden. Für die Ausführung einer neuen Debuggee-Anwendung muss jedoch immer manueller Aufwand getätigt werden, da eine individuelle Konfigurationsdatei notwendig ist, welche die benötigten Informationen und die gewünschten Break- oder Watchpoints enthält.
Die an den Prototypen gestellten Anforderungen, die Aufteilung in zwei Komponenten sowie eine Umsetzung der 'Visualisierer-Komponente' als Erweiterung für VSCode wurden erfüllt. Im Laufe der Ausarbeitung kamen weitere

Wünsche und Ideen hinzu, welche Funktionalitäten zusätzlich angeboten werden können. So können im finalen Prototypen beispielsweise zusätzlich JUnit-Tests ausgeführt und deren Ergebnisse visualisiert werden. Aufgrund der verschiedenen Komponenten kommen unterschiedliche Technologien zum Einsatz, deren Handhabung sich unterscheidet und dadurch die Arbeit verkompliziert hat. Zusätzlich wurde dies durch die mangelhafte Dokumentation des JDI erschwert. Als besonders schwierig stellte sich die korrekte Nutzung von Watchpoints heraus. Diese können im finalen Prototypen zwar gesetzt und genutzt werden, jedoch ist das Extrahieren und Aufbereiten von Daten aus der Debuggee-Anwendung über diese schwieriger als für herkömmliche Breakpoints. Auch für diese gestaltete sich die Datenverarbeitung als herausfordernder als erwartet, da je nach Datentyp individuelle Vorgehensweisen notwendig sind. Dieser Aufwand muss für jede weitere Programmiersprache wiederholt werden, wenn die Debuggerkomponente erweitert werden soll. Positiv hervorzuheben ist hierbei die Unabhängigkeit der Visualisiererkomponente. Durch die Nutzung der Visualisierungsdatei-Schnittstelle muss diese nicht angepasst werden, um weitere Programmiersprachen zu unterstützen.

Die Funktionalitäten dieses Prototypen müssen aufgrund der großen Auswahl an möglichen Datentypen und Programmiersprachen, sowie den unbegrenzten Möglichkeiten der Visualisierung als beispielhaft gesehen werden. Eine vollständige Abdeckung aller möglichen Datenstrukturen einer einzigen Programmiersprache erfordert bereits einen so großen Aufwand, dass dies nicht gewährleisten werden konnte. Gleiches gilt für den Aufwand der Umsetzung einer aufwendigen, gamifizierten Visualisierung, wie beispielsweise die des 'Codingame'. Der vorgestellte Prototyp stellt eine Grundlage für viele Erweiterungen bereit, ein Teil davon soll im Ausblick besprochen werden.

6.3 Ausblick

Zukünftig kann der hier entwickelte Prototyp an vielen Stellen erweitert oder verbessert werden. Zunächst könnten die gesetzten Einschränkungen aufgehoben werden. So könnte die Begrenzung der unterstützten Datentypen aufgehoben werden, indem neben `ArrayLists` auch andere Implementierungen von Listen oder Maps unterstützt werden. Zusätzlich kann die Debugger-Komponente durch eine Unterstützung von anderen Programmiersprachen erweitert werden. Zuletzt kann versucht werden, Objekte vollständig iterativ zu entpacken, so dass in Bezug auf das anfängliche 'Autorennen'-Beispiel vollständige Rennstrecken in der Visualisierung aufgenommen werden könnten.
Die Benutzeroberfläche der Visualisierungskomponente könnte durch zusätzliche Funktionen, wie eine Möglichkeit zu einem bestimmten Schritt zu springen, erweitert werden. Im Allgemeinen bedarf die Oberfläche einer besseren Aufbereitung.
Die einzelnen Reihen einer `grid`-Darstellung verfügen aufgrund der Basisfunktionalität der genutzten Visualisierungslösung über eine angezeigte Indexierung der Spalten. Diese Funktionalität wird nicht für die Reihen unterstützt und kann

auch nicht auf einfachem Wege deaktiviert werden. Daher wäre es wünschenswert dies zukünftig nachzurüsten. Allerdings kann dieser Schritt übersprungen werden, wenn die bisherige Visualisierungslösung komplett ausgetauscht wird.
Eine längerfristige Aufgabe ist die Ablösung der genutzten Visualisierungen. Die vom 'Visualizer-Bundle' angebotenen Funktionen sind sehr hilfreich, aber stark begrenzt und teilweise nicht voll funktionsfähig. Hier würde es sich anbieten, eine eigene Implementation zu schreiben, die es zusätzlich ermöglicht, die Visualisierung individueller zu gestalten, um so an das 'Codingame' heranzukommen und durch mehr Gamifizierung den Lernferfolg weiter zu verbessern.
Alternativ zur genutzten `grid`-Darstellung bietet die Plotly-Bibliothek eine andere Form der `grid`-Darstellung. Diese kann einen beliebigen Wert zur Identifizierung von Spalten und Reihen anzeigen, jedoch keine Veränderungen durch farbliche Unterschiede deutlich machen. Diese Visualisierungsart erinnert stark an eine herkömmliche Tabelle. Eine beispielhafte Darstellung dieser Visualisierung in in Abbildung 23 zu sehen.

	0	Header
0	341319	4488916
Row 2	281489	3918072

Abbildung 23: Visualisierung: Plotly-Grid

Hierbei sind die beiden Nullen, sowie 'Header' und 'Row 2' als Beispiele für die Identifizierung der Spalten und Reihen gesetzt worden.
Die Watchpoint-Funktionalität des JDI ist innerhalb der zur Verfügung stehenden Quellen relativ versteckt gewesen, dadurch kam die Idee zu deren Implementierung erst zu einem späten Zeitpunkt in der Ausarbeitung auf. Somit ist die derzeitige Watchpoint-Unterstützung nur rudimentär und sollte in Zukunft erweitert werden, um genauso funktional zu sein wie die Unterstützung der Breakpoints. Im speziellen gilt es ein Lösung zu finden, um Strings beobachten zu können, da ein `AccessWatchpoint` auf einer Stringvariable bei einem einzigen Zugriff sehr oft ausgelöst wird.
Zu Beginn der Arbeit sollten nicht nur Veränderungen in rot, sondern auch lesende Zugriffe in grün (wie in Abbildung Benutzeroberfläche Visualisierer) visualisiert werden. Diese Idee wurde schnell verworfen, da über die Breakpoints des JDI keine Möglichkeit besteht, über lesende Zugriffe benachrichtigt zu werden. Mit der Unterstützung der Watchpoints könnte diese anfängliche Idee wieder aufgegriffen und eventuell erfolgreich umgesetzt werden. Dies könnte zu einer weiteren Verbesserung des Programmverständnisses führen. So könnten die im Beispiel der Bildbearbeitung während des Weichzeichnens benutzen Felder hervorgehoben werden, um so zu zeigen, ob falsche Feldinformationen genutzt werden.
Abschließend kann auch die Unterstützung von Testfällen erweitert werden. Es könnten für jeden Schritt individuelle Tests ausgeführt und Daten angezeigt werden.

7 Fazit

Zu Beginn der Ausarbeitung stand die Idee durch die Entwicklung einer Debuggeranwendung die Programmabläufe von ausgeführten Anwendungen aufzuzeichnen und zu visualisieren. Die entstandene prototypische Applikation erreicht das gesetzte Ziel. Während der Entwicklung wurden weitere Funktionalitäten erdacht, die jedoch nicht alle realisiert werden konnten. So können Watchpoints grundlegend genutzt werden, jedoch wird keine Unterstützung zur Darstellung von lesenden Zugriffen auf Variablen angeboten. Auch die Idee, die Ausführung von Testfällen zu unterstützen und deren Ergebnisse zu visualisieren, wurde erfolgreich umgesetzt. Insgesamt kann die Arbeit daher als Erfolg bezeichnet werden. Eine Einbindung in das OPPSEE Projekt kann also erfolgen, erfordert aber noch einige Anpassungen an die endgültige Struktur der Plattform und die gewünschten Programmieraufgaben.

8 Literaturverzeichnis

[1] Gabriel Barata u. a. “Improving Participation and Learning with Gamification”. In: *Proceedings of the First International Conference on Gameful Design, Research, and Applications*. Gamification ’13. Toronto, Ontario, Canada: Association for Computing Machinery, 2013, S. 10–17. ISBN: 9781450328159. DOI: 10.1145/2583008.2583010. URL: https://doi.org/10.1145/2583008.2583010.

[2] Wim De Pauw u. a. “Visualizing the Execution of Java Programs”. In: Bd. 2269. Jan. 2001, S. 151–162. ISBN: 978-3-540-43323-1. DOI: 10.1007/3-540-45875-1_12.

[3] David J. Murray und Dale E. Parson. “Automated Debugging in Java Using OCL and JDI”. In: *Proceedings of the Fourth International Workshop on Automated Debugging, AADEBUG 2000, Munich, Germany, August 28-30th, 2000*. Hrsg. von Mireille Ducassé. 2000. URL: https://arxiv.org/abs/cs/0101002.

[4] Rainer Oechsle und Thomas Schmitt. “JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI)”. In: *Software Visualization*. Hrsg. von Stephan Diehl. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, S. 176–190. ISBN: 978-3-540-45875-3.

8.1 Online-Quellen

- Visual Studio Code: https://code.visualstudio.com/
- React.js Framework: https://reactjs.org/
- Dropzone React-API: https://react-dropzone.js.org/
- Visualization Framework: https://github.com/hediet/visualization
- Beispiel einer VSCode-Extension: https://github.com/microsoft/vscode-extension-samples/tree/main/webview-sample
- Beispiel VSCode-Webview: https://github.com/rebornix/vscode-webview-react
- VSCode Extension-API: https://code.visualstudio.com/api
- VSCode Webview-API: https://code.visualstudio.com/api/extension-guides/webview

Datum des letzten Abrufs aller Online-Quellen: 06.02.2022

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original