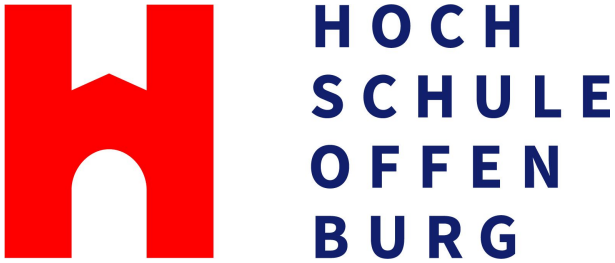


Ein generisches Framework zur Visualisierung von Programmabläufen

Matthias Reichenbach

MASTERTHESIS

zur Erlangung des akademischen Grades Master of Science (M.Sc.)

Studiengang Informatik Master

Fakultät Elektrotechnik, Medizintechnik und Informatik
Hochschule für Technik, Wirtschaft und Medien Offenburg

14.09.2023

Durchgeführt an der Hochschule Offenburg

Betreuer
Prof. Dr. Stefan Wehr, Hochschule Offenburg
Prof. Dr. Joachim Orb, Hochschule Offenburg

Vorwort

Diese Arbeit entstand im Rahmen meines Masterstudiums an der Hochschule Offenburg und befasst sich mit dem Thema *Ein generisches Framework zur Visualisierung von Programmabläufen*. Mein Ziel war es, eine Visualisierung für Programmabläufe zu entwickeln, die vor allem den Einstieg für Programmieranfänger erleichtern soll. Dabei war vor allem wichtig, dass das Framework nicht nur eine intuitive Nutzbarkeit hat, sondern auch in verschiedenen Programmiersprachen eingesetzt werden kann. Von März bis September 2023 beschäftigte ich mich intensiv mit der Forschung, Implementierung und dem Schreiben dieser Masterthesis.

Zuallererst möchte ich meinem Betreuer, Prof. Dr. Stefan Wehr, für seine wertvolle Unterstützung und sein Fachwissen während des gesamten Prozesses danken. Ihre konstruktiven Ratschläge und Ihr Engagement für meine akademische Entwicklung haben mir geholfen, über mich selbst hinauszuwachsen und diese Arbeit erfolgreich abzuschließen.

Außerdem möchte ich mich bei meiner Familie, Verwandten und meinen Freunden für ihre Unterstützung bedanken. Ihr habt mich immer ermutigt und an mich geglaubt.

Abschließend hoffe ich, dass diese Masterthesis nicht nur einen Beitrag zu meinem akademischen Fortschritt darstellt, sondern auch einen aktiven Beitrag für Professoren und Studierende liefern kann.

Vielen Dank an alle, die zu diesem Projekt beigetragen haben, sowie an die Leser, die sich die Zeit nehmen, meine Arbeit zu studieren. Ich wünsche Ihnen viel Freude beim Lesen dieser Masterthesis.

Mit herzlichsten Grüßen

Matthias Reichenbach
Offenburg, 14. September 2023

Eidesstattliche Erklärung

Hiermit versichere ich eidesstattlich, dass die vorliegende Masterthesis von mir selbstständig und ohne unerlaubte fremde Hilfe angefertigt worden ist, insbesondere, dass ich alle Stellen, die wörtlich oder annähernd wörtlich oder dem Gedanken nach aus Veröffentlichungen, unveröffentlichten Unterlagen und Gesprächen entnommen worden sind, als solche an den entsprechenden Stellen innerhalb der Arbeit durch Zitate kenntlich gemacht habe, wobei in den Zitaten jeweils der Umfang der entnommenen Originalzitate kenntlich gemacht wurde. Ich bin mir bewusst, dass eine falsche Versicherung rechtliche Folgen haben wird.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, d. h. dass die Arbeit elektronisch gespeichert, in andere Formate konvertiert, auf den Servern der Hochschule Offenburg öffentlich zugänglich gemacht und über das Internet verbreitet werden darf.

Offenburg, 14.09.2023

Matthias Reichenbach

Zusammenfassung

Die Visualisierung von Programmabläufen ist ein zentraler Aspekt für Programmieranfänger, um das Verständnis von Codeabläufen zu erleichtern und den Einstieg in der Softwareentwicklung zu unterstützen. In dieser Masterthesis wird ein speziell auf die Bedürfnisse von Einsteigern zugeschnittenes generisches Framework vorgestellt, wobei der Fokus auf einer einfachen, verständlichen aber auch korrekten Darstellung der Programmausführung liegt. Das Framework integriert das Debugger Adapter Protocol, um den Debugger unterschiedlicher Sprachen ansprechen und verwenden zu können.

In dieser Arbeit werden zunächst die Anforderungen für das generische Framework diskutiert. Anschließend werden bestehende Ansätze zur Visualisierung von Programmabläufen ausführlich untersucht und analysiert. Die Implementierung des Frameworks wird daraufhin detailliert beschrieben, wobei besonderer Wert auf die Erweiterbarkeit unterschiedlicher Sprachen gelegt wird.

Um die Eignung des Frameworks zu evaluieren, werden mehrere Aufgaben aus dem ersten Modul mit der jeweiligen Programmiersprache des Studiengangs Angewandte Informatik der Hochschule Offenburg betrachtet. Die Ergebnisse zeigen, dass das Framework mit den Aufgaben umgehen und diese korrekt und verständlich darstellen kann.

Abstract

The visualization of program sequences is a central aspect for programming beginners in order to facilitate the understanding of code sequences and to support the entrance into the software development. In this master thesis, a generic framework specifically tailored to the needs of beginners is presented, focusing on a simple, understandable but also correct representation of the program execution. The framework integrates the Debugger Adapter Protocol to be able to address and use the debugger of different languages.

In this paper the requirements for the generic framework are first discussed. Then, existing approaches for visualizing program flows are examined and analyzed in detail. The implementation of the framework is then described in detail, with special emphasis on the extensibility of different languages.

To evaluate the suitability of the framework, several tasks from the first semester of the *Angewandte Informatik* course are considered. The results show that the framework can handle the tasks and present them correctly and understandably.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
  - 1.1.1 Informationsüberfluss . . . 2
  - 1.1.2 Handhabung . . . 3
  - 1.1.3 Textuelle Darstellung . . . 4
- 1.2 Ziel . . . 5
- 1.3 Struktur der Arbeit . . . 6

2 Anforderungen **8**
- 2.1 Handhabung und Bedienbarkeit . . . 8
- 2.2 Darstellung . . . 9
- 2.3 Effizienz . . . 10
- 2.4 Unterstützung verschiedener Programmiersprachen . . . 11
- 2.5 Unterstützung unterschiedlicher Versionen von Programmiersprachen 11

3 Vergleich existierender Lösungen **13**
- 3.1 Debug Visualizer . . . 13
  - 3.1.1 Handhabung und Bedienbarkeit . . . 13
  - 3.1.2 Darstellung . . . 15
  - 3.1.3 Effizienz . . . 16
  - 3.1.4 Unterstützung verschiedener Programmiersprachen . . . 16
  - 3.1.5 Unterstützung unterschiedlicher Versionen von Programmiersprachen . . . 17
  - 3.1.6 Zwischenfazit Debug Visualizer . . . 17
- 3.2 Python Tutor . . . 18
  - 3.2.1 Handhabung und Bedienbarkeit . . . 19
  - 3.2.2 Darstellung . . . 21
  - 3.2.3 Effizienz . . . 23
  - 3.2.4 Unterstützung verschiedener Programmiersprachen . . . 24
  - 3.2.5 Unterstützung unterschiedlicher Versionen von Programmiersprachen . . . 24
  - 3.2.6 Zwischenfazit Python Tutor . . . 25
- 3.3 Fazit des Vergleichs existierender Lösungen . . . 25

4 Grundlagen **27**
4.1 Plattform Visual Studio Code 27
4.2 Debug Adapter Protocol 28
4.3 Vorarbeit 31
4.3.1 Bedienbarkeit, Handhabung und Darstellung 31
4.3.2 Unterstützung verschiedener Programmiersprachen 32
4.3.3 Komplexe Datentypen 33
4.3.4 Effizienz 34
4.3.5 User Experience 34

5 Architektur **35**
5.1 Einfache Darstellung 35
5.2 Detaillierte Darstellung 36
5.2.1 Extension 37
5.2.2 Backend 38
5.2.3 Frontend 42
5.2.4 View 42

6 Generisches Erweitern um neue Programmierprachen **44**
6.1 Vorgehensweise für das Erweitern 44
6.1.1 Kategorisierung von Datentypen 44
6.1.2 Debugger Output evaluieren 45
6.2 Veranschaulichung von Java und Python 45
6.2.1 Kategorisierung von Datentypen 45
6.2.2 Debugger Output evaluieren 46

7 Erweiterung und Verbesserung für die Programmiersprache Python **52**
7.1 Unterstützung unterschiedlicher Programmiersprachen 52
7.2 Komplexe Datentypen 53
7.3 Testen 55
7.4 User Experience 56
7.5 Bestandsaufnahme 56

8 Erweiterung für die Programmiersprache Java **58**
8.1 Generelle Erweiterungsschritte 58
8.2 Referenzen am Beispiel eines Strings 60
8.3 Wrapper-Typen und Strings 61
8.4 HashMaps 63
8.5 Testen 64
8.6 Bestandsaufnahme 65

9 Testen von Visual Studio Code Extensions **66**
9.1 Struktur der Testumgebung 66
9.2 Struktur eines Tests 67

9.3 Konkretes Testen . . . 69
9.3.1 Java und Python . . . 69
9.3.2 Hilfskomponenten . . . 71
9.3.3 Fehlende Tests . . . 71
9.4 Continuous Integration . . . 72
9.4.1 Workflow für GitHub Action . . . 72
9.4.2 Einschränkungen von GitHub Action . . . 74

10 Evaluierung mit Aufgaben aus Vorlesungen 76
10.1 Python Aufgaben . . . 76
10.1.1 Listen und Dictionaries . . . 76
10.1.2 Gemischte Daten . . . 81
10.2 Java Aufgaben . . . 84
10.2.1 Polynome . . . 85
10.2.2 Klassen . . . 87

11 Diskussion 90
11.1 Evaluation . . . 90
11.2 Ausblick . . . 92

12 Fazit 94

Abkürzungsverzeichnis viii

Tabellenverzeichnis ix

Abbildungsverzeichnis x

Listings xii

Literaturverzeichnis xiv

1 Einleitung

Die Programmierung ist eine wertvolle Fertigkeit, die in der heutigen technologiegetriebenen Welt zunehmend an Bedeutung gewinnt. In einer Zeit, in der Software nahezu jeden Aspekt unseres Lebens durchdringt, öffnet die Fähigkeit, Code zu schreiben, Türen zu einer Vielzahl von beruflichen und kreativen Möglichkeiten. Doch der erste Schritt in die faszinierende Welt der Programmierung kann für Neulinge eine beträchtliche Herausforderung darstellen. Die vielschichtigen Verbindungen zwischen Hardware und Software sowie die daraus resultierenden komplexen Programmabläufe können leicht zu Verwirrung und Frustration führen.

In diesem Zusammenhang erweist sich die Visualisierung von Programmabläufen als entscheidender Schlüssel, um das Verständnis von Code zu erleichtern und angehenden Programmierern einen unterstützenden Eintritt in die Welt der Softwareentwicklung zu ermöglichen. Die Fähigkeit, visuell nachvollziehen zu können, wie ein Programm Schritt für Schritt ausgeführt wird, kann den Lernprozess erheblich beschleunigen und die Hemmschwelle für Neulinge verringern.

1.1 Motivation

Der Einstieg in die Welt der Programmierung kann für Einsteiger oft wie das Betreten eines unbekannten Terrains erscheinen. Insbesondere abstrakte Konzepte wie *Stack*, *Heap*, *Map*, *Array* oder *Dictionary* können anfangs eine beeindruckende Herausforderung darstellen, wenn es darum geht, die ersten Verbindungen zwischen eigenem Code und seiner Ausführung auf einem Computer zu verstehen. Fragen wie zum Beispiel, wann Daten im *Heap* oder im *Stack* abgelegt werden oder wie sich eine erneute Zuweisung an ein bestehendes Array auf den *Heap* auswirkt, lassen sich oft nicht allein durch eine Überprüfung des Codes beantworten. Hier spielt der Debugger eine entscheidende Rolle und bietet die Möglichkeit, solche Zusam-

menhänge aufzuzeigen. Bei der Betrachtung der Erkenntnisse von [Gretsch und Holzäpfel 2016] wird deutlich, dass das Lernen und Verstehen durch Visualisierung einen deutlichen Einfluss auf den Lernerfolg hat. Dies wirft die Frage auf: Warum nicht Visualisierung und den Debugger kombinieren, um den Einstieg für Programmieranfänger zu erleichtern?

1.1.1 Informationsüberfluss

Die Verwendung eines Debuggers führt nicht immer sofort zu klaren Erkenntnissen. Um dies zu verdeutlichen, wird ein Python-Node betrachtet, der exemplarisch im folgenden Listing 1.1 von Zeile 1 bis 5 definiert ist.

```
1  class Node:
2      def __init__(self, data):
3          self.data = data
4          self.prev = None
5          self.next = None
6
7  node = Node(10)
```

Listing 1.1: Python-Code einer einfachen Nodestruktur

Ein solcher Node wird zum Beispiel bei doppelt verketten Listen verwendet. Er besitzt drei Attribute: *data*, *prev* und *next*. Beim Erzeugen eines Nodes, wie in Zeile 7 dargestellt, wird der Wert für das Attribut *data* übergeben. Der Speicherzustand nach Zeile 7 kann mittels eines Debuggers untersucht werden. Der dargestellte Output in VSC[1] ist in Abbildung 1.1 zu sehen.

Auf den ersten Blick sollte erkennbar sein, dass eine lokale Variable vorhanden ist, die einen Node mit den oben genannten Attributen repräsentiert, und dass *data* auf 10 gesetzt wurde. Allerdings werden zahlreiche weitere Werte dargestellt, wie zum Beispiel *class variables* oder *special variables*. Wenn der Node sowohl unter *Locals* als auch unter *Globals* aufgeführt ist, kann dies zu Missverständnissen führen.

[1] Visual Studio Code

Abbildung 1.1: Debugger-Output für die Klasse Node

1.1.2 Handhabung

Die Verwendung eines Debuggers ist für Anfänger nicht immer selbsterklärend. Zum Beispiel unterstützt VSC die Debugger-Steuerelemente standardmäßig mit den Funktionen *Continue / Pause*, *Step Over*, *Step Into*, *Step Out*, *Restart* und *Stop*, wie auch in Abbildung 1.2 in selbiger Reihenfolge aufgeführt.

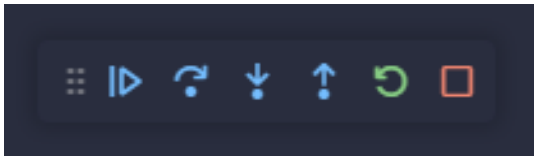


Abbildung 1.2: Debugger-Steuerelemente in VSC

Wie in der Dokumentation von [Microsoft 2023b] beschrieben, werden in der folgenden Tabelle 1.1 die Steuerelemente beschrieben.

Schritt	Beschreibung
Continue	Normale Programm-/Skriptausführung fortsetzen bis zum nächsten Breakpoint
Pause	Überprüfung der aktuell ausgeführten Zeile
Step Over	Nächste Methode als einen einzigen Befehl ausführen, ohne die einzelnen Schritte zu überprüfen oder zu verfolgen
Step Into	In nächste Methode eintreten, um ihre Ausführung Zeile für Zeile zu verfolgen
Step Out	Wenn in einer Methode oder einem Unterprogramm, zum vorherigen Ausführungskontext zurückkehren, indem die verbleibenden Zeilen der aktuellen Methode wie einen einzelnen Befehl ausführen
Restart	Beendet die aktuelle Programmausführung und startet das Debugging erneut mit der aktuellen Laufkonfiguration
Stop	Beendet die aktuelle Programmausführung

Tabelle 1.1: Beschreibung der Debugger-Steuerelemente in VSC

Aus der Perspektive eines Programmieranfängers kann es jedoch unklar sein, wann welche dieser Funktionen konkret verwendet werden soll. Ein anderes Problem ist das Zurückspringen im Code. Sobald ein Schritt ausgeführt wurde, ist es mit dem Standard-Debugger nicht mehr möglich in der gleichen Debugger-Session an dieselbe Stelle erneut zu navigieren. Falls also ein wichtiger Punkt innerhalb des Codes erreicht wurde und über diesen gesprungen wird, muss erneut der ganze Debugging-Prozess gestartet werden, um die vorherige Stelle zu erreichen.

1.1.3 Textuelle Darstellung

Wie bereits in der Einleitung des Kapitels erläutert wurde, ist es effizienter, mit einer visuellen Darstellung zu arbeiten, anstelle einer rein textuellen Darstellung. Abbildung 1.1 verdeutlicht anschaulich, wie herausfordernd es sein kann, sich Datenstrukturen wie Klassen, aber auch *Maps* oder *Dictionaries*, rein textuell vorzustellen. Das Beispiel in Abbildung 1.3 vermittelt dagegen eine viel bessere Vorstellung von der Instanz der Klasse Node.

Dieses Bild wurde dem *Python Tutor* entnommen, welcher durch die Visualisierung von Programmausführung das Lernen einer Programmiersprache erleichtern möchte.

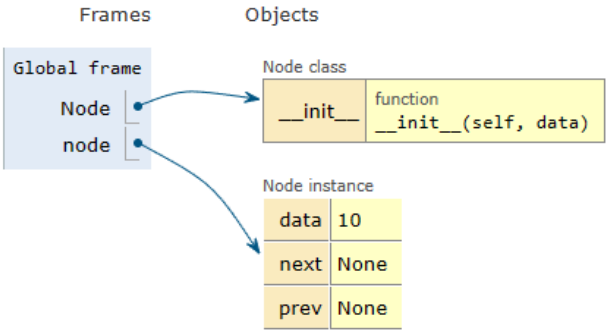


Abbildung 1.3: Beispielhafte visuelle Darstellung eines Node aus [Python Tutor 2023a]

1.2 Ziel

Wie bereits in Kapitel 1.1 dargelegt, bestehen diverse Herausforderungen beim konventionellen Einsatz von Debuggern. Das übergeordnete Ziel dieser Arbeit ist die Entwicklung eines anfängerfreundlichen und flexiblen Frameworks zur Visualisierung von Programmabläufen. Dieses Framework soll angehenden Programmierenden dabei helfen, die komplexen Verbindungen zwischen ihrem geschriebenen Code und dessen tatsächlicher Ausführung auf dem Computer besser zu verstehen.

Entwicklung eines Generischen Frameworks Die Hauptzielsetzung besteht in der Konzeption und Implementierung eines Frameworks, das unabhängig von der verwendeten Programmiersprache eingesetzt werden kann. Das Framework soll die Interaktion unterschiedlicher Sprachen ermöglichen und gleichzeitig eine klare und verständliche visuelle Darstellung von Programmabläufen bieten.

Verbesserung des Verständnisses Das Framework soll dazu beitragen, das Verständnis von Codeabläufen bei Programmieranfängern zu fördern. Durch die Möglichkeit, den Programmfluss schrittweise zu verfolgen und visuelle Darstellungen von Variablen, Datenstrukturen und Speicherzuweisungen zu erhalten, sollen komplexe Konzepte besser erfassbar und begreifbar gemacht werden.

Erweiterbarkeit und Anpassungsfähigkeit Ein weiteres Ziel ist die Schaffung eines flexiblen Frameworks, das leicht erweitert und an die spezifischen Anforderungen unterschiedlicher Programmiersprachen und Lernumgebungen angepasst werden kann.

Evaluation der Effektivität Die Effizienz des entwickelten Frameworks soll anhand praktischer Beispiele aus dem ersten Semester des Studiengangs AI[2] bewertet werden. Hierbei werden nicht nur technische Aspekte berücksichtigt, sondern auch die tatsächliche Verbesserung des Verständnisses und des Lernfortschritts der Nutzer werden untersucht.

Beitrag zur Bildung Letztendlich zielt diese Arbeit darauf ab, einen Beitrag zur Verbesserung der Bildungslandschaft im Bereich der Programmierung zu leisten. Das entwickelte Framework, das das Lernen und Verstehen unterstützt, soll angehenden Programmierenden den Einstieg in die Welt der Softwareentwicklung erleichtern und Hemmschwellen verringern. Besonders erstrebenswert ist es, das Framework in den ersten Lehrveranstaltungen zu einer Programmiersprache im Studium einzusetzen.

1.3 Struktur der Arbeit

Die Struktur dieser Arbeit wird im Folgenden erläutert.

Nach der Einleitung folgt im zweiten Kapitel *Anforderungen* die Definition der spezifischen Anforderungen an die zu entwickelnde Lösung. Dies umfasst Aspekte wie Handhabung und Bedienbarkeit, Darstellung, Effizienz, Unterstützung verschiedener Programmiersprachen und Unterstützung unterschiedlicher Versionen von Programmiersprachen. Die klare Formulierung dieser Anforderungen legt den Rahmen für die spätere Evaluierung und den Vergleich bestehender Lösungen.

Im dritten Kapitel *Vergleich existierender Lösungen* werden zwei wichtige bestehende Lösungen, der *Debug Visualizer* und *Python Tutor*, detailliert analysiert und den Anforderungen entsprechend bewertet. Dies ermöglicht einen Überblick über aktuelle Lösungen und zeigt die Notwendigkeit eines eigen implementierten Frameworks auf.

Das Kapitel *Grundlagen* thematisiert die Bachelorthesis von [Velten 2023]. Die Arbeit *Visualisierung von Python Programmen in der Entwicklungsumgebung Visual Studio Code* handelt von einem Framework, das innerhalb von VSC verwendet wird, um die Programmabläufe visuell darzustellen. Die Erläuterung der wichtigs-

[2] Angewandte Informatik

ten Aspekte dieser Arbeit dient dazu, den Kontext für die weitere Entwicklung und Verbesserung der Lösung zu schaffen.

Die Architektur der entwickelten Lösung wird in dem Kapitel *Architektur* beschrieben. Hier werden sowohl eine einfache als auch eine detaillierte Darstellung der Architektur präsentiert, um dem Leser ein klares Verständnis davon zu vermitteln, wie die Lösung funktioniert und aufgebaut ist.

Das Kapitel *Generisches Erweitern um neue Programmierprachen* beschreibt die Methodik und Vorgehensweise, wie die Lösung für verschiedene Programmiersprachen erweitert werden kann. Dieses Vorgehen wird durch Java und Python veranschaulicht.

In den darauf folgenden Kapiteln *Erweiterung und Verbesserung für die Programmiersprache Python* sowie *Erweiterung für die Programmiersprache Java* wird die spezifische Implementierung und Integration der Lösung für die Programmiersprachen Python und Java behandelt. Hier werden Anpassungen und mögliche Limitationen für jede Programmiersprache diskutiert.

Das Kapitel *Testen von Visual Studio Code Extensions* beschreibt die Möglichkeiten des Testens von Extensions in VSC. Dabei wird auf die generelle Struktur eingegangen, aber auch die Integration in eine CI[3].

Die Evaluierung der Lösung anhand von realen Vorlesungsaufgaben für Java und Python wird im Kapitel *Evaluierung mit Aufgaben aus Vorlesungen* präsentiert. Hier werden die Ergebnisse und Erkenntnisse aus den Tests analysiert und interpretiert.

Die *Diskussion* bietet Raum für die Bewertung der erreichten Ergebnisse und für einen Ausblick, um mögliche Weiterentwicklungen und Anwendungsbereiche der entwickelten Lösung zu diskutieren.

Abschließend fasst das *Fazit* die wichtigsten Erkenntnisse der Arbeit zusammen.

Das erarbeitete Framework ist als öffentliches GitHub Repository von [Reichenbach 2023] zu finden.

[3] Continuous Integration

2 Anforderungen

Dieses Kapitel dient der Festlegung des Rahmens für ein Framework und definiert die zu erfüllenden Anforderungen. Dabei werden fünf Hauptkriterien betrachtet: Handhabung und Bedienbarkeit, Darstellung, Effizienz, Unterstützung unterschiedlicher Versionen von Programmiersprachen und Unterstützung verschiedener Programmiersprachen. Im Folgenden werden diese Hauptkriterien näher erläutert, um ihre Bedeutung hervorzuheben.

2.1 Handhabung und Bedienbarkeit

Die Nutzung eines Debugger ist zunächst nicht immer so intuitiv, wie es erhofft wird. Dies wurde bereits in Kapitel 1.1.2 näher erläutert. Das Ziel eines passenden Frameworks ist es, den Benutzer so zu unterstützen, dass die richtigen Schritte automatisch erkannt und ausgeführt werden. Dadurch wäre lediglich ein Vor- und Zurückgehen zur Steuerung notwendig. Abbildung 2.1 zeigt beispielhafte Steuerelemente, die die Bedienbarkeit vereinfachen sollen.

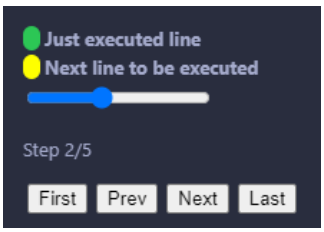


Abbildung 2.1: Bedienelemente der Visualisierung

Über die *First*, *Prev*, *Next* und *Last* Buttons kann der erste, vorherige, nächste oder letzte Schritt aufgerufen werden. Darüber ist ein Schieberegler erkennbar, der eine variable Veränderung des aktuell angezeigten Schrittes ermöglicht. Zusätzlich ist

eine Legende vorhanden, die für den Programmcode selbst gedacht ist. Ähnlich wie beim Debuggen selbst soll die aktuelle Zeile hervorgehoben werden. Um das Verständnis des Ablaufs zu verbessern, sollte auch die als nächstes auszuführende Zeile markiert werden. In Abbildung 2.2 ist zu erkennen, dass Zeile 5, die aktuell ausgeführt wird, grün hervorgehoben wird und die danach auszuführende Zeile 7 gelb markiert ist. Diese Farben entsprechen denen in der Legende aus Abbildung 2.1.

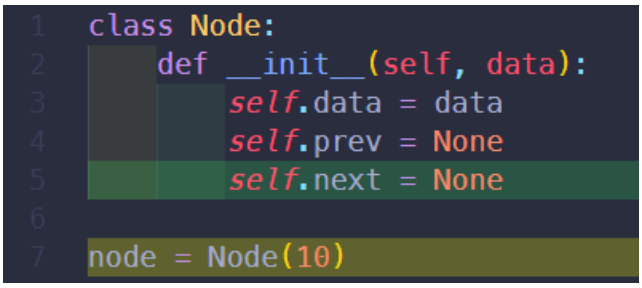


Abbildung 2.2: Hervorheben der aktuellen und der nächst auszuführenden Zeile im Programmcode

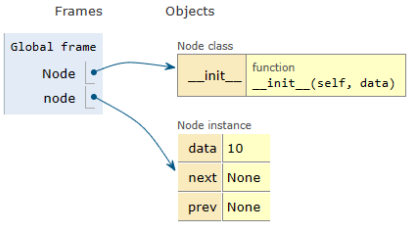
Zudem sollte die Handhabung des Frameworks einfach gehalten werden. Dies kann beispielsweise durch die Integration in eine Entwicklungsumgebung oder einen Quelltexteditor erreicht werden. Entwicklungsumgebungen wie Visual Studio, IntelliJ IDEA, VSC und Eclipse unterstützen Plugins beziehungsweise Erweiterungen, die eine nahtlose Integration der Visualisierung in die vertraute Arbeitsumgebung ermöglichen. Dies steigert nicht nur die Bequemlichkeit, sondern erleichtert auch den Übergang zu fortgeschrittenen Texteditoren. Dadurch ist es möglich, überflüssige Schritte bis zur Visualisierung zu vermeiden und die Visualisierung durch einen Button neben dem Programmcode zu starten.

2.2 Darstellung

Um dem Informationsüberfluss entgegenzuwirken, wie in Kapitel 1.1.1 herausgearbeitet, muss eine Reduktion auf das Wesentliche angestrebt werden. In Abbildung 2.3 sind nochmals die textuelle als auch visuelle Darstellung des Debugger-Outputs gegenüber gestellt.

Dadurch wird eine wesentlich ansprechendere und anschaulichere Darstellung erreicht. In dieser visuellen Darstellung sind keine *special* oder *class variables* sichtbar. Die vom Debugger gelieferten Daten werden übersichtlich zusammengefasst

(a) Textuelle Darstellung



(b) Visuelle Darstellung aus [Python Tutor 2023a]

Abbildung 2.3: Gegenüberstellung von textuellem und visuellem Debugger-Output

und auf eine verständliche Art und Weise dargestellt, was zu einer Verbesserung der Lesbarkeit führt. Durch die Verwendung von grafischen Elementen wie Kästen und Pfeilen sowie die Gegenüberstellung von *Frames* und *Objects* können Datenstrukturen und Datentypen übersichtlicher und verständlicher dargestellt werden. Aus diesem Grund ist es von entscheidender Bedeutung, dass die Darstellung sowohl sinnvoll als auch nachvollziehbar ist.

2.3 Effizienz

Um die UX[1] bei der Betrachtung der Programmablaufvisualisierung zu optimieren, ist es am besten, wenn die Visualisierung direkt durch das Drücken eines Buttons oder einer ähnlichen Aktion angezeigt wird. Im Vergleich dazu würde die schrittweise textuelle Anzeige durch den Debugger sowohl die Entscheidung, welcher Schritt als Nächstes auszuführen ist, als auch die Reaktionszeit des Debuggers nach Drücken eines Schrittes erheblich verzögern. Dabei wird nicht nur die Zeit des Nutzers betrachtet, sondern auch die Zeit des Debuggers selbst, welche dieser benötigt, um die Ansicht zu aktualisieren und die gegebene Zeile auszuführen. Das überge-

[1] User Experience

ordnete Ziel besteht darin, die Effizienz zu maximieren und diese Verzögerungen zu minimieren. Ein weiterer Punkt, an welcher Effizienz eine Rolle spielt, ist, wenn dasselbe Programm mehrmals betrachtet wird. Dabei sollte dies im Vergleich zur ersten Betrachtung der Visualisierung spürbare Zeitersparnisse bieten.

2.4 Unterstützung verschiedener Programmiersprachen

Nicht alle Anfänger in der Programmierung starten mit derselben Sprache. Außerdem verhält sich nicht jede Programmiersprache gleich. Um die Notwendigkeit der Verwendung mehrerer verschiedener Anwendungen oder Erweiterungen zu vermeiden, sollte das Framework nicht auf eine spezifische Programmiersprache beschränkt sein. Bei der Betrachtung des Modulhandbuchs der [Hochschule Offenburg 2023] wird deutlich, dass Studierende des Studiengangs AI zuerst mit Java, C, C++, C# und .NET in Berührung kommen. Zusätzlich sollte der aktuelle Trend im Bereich Künstlicher Intelligenz und Maschinellen Lernens berücksichtigt werden. In diesem Bereich hat Python eine wichtige Rolle.

Es ist festzuhalten, dass die Sprachen Java, Python, C und C++ am häufigsten als erste Programmiersprachen verwendet werden und somit Programmieranfänger zuerst mit diesen konfrontiert sind. Aus diesem Grund sollte das Framework eine Unterstützung für genau diese Programmiersprachen bieten.

2.5 Unterstützung unterschiedlicher Versionen von Programmiersprachen

Programmiersprachen werden kontinuierlich weiterentwickelt. Der *Python Release Cycle* der [Python Software Foundation 2023] verdeutlicht in Abbildung 2.4, dass im Jahr 2023 die Python-Versionen 3.7 bis 3.12 aktuell sind.

Dies lässt vermuten, dass es durchaus möglich ist, dass eine Person Version 3.10 nutzt, während eine andere Person bereits Version 3.11 verwendet. Da Anfänger in der Programmierung wahrscheinlich weniger vertraut mit den Unterschieden zwischen verschiedenen Versionen einer Programmiersprache sind und sich eher auf die grundlegenden Funktionen konzentrieren möchten, sollte das Framework genau diese Anforderung unterstützen.

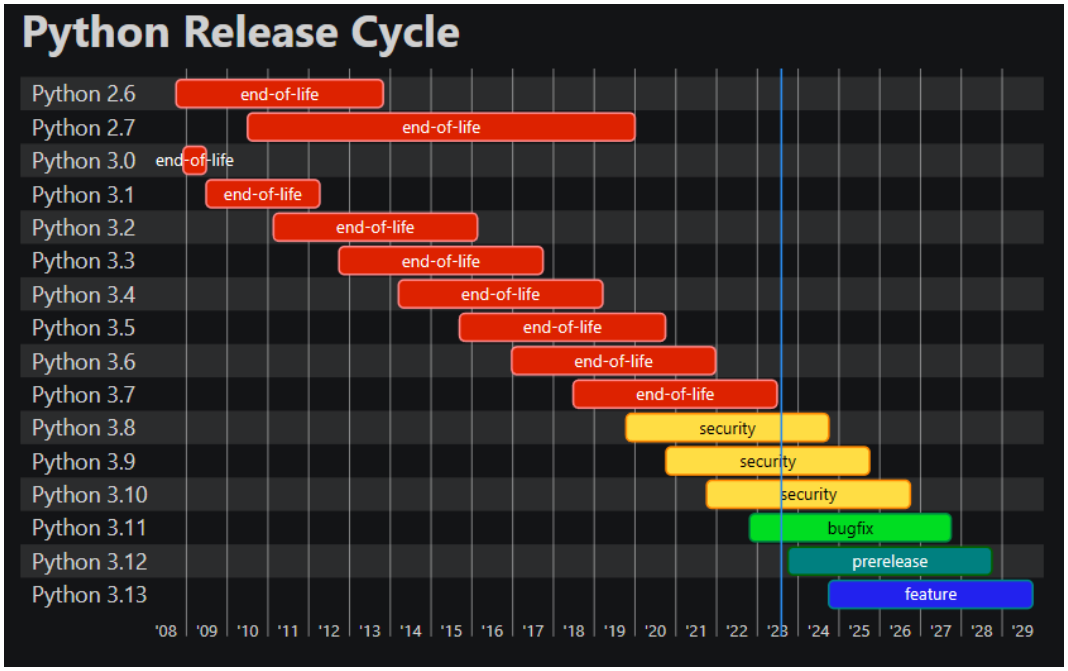


Abbildung 2.4: Python Release Cycle aus [Python Software Foundation 2023]

3 Vergleich existierender Lösungen

Bei der Recherche nach bestehenden Lösungen zur Visualisierung des Programmablaufs wird deutlich, dass dieser Ansatz keineswegs neu ist. Im Verlauf dieser Untersuchung wurde ein entscheidendes Kriterium berücksichtigt: Die Unterstützung mehrerer Programmiersprachen, wie bereits in Kapitel 2.4 beschrieben. Auf der Grundlage dieses Kriteriums wurden zwei aktuelle Lösungen identifiziert: *Debug Visualizer*, eine Erweiterung für VSC, und *Python Tutor*, eine webbasierte Plattform. Im kommenden Abschnitt werden diese beiden Lösungen ausführlich betrachtet und mit den Anforderungen verglichen, die in Kapitel 2 formuliert sind.

3.1 Debug Visualizer

Die VSC-Erweiterung *Debug Visualizer* von [Henning Dieterichs 2023] ermöglicht es Entwicklern, Daten während des Debuggens in ansprechenden visuellen Darstellungen anzuzeigen. Diese Erweiterung bietet eine Möglichkeit zur Datenvisualisierung innerhalb des Debugging-Prozesses. Dabei stehen anpassbare und interaktive Darstellungen zur Verfügung, die verschiedene Datentypen unterstützen. Die Erweiterung betont eine einfache Integration und Erweiterbarkeit. Das Hauptziel besteht darin, das Verständnis und die Analyse von Daten im Debugging-Prozess zu erleichtern. Die Erweiterung kann direkt aus dem VSC-Marketplace heruntergeladen werden.

3.1.1 Handhabung und Bedienbarkeit

Wie bereits erwähnt, ist die Erweiterung in VSC integriert und ermöglicht somit die Nutzung direkt innerhalb einer Entwicklungsumgebung. Jedoch gibt es hierbei eine wichtig zu nennende Hürde: Eine ähnliche Darstellung, wie aus den vorge-

stellten Beispielen aus dem Marketplace von [Henning Dieterichs 2023], gestaltet sich nicht so einfach, wie es auf den ersten Blick scheint. Es fehlt an einer klaren und verständlichen Dokumentation. Tatsächlich erfordert die Verwendung der Erweiterung in den meisten Fällen eine umfassende Anpassung des Codes, um die Visualisierung zu erreichen.

Beim Betrachten von zum Beispiel Python-Code wird für die Visualisierung in *Debug Visualizer* eine Variable, welche Daten im JSON[1]-Format speichert, benötigt. Die Methode `serialize`, in folgendem Listing 3.1 dargestellt, ist ein Beispiel dafür, wie ein Array zur Visualisierung umgewandelt werden kann.

```
1   def serialize(arr):
2       formatted = {
3           "kind": {"grid": True},
4           "rows": [
5               {
6                   "columns": [
7                       {"content": str(value), "tag": str(value)} for value in arr
8                   ],
9               },
10          ],
11      }
12      return formatted
```

Listing 3.1: `serialize`-Methode für die Visualisierung eines Arrays in *Debug Visualizer*

Die Methode `serialize` wandelt einen Array in JSON um und ermöglicht *Debug Visualizer* diesen darzustellen. Der Gebrauch wird in Listing 3.2 gezeigt.

```
1   arr = [6, 9, 3, 12, 1, 11, 5, 13, 8, 14, 2, 4, 10, 0, 7]
2   visualization = serialize(arr)
```

Listing 3.2: Beispiel Code für die Visualisierung eines Arrays mit *Debug Visualizer*

Das Beispiel zeigt einen Array, der in Zeile 2 mithilfe von `serialize` der Variable *visualization* als JSON zugewiesen wird. Das Gesamtbild innerhalb VSC wird nach der Ausführung in der Abbildung 3.1 dargestellt.

Auf der linken Seite ist der Programmcode innerhalb einer Debugging-Session zu sehen, während auf der rechten Seite die Ansicht des *Debug Visualizer* dargestellt ist. Unter dem geöffneten Tab ist ein Textfenster zu erkennen (rot umrandet), in welchem der Variablenname der in Listing 3.2 Zeile 2 definierten Variable steht. Die Visualisierung funktioniert über die selbst erstellte Variable, welche das JSON

[1] JavaScript Object Notation

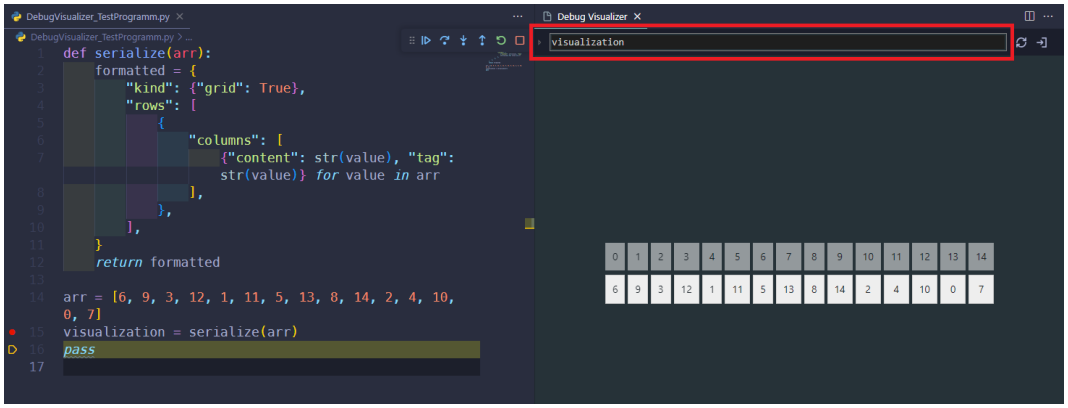


Abbildung 3.1: Nebeneinanderstellung von Programmcode und *Debug Visualizer* mit Visualisierung eines Arrays

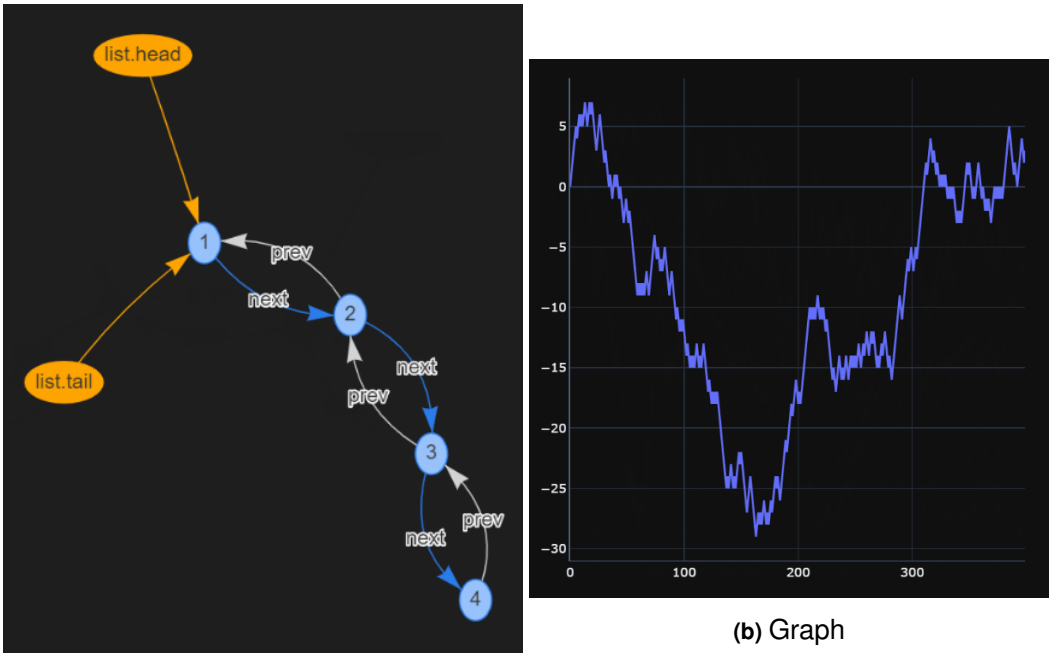
enthält. Innerhalb des Fensters ist der Array sichtbar, der mithilfe der `serialize`-Methode erstellt wurde.

Wie das Beispiel zeigt, ist die Handhabung und Bedienbarkeit eher eingeschränkt. Es fehlt an einer nutzerfreundlichen Vereinfachung und die Anwendung erfordert zusätzlichen selbst geschriebenen Code, was das Verständnis und die Nutzung vorerst erschwert. Einzig, wie [Henning Dieterichs 2023] schreibt, ist es mit TypeScript, JavaScript oder einer ähnlichen Programmiersprache möglich, eine benutzerfreundliche Handhabung ohne manuelles Schreiben von JSON zu erreichen.

3.1.2 Darstellung

Wie bereits im vorherigen Abschnitt *Handhabung und Bedienbarkeit* erläutert, erfolgt die Auswahl der passenden Visualisierung für den Programmcode durch die Verwendung von JSON. Die Vielfalt der Möglichkeiten wird auf der Marketplace-Seite von [Henning Dieterichs 2023] hervorgehoben. Als Beispiele werden eine doppelt verkettete Liste und ein Graph aufgeführt, wobei jeweils eine beispielhafte Visualisierung in Abbildung 3.2 dargestellt ist.

Die Erstellung der Visualisierung selbst erfolgt durch das Schreiben einer JSON-Definition. Dadurch sind der Darstellung praktisch keine Grenzen gesetzt und es besteht die Möglichkeit, jede gewünschte Visualisierung zu verwenden, die den individuellen Vorstellungen und Bedürfnissen entspricht.



(a) Double Linked List

(b) Graph

Abbildung 3.2: Beispieldarstellungen von *Debug Visualizer* aus [Henning Dieterichs 2023]

3.1.3 Effizienz

Wie bereits weiter oben in Abschnitt 3.1.1 beschrieben, verwendet *Debug Visualizer* eigens erstellte Variablen als Mittel zur Visualisierung. Jeder relevante Schritt wird durch die Aktualisierung dieser Variable visualisiert. In diesem Sinne bietet die Erweiterung eine Effizienz, die der manuellen Debug-Ausführung entspricht. Allerdings kommt zusätzlich die Zeit hinzu, die benötigt wird, um den Programmcode kompatibel mit der Erweiterung zu gestalten. Dies wirkt sich negativ auf die Effizienz aus. Die zusätzliche Zeit, die für die Erstellung einer passenden JSON-Definition sowie das Abdecken jeder relevanten Stelle erforderlich ist, spricht nicht für diese Erweiterung. Auch wird nicht die Wahl des nächsten Debbuger-Schrittes beeinflusst, wodurch der Nutzer diesen selbst wählen muss. Der zusätzliche Aufwand wird nur bei der Programmiersprache TypeScript abgenommen, welche aber nicht durch die in den Abschnitt 2.4 gestellten Anforderungen priorisiert wird.

3.1.4 Unterstützung verschiedener Programmiersprachen

Debug Visualizer nutzt die Debugger der jeweiligen Programmiersprachen. So führt [Henning Dieterichs 2023] die getesteten Programmiersprachen mit *Basic Support*

und *Full Support* auf. Für vollständig unterstützte Sprachen stehen Datenextraktoren zur Verfügung, die bestimmte bekannte Datenstrukturen automatisch in JSON konvertieren können. Bei weniger umfangreich unterstützten Sprachen muss dieser Vorgang manuell durchgeführt werden. Die folgende Tabelle 3.1 zeigt die aufgeführten Sprachen sowie den entsprechenden Unterstützungsgrad und Debugger.

Programmiersprache	Debugger	Supportlevel
JavaScript/TypeScript/...	node/node2/extensionHost/chrome/ pwa-chrome/pwa-node	Full
Dart/Flutter	dart	Basic
Go	go	Basic
Python	python	Basic
C#	correclr	Basic
PHP	php	Basic
Java	java	Basic
Swift	lldb	Basic
Rust	lldb	Basic
Ruby	rdbg	Basic

Tabelle 3.1: Auflistung der Programmiersprache und deren Debugger mit dem jeweiligen Supportlevel aus [Henning Dieterichs 2023]

Zu sehen ist, dass mit einfacher Unterstützung mehr als die in Abschnitt 2.4 gestellten Anforderungen erfüllt sind. Jedoch fehlt es an der Menge von Programmiersprachen mit voller Unterstützung.

3.1.5 Unterstützung unterschiedlicher Versionen von Programmiersprachen

Die Verwendung der Programmiersprachen eigenen Debugger, wie in dem vorherigen Abschnitt 3.1.4 beschrieben, ermöglicht die Unterstützung jeder Sprachversion, solange der Debugger nicht wesentlich verändert wurde.

3.1.6 Zwischenfazit Debug Visualizer

Die Anwendung von *Debug Visualizer* für die Visualisierung von Programmabläufen ist nur in gewissem Maße geeignet. Mit der Einschränkung, selbstständig die Visualisierung in JSON zu schreiben und keine Vereinfachung der Bedienelemente zu erhalten, können Programmieranfänger schnell aus dem Konzept gebracht wer-

den. Jedoch könnte diese Erweiterung für erfahrene Anwender wie Dozenten nützlich sein, um Datenstrukturen zu visualisieren und in Vorlesungen zu präsentieren. Auch wenn es möglich ist, jeden Schritt des *Heaps* und *Stacks* aufzuführen, scheint der Nutzen dieser Erweiterung eher auf die Visualisierung von Datenstrukturen als Programmabläufen ausgelegt zu sein.

3.2 Python Tutor

Eine weitere vorhandene Lösung ist *Python Tutor*, welche auf der Website *https://pythontutor.com/* zur Verfügung steht. Um einen Einblick in das Design, die Funktionen und die Nutzbarkeit zu erhalten, zeigt die folgende Abbildung 3.3 die Startseite.

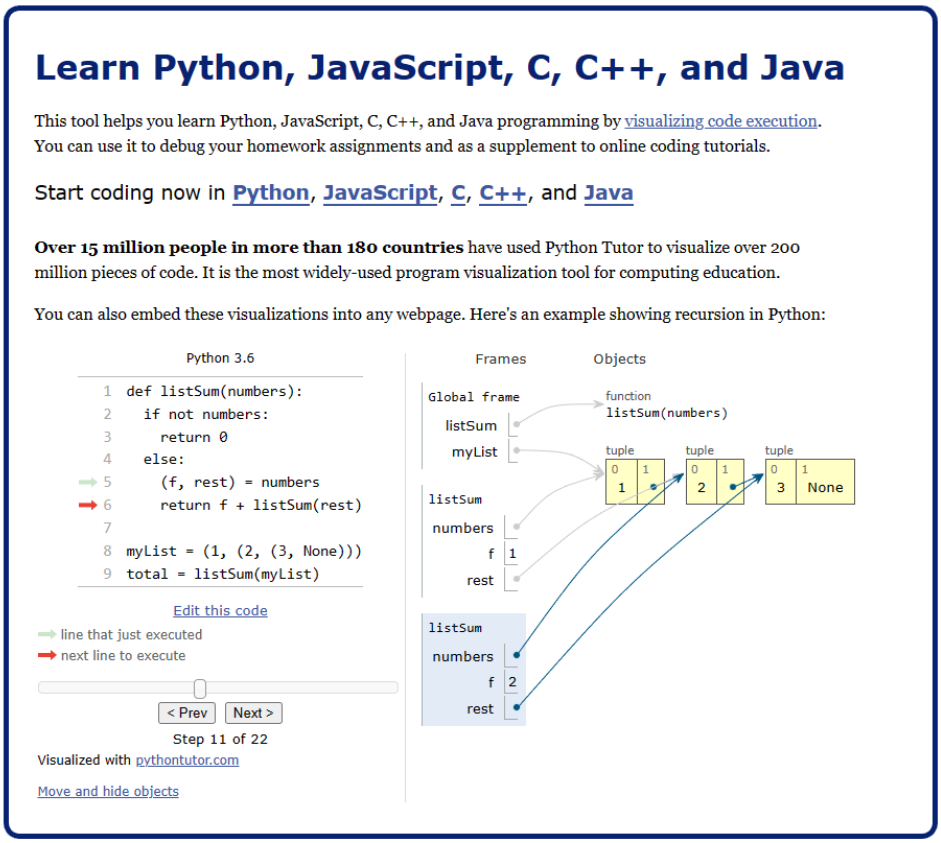


Abbildung 3.3: Startseite von *Python Tutor* aus https://pythontutor.com/

Auf der Startseite ist ein Beispielprogramm zu sehen, das mithilfe von *Frames* und *Objects* visuell dargestellt wird, begleitet von einer entsprechenden Benutzerober-

fläche. In den folgenden Abschnitten wird wiederholt auf diese Abbildung verwiesen und näher auf weitere Details eingegangen.

3.2.1 Handhabung und Bedienbarkeit

Das Visualisieren erfolgt durch das Schreiben oder Einfügen von Programmcode in ein Editorfenster, wie in Abbildung 3.4 dargestellt.

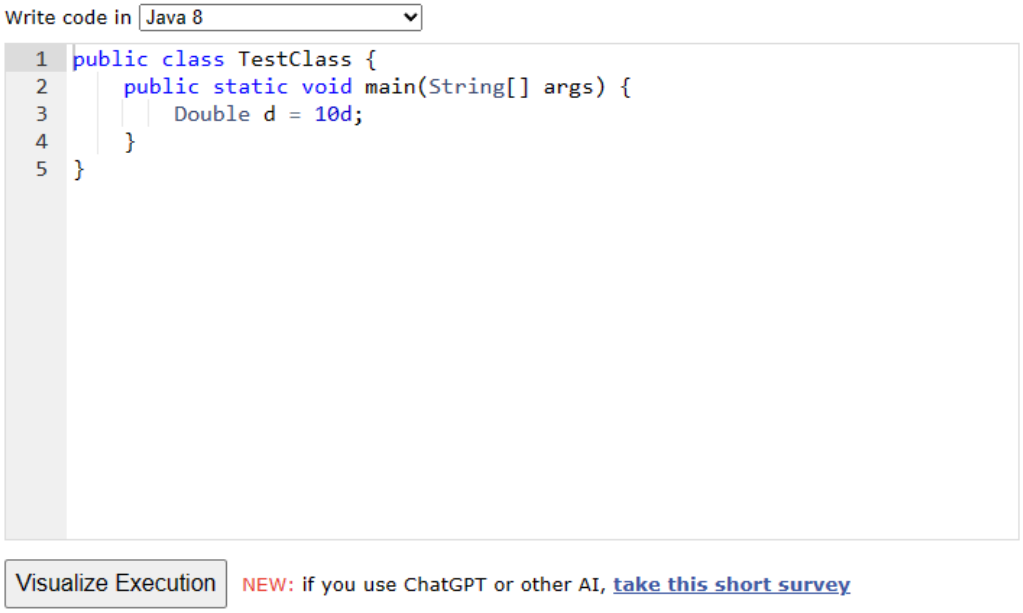


Abbildung 3.4: Editorfenster und Button zur Visualisierung in *Python Tutor*

Das Editorfenster verfügt über Code-Highlighting und Zeilennummerierung. Über dem Editorfenster kann die gewünschte Programmiersprache ausgewählt werden. In diesem Fall ist es Java 8. Genaueres dazu wird in den späteren Abschnitten *Unterstützung verschiedener Programmiersprachen* und *Unterstützung unterschiedlicher Versionen von Programmiersprachen* erläutert. Durch einen Button mit der Beschriftung *Visualize Execution* kann der Programmcode visualisiert werden. Das sich öffnende Fenster wird in Abbildung 3.5 gezeigt.

In diesem Fenster sind Bedienelemente und Visualisierungshilfen sichtbar. Es ist möglich, Schritt für Schritt voranzugehen, zum ersten oder letzten Schritt zu springen und mithilfe eines Schiebereglers schnell vor- und zurückzugehen oder einen bestimmten Schritt auszuwählen. Im dargestellten Programmcode auf der linken

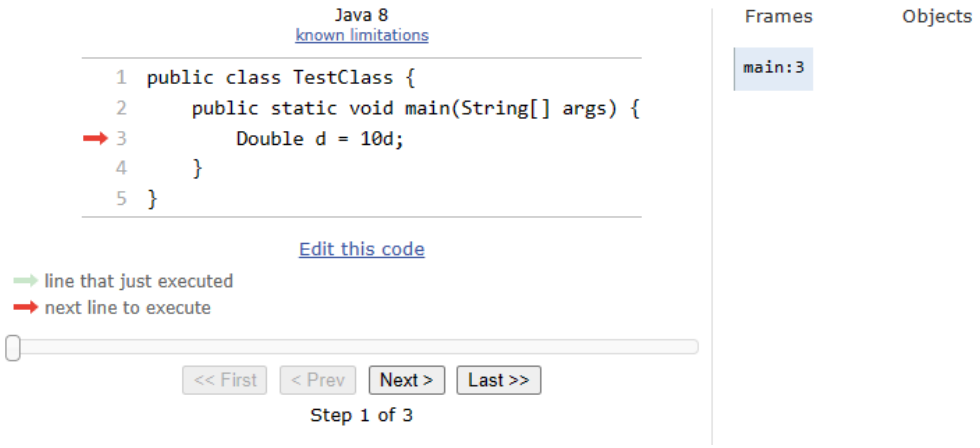


Abbildung 3.5: Visualisierung und Bedienelemente in *Python Tutor*

Seite der Visualisierung sind die aktuelle Zeile und die nächste auszuführende Zeile durch unterschiedlich farbige Pfeile hervorgehoben.

Ein weiterer Vorteil ist die Möglichkeit, eine URL[2] zu teilen. Nachdem ein Programmcode visualisiert wurde, kann eine URL generiert werden. Diese URL enthält den Programmcode als Query-String, was dazu führen kann, dass die URL sehr lang wird. Jedoch erfolgt keine Zwischenspeicherung (Caching) durch die URL. Stattdessen wird der Programmcode im Editorfenster erneut angezeigt und visualisiert. Ein Beispiel für eine solche URL wird folgend dargestellt:

https://pythontutor.com/render.html#code=public%20class%20TestClass%20%7B%0A%20%20%20%20public%20static%20void%20main%28String%5B%5D%20args%29%20%7B%0A%20%20%20%20%20%20%20%20Double%20d%20%3D%2010d%3B%0A%20%20%20%20%7D%0A%7D&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=java&rawInputLstJSON=%5B%5D&textReferences=false

In diesem Beispiel wurde der Code aus Abbildung 3.4 verwendet. Der Code ist nach *code*= in der URL erkennbar. Der Wert des Parameters *mode*= kann zwischen *editor* und *display* gewählt werden. Mit *editor* bleibt man im bereits vorgestellten Editorfenster, während *display* das *Visualize Execution* direkt ausführt.

Eine Einschränkung von *Python Tutor* ist, dass es ausschließlich im Browser genutzt werden kann und eine Internetverbindung benötigt wird. Die Nutzung ist daher von der Verfügbarkeit einer Internetverbindung abhängig. Zudem steht die An-

[2] Uniform Resource Locator

wendung nicht innerhalb der gewohnten Entwicklungsumgebung zur Verfügung, was einen Wechsel der Anwendung erforderlich macht. Dies kann den Arbeitsfluss unterbrechen und die Effizienz beeinträchtigen.

Möchte man innerhalb des Editorfensters programmieren, ergibt sich eine weitere Einschränkung. In diesem Fall bietet *Python Tutor* keine Autovervollständigung oder andere Hilfestellungen, die in modernen Entwicklungsumgebungen oder in Website Compilern wie [One Compiler 2023] üblicherweise vorhanden sind. Auch können keine Bibliotheken verwendet werden. Möchte man Teile einer externen Bibliothek verwenden, müssen diese Codeabschnitte mitübertragen werden, um eine Funktionsfähigkeit zu ermöglichen.

3.2.2 Darstellung

Die Startseite von *Python Tutor*, wie bereits in Abbildung 3.3 dargestellt, bietet eine einfache und verständliche Darstellung von Datentypen und deren Beziehungen. Zum Beispiel ist deutlich erkennbar, dass das Tupel zwei Elemente hat, die die Indizes 0 und 1 haben. Die Visualisierung verdeutlicht die Referenzen anschaulich, beispielsweise wie der Wert an Index 1 des ersten Tupels auf das zweite Tupel verweist. Zudem zeigt die Darstellung, welche *Frames* welche *Objects* im *Heap* haben und welche Werte im *Stack* gespeichert werden. Dabei wird aber eine Vereinfachung der Zahlen in Python vorgenommen und diese werden direkt im *Stack* angezeigt.

Python Tutor bietet auch die Möglichkeit, die Darstellung leicht anzupassen. Elemente können beispielsweise per Drag-and-Drop verschoben werden. Zusätzlich können Pfeile in Stil, Biegung, Länge, Breite und Faltung angepasst werden. Diese Einstellungen werden in Abbildung 3.6 gezeigt.

Jedoch gibt es Fälle, wie zum Beispiel bei Wrapper-Typen in Java, in denen die Darstellung zu vereinfacht ist. Ein solcher Fall wird durch den folgenden Java-Code in Listing 3.3 illustriert.

```java
class TestClass {
    public static void main(String[] args) {
        Double d = 10d;
    }
}
```

Listing 3.3: Wrapper *Double* in Java

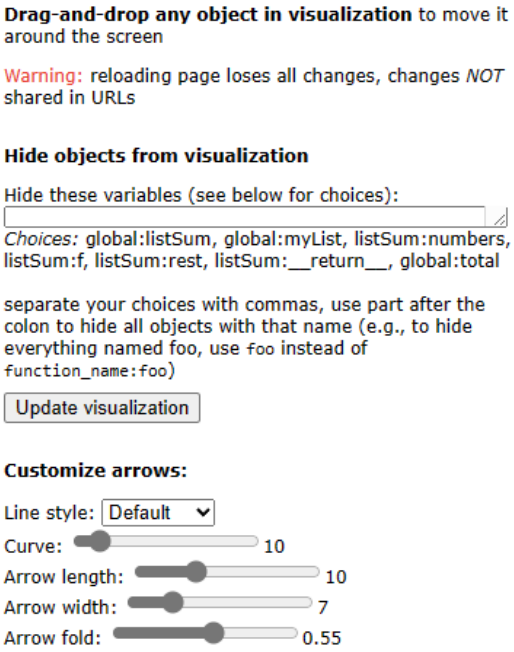


Abbildung 3.6: Möglichkeiten der Veränderung der Visualisierung in *Python Tutor*

Der gezeigte Java-Code erstellt ein *Double*-Objekt mit dem Wert 10, wie in Zeile 3 zu sehen ist. Das Ergebnis dieser Darstellung wird in Abbildung 3.7 präsentiert.

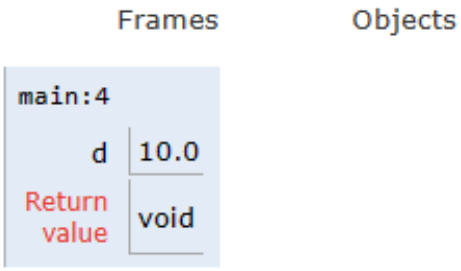


Abbildung 3.7: Vereinfachte Darstellung eines *Doubles*

Es könnte irreführend sein anzunehmen, dass ein *Double* im Stack gespeichert wird, anstatt zu erkennen, dass es sich um ein eigenes Objekt handelt. Um solche Missverständnisse zu vermeiden, wäre eine Darstellung wie in Abbildung 3.8 für Programmieranfänger hilfreicher.

Diese Darstellung zeigt eine eindeutige Referenz vom *Stack* auf den *Heap* und verdeutlicht den klaren Unterschied zwischen einem primitiven *double* und einem Wrapper-Objekt auf den ersten Blick. Dies kann dazu beitragen, Missverständnisse zu vermeiden und das Verständnis für solche Konzepte zu verbessern.

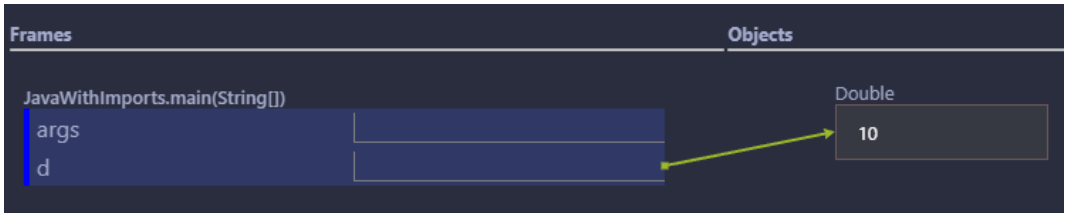


Abbildung 3.8: Korrekte Darstellung eines *Doubles*

3.2.3 Effizienz

In Bezug auf die Effizienz weist *Python Tutor* bestimmte Einschränkungen auf, die die Visualisierung von Code betreffen. Wie in den von [Python Tutor 2023b] beschrieben *unsupported features*, können nur Codeabschnitte die

- keinen zu langen Programmcode haben
- nicht zu viele Schritte oder eine zu lange Laufzeit haben
- nicht zu viele Variablen oder Objekte definieren

visualisiert werden. Für die Länge des Programmcodes wird die Größe einer Präsentationsfolie oder die Menge, die auf einer Tafel dargestellt werden kann, genannt. Es werden auch längere Codeabschnitte unterstützt, jedoch nicht empfohlen. Bei der Anzahl der Schritte wird ab einem Wert von etwa 100 von zu viel gesprochen. Auch wenn die Laufzeit zehn Sekunden überschreitet, ist dies ein Indiz, dass der Code zu lange ist. Aufgrund dieser Einschränkungen kann eine schnelle Visualisierung gewährleistet werden. Für das Beispiel der Startseite aus Abbildung 3.3 wird zum Beispiel direkt nach dem Drücken von *Visuzalize Execution* die Visualisierung überliefert. Allerdings variiert die Geschwindigkeit der Visualisierung je nach Programmiersprache. Die Java-Version von *Python Tutor* benötigt im Vergleich zu Python deutlich länger für die Visualisierung, wie anhand der Beispiele im Code-Listing 3.4 für Python und 3.5 für Java gezeigt wird.

```
1    float = 1.0
```

Listing 3.4: Float in Python

```
1  public class TestClass {
2      public static void main(String[] args) {
3              float f = 1f;
4      }
5  }
```

Listing 3.5: Float in Java

Um die Geschwindigkeit der Ausführung zu vergleichen, wurden mehrere Durchläufe durchgeführt und die Zeit vom Drücken des *Visuzalize Execution*-Buttons bis

zum vollständigen Laden der Visualisierungsseite gemessen. Die Ergebnisse sind in der folgenden Tabelle 3.2 dargestellt.

Durchlauf	Python	Java
1	0.67 s	3.28 s
2	0.69 s	2.73 s
3	0.61 s	2.83 s
4	0.62 s	2.74 s
5	0.66 s	2.92 s
Durchschnitt	0.65 s	2.90 s

Tabelle 3.2: Vergleich der Ausführungszeit in *Python Tutor* von Java und Python anhand einer einfachen float-Zuweisung

Wie aus der Tabelle hervorgeht, ist die Visualisierung in Python im Durchschnitt um mehr als das Vierfache schneller als in Java. Dies kann teilweise auf die externe Java-Erweiterung zurückgeführt werden, die nicht mehr aktiv unterstützt wird und von einem externen Entwickler bereitgestellt wird.

Es ist wichtig zu beachten, dass die Effizienz von *Python Tutor* von verschiedenen Faktoren abhängt, einschließlich der Programmiersprache und der Komplexität des Codes. Bei der Evaluierung von *Python Tutor* ist dies daher im Gesamten zu berücksichtigen.

3.2.4 Unterstützung verschiedener Programmiersprachen

Anders als der Name *Python Tutor* vermuten lässt, unterstützt diese Plattform mehrere Sprachen neben Python. Wie in Abbildung 3.3 dargestellt, umfasst *Python Tutor* die Programmiersprachen Python, JavaScript, C, C++ sowie Java. Dadurch werden die wesentlichen Programmiersprachen für Einsteiger abgedeckt.

3.2.5 Unterstützung unterschiedlicher Versionen von Programmiersprachen

Die Unterstützung der verschiedenen Programmiersprachen durch *Python Tutor* ist von den jeweiligen Sprachversionen abhängig. In Tabelle 3.3 sind die unterstützten Programmiersprachen mit ihren jeweiligen Versionen aufgeführt.

Programmiersprache	Unterstützte Sprachversion
C	gcc 9.3, C17 + GNU
C++	g++ 9.3, C++20 + GNU
Java	8
JavaScript	ES6
Python	3.6

Tabelle 3.3: Unterstütze Sprachversionen von *Python Tutor*, herausgearbeitet aus [Python Tutor 2023a]

Besonders hervorzuheben ist, dass die unterstützte Python-Version 3.6 bereits seit dem 23.12.2021 nicht mehr aktuell ist, wie im *Status of Python Versions* der [Python Software Foundation 2023] beschrieben wird. Dies wurde bereits in Kapitel 2.5 mit Abbildung 2.4 hervorgehoben. Konkret bedeutet dies, dass keine Features aus neueren oder älteren Versionen genutzt werden können. Dieses Beispiel verdeutlicht, wie die Anforderung der Unterstützung unterschiedlicher Versionen von Programmiersprachen nicht erfüllt wird.

3.2.6 Zwischenfazit Python Tutor

Zusammenfassend lässt sich feststellen, dass *Python Tutor* einige der Anforderungen erfüllt. Handhabung, Bedienbarkeit, Effizienz und Unterstützung mehrerer Programmiersprachen sind weitgehend gegeben. Allerdings fehlt es *Python Tutor* an Unterstützung unterschiedlicher Versionen von Programmiersprachen, wie im Abschnitt *Unterstützung unterschiedlicher Versionen von Programmiersprachen* festzustellen ist. Es wird lediglich eine veraltete Python-Version unterstützt. Zudem entspricht die Effizienz und Korrektheit nicht in jeder Sprache den idealen Anforderungen, insbesondere im Hinblick auf die Zielgruppe. Vor allem aber auch, dass externe Bibliotheken nicht unterstützt werden, ist ein großer Nachteil. Aus diesen Gründen kann *Python Tutor* nicht als ideale Plattform betrachtet werden. Daher bleibt die Entwicklung eines eigenen Frameworks nach wie vor notwendig.

3.3 Fazit des Vergleichs existierender Lösungen

Nach der Untersuchung von *Debug Visualizer* und *Python Tutor* ist festzustellen, dass keine der existierenden Lösungen den Anforderungen gerecht wird, welche in

Kapitel 2 formuliert wurden. Daher ist es notwendig, ein eigenes Framework zu entwickeln, das alle diese Anforderungen erfüllt.

In der folgenden Tabelle 3.4 ist übersichtlich nochmals das Ergebnis der Untersuchung der Lösungen dargestellt.

Anforderung	Debug Visualizer	Python Tutor
Handhabung und Bedienbarkeit	nein	bedingt
Darstellung	bedingt	bedingt
Effizienz	bedingt	ja
Unterstützung verschiedener Programmiersprachen	ja	ja
Unterstützung unterschiedlicher Versionen von Programmiersprachen	ja	nein

Tabelle 3.4: Erfüllung der Anforderung von existierender Lösungen

4 Grundlagen

Wie in dem Fazit des Vergleichs existierender Lösungen im Abschnitt 3.3 dargelegt, gibt es keine Lösung, die den in Kapitel 2 formulierten Anforderungen gerecht wird. Aus diesem Grund wurde bereits Vorarbeit für ein Framework geleistet, das genau diesen Anforderungen entsprechen soll. Die Bachelorthesis mit dem Titel *Visualisierung von Python Programmen in der Entwicklungsumgebung Visual Studio Code* von [Velten 2023] wird im Folgenden genauer betrachtet. Dabei werden die verwendeten Werkzeuge näher erläutert und diskutiert. Anschließend wird der aktuelle Stand des entwickelten Frameworks analysiert und hinsichtlich der Anforderungen aus Kapitel 2 untersucht.

4.1 Plattform Visual Studio Code

Die Entwicklung des Frameworks erfolgte für die Entwicklungsumgebung VSC, wie bereits im Titel der Bachelorthesis von [Velten 2023] erwähnt. Die Wahl von VSC als Entwicklungsplattform wurde durch seine einfache Erweiterbarkeit, die kostenlose Open Source Nutzung und die Plattformunabhängigkeit begründet. Dabei werden Erweiterungen mit TypeScript implementiert. Um die Relevanz dieser Entscheidung zu untermauern, werden verschiedene bekannte Entwicklungsumgebungen betrachtet. Diese werden anhand der Häufigkeit der Suchanfragen in Google Trends verglichen. Die Abbildung 4.1 zeigt diesen Vergleich.

Der Betrachtungszeitraum erstreckt sich über die letzten drei Jahre. Deutlich wird, dass VSC und Visual Studio die meisten Suchanfragen verzeichnen, was die Relevanz von Microsoft-Produkten hervorhebt. Besonders auffällig ist, dass VSC seit Mitte 2021 kontinuierlich die meisten Suchanfragen verzeichnet, was seine Relevanz im Vergleich zu Eclipse, IntelliJ IDEA, Notepad++ und Visual Studio betont.

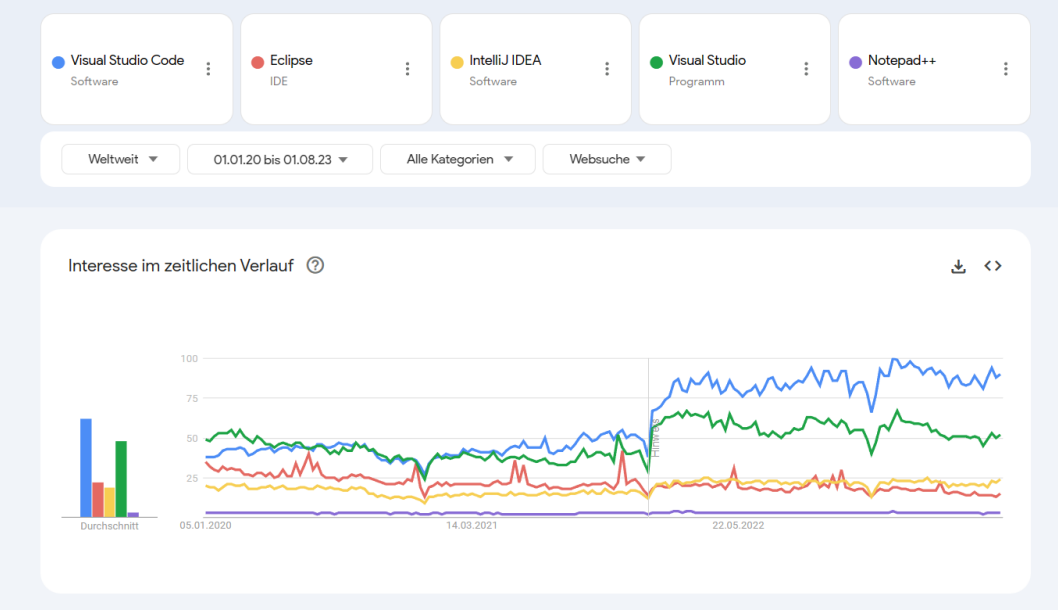


Abbildung 4.1: Suchanfragen in Google für Visual Studio Code, Eclipse, IntelliJ IDEA, Visual Studio und Notepad++ im Vergleich aus [Google 2023]

Aufgrund dieser Erkenntnisse ist es durchaus sinnvoll, die Entwicklung des Frameworks für die Visualisierung von Programmabläufen in VSC fortzusetzen.

4.2 Debug Adapter Protocol

Das Framework kommuniziert über das DAP[1], um auf die Daten des Debuggers zuzugreifen. Da viele Programmiersprachen dieses Protokoll unterstützen, wie von [Microsoft 2021] in der Dokumentation des DAP angegeben wird, ermöglicht dies eine Anbindung an verschiedene Debugger. [Velten 2023, Seite 22] beschreibt in seiner Bachelorthesis, dass dies sehr einfach durch den Austausch des anzusprechenden Debug-Adapters erreicht werden kann. Die Abbildung 4.2 zeigt das Zusammenspiel des Debug Adapter Protocols, des Debug Adapters und des sprachspezifischen Debuggers.

Die Bachelorthesis hebt hervor, dass das DAP ausreichend Daten für die Visualisierung liefert. Daher ist gewährleistet, dass das DAP auch in einem generischen Ansatz zur Unterstützung mehrerer Programmiersprachen durch das Framework verwendet werden kann. Zur weiteren Veranschaulichung wird ein Beispiel aus Java, in

[1] Debugger Adapter Protocol

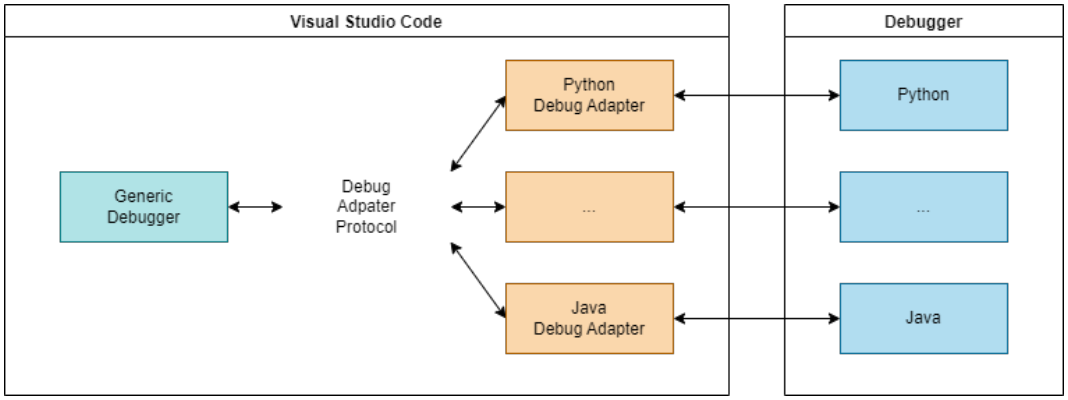


Abbildung 4.2: Debug Adpater Protocol in Visual Studio Code

dem Informationen zu einem *int*-Array `int[] intArr = {1, 2, 3};` abgefragt wurden, näher betrachtet.

```
{
    evaluateName: 'intArr',
    name: 'intArr',
    type: 'int[]',
    value: 'int[3]@10',
    variablesReference: 5,
    namedVariables: 0,
    indexedVariables: 3
}
```

Listing 4.1: Beispiel des DAP für *int*-Array in Java

```
{
    evaluateName: 'intArr[0]',
    name: '0',
    type: 'int',
    value: '1',
    variablesReference: 0,
    namedVariables: 0,
    indexedVariables: 0
}
```

Listing 4.2: Auflösung des Beispiels für *int*-Array in Java

Generell wird lediglich eine Variable von dem DAP geliefert. Diese umfasst in diesem Java-Beispiel sieben Werte. Diese werden im Folgenden genauer erläutert:

variablesReference Einer der wichtigsten Parameter ist die *variablesReference*. Einige komplexe Strukturobjekte werden nicht direkt mit ihren Containern (*Stack Frames*, *Scopes*, *intVariablen*) zurückgegeben, sondern müssen mit separaten Anfragen auf der Basis von *variablesReference* abgerufen werden. Eine *variablesReference* wird durch eine ganze Zahl definiert, welche vom Debug-Adapter zugewiesen wird und nichts mit den Referenzen innerhalb der Programmiersprache zu tun hat. Bei Variablen, die keine weiteren Referenzen haben, ist dieser Wert `0`, sonst eine positive ganze Zahl.

evaluateName Dieser Wert beschreibt den aktuellen auswertbaren Namen. Dieser ändert sich je nach Auflösungstiefe. In Listing 4.2 ist zu erkennen, dass der Name

nach der Auflösung mit `[0]` ergänzt wird. Da der Beispiel-Array drei Elemente speichert, gibt es bei der Auflösung drei Objekte. Das im Listing dargestellte ist das Objekt für Index 0. Bei Betrachtung von Index 1 ist der *evaluateName* `[1]` und so weiter. Mit diesem Parameter kann eine *evaluate*-Anfrage getätigt werden, um den Wert der Variablen abzurufen.

name Mit *name* ist der Name der Variable gemeint. Wenn also die Variable `intArr` betrachtet wird, ist dies auch der *name*. Wenn jedoch die Variable aufgelöst wird, ändert sich dieser in einen internen Wert, wie zum Beispiel `0`, und bietet ohne Kontext keinen weiteren Mehrwert.

type Der *type* beschreibt den Typ der Variable. Zum Beispiel ist bei einem *int*-Array `int[]` der Typ oder bei einer *int*-Variable `int`.

value Um den Wert der Variable zu erhalten, wird der Parameter *value* betrachtet. Dies kann ein mehrzeiliger Text sein, zum Beispiel bei einer Funktion der Funktionskörper, oder wie in diesem Beispiel eine Angabe über den *int*-Array. Mit `int[3]@10` wird dargestellt, dass der *int*-Array drei Elemente hat und die in Java verwendete Referenz `10` ist.

namedVariables Die Anzahl der benannten Variablen in diesem Scope wird mit *namedVariables* angegeben.

indexedVariables In diesem Beispiel hat der *int*-Array drei Elemente, weshalb *indexedVariables* drei angibt. Das bedeutet, dass die Anzahl der indizierten Variablen in diesem Scope mit diesem Parameter angegeben wird.

Da ein Debugger im Allgemeinen verwendet wird, ermöglicht er auch die Verwendung beliebiger Versionen der unterstützten Programmiersprachen, solange der Debugger in seiner Funktion und seinem Datenformat nicht verändert wird.

4.3 Vorarbeit

Nachdem ein grundlegendes Verständnis über die Werkzeuge und die Umgebung der Framework-Implementierung vorliegt, werden in diesem Abschnitt die Funktionen und Einschränkungen des Frameworks analysiert.

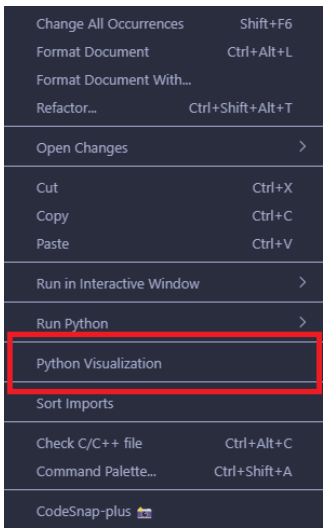
4.3.1 Bedienbarkeit, Handhabung und Darstellung

Am Beispiel des Codeausschnitts in Listing 4.3 wird das Framework weiter analysiert.

```
1  nestedTuple = (1, (2, None))
2  sampleDict = {"one": 1, "two": 2, "three": 3}
3  x = sampleDict.get("one")
```

Listing 4.3: Beispielprogramm zur Untersuchung des Frameworks

Die Bedienung des Frameworks innerhalb von VSC ist durchdacht und zielführend gestaltet. Die Ausführung der Visualisierung erfolgt durch einen Klick auf den Button *Python Visualization* oder durch das Kontextmenü. Eine Darstellung dieses Buttons und des Kontextmenüs innerhalb von VSC ist in der folgenden Abbildung 4.3 zu sehen.



(a) Kontextmenü

(b) Ausführungsbutton

Abbildung 4.3: Ausführungsmöglichkeiten für Visualisierung des Frameworks

Python Visualization, rot umrandet, wird nur angezeigt, wenn eine Datei im unterstützten Dateiformat geöffnet ist, was in diesem Fall auf Python-Dateien mit der

Erweiterung *.py* zutrifft. Nach der Ausführung der Visualisierung wird eine Side-by-Side-Ansicht angezeigt, wie in Abbildung 4.4 dargestellt.

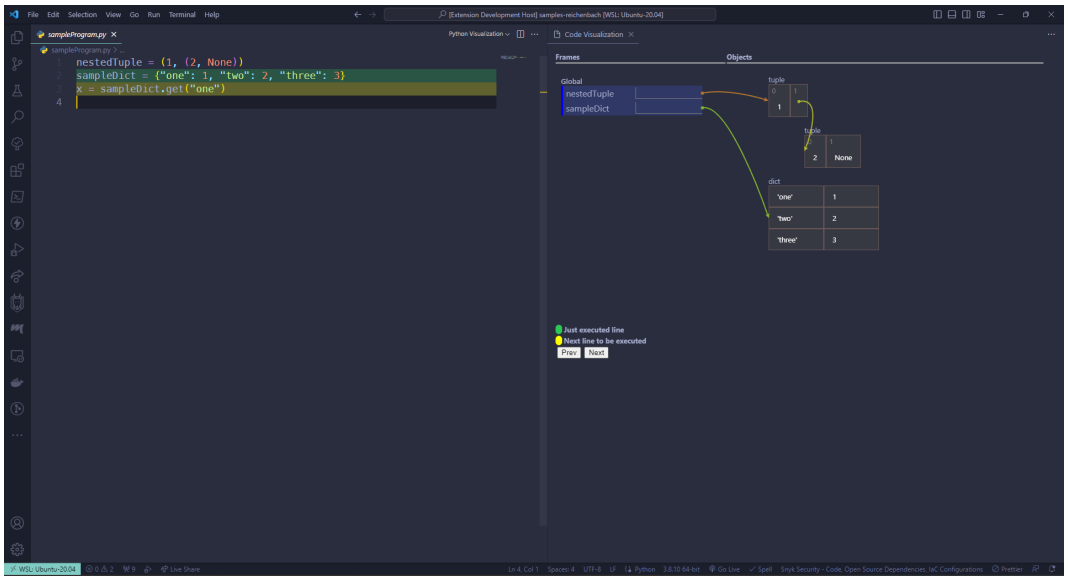


Abbildung 4.4: Anzeige in Visual Studio Code nach Ausführung der Visualisierung von Python Visualization

Die Side-by-Side-Ansicht ermöglicht eine einfache Darstellung der Visualisierung innerhalb von VSC. Die Darstellung der Datentypen erfolgt übersichtlich und verständlich. Das verschachtelte Tupel aus Zeile 1 des Codeausschnitts aus Listing 4.3 wird korrekt mit seinen Referenzen auf ein weiteres Tupel dargestellt. Das Dictionary aus Zeile 2 wird ebenfalls übersichtlich mit Key-Value-Paaren angezeigt. Unterhalb der Darstellung befindet sich eine Legende, die die Markierungen innerhalb des Programmcodes erläutert. Die aktuelle Zeile wird grün markiert, während die Zeile des nächsten Schrittes gelb markiert ist. Mithilfe der Schaltflächen *Next* und *Prev* unter der Legende kann zwischen den Schritten navigiert werden, was eine flexible und benutzerfreundliche Visualisierung ermöglicht. Dadurch fällt auch die manuelle Steuerung des Debuggers weg.

4.3.2 Unterstützung verschiedener Programmiersprachen

Wie bereits der Titel der Bachelorthesis nahelegt, wird ausschließlich die Programmiersprache Python unterstützt. Es ist jedoch nicht klar ersichtlich, wie erweiterbar die Anwendung basierend auf ihrer Architektur ist. Der Code wurde in grobe Ab-

schnitte unterteilt, was bedeutet, dass es einen erheblichen Spielraum für mögliche Erweiterungen gibt.

4.3.3 Komplexe Datentypen

Die Darstellung komplexerer Datentypen stößt auf Einschränkungen. Ein Beispiel dafür ist in Listing 4.4 dargestellt.

```
1   (1, (2, (3, None)))
2   (1, (2, (3, (4, None))))
```

Listing 4.4: Zwei verschachtelte Tupel

Obwohl das verschachtelte Tupel in der ersten Zeile innerhalb der Erweiterung dargestellt wird, ist dies nur bis zur zweiten Ebene möglich. Die Visualisierung wird in Abbildung 4.5 veranschaulicht.

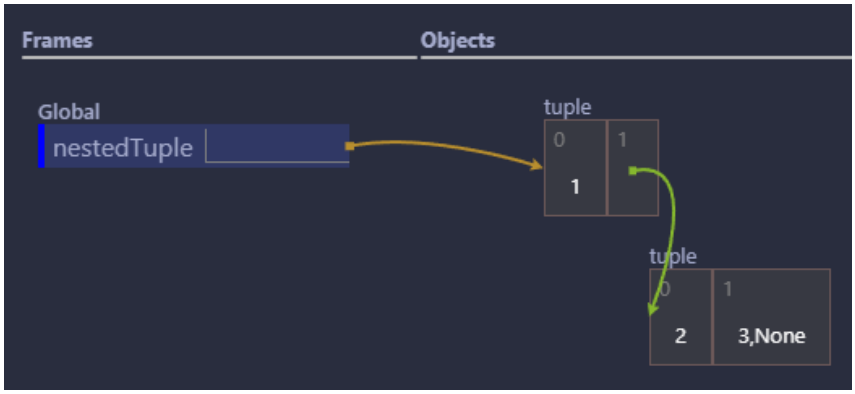


Abbildung 4.5: Beispiel eines verschachtelten Tupel

Man erkennt, dass das zweite Tupel, `(2, (3, None))`, den Wert des ersten Index als eigenständigen Wert speichert, anstatt wie beim ersten Tupel `(1, (...))` eine Referenz auf den ersten Index zu verwenden. Bei Betrachtung des Beispiels in Zeile 2 bricht die Ausführung ab und es kann keine Visualisierung erstellt werden. Das Framework unterstützt also generell nur Rekursionen bis zur ersten Ebene, welche in der gezeigten Abbildung dargestellt ist.

4.3.4 Effizienz

Grundsätzlich lässt sich feststellen, dass die Ausführungszeit bei sehr kurzen Codeabschnitten durchaus akzeptabel ist. Jedoch wird bei längeren Codeabschnitten, insbesondere solchen, die Daten aus vorherigen Zeilen beibehalten, die Ausführungszeit für jeden Schritt zunehmend langsamer. Dies ist auch darauf zurückzuführen, dass selbst bei einer unveränderten Variablen keine Informationen vom Debugger geliefert werden, die einen schnelleren Durchlauf ermöglichen würden. Jeder Schritt muss also vollständig alle Variablen auflösen.

4.3.5 User Experience

Um das Framework zur Visualisierung von Programmcode zu nutzen, muss VSC einen geöffneten Ordner haben. Es ist nicht möglich, das Framework direkt für einzelne Dateien zu verwenden.

Die Ausführung des Frameworks verhält sich ebenfalls unerwartet. Wenn die Visualisierung durch Klicken auf *Python Visualization* gestartet wird, wie in Abbildung 4.3 gezeigt, wird die aktuell geöffnete Datei geschlossen. Ein neuer Tab wird geöffnet, der den gleichen Code enthält, aber mit einem *pass* am Ende erweitert ist. Anschließend wird dieser temporäre Tab wieder geschlossen und der ursprüngliche Tab mit der originalen Datei wird wieder angezeigt. Dies führt zu Unklarheit darüber, was genau in diesem Moment geschieht. Bei einem Fehler, wie dem zu tief verschachtelten Tupel aus Abschnitt 4.3.3, bleibt der Debugger stehen und muss manuell durch einen beliebigen Schritt beendet werden. Erst danach wird wieder die originale Datei angezeigt.

5 Architektur

Um ein umfassendes Verständnis für die internen Zusammenhänge und Abläufe innerhalb des Frameworks zu erlangen, konzentriert sich dieses Kapitel auf die Architektur des Frameworks. Der Abschnitt beginnt mit einer grundlegenden und vereinfachten Darstellung, um ein erstes Verständnis zu vermitteln, bevor im Detail auf die einzelnen Komponenten eingegangen wird. Dieses Kapitel dient als Grundlage, um den Aufbau und die Funktionsweise des Frameworks besser nachvollziehen zu können. Dabei wird bereits auf die während dieser Thesis neu strukturierte Architektur eingegangen.

5.1 Einfache Darstellung

Die Architektur des Frameworks umfasst grundlegend zwei Komponenten: VSC und den Debugger. Diese sind in der Abbildung 5.1 dargestellt. Dabei sind die selber implementierten Komponenten orange hervorgehoben.

Der Debugger wird von der jeweiligen Programmiersprache bereitgestellt und kann über das DAP angesprochen werden. Innerhalb von VSC existiert eine eigens entwickelte Erweiterung (Extension), die sowohl ein *Backend* als auch ein *Frontend* umfasst. Diese klassische *Frontend-Backend*-Architektur ermöglicht die Kommunikation zwischen dem *Backend* und *Frontend*.

Es ist jedoch zu beachten, dass die Bezeichnungen *Backend* und *Frontend* in diesem Kontext nicht im herkömmlichen Sinne verwendet werden. Das *Backend* ist hauptsächlich für die Erzeugung eines *Trace* verantwortlich, der alle erforderlichen Daten für die Visualisierung enthält. Es kommuniziert mit einem *Debug Adapter Tracker*. Der *Debug Adapter Tracker* dient dazu, die Daten vom Debugger zu empfangen und an diesen weiterzuleiten.

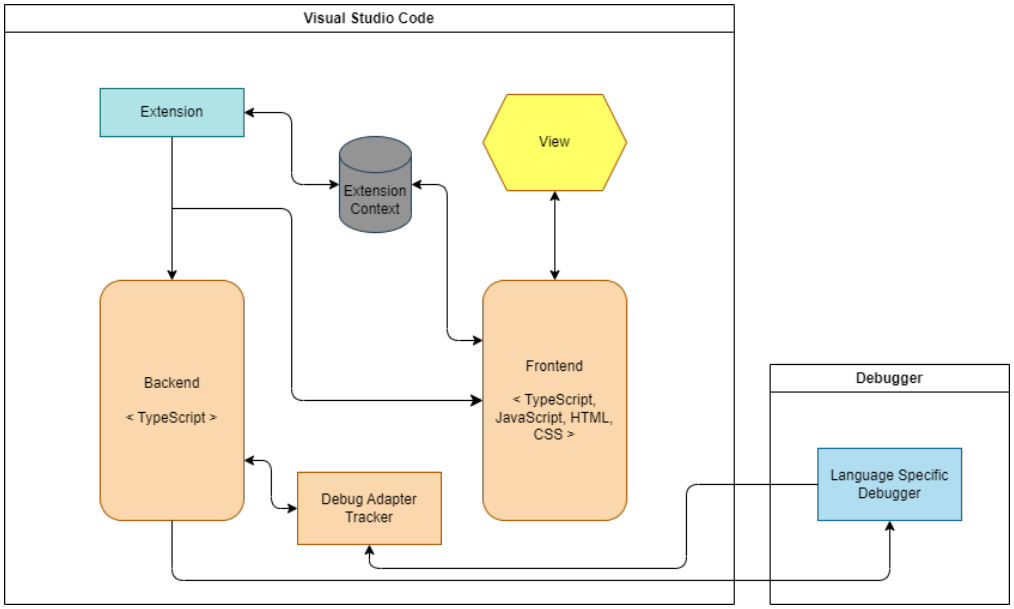


Abbildung 5.1: Einfache Darstellung der Framework-Architektur

Das *Frontend* greift auf die Benutzeroberfläche von VSC zu und wandelt das im *Backend* generierten *Trace* in HTML[1] um. Dieser Prozess beinhaltet auch die Anwendung von CSS[2], um die Darstellung zu gestalten. Die *Backend*- und *Frontend*-Komponenten interagieren nicht direkt miteinander. Das *Backend* speichert das *Trace* im Kontext der Extension ab und weist ihm eine eindeutige ID[3] zu. Diese ID kann später von dem *Frontend* abgerufen werden, um auf das gespeicherten *Trace* zuzugreifen.

Diese zentrale Speicherung ermöglicht auch das Durchführen von Tests, wie im späteren Kapitel *Testen von Visual Studio Code-Erweiterungen* näher erläutert wird.

5.2 Detaillierte Darstellung

Nachdem die Zusammenhänge der einzelnen Komponenten erläutert wurden, erfolgt nun eine detaillierte Betrachtung ihrer Funktionsweise. Dabei wird die Abbildung 5.2 als Referenz genommen.

[1] Hypertext Markup Language
[2] Cascading Style Sheets
[3] Identifikationsnummern

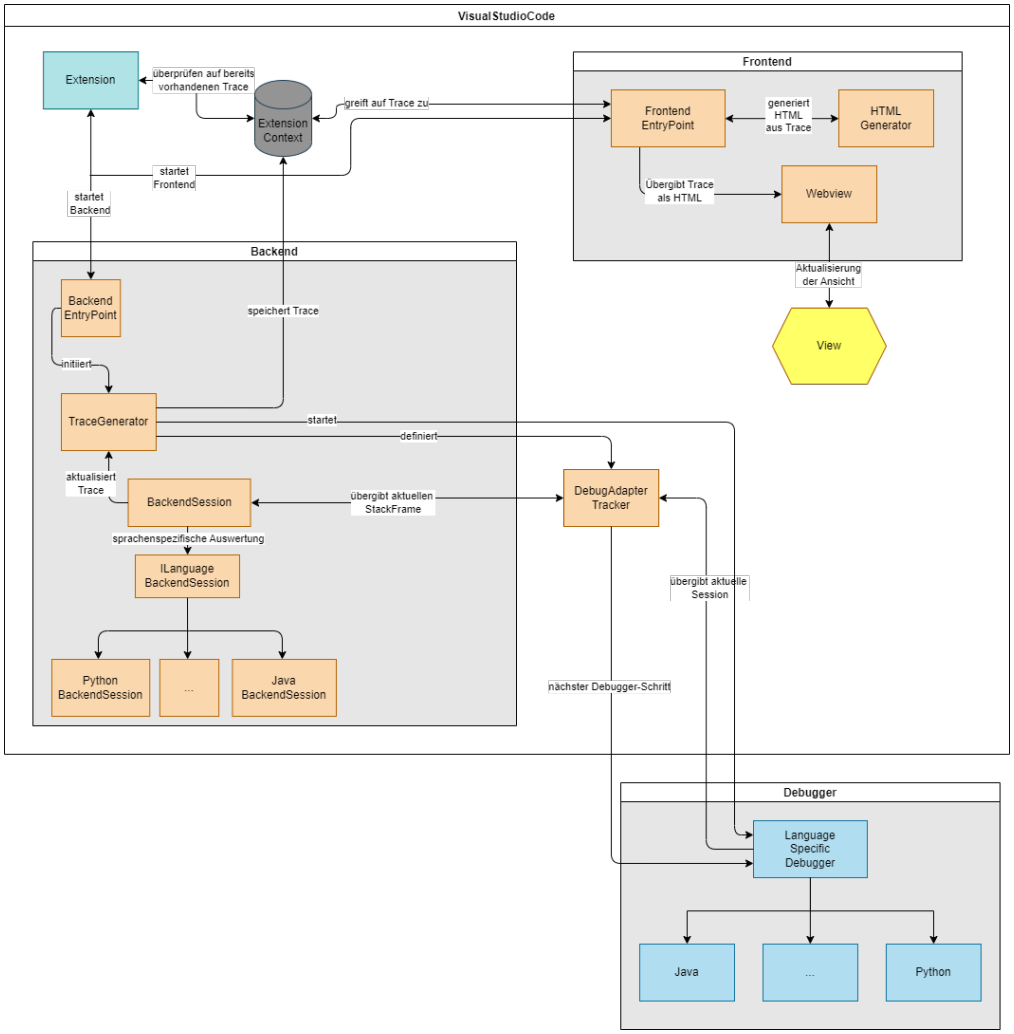


Abbildung 5.2: Detaillierte Darstellung der Architektur des Frameworks

5.2.1 Extension

Die Erweiterung kann so konfiguriert werden, dass sie nur für bestimmte Dateitypen gestartet werden kann. Dadurch lässt sich eine Art Whitelisting realisieren, um das Starten für nicht unterstützte Sprachen zu verhindern. Sobald die Extension gestartet wird, überprüft sie, ob für die aktuelle Datei bereits ein *Trace* vorhanden ist. Hierbei wird ein Hashwert aus dem Programmcode innerhalb der Datei generiert und auf Gleichheit überprüft. Falls kein passendes *Trace* gefunden wird, wird das *Backend* aufgerufen.

Extension Context Der *Extension Context* dient der spezifischen Speicherung von Daten innerhalb der Erweiterung. Dies umfasst auch das Speichern von Dateien,

wodurch Caching ermöglicht wird. Zudem gewährleistet der *Extension Context* die Unabhängigkeit der Komponenten, da diese nicht untereinander Daten abfragen müssen, sondern auf den Kontext zugreifen können.

5.2.2 Backend

Wenn die Extension kein passendes *Trace* findet, wird das *Backend* aktiviert. Es besteht aus zwei Hauptkomponenten: Dem *Trace Generator* und der *BackendSession*.

Trace Das *Trace* speichert alle erfassten Zustände aus den Daten des Debuggers in einem JSON-Format. Um den Inhalt des JSON näher zu erläutern, wird das Python-Beispiel `simpleList = [1, 2, 3, 4]` näher betrachtet. Das für diesen Code erzeugte *Trace* ist in Listing 5.1 zu sehen.

Im Allgemeinen wird für jeden Schritt ein Objekt in Form eines Arrays erstellt. Jedes Objekt setzt sich aus vier Elementen zusammen:

- *line* beschreibt die ausgeführte Zeile.
- *filePath* gibt den vollständige Pfad zur aktuell ausgeführten Datei an.
- *stack* repräsentiert den aktuellen Zustand des *Stacks*.
- *heap* repräsentiert den aktuellen Zustand des *Heaps*.

Dieses Beispiel zeigt 2 Schritte. Der erste Schritt erstreckt sich von Zeile 2 bis Zeile 7. In diesem Schritt ist der aktuelle Zustand ein leerer *Stack* und *Heap*. Um den tatsächlichen Wert aus Zeile 1 lesen zu können, wird *pass* am Ende der Datei hinzugefügt. Weitere Details zur Notwendigkeit von *pass* sind in [Velten 2023, Seite 34] zu finden. Aus diesem Grund gibt es ein zweites Objekt von Zeile 8 bis Zeile 45.

Dieses zweite Objekt stellt die Beispielliste dar. Es ist zu erkennen, dass der *Stack* mit Hilfe von *frameName* und *locals* definiert wird. Innerhalb von *locals* werden die im *Stack* vorhandenen Variablen gespeichert. In diesem Fall ist dies die Variable *simpleList*. Da es sich um eine Liste handelt, die Referenzen auf den *Heap* enthält, werden die einzelnen gespeicherten Werte nicht direkt angezeigt, sondern mittels *ref* auf *7*. Betrachtet man den *Heap*, so sieht man, dass in Zeile 23 auch eine *7* vorkommt. Unter diesem Verweis wird gespeichert, dass eine *list* mit den zugehörigen

Werten *values* existiert. Unter *values* sind dann die Einträge der Liste *1*, *2*, *3* und *4* als *int* zu sehen.

```
[
  {
      "line": 1,
      "filePath": "<path>/sampleProgram.py",
      "stack": [],
      "heap": {}
  },
  {
      "line": 2,
      "filePath": "<path>/sampleProgram.py",
      "stack": [
          {
              "frameName": "<module>",
              "locals": {
                  "simpleList": {
                      "type": "ref",
                      "value": 7
                  }
              }
          }
      ],
      "heap": {
          "7": {
              "type": "list",
              "value": [
                  {
                      "type": "int",
                      "value": 1
                  },
                  {
                      "type": "int",
                      "value": 2
                  },
                  {
                      "type": "int",
                      "value": 3
                  },
                  {
                      "type": "int",
                      "value": 4
                  }
              ]
          }
      }
  }
]
```

Listing 5.1: Beispiel eines Trace

Trace Generator Der *Trace Generator* ist dafür zuständig, ein *Trace* zu erstellen. Er bereitet die Ausführung vor, erzeugt die notwendigen Komponenten und stellt eine Verbindung zum Debugger her. Zunächst wird, falls es sich bei der Programmiersprache um Python handelt, eine temporäre Datei mit einem *pass*-Schlüsselwort am Ende der Datei erzeugt. Dies ist in Python erforderlich, da der Debugger andernfalls keine Daten für den letzten Schritt liefert, sondern direkt endet. Anschließend wird ein *Debug Adapter Tracker* erstellt, der Lesezugriff auf die Kommunikation zwischen dem Editor und einem Debug-Adapter ermöglicht. Durch die Implementierung der Funktion `async onDidSendMessage(message)` wird dies realisiert. Diese Funktion unterbricht die Kommunikation und enthält die wichtigen Informationen im Parameter `message`. Danach kann der Debugger gestartet werden. Hierfür wird eine Konfiguration benötigt. Das Interface zu dieser Konfiguration ist im Listing 5.2 dargestellt. Diese Konfiguration legt die Parameter fest, die für die Debug-Session benötigt werden.

```
interface DebugConfiguration {
    type: string;

    name: string;

    request: string;

    [key: string]: any;
}
```

Listing 5.2: Interface einer *DebugConfiguration*

```
{
    name: 'Debugging File',
    type: language,
    request: 'launch',
    program: file?.fsPath ?? '${file}',
    console: 'integratedTerminal',
    stopOnEntry: true
}
```

Listing 5.3: Generische Konfiguration einer *DebugConfiguration*

Entsprechend der in Listing 5.3 gezeigten Konfiguration des Frameworks werden die einzelnen Parameter im Folgenden genauer erläutert, wie von [Microsoft 2023b] beschrieben.

- *type*: Dies ist der Typ der Debug-Session, der angibt, um welche Art von Debugger es sich handelt. Beispiele sind `java` oder `python`, abhängig von der zu untersuchenden Programmiersprache.
- *name*: Hierbei handelt es sich um eine selbst gewählte Bezeichnung für die Konfiguration. Sie dient dazu, die Konfigurationen innerhalb der Entwicklungsumgebung zu identifizieren.
- *request*: Mit *request* wird die auszuführende Aktion beschrieben. Da hier die Debug-Session gestartet werden soll, wird der Wert `launch` angegeben.

Neben den drei Hauptparametern ermöglicht das Interface zusätzliche Parameter, welche durch `[key: string]: any` dargestellt sind. Die zusätzlich verwendeten Parameter werden folgend erläutert.

- *program*: Mit *program* wird die auszuführende Datei angegeben, also die Datei, die visualisiert werden soll.
- *console*: Durch *console* wird festgelegt, ob das integrierte Terminal verwendet werden soll.
- *stopOnEntry*: Dieser Parameter gibt an, ob der Debugger nach dem Starten der Debug-Session anhalten soll, vergleichbar mit einem Breakpoint in der ersten Zeile.

Nach Abschluss des Debugging-Prozesses wird die Umgebung durch das Beenden der Debug-Session sowie das Löschen potentiell erzeugter temporärer Dateien bereinigt.

BackendSession Die *BackendSession* verarbeitet die empfangenen Daten in der Funktion `async onDidSendMessage(message)`. Außerdem entscheidet die *BackendSession*, welcher Schritt als nächstes ausgeführt werden soll, zum Beispiel *stepIn* oder *stepOut*. Da die Programmiersprachen wie Java und Python unterschiedlich funktionieren, wie in Abschnitt 6.2.2 beschrieben wird, erfolgt die Auswertung der Daten auf unterschiedliche Weise. Aus diesem Grund existiert eine variable Schnittstelle, die je nach Programmiersprache ausgetauscht werden kann. Das Interface ist in Listing 5.4 dargestellt.

```
1 interface ILanguageBackendSession {
2     createStackAndHeap: (
3         session: DebugSession,
4         stackFrames: Array<StackFrame>
5     ) => Promise<[Array<StackElem>, Map<Address, HeapValue>, DebuggerStep]>
6 }
```

Listing 5.4: Interface der programmiersprachenspezifischen *BackendSession*

Das Interface enthält eine notwendige Funktion namens `createStackAndHeap()`. Um weitere Variablen aufzulösen, wird die *DebugSession* benötigt. Die *stackFrames* können bereits für jede Programmiersprache definiert werden, weshalb sie hier übergeben werden können. Die Funktion gibt den *Stack*, den *Heap* und den

nächsten auszuführenden *DebuggerStep* zurück. Dies sind die Werte, welche nicht von allen Programmiersprachen auf die gleiche Weise ausgewertet werden können.

Im Allgemeinen generiert die *BackendSession* anhand der aktuellen Debug-Session und des aktuellen *stackFrame* das *Trace*. Die zurückgegebenen Daten umfassen sowohl die Informationen im *Stack* und *Heap* als auch den nächsten auszuführenden Schritt. Nach Abschluss des Prozesses speichert das *Backend* die generierten Daten im JSON-Format im *Extension Context*.

5.2.3 Frontend

Nachdem ein *Trace* für die zu untersuchende Datei mit Programmcode erstellt wurde, kann das *Frontend* aufgerufen werden. Dieses lädt den Trace aus dem *Extension Context*. Anschließend wird die Visualisierung mithilfe des *HTML Generators* erstellt. Die Anbindung an die von VSC bereitgestellte Oberfläche erfolgt über die Komponente *Webview*.

HTML Generator Der *HTML Generator* erzeugt für jeden im *Trace* gespeicherten Schritt eine entsprechende HTML-Darstellung. Hierbei werden HTML-Elemente generiert und durch die Benennung von `id` die Start- und Endpunkte für die später dargestellten Pfeile festgelegt. Die Bibliothek LeaderLine von [anseki 2023] wird verwendet, um diese Pfeile grafisch darzustellen.

Webview Innerhalb der *Webview* wird eine Verbindung zur View von VSC hergestellt. Dabei wird die Darstellung nach dem aktuell darzustellenden Schritt angepasst. So wird beispielsweise auf das Drücken von Schaltflächen wie *Next* reagiert, woraufhin die Referenzpfeile und die Darstellung entsprechend angepasst werden. Die gesamte Darstellung wird durch eine CSS-Datei angepasst.

5.2.4 View

Die View ist die von VSC bereitgestellte Oberfläche, um Daten anzuzeigen. Sie funktioniert ähnlich wie ein Browser, weil diese unter anderem die Möglichkeit bietet, die Darstellung mithilfe von JavaScript, CSS und HTML anzupassen. Zu-

sätzlich wird eine Entwicklerkonsole bereitgestellt, ähnlich der Konsole, die in den meisten Browsern über die F12-Taste aufgerufen werden kann.

6 Generisches Erweitern um neue Programmierprachen

Dieses Kapitel widmet sich dem Hinzufügen weiterer Programmiersprachen zu dem Framework. Es behandelt die empfohlenen Schritte und illustriert die Vorgehensweise anhand von Beispielen mit Java und Python.

6.1 Vorgehensweise für das Erweitern

Generell sind zwei Schritte vor der eigentlichen Implementierung erforderlich. Zunächst müssen die Datentypen kategorisiert werden und anschließend muss der Output dieser Datentypen durch den Debugger ausgewertet werden. Erst danach kann eine weitere, sprachenspezifische *BackendSession* erstellt und der Extension als unterstützte Programmiersprache hinzugefügt werden.

6.1.1 Kategorisierung von Datentypen

Die Datentypen innerhalb einer Programmiersprache werden im Rahmen dieses Frameworks in drei Kategorien unterteilt.

Referenzlose Datentypen Dies sind die normalen, primitiven aber auch referenzlosen Datentypen einer Programmiersprache. Sie werden im *Stack* gespeichert und erfordern keine explizite Visualisierung innerhalb des *Heaps*.

Gängige Datentypen Hierbei handelt es sich um die am häufigsten verwendeten Referenz-Datentypen. Dazu können Arrays, Listen, Klassen oder HashMaps gehören. Diese werden identifiziert, um eine spezifische Visualisierung zu ermöglichen.

Referenz Datentypen Diese Kategorie umfasst alle Datenstrukturen und Datentypen, die nicht zu den gängigen oder referenzlosen gehören. Sie werden einheitlich behandelt und erhalten keine spezielle Visualisierung. Stattdessen erfolgt eine standardmäßige Auswertung.

6.1.2 Debugger Output evaluieren

Nach der Kategorisierung der Datentypen einer Programmiersprache können diese mithilfe des Debuggers genauer untersucht werden. Eine einfache Methode besteht darin, ein Beispielprogramm zu erstellen, das genau diesen Datentyp verwendet, und anschließend die Extension im Debug-Modus auszuführen. Dabei werden die Daten, die der Debugger an den *Debug Adapter Tracker* übermittelt, analysiert. Dies ermöglicht Einblicke in die Struktur und den Inhalt der Daten, die zur Visualisierung benötigt werden.

6.2 Veranschaulichung von Java und Python

Im Folgenden werden die aus den vorherigen Abschnitten erarbeiteten Schritten an der Programmiersprache Java und Python verdeutlicht.

6.2.1 Kategorisierung von Datentypen

Zuerst werden die Datentypen, wie in Abschnitt 6.1.1 beschrieben, kategorisiert. Dabei werden die Referenz-Datentypen ausgelassen, da diese als allgemein behandelt werden und daher nicht gesondert erwähnt werden müssen. Diese werden in Tabelle 6.1 gegenübergestellt.

Zu erkennen ist, dass Java hierbei sowohl mit mehr referenzlosen als auch gängigen Datentypen kategorisiert wird. Zudem erfordert Java aufgrund externer Imports eine getrennte Betrachtung einiger Datentypen. Im Gegensatz dazu gestaltet sich vieles in Python einfacher, da viele Funktionalitäten bereits direkt in die Sprache integriert sind.

Anforderung	Java	Python
Referenzlose Datentypen	byte	int
	short	float
	int	str
	long	bool
	float	NoneType
	double	
	char	
	boolean	
Gängige Datentypen	String	list
	StringBuilder	tuples
	StringBuffer	dict
	array	set
	ArrayList	type
	LinkedList	
	Wrapper-Klassen	
	BigDecimal	
	HashMap	
	HashSet	

Tabelle 6.1: Kategorisierung der Datentypen von Java und Python

6.2.2 Debugger Output evaluieren

Nachdem die Datentypen festgelegt wurden, ist es notwendig, ihre Ausgabe zu untersuchen. Hier werden einige Beispiele betrachtet, um ein besseres Verständnis für die Ausgabe zu entwickeln.

Referenzlose Datentypen Als Erstes werden primitive Datentypen betrachtet. Dabei wird als Beispiel der Code `f = 1.0` in Python und `float f = 1.0f` in Java betrachtet. Die jeweilige Ausgabe des Debuggers für diese Beispiele wird in Listing 6.1 für Python und in Listing 6.2 für Java dargestellt. Das gegebene Datenformat wird mittels selbst definiertem Typ *Variable* definiert.

```
{
    evaluateName: 'f',
    name: 'f',
    type: 'float',
    value: '1.0',
    variablesReference: 0
}
```

Listing 6.1: Beispiel Debugger-Output eines primitiven Datentyps in Python

```
{
    evaluateName: 'f',
    name: 'f',
    type: 'float',
    value: '1.0',
    variablesReference: 0,
    namedVariables: 0,
    indexedVariables: 0
}
```

Listing 6.2: Beispiel Debugger-Output eines primitiven Datentyps in Java

Es fällt auf, dass Java im Gegensatz zu Python zwei zusätzliche Werte zurückliefert: *namedVariables* und *indexedVariables*. Diese Werte werden jedoch im Folgenden keine Relevanz haben, weshalb nicht weiter auf diese eingegangen wird. Abgesehen davon liefern beide Programmiersprachen die gleichen grundlegenden Informationen.

Strings Als nächstes werden Strings betrachtet. Hierzu wird in Python mit `helloWorld = "Hello World!"` und in Java wird mit `String helloWorld = "Hello World!"` das Verhalten untersucht. Beim Betrachten des Debugger-Outputs für diesen String in Python, ergibt sich der Output aus Listing 6.3.

```
{
    evaluateName: 'helloWorld',
    name: 'helloWorld',
    type: 'str',
    value: 'Hello World!',
    variablesReference: 0
}
```

Listing 6.3: Beispiel Debugger-Output eines Strings in Python

In diesem Output ist zu erkennen, dass der String mit einer *variablesReference* von 0 bereitgestellt wird, was bedeutet, dass keine weiteren Variablen referenziert werden. Der Wert des Strings wird direkt angezeigt und kann so verwendet werden. Java jedoch handhabt Strings anders als Python. In Abbildung 6.1 wird dies verdeutlicht.

In (a) ist der Ausgabewert für den String dargestellt. Dabei fällt auf, dass es eine *variablesReference* größer *0* gibt. Diese *variablesReference* zeigt auf, dass im Gegensatz zu Python ein String in Java weiter aufgelöst werden kann. Nach der Auflösung ergibt sich der Output in (b). Dieser zeigt, wie Java den String aus Bytes zusammensetzt. In diesem Fall sind es zwölf Bytes, die notwendig sind, um

```
{
    evaluateName: 'helloWorld',
    name: 'helloWorld',
    type: 'String',
    value: 'Hello World!',
    variablesReference: 31,
    namedVariables: 0,
    indexedVariables: 0
}
```

(a) Erster Output des Debuggers von String in Java

```
{ // Auflösung Referenz 31
    evaluateName: 'helloWorld.value',
    name: 'value',
    type: 'byte[]',
    value: 'byte[12]@22',
    variablesReference: 10,
    namedVariables: 0,
    indexedVariables: 12
}
```

(b) Auflösung des ersten Outputs von String in Java

Abbildung 6.1: Beispiel Debugger-Output eines Strings in Java

den String `"Hello World!"` darzustellen. Zusätzlich zu `helloWorld.value` gibt es in der Auflösung von (b) auch `helloWorld.coder` und `helloWorld.hash`, die jedoch in diesem Kontext keine Relevanz haben und daher nicht weiter betrachtet werden.

Tupel in Python Als Nächstes wird das Beispiel eines Tupels in Python betrachtet, da Java diese nicht ohne zusätzliche Bibliotheken unterstützt. Der Output der Variable `testTuple = (1, 2)` wird in Abbildung 6.2 dargestellt.

```
{
    evaluateName: 'testTuple',
    name: 'testTuple',
    type: 'tuple',
    value: '(1, 2)',
    variablesReference: 13,
}
```

(a) Erster Output des Debuggers von Tupel in Python

```
{ // Auflösung Referenz 13
    evaluateName: 'testTuple[0]',
    name: '0',
    type: 'int',
    value: '1',
    variablesReference: 0,
}
```

(b) Auflösung des ersten Outputs von Tupel in Python

Abbildung 6.2: Beispiel Debugger-Output eines Tupel in Python

In (a) wird der erste Output des Debuggers für das Tupel in Python gezeigt. Hier wird der gesamte Wert des Tupels innerhalb des *value*-Parameters gespeichert. Erst nach Auflösen dieses Outputs, in (b) dargestellt, werden die Werte der jeweiligen Indizes deutlich. Für den Index 0 des Tupels liefert der Debugger den dargestellten Output in (b). Dabei ist zu erkennen, dass dort dann der tatsächliche Wert des Index 0 zur Verfügung gestellt wird. Der Output für Index 1 wird aufgrund des gleichen Schemas wie Index 0 nicht mit aufgeführt.

Wrapper in Java Da Wrapper-Typen nur in Java existieren, werden sie ebenfalls ohne einen direkten Vergleich mit Python betrachtet. Das Listing 6.4 zeigt den ersten Output des Debuggers, wenn die Variable `Byte b = 12` getestet wird.

```
{
    evaluateName: 'b',
    name: 'b',
    type: 'Byte',
    value: 'Byte@19',
    variablesReference: 15,
    namedVariables: 0,
    indexedVariables: 0
}
```

Listing 6.4: Erster Output des Debuggers von Byte in Java

Hier ist zu erkennen, dass ohne eine weitere Auflösung kein Wert für die Variable bereitgestellt wird, sondern lediglich eine Information darüber, dass es sich um ein Byte handelt. In Abbildung 6.3 sind die weiteren Auflösungen dargestellt.

```
{ // Auflösung Referenz 15
    evaluateName: 'b',
    name: '',
    type: '',
    value: 'Byte@19 "12"',
    variablesReference: 17,
    namedVariables: 0,
    indexedVariables: 0
}
```

(a) Auflösung des ersten Outputs eines Wrappers in Java

```
{ // Auflösung Referenz 17
    evaluateName: 'b',
    name: 'value',
    type: 'byte',
    value: '12',
    variablesReference: 0,
    namedVariables: 0,
    indexedVariables: 0
}
```

(b) Auflösung des zweiten Outputs eines Wrappers in Java

Abbildung 6.3: Beispiel Debugger-Output eines Wrappers in Java

In (a) ist zu erkennen, dass innerhalb des *value*-Parameters nun der Wert `12` steht. Nach der Auflösung dieses Wertes, wie in (b) dargestellt, wird der primitive Datentyp des Wrappers mit dem entsprechenden Wert angegeben.

HashMaps in Java In dem letzten Beispiel wird eine HashMap in Java betrachtet. Der verwendete Code (a) als auch der Debugger-Output (b) für diese HashMap wird in Abbildung 6.4 dargestellt.

Hier werden zuerst nur Informationen über die Größe der HashMap im Parameter *value* durch `size=1` angezeigt. Dies ist jedoch nicht ausreichend, um die Darstel-

```
Map<String, Integer> hashMap =
    new HashMap<>();
hashMap.put("one", 1);
```

(a) Verwendeter Java-Code für Visualisierung

```
{
    evaluateName: 'testHashMap',
    name: 'testHashMap',
    type: 'HashMap',
    value: 'HashMap@9 size=1',
    variablesReference: 7,
    namedVariables: 0,
    indexedVariables: 1
}
```

(b) Erster Output des Debuggers von HashMap in Java

Abbildung 6.4: Veranschaulichung von HashMap in Java

lung einer HashMap zu ermöglichen. Daher wird die HashMap weiter aufgelöst, was in Abbildung 6.5 dargestellt wird.

```
{ // Auflösung Referenz 7
    evaluateName: 'testHashMap.
        entrySet().toArray()[0]',
    name: '0',
    type: 'HashMap$Node',
    value: 'HashMap$Node@29',
    variablesReference: 8,
    namedVariables: 0,
    indexedVariables: 0
}
```

(a) Auflösung des ersten Outputs von HashMap in Java

```
{ // Auflösung Referenz 8
    evaluateName: 'testHashMap.
        entrySet().toArray()[0]',
    name: '',
    type: '',
    value: 'HashMap$Node@29 "one":"1"'
        ,
    variablesReference: 9,
    namedVariables: 0,
    indexedVariables: 0
}
```

(b) Auflösung des zweiten Outputs von HashMap in Java

```
{ // Auflösung Referenz 9
    evaluateName: '((java.util.Map
        $Entry)variableName.
        entrySet().toArray()[0]).
        getKey()',
    name: 'key',
    type: 'String',
    value: '"one"',
    variablesReference: 10,
    namedVariables: 0,
    indexedVariables: 0
}
```

(c) Auflösung des dritten Outputs von HashMap in Java

```
{ // Auflösung Referenz 9
    evaluateName: '((java.util.Map$
        Entry)variableName.entrySet().
        toArray()[0]).getValue()',
    name: 'value',
    type: 'Integer',
    value: 'Integer@33',
    variablesReference: 11,
    namedVariables: 0,
    indexedVariables: 0
}
```

(d) Auflösung des dritten Outputs von HashMap in Java

Abbildung 6.5: Beispiel Debugger-Output einer HashMap in Java

In (a) und (b) ist zu erkennen, dass die HashMap einen Node am Anfang hat. In (b) ist der Wert `"one":"1"` im Parameter *value* zu sehen. Wenn diese Auflösung in (b) weiter aufgelöst wird, erhält man in (c) den *key* und in (d) den *value* des gegebenen Index der HashMap. Da der *key* in diesem Beispiel ein String ist, kann dieser, wie oben bereits erläutert, aufgelöst werden. An diesem Punkt sind alle notwendigen Informationen über die Werte der HashMap gesammelt.

Im Allgemeinen würde bei einer HashMap mit mehr als einem Element die Auflösung von Referenz *7* aus Abbildung 6.4 (b) für jeden Eintrag einen weiteren Node wie in Abbildung 6.5 (a) erstellt werden. Für jedes dieser Elemente folgen dann wiederum (b) als auch (c) und (d).

7 Erweiterung und Verbesserung für die Programmiersprache Python

Die vorherigen Kapitel haben bereits einen umfassenden Einblick in den aktuellen Stand der VSC-Erweiterung und deren Aufbau gewährt. In der folgenden Tabelle 7.1 wird das Ergebnis der Vorarbeit aus Kapitel 4 in Bezug zu den Anforderungen aus Kapitel 2 gestellt.

Anforderung	Framework vor dieser Arbeit
Handhabung und Bedienbarkeit	bedingt
Darstellung	bedingt
Effizienz	bedingt
Unterstützung verschiedener Programmiersprachen	nein
Unterstützung unterschiedlicher Versionen von Programmiersprachen	ja

Tabelle 7.1: Erfüllung der Anforderung des Frameworks vor dieser Arbeit

Da einige dieser Anforderungen entweder gar nicht oder nur bedingt erfüllt werden, sind Anpassungen notwendig, um den Anforderungen gerecht zu werden. In diesem Kontext widmet sich das folgende Kapitel den erforderlichen Anpassungen und Erweiterungen für die Python-Programmiersprache.

7.1 Unterstützung unterschiedlicher Programmiersprachen

Um die Unterstützung anderer Programmiersprachen zu ermöglichen, ist eine Neustrukturierung des Codes erforderlich. Wie bereits in Abschnitt 4.3.2 erläutert, bietet der aktuelle Code des Frameworks nur begrenzt eine geeignete Schnittstelle für die Erweiterung anderer Programmiersprachen. In Kaptiel 5 wurde in Abschnitt 5.2 genauer auf die überarbeitete Struktur eingegangen, die in dieser Hinsicht angepasst

wurde. Diese Neustrukturierung ermöglicht eine einfachere Erweiterung für verschiedene Programmiersprachen. Auf weitere Einzelheiten wird in Kapitel 8 eingegangen, da sich dieses Kapitel auf die aktuelle Umsetzung mit Python konzentriert.

7.2 Komplexe Datentypen

Das Problem bei der Verarbeitung komplexer Datentypen ist auf die Art und Weise zurückzuführen, wie das Framework Variablen auflöst. Um dieses Problem näher zu erläutern, dient das Beispiel des Tupels `(1, (2, (3, None)))` aus Listing 4.4. In Listing 7.1 ist das Datenformat dargestellt, das zur Auflösung der Variable verwendet wird.

```
{
    name: 'nestedTuple',
    value: '(1, (2, (...)))',
    type: 'tuple',
    evaluateName: 'nestedTuple',
    variablesReference: 7
}
```

Listing 7.1: Datenformat der aufzulösenden Variable

```
{
    name: '1',
    value: '(2, (3, None))',
    type: 'tuple',
    evaluateName: 'nestedTuple[1]',
    variablesReference: 11
}
```

Listing 7.2: Auflösung der *variablesReference*

Der Parameter *variablesReference* gibt an, ob die Variable eine Referenz auf ein anderes Objekt hat. Wenn die *variablesReference* `7` aufgelöst wird, können die Informationen in Listing 7.2 abgelesen werden. Hier wurde die erste Verschachtelung `(1, (...))` aufgelöst und die Referenz auf Index `1` ist nun verfügbar. Da es insgesamt drei Verschachtelungen gibt, liegt es nahe, auch die *variablesReference* `11` aufzulösen. Das Problem besteht jedoch darin, dass die bisherige Lösung den *value*-Wert direkt verwendet und diesen als einzelnen Wert interpretiert, indem er in `2` und `3,None` aufgeteilt wird. Dies führt zu unvollständigen Darstellungen von verschachtelten Datentypen, wie bereits in Abbildung 4.5 näher erläutert. Um dieses Problem anzugehen, ist eine grundlegend andere Methode erforderlich.

Der verfolgte Ansatz basiert auf dem Konzept der Tiefensuche. Dies ermöglicht die Erkennung komplexer Strukturen und die Vermeidung von Zyklen. Der Pseudocode in Listing 7.3 beschreibt den grundlegenden Prozess der Tiefensuche, der ausgehend von der zu untersuchenden Variable durchgeführt wird.

```
1   depthSearch(variable) {
2       listForDepth = resolve variablesReference of variable;
3
4       while (listForDepth not empty) {
5           actualVariable = popFirst(listForDepth);
6           mark actualVariable as visited;
7
8           if (actualVariable has variablesReference) {
9               save result of depthSearch(actualVariable);
10          }
11          save actualVariable;
12      }
13  }
```

Listing 7.3: Pseudocode der Tiefensuche innerhalb des Frameworks

Der Prozess beginnt mit der Auflösung der *variablesReference* der zu untersuchenden Variable in Zeile 2. Diese Referenz hat zum Beispiel fünf Referenzen, wenn die Liste fünf Elemente enthält und nur eine Referenz, wenn die Liste nur ein Element enthält. Anschließend wird nacheinander überprüft, ob jede dieser Referenzen selbst wieder eine *variablesReference* aufweist, wie in Zeile 8 dargestellt. Falls ja, wird der Prozess rekursiv für diese Variable fortgesetzt, wie in Zeile 9 beschrieben. Schließlich werden die Ergebnisse in den Zeilen 9 und 11 gespeichert und können später verwendet werden, um die Visualisierung korrekt aufzubauen. Beim Speichern der Variablen wird direkt ausgewertet, um welchen Typ es sich handelt. Wenn zum Beispiel bei dem verschachtelten Tupel `(1, (2, 3))` die 1 untersucht wird, wird diese direkt als 1 gespeichert. Für den zweiten Teil des Tupels, also `(2, 3)`, wird hingegen eine Referenz gespeichert und zusätzlich das Ergebnis als eigener Wert. Im weiteren Verlauf können die gespeicherten Referenzen genutzt werden, um die Visualisierung korrekt aufzubauen.

Ein weiterer Vorteil dieses Ansatzes ist, dass Schritte sehr einfach zwischengespeichert werden können. Dies vermeidet zusätzliche Aufrufe zur Auflösung der nächsten Referenz und erspart somit Zeit. Bei jeder Auflösung der Variable kann gespeichert werden, welches Ergebnis erzielt wurde. Dies ist jedoch nicht für jeden Schritt im Debugging-Prozess übertragbar, da die *variablesReference* kein dauerhafter Wert während des gesamten Debugging-Prozesses ist.

Die korrekte Darstellung der Beispiele aus Listing 4.4 und die Funktionalität des Ansatzes zur Tiefensuche werden in Abbildung 7.1 verdeutlicht.

Während dieser Erweiterung wurden auch Probleme mit den Pfeilen festgestellt. Bisher konnten Pfeile nur für eine einzige Referenz dargestellt werden. Wenn je-

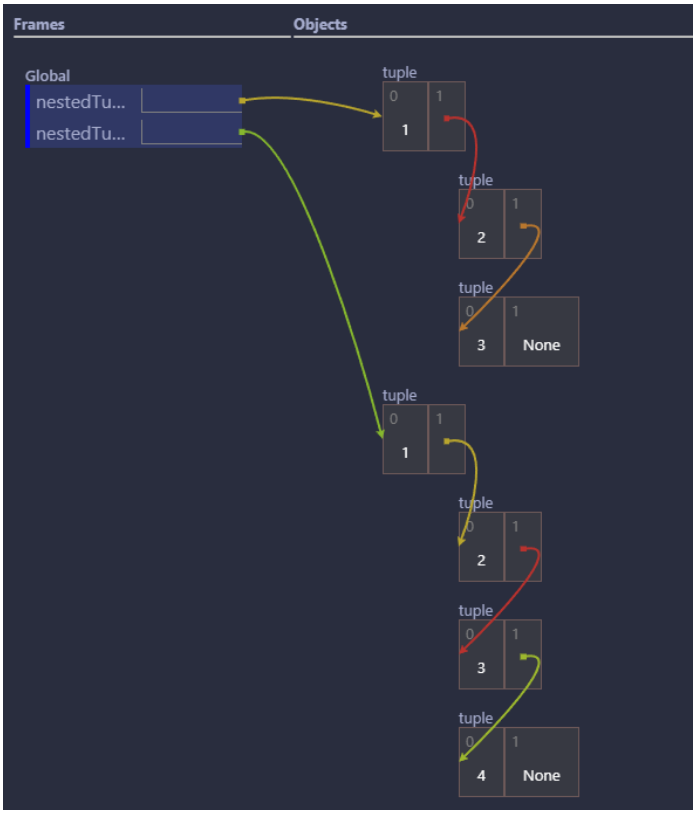


Abbildung 7.1: Korrekte und vollständige Darstellung von verschachtelten Tupel

doch mehrere Objekte auf dasselbe Objekt verweisen, wurde nur ein Pfeil angezeigt. Um dieses Problem zu lösen, wurde eine zusätzliche Kennung für jeden Start- und Endpunkt eines Pfeils eingeführt. Dies verhindert, dass mehrere identische Start- oder Endpunkte existieren, die nur einmal ausgewertet werden können.

7.3 Testen

In der Vorarbeit fehlen funktionierende und umfassende Tests, weshalb dies nachgebessert werden muss. Eine detaillierte Untersuchung des Testens von VSC-Erweiterungen erfolgt später in Kapitel 9, daher wird in diesem Zusammenhang nicht näher darauf eingegangen.

7.4 User Experience

Wie bereits in Abschnitt 4.3.5 erläutert, gibt es einige Aspekte der Benutzerfreundlichkeit, die verbessert werden können. Beispielsweise wurde das vollständige Schließen des Fensters und das erneute Öffnen der ursprünglichen Datei nach der Visualisierung vereinfacht werden. Zudem ist es nun möglich, Dateien ohne geöffneten Ordner zu visualisieren. Dies ermöglicht es, dass die vor der Visualisierung geöffnete Umgebung auch nach einem Fehler noch geöffnet ist und keine Änderungen vorgenommen werden. Eine ebenfalls zu nennende größere Verbesserung ist die Ermöglichung, andere Dateien zusätzlich aufzurufen und die dort verwendeten Zeilen Code ebenfalls in die Visualisierung zu integrieren. Wenn eine selbst geschriebene Methode in einer anderen Datei aufgerufen wird, kann nun auch innerhalb dieser Datei angezeigt werden, welche Zeile durchlaufen wird.

Insgesamt sollen diese Anpassungen dazu beitragen, die gestellten Anforderungen besser zu erfüllen und die Nutzererfahrung mit dem Framework zu verbessern.

7.5 Bestandsaufnahme

Nach den genannten Erweiterungen und Verbesserungen kann die Tabelle 7.1 aktualisiert werden. Dies wird im Folgenden in der Tabelle 7.2 dargestellt.

Anforderung	Eigenes Framework
Handhabung und Bedienbarkeit	ja
Darstellung	ja
Effizienz	bedingt
Unterstützung verschiedener Programmiersprachen	bedingt
Unterstützung unterschiedlicher Versionen von Programmiersprachen	ja

Tabelle 7.2: Aktualisierung der Erfüllung der Anforderungen des Frameworks nach Verbesserung und Erweiterung für die Programmiersprache Python

Nun sind Handhabung, Darstellung sowie die Unterstützung unterschiedlicher Versionen von Programmiersprachen gegeben. Auf die Effizienz des Frameworks wurde nicht eingegangen, da der Fokus zu diesem Zeitpunkt auf Funktionalität und Richtigkeit gelegt wurde. Die Unterstützung verschiedener Programmiersprachen ist nur bedingt gegeben, da bisher lediglich der Grundstein gelegt wurde, um ein

einfaches Erweitern für neue Programmiersprachen zu ermöglichen. Darauf wird im folgenden Kapitel 8 näher eingegangen.

8 Erweiterung für die Programmiersprache Java

Nach Abschluss des Kapitels zur Erweiterung und Verbesserung der Programmiersprache Python, wie in Kapitel 7 beschrieben, erfolgt im Folgenden die Erweiterung des Frameworks mit der Programmiersprache Java. Hierbei wird Bezug auf den Abschnitt 6.2 genommen. Zuerst werden generelle Schritte erläutert, welche zur Integration einer neuen Programmiersprache nötig sind. Anschließend wird auf Referenzen in Java anhand von Strings eingegangen. Daraufhin werden Wrapper-Typen und Strings genauer beschrieben. Danach wird auf die Visualisierung von HashMaps eingegangen. Zuletzt erfolgt eine erneute Bestandsaufnahme.

8.1 Generelle Erweiterungsschritte

Um Java als Programmiersprache dem Framework hinzuzufügen, sind einige Schritte notwendig, die im Folgenden genauer beschrieben werden.

Zunächst muss es der Erweiterung erlaubt werden, auch bei Java-Dateien zu erscheinen. Bislang war die Erweiterung so konfiguriert, dass der Start-Button *Python-Visualization*, wie bereits in Abschnitt 4.3.1 vorgestellt, nur angezeigt wurde, wenn die Datei die Erweiterung *.py* hatte. Diese Konfiguration wird in der Datei *package.json* vorgenommen. Der relevante, erweiterte Ausschnitt ist in Listing 8.1 dargestellt.

```
"menus": {
    "editor/context": [
      {
        "command": "programflow-visualization.startDebugSession",
        "when": "(editorLangId == python && resourceExtname == .py) || (
            editorLangId == java && resourceExtname == .java)",
        "group": "programflowViz@0"
      }
    ],
    "editor/title/run": [
      {
        "command": "programflow-visualization.startDebugSession",
        "when": "resourceExtname == .py || resourceExtname == .java",
        "group": "programflowViz@0"
      }
    ]
}
```

Listing 8.1: Ausschnitt Konfiguration der Extension

In diesem Ausschnitt sind unter dem Abschnitt *menus* die Abschnitte *editor/context* und *editor/title/run* zu sehen. Da die Erweiterung an zwei verschiedenen Stellen gestartet werden kann, wie zuvor erläutert, werden beide Optionen im Menü erweitert. Die Bedingung *when* bestimmt, wann die Erweiterung angezeigt wird. Mit *command* wird festgelegt, welche Aktion bei der Interaktion ausgeführt wird. In diesem Fall handelt es sich um einen einfachen booleschen Ausdruck, der überprüft, ob die Datei die Erweiterung *.py* oder *.java* hat.

Da nun die Erweiterung auch für Java-Dateien gestartet werden kann, müssen im Framework interne Anpassungen vorgenommen werden. Dafür gibt es einen internen Datentyp *SupportedLanguages*, welcher die Handhabung mehrerer Sprachen erleichtern soll. Dieser Datentyp wird mittels *type* von TypeScript definiert: `type SupportedLanguages = 'python' | 'java';`. Der Trace Generator verwendet dann diese Information, um zu entscheiden, welches Interface der *Backend-Session* übergeben werden soll. Dafür muss zunächst eine neue Implementierung dieses Interfaces für Java erstellt werden. Innerhalb dieser Datei wird dann die gesamte Logik für die Programmiersprache Java implementiert. Durch die Verwendung des DAP wird sichergestellt, dass bis zur Auswertung der Daten des Debuggers die gleichen Schritte wie bei Python durchgeführt werden.

Abschließend müssen noch neue Datentypen hinzugefügt werden. Für eine einfachere Nutzung im Code werden entsprechende *type*-Definitionen erstellt. Ein Beispiel ist der `type Value`, der primitive Datentypen speichert. Ein *int* wird beispielsweise mit `type: 'int'; value: number` definiert und ein String mit `type:`

`'str'; value: string` . Da Java mehr primitive Datentypen unterstützt als Python, wie in Tabelle 6.1 ersichtlich, müssen an dieser Stelle auch neue *Value*-Typen für Java hinzugefügt werden.

8.2 Referenzen am Beispiel eines Strings

Um ein Verständnis für Referenzen in Java zu erhalten, wird das in Listing 8.2 dargestellte Beispiel verwendet.

```
1  public class SampleJavaClass {
2      public static void main(String[] args) {
3          String s1 = "Hello42";
4          String s2 = "Hello42";
5          String s3 = "Hello" + 42;
6          String s4 = "Hello" + get42();
7      }
8
9      static int get42() {
10         return 42;
11     }
12 }
```

Listing 8.2: Java-Code für das Verständnis von Referenzen

Dieses Beispiel dient dazu, das Konzept der Referenzen in Java besser zu verstehen und wie diese in der Ausgabe des Debuggers zu behandeln sind. Der Debugger-Output wurde bereits in Abschnitt 6.2 dargelegt. Dabei wird ein String mit dem Wert `"Hello42"` auf verschiedene Arten definiert. In Zeile 3 und 4 werden sie identisch definiert. In Zeile 5 werden ein String und eine Ganzzahl, in diesem Beispiel als *int*, miteinander addiert. Das letzte Beispiel in Zeile 6 ruft eine Methode auf, die eine Ganzzahl zurückliefert und diese mit dem String addiert. Bei der Visualisierung sollten die Strings mit der gleichen Referenz, also s1, s2 und s3, denselben Speicherplatz im *Heap* nutzen, sodass die Variablen die gleiche Referenz aufweisen. s4 sollte eine eigene Referenz haben. In Abbildung 8.1 werden die Debugger-Outputs von s1 bis s4 dargestellt.

Bei der Betrachtung der einzelnen Debugger-Outputs fällt jedoch auf, dass nicht die gleiche *variablesReference* geteilt wird. In Python ist ein eindeutiges Indiz für die gleiche Referenz, wenn die *variablesReference* gleich ist. Der Java-Debugger liefert jedoch trotz gleicher Referenz unterschiedliche Werte. Wenn man die Auflösung

```
{
    evaluateName: 's1',
    name: 's1',
    type: 'String',
    value: '"Hello42"',
    variablesReference: 31,
    namedVariables: 0,
    indexedVariables: 0
}
```

(a) Debugger-Output für s1

```
{
    evaluateName: 's2',
    name: 's2',
    type: 'String',
    value: '"Hello42"',
    variablesReference: 32,
    namedVariables: 0,
    indexedVariables: 0
}
```

(b) Debugger-Output für s2

```
{
    evaluateName: 's3',
    name: 's3',
    type: 'String',
    value: '"Hello42"',
    variablesReference: 33,
    namedVariables: 0,
    indexedVariables: 0
}
```

(c) Debugger-Output für s3

```
{
    evaluateName: 's4',
    name: 's4',
    type: 'String',
    value: '"Hello42"',
    variablesReference: 34,
    namedVariables: 0,
    indexedVariables: 0
}
```

(d) Debugger-Output für s4

Abbildung 8.1: Debugger-Output für s1 bis s4

der jeweiligen Outputs betrachtet, wie in Abbildung 8.2 zu sehen ist, erkennt man Gemeinsamkeiten.

Innerhalb des *value*-Felds der Bytes steht eine Zahl nach dem @-Symbol. Diese Zahl gibt die entsprechende Referenz im *Heap* an. Dies bedeutet, wenn die gleiche Referenz geteilt wird, ist auch der Wert hinter dem @-Symbol derselbe. Um also gleiche Referenzen von Java-Objekten zu erkennen, muss dieser Wert betrachtet und evaluiert werden. Wie Strings im Allgemeinen ausgewertet werden, wird im folgenden Abschnitt näher erläutert.

8.3 Wrapper-Typen und Strings

Das Schema des Debuggers für Wrapper-Typen wurde bereits in Abschnitt 6.2.2 erläutert. Bei der Erstellung der Visualisierung muss jedoch darüber nachgedacht werden, wie diese dargestellt werden sollen. Der Debugger-Output erzeugt drei Referenzen, die dargestellt werden müssen: Die Referenz aus Listing 6.4 sowie die beiden Auflösungen (a) und (b) aus Abbildung 6.3. An dieser Stelle kann die Darstellung jedoch vereinfacht werden, ohne die Richtigkeit zu verlieren. Der Schlüs-

```
{ // Auflösung Referenz 31
    evaluateName: 's1.value',
    name: 'value',
    type: 'byte[]',
    value: 'byte[7]@22',
    variablesReference: 35,
    namedVariables: 0,
    indexedVariables: 7
}
```

(a) Auflösung von s1

```
{ // Auflösung Referenz 32
    evaluateName: 's2.value',
    name: 'value',
    type: 'byte[]',
    value: 'byte[7]@22',
    variablesReference: 36,
    namedVariables: 0,
    indexedVariables: 7
}
```

(b) Auflösung von s2

```
{ // Auflösung Referenz 33
    evaluateName: 's3.value',
    name: 'value',
    type: 'byte[]',
    value: 'byte[7]@22',
    variablesReference: 37,
    namedVariables: 0,
    indexedVariables: 7
}
```

(c) Auflösung von s3

```
{ // Auflösung Referenz 34
    evaluateName: 's4.value',
    name: 'value',
    type: 'byte[]',
    value: 'byte[7]@42',
    variablesReference: 38,
    namedVariables: 0,
    indexedVariables: 7
}
```

(d) Auflösung von s4

Abbildung 8.2: Auflösungen der Variablen s1 - s4

selpunkt bei der Darstellung eines Wrappers ist, dass er nicht wie ein einfacher Wert im *Stack* gespeichert wird, sondern als eigenes Objekt im *Heap* existiert. Diese Darstellung kann mit einer einzigen Referenz vereinfacht visualisiert werden, wie in Abbildung 8.3 dargestellt.

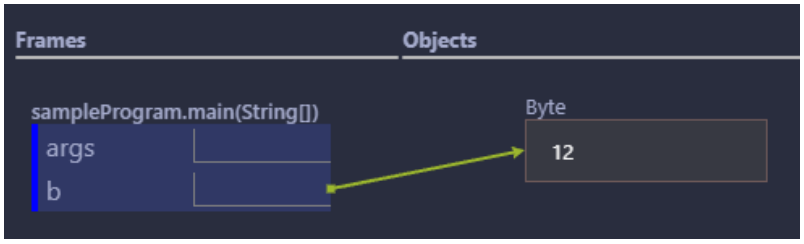


Abbildung 8.3: Darstellung eines Wrapper in der Visualisierung

Auch Strings müssen anders dargestellt werden als es der Debugger-Output vorgibt. Anstatt auf jedes einzelne Byte im String mit einer eigenen Referenz zu verweisen, wird der gesamte String als Ganzes im *Heap* visualisiert. Dies ist in Abbildung 8.4 dargestellt.

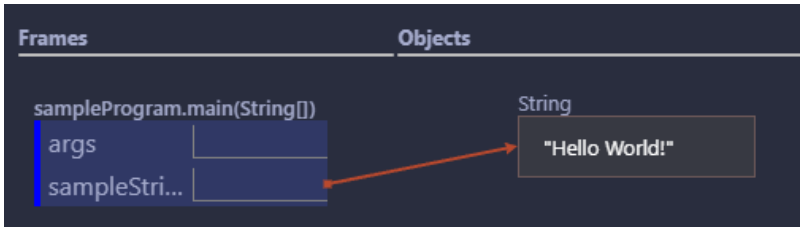


Abbildung 8.4: Darstellung eines String in der Visualisierung

Bei der richtigen Auswertung musste entschieden werden, in welchem Umfang eine Variable aufgelöst werden sollte, um die benötigten Informationen zu erhalten. Um beispielsweise bei einem String das Auflösen jedes einzelnen Bytes zu vermeiden und viele zeitintensive Anfragen zu verhindern, wurde entschieden, den *value*-Parameter als Referenz für den benötigten Wert zu benutzen. Auf diese Weise wird ein String einmal aufgelöst, wie es aufgrund der Referenz erforderlich ist, aber es ist nicht notwendig, weitere Auflösungen vorzunehmen. Der Wert der Variable wird mithilfe einer einfachen `split()`-Funktion aus dem *value*-Parameter extrahiert. Dies ist möglich, da das Schema immer gleich ist und gewährleistet ist, dass der Wert der Variable nach einem Anführungszeichen kommt. Auch bei den Wrapper-Typen wird aus dem *value*-Parameter der eigentliche Wert entnommen.

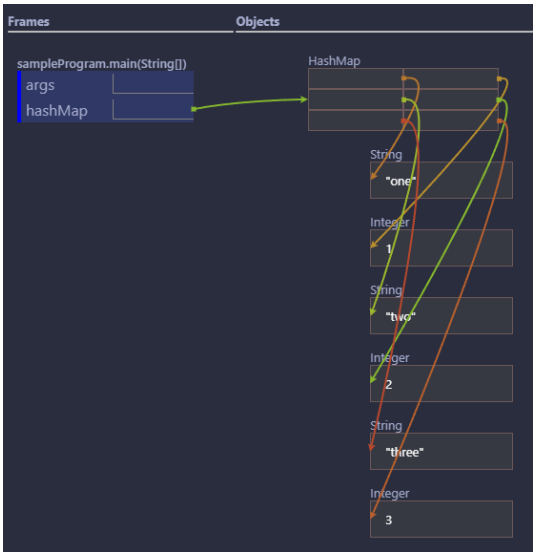
8.4 HashMaps

Die Komplexität von HashMaps und die Anzahl der Schritte zur Auflösung wurden bereits in Abschnitt 6.2.2 aufgezeigt. Für die Visualisierung wurde entschieden, sich auf das Wesentliche zu konzentrieren. Es werden nur die vorhandenen Werte innerhalb der HashMap dargestellt. Da Objekte innerhalb einer HashMap gespeichert werden, werden die Key-Value-Paare ebenfalls mit Referenzen dargestellt und nicht vereinfacht, wie beispielsweise bei Strings. Dies ist in Abbildung 8.5 dargestellt.

In (a) ist der Code dargestellt, der für die Visualisierung in (b) verwendet wurde. Bei der Betrachtung der Darstellung wird deutlich, dass es bei einer großen HashMap sehr schnell unübersichtlich werden kann. Die Darstellung weist Schwächen auf, da die Objekte untereinander angeordnet sind und die Referenzpfeile die Objekte durchqueren. Da der Fokus auf der Funktionalität liegt, wird mehr Wert auf die korrekte Darstellung gelegt, weshalb die visuelle Darstellung vorerst so beibehalten wird.

```java
Map<String, Integer> hashMap = new
    HashMap<>();
hashMap.put("one", 1);
hashMap.put("two", 2);
hashMap.put("three", 3);
```

(a) Verwendeter Java-Code für Visualisierung



(b) Darstellung einer HashMap in der Visualisierung

Abbildung 8.5: Gegenüberstellung Java-Code einer HashMap und deren Visualisierung

Die Implementierungslogik beruht darauf, für jeden Knoten in der HashMap eine eigene Evaluation nach dem Prinzip der Wrapper durchzuführen, wie oben beschrieben. Da eine HashMap auf die gleichen Objekte zugreift, die bereits behandelt wurden, kann dies einfach umgesetzt werden. Jedoch kann nicht dieselbe Vorgehensweise wie bei den Wrappern verfolgt werden, da zuerst der *key* und der *value* extrahiert werden müssen. Da dies einen zusätzlichen Schritt erfordert im Vergleich zum Beispiel bei einem Byte, ist eine einfache Umsetzung ohne komplexere Logik nicht möglich. Deshalb werden zunächst die einzelnen Key-Value-Paare extrahiert und anschließend einzeln, nach dem gleichen Schema wie Wrapper, ausgewertet.

8.5 Testen

Wie in Kapitel 7 wird auch hier für das Testen von VSC-Erweiterungen auf Kapitel 9 verwiesen. Daher wird in diesem Zusammenhang nicht näher darauf eingegangen.

8.6 Bestandsaufnahme

Nach der Erweiterung des Frameworks um die Programmiersprache Java lässt sich die Tabelle 7.2 aktualisieren. Dies wird im Folgenden in der Tabelle 8.1 dargestellt.

Anforderung	Eigenes Framework
Handhabung und Bedienbarkeit	ja
Darstellung	ja
Effizienz	bedingt
Unterstützung verschiedener Programmiersprachen	ja
Unterstützung unterschiedlicher Versionen von Programmiersprachen	ja

Tabelle 8.1: Aktualisierung der Erfüllung der Anforderungen des Frameworks nach Erweiterung um die Programmiersprache Java

Die Erweiterung des Frameworks um die Programmiersprache Java betrifft lediglich die Anforderung *Unterstützung verschiedener Programmiersprachen*. Da nun neben Python auch Java unterstützt wird, kann diese Anforderung von *bedingt* auf *ja* geändert werden. Es muss jedoch darauf hingewiesen werden, dass das Framework, wie im Abschnitt 2.4 beschrieben, nicht den idealen Umfang an unterstützten Programmiersprachen erzielt.

9 Testen von Visual Studio Code Extensions

Dieses Kapitel widmet sich dem Testen, nachdem die Architektur und die Implementierungen ausführlich diskutiert wurden. Zunächst wird die Struktur der Testumgebung diskutiert. Anschließend auf die Struktur eines Tests als auch die konkrete Ausführen eingegangen. Schließlich wird die Integration in GitHub Actions näher betrachtet.

9.1 Struktur der Testumgebung

Das Testen von VSC-Extensions basiert auf den Bibliotheken *test-electron* und *Mocha*. Um die Tests durchzuführen, muss eine Testumgebung aufgebaut werden, in der definiert wird, welche Version von VSC verwendet wird und welche Erweiterungen benötigt werden. In Listing 9.1 wird eine solche Konfiguration gezeigt.

```
1  const vscodeExecutablePath = await downloadAndUnzipVSCode('stable');
2  const [cliPath, ...args] = resolveCliArgsFromVSCodeExecutablePath(
       vscodeExecutablePath);
3  cp.spawnSync(
4      cliPath,
5      [...args,
6          '--install-extension', 'vscjava.vscode-java-debug',
7          '--install-extension', 'redhat.java',
8          '--install-extension', 'ms-python.python',
9          '--install-extension', 'ms-toolsai.jupyter'],
10     {
11         encoding: 'utf-8',
12         stdio: 'inherit'
13     }
14 );
```

Listing 9.1: Konfiguration der VSC-Testumgebung

In Zeile 1 wird VSC heruntergeladen. Dabei kann eine bestimmte Version wie `1.81.1` ausgewählt werden, mit `stable` die aktuellste Version oder mit `insider` die Insider-Version verwendet werden. In Zeile 2 werden die Befehlszeilenargumente von VSC ausgewertet, die für die Unterstützung von Subprozessen benötigt werden. Von Zeile 3 bis 13 wird eine solche Instanz gestartet, die dafür sorgt, dass die benötigten Erweiterungen installiert werden. Da das eigen entwickelte Framework Abhängigkeiten von Erweiterungen hat, um den Debugger nutzen zu können, müssen diese explizit installiert werden. Dies stellt sicher, dass das Framework ordnungsgemäß funktioniert.

Zusätzlich zur VSC-Umgebung wird auch der *Mocha*-Test konfiguriert. In Listing 9.2 wird diese Konfiguration gezeigt. Anzumerken ist, dass aus Gründen der Übersichtlichkeit die Fehlerbehandlungen des Codes weggelassen wurden.

```
1   new Promise((c, e) => {
2       glob('**/**.test.js', { cwd: testsRoot }, (err, files) => {
3           files.forEach((f) => mocha.addFile(path.resolve(testsRoot, f)));
4
5           mocha.run((failures) => {
6               if (failures > 0) {
7                 e(new Error(`${failures} tests failed.`));
8               } else {
9                 c();
10              }
11          });
12      });
13  });
```

Listing 9.2: Konfiguration eines Mocha-Tests

Die Tests werden innerhalb eines Promise ausgeführt. Dabei wird für jede Datei, die im angegebenen Verzeichnis mit der Endung `.test.js` vorhanden ist, ein Test ausgeführt.

9.2 Struktur eines Tests

Die Bibliothek *Mocha* bietet die Syntax einer *suite* mit *describe* und *it*. Mit dieser Struktur können Tests aufgebaut werden. Ein Beispiel für diese Struktur ist in Listing 9.3 zu sehen.

```
1  suite('The Python BackendSession', () => {
2      describe('when evaluating primitive Variables', () => {
3          it('should contains an int value', () => {
4
5          });
6      });
7  });
```

Listing 9.3: Mocha-Teststruktur

In diesem Beispiel beschreibt die *suite* den allgemeinen Testfall. In diesem Fall handelt es sich um die gesamte *BackendSession* von Python. Mit *describe* wird es konkreter, und ein bestimmter Fall innerhalb der *BackendSession* wird betrachtet. In diesem Beispiel geht es um die Auswertung von primitiven Variablen. Mit *it* wird der genaue Test durchgeführt, wie zum Beispiel in Zeile 3 mit einem *int*. Nach diesem Schema wird weiterhin für verschiedene Fälle getestet. Da dies repetitiv sein kann, bietet sich die Verwendung von `forEach` an. Dies wird in Listing 9.4 dargestellt.

```
1  const variables: Array<[string, string, number | string]> = [
2      ['positiveInt', 'int', 1],
3      ['positiveFloat', 'float', 1.0]
4  ];
5
6  variables.forEach(([name, type, value], index) => {
7      it(`should contain the variable ${name} as ${type} with value ${value}`, ()
           => {
8
9      });
10 });
```

Listing 9.4: Mehrere Testfälle als Schleife definieren

In diesem Beispiel werden zwei Variablen verwendet, die jedoch leicht erweitert werden können. Ein Array mit den gewünschten Parametern wird erstellt und für jeden Eintrag wird ein *it* innerhalb des `forEach` erstellt. Dadurch kann sehr einfach auf neue Werte getestet und der Test erweitert werden. Da zum Testen jeweils Dateien benötigt werden, die zur Erzeugung der Traces verwendet werden, müssen diese Dateien anschließend gelöscht werden, um unnötigen Speicherplatz einzusparen. Dies wird mit einer einfachen *after()*-Funktion gelöst, wie in Listing 9.5 dargestellt.

```
1    after(() => {
2        fs.rm(TESTFILE_DIR_PYTHON, { recursive: true }, err => {
3            if (err) { throw err; }
4        });
5    });
```

Listing 9.5: After-Methode zum Löschen von temporären Files

Dabei wird der gesamte Ordner, in dem die Dateien gespeichert sind, nach dem Ausführen der Tests gelöscht. Wichtig ist anzumerken, dass die Testdauer aufgrund des Startens des Debuggers länger als die standardmäßig definierten zwei Sekunden sein kann, weshalb das Zeitlimit angehoben werden muss. Dieses Zeitlimit kann auch als Erfüllung der Anforderung für die Effizienz gesehen werden. Dabei kann festgelegt werden, dass ein Test zum Beispiel nicht länger als eine Minute dauern soll. Falls diese Zeit überschritten wird und dadurch der Tests fehlschlägt, ist dies ein Indiz dafür, dass der Code überarbeitet werden muss.

9.3 Konkretes Testen

Nach der Erläuterung der Teststruktur wird im Folgenden näher auf die konkrete Durchführung der Tests eingegangen.

9.3.1 Java und Python

Das Listing 9.4 zeigt bereits die Struktur für mehrere Tests innerhalb derselben Datei auf. Da diese Vorgehensweise auch in diesem Fall angewendet werden kann, kann das *Trace* für alle Tests mithilfe von `beforeAll` erzeugt werden, wie in Listing 9.6 dargestellt.

```
1  let result: BackendTrace | undefined;
2  this.beforeAll(async function () {
3      const testFile = await TestExecutionHelper.createTestFileWith(
           TESTFILE_DIR_JAVA, "JavaPrimitiveVariableTestClass", "java",
           TestFileContents.ALL_PRIMITIVE_VARIABLES);
4
5      const context = await executeExtension(testFile);
6      result = await loadTraceFromContext(testFile, context);
7  });
```

Listing 9.6: Vorbereitung für explizites Testen von primitive Variablen

In Zeile 3 ist die Zusammensetzung der Testdatei zu sehen. Hierbei wird eine externe Datei genutzt, in der ein Beispielprogramm als String definiert ist. Diese wird dann in die eigentliche Datei eingefügt, um die Übersichtlichkeit zu wahren. Der Kontext wird in Zeile 5 nach der Ausführung des Tests gespeichert. Das vorliegende Beispiel zeigt die Vorgehensweise für Java, für Python ist die Vorgehensweise jedoch identisch. Mithilfe des Kontexts kann das Ergebnis geladen werden, wie in Zeile 6 gezeigt. Innerhalb des *it*-Blocks wird das *Trace*, gespeichert in der Variable `result`, detaillierter untersucht, wie in Listing 3 dargestellt.

```
1  if (!result) {
2    assert.fail("No result was generated!");
3  }
4
5  const stack = result.at(index + 1)?.stack[0];
6  if (!stack) {
7    assert.fail("No Stack Elements");
8  }
9
10 const keys = Array.from(Object.keys(stack.locals));
11 const values = Array.from(Object.values(stack.locals));
12
13 assert(keys.includes(name));
14 assert.deepEqual(values[keys.indexOf(name)], { type: type, value: value });
```

Listing 9.7: Expliziter Test für primitive Variablen

In den Zeilen 1 bis 3 wird sichergestellt, dass das *Trace* erzeugt ist. In Zeile 5 wird der entsprechende Schritt auf dem *Stack* herausgefiltert und in den Zeilen 6 bis 8 dessen Vorkommen überprüft. Die *keys* und *values* des *Stacks* werden in den Zeilen 10 und 11 betrachtet. Dabei entsprechen die *keys* den Variablennamen und die *values* enthalten die gespeicherten Variablenwerte. Wenn dieser Test erfolgreich durchgeführt wurde, ist sichergestellt, dass sich die gewünschte Variable an der richtigen Stelle im *Stack* befindet. Erwähnenswert dabei ist die Verwendung von `assert.deepEqual`, womit der Vergleich von zwei Objekten möglich ist. Dies ist in TypeScript mit dem normalen `assert` oder `===` nicht möglich.

Für Referenzvariablen wird ein ähnliches Prinzip angewendet. Allerdings erfolgt zusätzlich eine Betrachtung und Bewertung des *Heaps*.

9.3.2 Hilfskomponenten

Die Tests umfassen nicht nur spezifische Java- und Python-Tests, sondern auch Tests für Hilfskomponenten, die unabhängig von der eigentlichen *Trace*-Generierung sind. Konkret setzen sich die Hilfskomponenten aus einem *Completer*, einem *FileHandler* und einem *VariableMapper* zusammen. Der *Completer* ermöglicht das manuelle Festlegen des Starts und des Endes eines TypeScript-Promises, welches in den Tests überprüft wird. Durch die Verwendung des *FileHandlers* können zusätzliche Dateien hinzugefügt werden, wie zum Beispiel das Erstellen und Einfügen von *pass* am Ende einer Python-Datei, wie es in Abschnitt 5.2 beschrieben ist. Ebenso kann eine JSON-Datei erstellt werden, die das *Trace* enthält. Im Rahmen der Tests wird überprüft, ob der *FileHandler* seine Aufgaben korrekt ausführt. Die letzte genannte Komponente, der *VariableMapper*, ist dafür verantwortlich, eine im Debugger-Output gefundene Variable einem später im *Stack* gespeicherten Objekt zuzuordnen. Ziel der Tests ist die Überprüfung der Korrektheit dieser Zuordnung durch den *VariableMapper*.

Diese Tests gewährleisten die ordnungsgemäße Funktion der Hilfskomponenten, sodass sie zur nahtlosen Durchführung des *Trace*-Prozesses beitragen. Dies ist von entscheidender Bedeutung, da diese Komponenten dazu beitragen, dass das *Trace* unabhängig von der verwendeten Programmiersprache korrekt erstellt und verarbeitet wird.

9.3.3 Fehlende Tests

Nach Erklärung der implementierten Tests wird im Folgenden auf noch zu implementierende Tests eingegangen.

In diesem Zusammenhang sind vor allem die Tests des *Frontends* zu erwähnen. Es werden keine Tests der Frontendkomponenten, wie beispielsweise des *HTMLGenerator*, durchgeführt. Auch die Kommunikation zwischen *view* und *webview* sowie die tatsächliche Darstellung durch HTML, CSS und JavaScript, werden momentan nicht getestet.

Die *Trace*-Tests können ebenfalls weiter ausgebaut werden. Wie bereits gezeigt, testen diese eher auf die korrekte Anzahl der vorhandenen Elemente, aber nicht auf deren Reihenfolge oder andere im *Trace* gespeicherte Informationen. Zudem existieren nur positive Tests und keine Gegen-Tests. Wenn also getestet wird, ob ein

Element an der richtigen Position vorhanden ist, sollte auch geprüft werden, ob es nicht an der falschen Stelle existiert. Genau dies könnte in den Tests noch ergänzt werden.

9.4 Continuous Integration

Das Testen kann manuell innerhalb von VSC durchgeführt werden. Es besteht jedoch die Möglichkeit CI in GitHub einzusetzen. Dies ist mithilfe von GitHub Actions möglich. Auf die Integration in GitHub Action wird im Folgenden eingegangen.

9.4.1 Workflow für GitHub Action

Die gesamte Konfiguration des Workflows ist in Listing 9.8 dargestellt.

In Zeile 1 ist der Name des Workflows festgelegt, der zur Identifikation bei mehreren aktiven Workflows dient. Die Zeilen 3 bis 6 definieren den Zeitpunkt, zu dem dieser Workflow durchzuführen ist. In diesem Fall bei einem *push* auf den Branch *main*. Ab Zeile 8 folgen dann die eigentlichen Aufgaben des Workflows. In den Zeilen 10 bis 14 werden unterschiedliche Strategien anhand der Matrix definiert. Mit den in Zeile 12 definierten OS[1]-Variablen wird in Zeile 15 mit `runs-on` festgelegt, welches Betriebssystem der Container haben soll. Da in diesem Fall MacOS, Ubuntu und Windows angegeben sind, werden drei Container erstellt, jeweils mit einem dieser Betriebssysteme. Ein ähnliches Verfahren wird in Zeile 13 für Java und in Zeile 14 für Python verwendet. Um sicherzustellen, dass die erforderlichen Programmiersprachen innerhalb des Containers installiert sind, werden diese aktiviert. Die Zeilen 23 bis 26 laden Python herunter und konfigurieren die gewünschte Version. In den Zeilen 27 bis 31 wird die gleiche Aktion für Java durchgeführt. Dabei werden bei den Versionen ebenfalls die Matrix-Variablen verwendet. Für jede der Java-Versionen (8, 11, 17) wird ein eigener Container erstellt, jeweils mit einem der Betriebssysteme. Das Testen der verschiedenen Versionen der Programmiersprachen entspricht der Anforderung der Unterstützung unterschiedlicher Versionen von Programmiersprachen. Das Gleiche gilt für Python. Da eine VSC-Extension auf Node.js basiert, wird Node.js ebenfalls in den Zeile 19 bis 22 in-

[1] Betriebssystem

stalliert. Nach Durchführung der notwendigen Vorbereitungen wird ab Zeile 32 die eigentliche Testung durchgeführt. Mittels `npm install` können alle notwendigen Komponenten installiert werden. Unter Linux ist die Verwendung eines X-Servers erforderlich, um grafische Anwendungen auszuführen. Ein Framebuffer mit virtuellem Speicher kann mithilfe von *xvfb* emuliert werden, wie im Manual von [Levente Polyák, Aaron Griffin und Judd Vinet 2023] beschrieben. Unter MacOS und Windows ist diese Emulation nicht notwendig, daher reicht dort der einfache Befehl `npm test` aus.

```yaml
name: VSCode Extension Action

on:
  push:
    branches:
      - main

jobs:
  build:
    strategy:
      matrix:
        os: [macos-latest, ubuntu-latest, windows-latest]
        java: [ '8', '11', '17' ]
        python: [ '3.8', '3.10', '3.11' ]
    runs-on: ${{ matrix.os }}
    steps:
    - name: Checkout
      uses: actions/checkout@v3
    - name: Install Node.js
      uses: actions/setup-node@v3
      with:
        node-version: 19.x
    - name: Set up Python
      uses: actions/setup-python@v3
      with:
        python-version: ${{ matrix.python }}
    - name: Set up JDK
      uses: actions/setup-java@v3
      with:
        java-version: ${{ matrix.java }}
        distribution: 'temurin'
    - run: npm install
    - run: xvfb-run -a npm test
      if: runner.os == 'Linux'
    - run: npm test
      if: runner.os != 'Linux'
```

Listing 9.8: GitHub Action Workflow

9.4.2 Einschränkungen von GitHub Action

Beim Testen mithilfe dieses Workflows treten jedoch einige Schwierigkeiten auf. Als eines der wichtigsten Probleme ist zu erwähnen, dass Windows nicht innerhalb des Containers funktioniert. Beim Erzeugen des Containers tritt folgender Fehler auf:

```
This version of D:\a\ProgramFlow-Visualization\ProgramFlow-
Visualization\.vscode-test\vscode-win32-x64-archive-1.81.1
\Code.exe is not compatible with the version of Windows you're
running. Check your computer's system information and then
contact the software publisher.
```

Da auch in der Dokumentation von [Microsoft 2023a] `windows-latest` für Windows verwendet wird, wie im verwendeten Workflow, scheint es sich um einen temporären Fehler zu handeln, der wahrscheinlich mit zukünftigen Updates behoben wird.

Auch gibt es teilweise unerwartete Ausfälle des Dienstes, wie in Abbildung 9.1 zu sehen ist.

Abbildung 9.1: Downloadfehler des Repository innerhalb der Container

Die Tests konnten aufgrund von Problemen beim Herunterladen der Aktion `actions/checkout` nicht durchgeführt werden. Dieses Problem erwies sich jedoch als vorübergehend und wurde innerhalb weniger Stunden von GitHub selbst behoben. Während der Entwicklungsphase und des Vertrauens in die CI, um zuverlässige Statusberichte zu liefern, war dies dennoch nicht positiv hervorzuheben.

Ein weiteres Problem ist, dass einige Testcontainer gelegentlich fehlschlagen. Dies tritt jedoch nicht bei jedem Durchlauf des Workflows auf. Es gibt Zeiträume, in denen der Workflow problemlos durchgeführt wird. In anderen Fällen laufen nur

wenige Testcontainer durch und erst nach erneuter Ausführung der fehlgeschlagenen Container können diese erfolgreich abgeschlossen werden.

Ebenfalls gibt es ein Problem beim Testen des Java-Debuggers. Die Java-Tests können zwar durchgeführt werden, jedoch nur dann, wenn keine anderen Tests parallel ausgeführt werden. Wenn mehrere *BackendSessionen* getestet werden, wie beispielsweise sowohl Java als auch Python, schlägt der Java-Debugger fehl. Wenn jedoch alle Tests außer den Java-Backend-Tests ausgelassen werden, verlaufen sie problemlos. Dieses Problem tritt lokal nicht auf, was darauf hinweist, dass das Debuggen in einem nicht eigenen Container Schwierigkeiten bereitet. An dieser Stelle wird nicht weiter auf das Problem eingegangen. Vorübergehend wird die Lösung angewendet, die Java-spezifischen Tests mithilfe von `.skip` auszulassen. Bei der lokalen Entwicklung können diese Tests durch Entfernen von `.skip` wieder aktiviert und getestet werden.

10 Evaluierung mit Aufgaben aus Vorlesungen

In diesem Kapitel wird, nachdem auf die Architektur, die verschiedenen Implementierungen und das Testen des Frameworks eingegangen wurde, die praktische Funktionsweise des Frameworks erläutert. Dazu werden Aufgaben aus der Vorlesung von Prof. Dr. Stefan Wehr mit dem Framework getestet. Zunächst werden zwei Aufgaben in Python betrachtet, gefolgt von zwei Aufgaben in Java. Die Aufgaben sollen einen umfassenden Einblick in verschiedene komplexe Verhaltensweisen des Frameworks geben.

10.1 Python Aufgaben

In diesem Abschnitt werden zwei Python-Aufgaben behandelt. Die erste Aufgabe befasst sich mit Listen und Dictionaries, während die zweite Aufgabe gemischte Datentypen behandelt.

10.1.1 Listen und Dictionaries

Bei der Aufgabe zur Verwendung von Listen und Dictionaries geht es darum, das Verständnis von Key-Value-Paaren zu vermitteln und die wichtigsten Operationen zu demonstrieren. Diese Thematik wird anhand eines praxisnahen Beispiels eines vereinfachten Online-Shops erläutert. Zuerst werden alle Artikel in einer Liste gespeichert und anschließend in einem Dictionary. Der zugrunde liegende Code basiert auf einer intern an der Hochschule entwickelten Bibliothek namens *wypp*, welche von [Wehr 2022] entwickelt wird. *Write Your Python Program* bietet eine einfache und benutzerfreundliche Umgebung für Python-Anfänger, die durch eine ver-

einfachte Syntax die Grundlagen der Programmierung leichter verständlich macht. Der verwendete Code zur Veranschaulichung der Arbeit mit Dictionaries ist im folgenden Listing 10.1 dargestellt.

```
1  from __future__ import annotations
2  from wypp import *
3
4  @record
5  class Article:
6      articleId: str
7      title: str
8      description: str
9      price: float
10
11 trampolin = Article('871A05', 'Trampolin', 'Ein exklusives Trampolin', 76.54)
12 fussball = Article('17X81', 'Derbystar Fussball', 'Bundesliga Spielball', 21.8)
13 bohrstaender = Article('091GH7', 'Bohrstaender', 'Profi-Bohrstaender der Firma
       Wabeco', 120.9)
14 articleCatalog = [trampolin, fussball, bohrstaender]
15
16 def searchForArticle(catalog: list[Article], articleId: str) -> Optional[Article
       ]:
17     for x in catalog:
18         if x.articleId == articleId:
19             return x
20
21 check(searchForArticle(articleCatalog, '4711'), None)
22 check(searchForArticle(articleCatalog, '17X81'), fussball)
23
24 def searchForArticleWithDict(catalog: dict[str, Article], x: str) -> Optional[
       Article]:
25     return catalog.get(x)
26
27 articleDb = dict([(a.articleId, a) for a in articleCatalog])
28
29 check(searchForArticleWithDict(articleDb, '4711'), None)
30 check(searchForArticleWithDict(articleDb, '17X81'), fussball)
```

Listing 10.1: Beispielprogramm für Listen und Dictionaries als Aufgabe aus Vorlesung

Diese Aufgabe dient dazu, zu zeigen, dass externe Bibliotheken importiert werden können. In diesem Fall handelt es sich um *wypp*. Darüber hinaus wird eine Liste mit komplexeren Datentypen erstellt, in diesem Fall Klassen als *Records*. Neben der Liste wird auch ein Dictionary erstellt, das ebenfalls *Records* enthält. Im Anschluss werden nach und nach die Visualisierung und der damit verbundene Code aufeinander abgestimmt.

Zuerst werden die Zeilen 4 bis 9 des Codes betrachtet. Hier wird ein Artikel als *Records*-Objekt definiert. Nachfolgend werden drei Artikel mit den entsprechenden

Parametern erstellt und in einer Liste gespeichert. Die Visualisierung dieses Zustands in Zeile 16, nachdem all diese Objekte erzeugt wurden, ist in Abbildung 10.1 zu sehen.

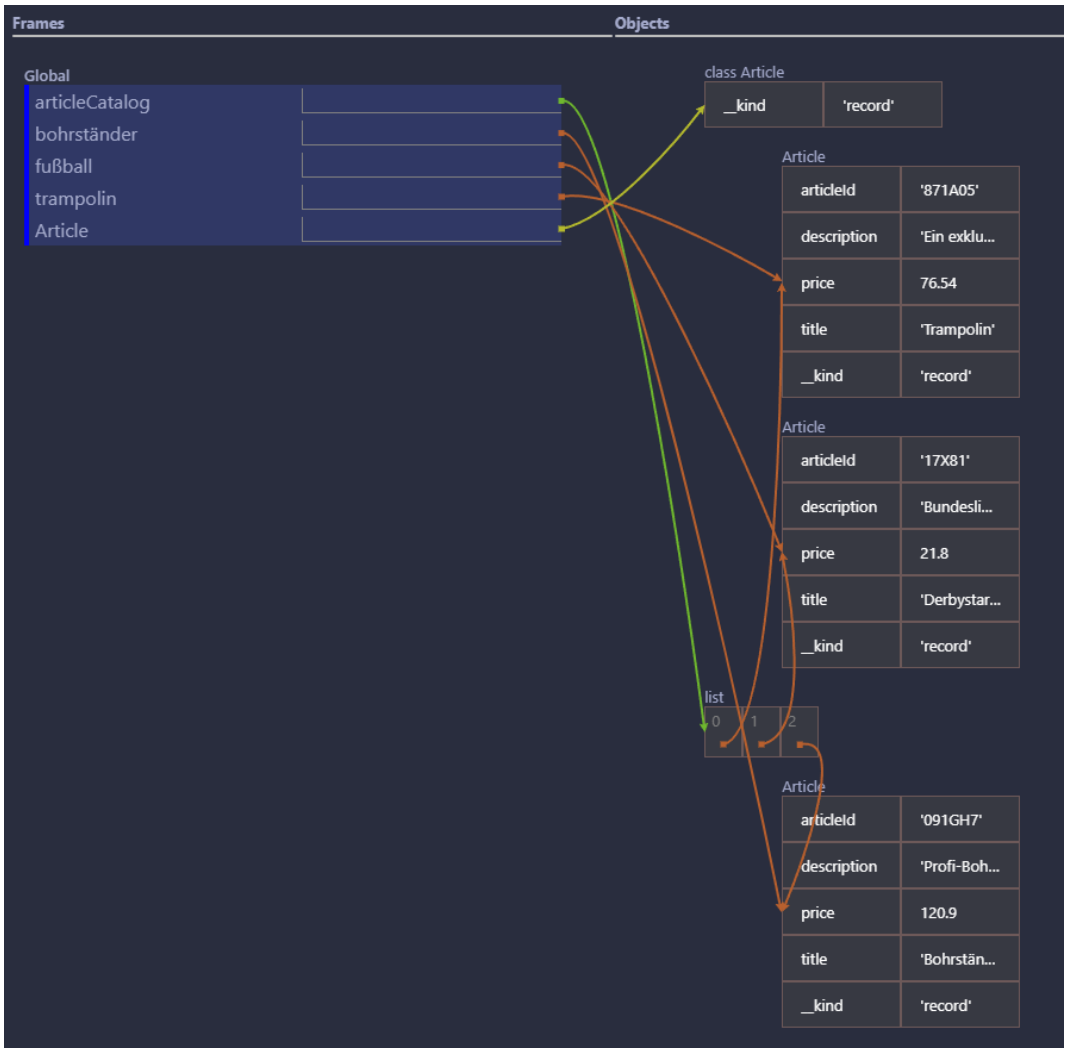
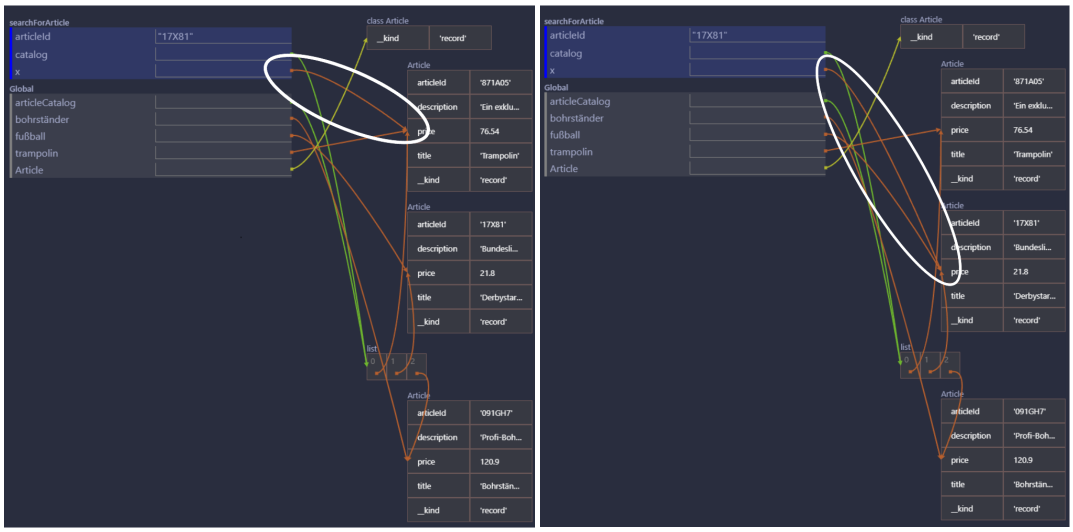


Abbildung 10.1: Darstellung Zeile 16 aus Listing 10.1

Hier ist zu erkennen, dass der *Stack* die Artikel *bohrstaender*, *fussball* und *trampolin* sowie die Liste *articleCatalog*, in der diese Artikel gespeichert sind, enthält. Neben diesen selbst erstellten Variablen ist auch die Definition des *Records*-Objekts als eigenes Objekt im *Heap* erkennbar. Der Parameter `__kind` gibt an, dass es sich um ein *Records*-Objekt handelt. Dies ist auch in jeder der definierten Artikel zu sehen, wobei der letzte Parameter darauf hinweist. Schließlich wird durch die Pfeile der Referenz verdeutlicht, dass es sich bei den Objekten in der Liste um dieselben handelt wie in den zuvor definierten Variablen. Der Grund dafür ist, dass die *Stack*-Variable auf denselben Artikel verweist wie der angegebene Index in der Liste. Es

ist also deutlich ersichtlich, dass Klassen, Listen und mehrfache Referenzen korrekt und verständlich dargestellt werden können.

Als Nächstes wird der Aufruf in Zeile 27 der Funktion `searchForArticle` betrachtet. Hier wird eine Schleife durchlaufen, die die Elemente der Liste der Artikel enthält. Da das zweite Element das gesuchte ist, wird diese Schleife zweimal durchlaufen, aber mit mehr als zwei Schritten. Um zu zeigen, wie diese Funktion im Allgemeinen dargestellt wird, werden zwei Schritte betrachtet. Diese finden nach der Überprüfung der *if*-Anweisung in Zeile 23 statt, wenn wieder zu Zeile 22 gesprungen wird, da dies nicht das gesuchte Element ist. Diese beiden Schritte von Zeile 23 zu Zeile 22 sind in Abbildung 10.2 dargestellt.



(a) Darstellung Zeile 23 aus Listing 10.1 **(b)** Darstellung Zeile 22 aus Listing 10.1

Abbildung 10.2: Darstellung Zeile 23 und 22 aus Listing 10.1

Hier ist zu sehen, dass neben dem *Global Stack*, der bereits in Abbildung 10.1 zu sehen ist, zusätzlich der *Stack* für `searchForArticle` angezeigt wird. Dieser enthält die relevanten Variablen, die für diese Funktion erforderlich sind. Dazu gehören die übergebenen Parameter `articleId` und `catalog` sowie das Element der Schleife `x`. Auch hier ist zu erkennen, dass der übergebene `catalog` innerhalb der Funktion dieselbe Referenz teilt wie die erstellte Variable. Der einzige, aber relevante Unterschied zwischen den Abbildungen (a) und (b) ist, dass `x` die Referenz wechselt. Die gewechselte Referenz ist durch die weiße Umrandung des Pfeils hervorgehoben.

Der letzte darzustellende Schritt, der Endzustand, ist in Abbildung 10.3 dargestellt.

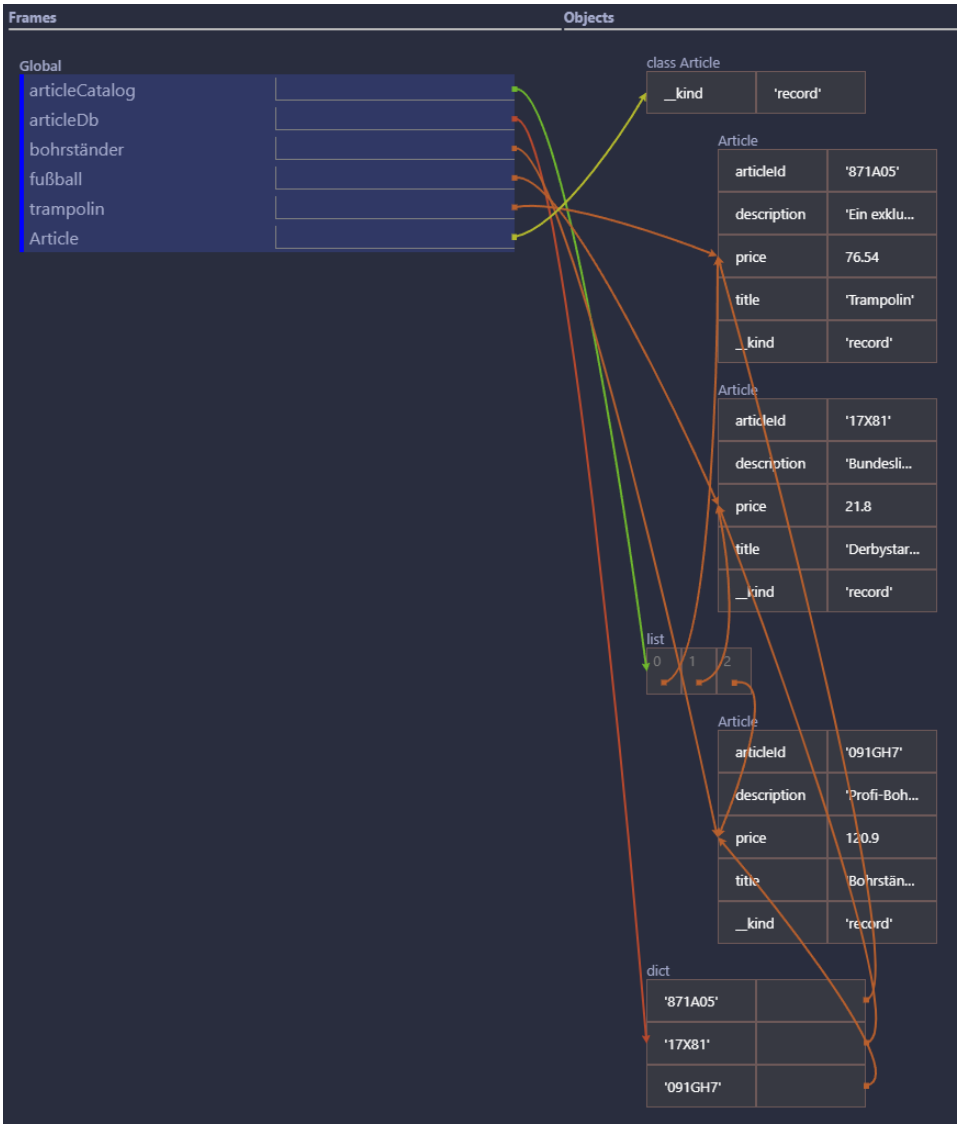


Abbildung 10.3: Darstellung Zeile 30 aus Listing 10.1

Hier ist zu erkennen, dass nun auch ein Dictionary dargestellt wird. Dieses enthält, wie auch die Liste, eine Referenz auf die einzelnen Artikel. Die präsentierte Darstellung verdeutlicht die Key-Value-Paare.

Im Gesamten zeigt dieses Beispiel auf, dass komplexe Datentypen und mehrfache Referenzen zu einem Objekt korrekt und verständlich dargestellt werden können. Es zeigt auch, dass die Integration von externen Bibliotheken die Visualisierung nicht beeinträchtigt. Darüber hinaus verdeutlicht dieses Beispiel, dass auch Funktionen und deren Abläufe dargestellt werden können. Das Hauptproblem besteht jedoch in der Effizienz. Um diese Visualisierung mit 38 Schritten zu erstellen, hat das Framework knapp eine Minute benötigt.

10.1.2 Gemischte Daten

Das zweite Beispiel behandelt gemischte Daten. Der Fokus liegt auf der Modellierung von Objekten mit verschiedenen Ausprägungen und wie mit einer Fallunterscheidung über gemischte Daten in der Programmierung umgegangen werden kann. Als Beispiel werden hier Tiere verwendet. Im folgenden Listing 10.2 wird das Beispielprogramm dargestellt.

```
from __future__ import annotations
from wypp import *

@record
class Papagei:
    gewichtsklasse: Gewichtsklasse
    satz: str

Gewichtsklasse = Literal['leicht', 'mittel', 'schwer']

mina = Papagei('mittel', "Let's go to the punkrock show")
robin = Papagei('leicht', 'oh no, not again')

@record
class Gürteltier:
    gewicht: int
    totOderLebendig: Status

Status = Literal['tot', 'lebendig']

dora = Gürteltier(25000, 'lebendig')
daniel = Gürteltier(30000, 'tot')

Tier = Union[Gürteltier, Papagei]

def wiegeTier(animal: Tier) -> int:
    if isinstance(animal, Gürteltier):
        return animal.gewicht
    elif isinstance(animal, Papagei):
        return gewichtFürGewichtsklasse(animal.gewichtsklasse)

def gewichtFürGewichtsklasse(weightClass: Gewichtsklasse) -> int:
    if weightClass == 'leicht':
        return 900
    elif weightClass == 'mittel':
        return 1200
    else:
        return 1600

check(gewichtFürGewichtsklasse('leicht'), 900)
check(gewichtFürGewichtsklasse('mittel'), 1200)
check(gewichtFürGewichtsklasse('schwer'), 1600)

check(wiegeTier(dora), 25000)
```

```
45  check(wiegeTier(robin), 900)
```

Listing 10.2: Beispiel für gemischte Daten als Aufgabe aus Vorlesung

Es ist zu erkennen, dass zwischen zwei Tierarten unterschieden wird: Papageien, Zeilen 4 bis 7 und Gürteltiere, Zeilen 14 bis 17. Diese werden in Zeile 24 zu einem Tier in Form einer *Union* zusammengeführt. Weitere Besonderheiten sind die Literale. In Zeile 9 wird die Gewichtsklasse und in Zeile 19 der Status definiert. Zuerst wird betrachtet, wie der Datentyp *Literal*, der durch die Bibliothek *wypp* hinzugefügt wurde, behandelt wird. Dies ist in Abbildung 10.4 dargestellt.

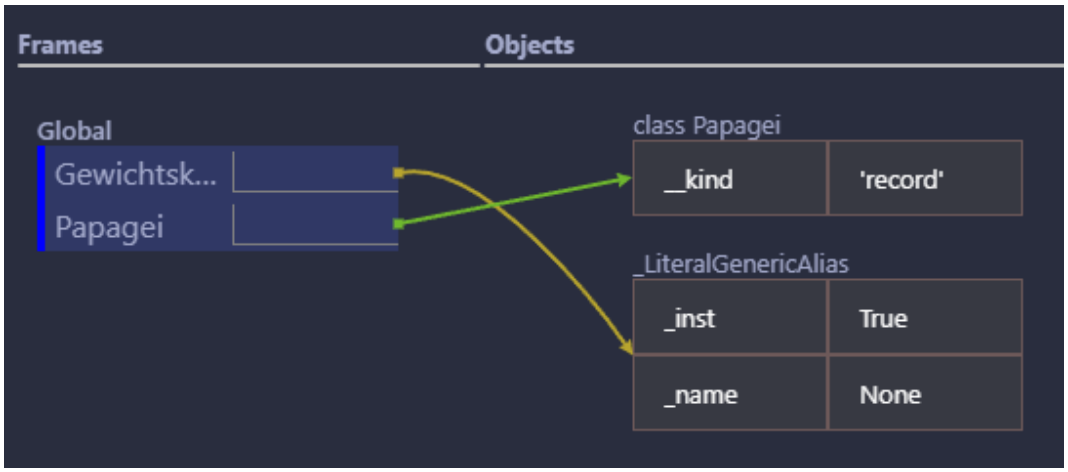


Abbildung 10.4: Darstellung Zeile 9 aus Listing 10.2

Es fällt sofort auf, dass die erwarteten Werte *leicht*, *mittel* und *schwer* nicht dargestellt werden, sondern ein *_LiteralGenericAlias*. Dieser hat zwei Parameter, `_inst` und `_name`, die jedoch nichts mit der Gewichtsklasse zu tun haben. Für genauere Betrachtung des Grundes, wird der Debugger-Output betrachtet. Dieser ist in Abbildung 10.5 dargestellt.

```
1  {
2      evaluateName: 'Gewichtsklasse',
3      name: 'Gewichtsklasse',
4      type: '_LiteralGenericAlias',
5      value: 'typing.Literal[
6          'leicht', 'mittel', 'schwer'
7        ]',
8      variablesReference: 57
9  }
```

(a) Debugger-Output Literal

```
1  {
2      name: '_LiteralGenericAlias',
3      type: 'instance',
4      value: Map(2) {
5        _inst => {type: 'bool', value: '
             True'},
6        _name => {type: 'none', value: '
             None'}},
7  }
```

(b) Auflösung Debugger-Output Literal

Abbildung 10.5: Auswertung des Debugger-Outputs für ein Literal

Hierbei lässt sich feststellen, dass nur im ersten Debugger-Output der gewünschte Wert als String vorhanden ist. Die Auflösung beinhaltet jedoch alle bereits angesprochenen Parameter. Das bedeutet, dass das Framework richtig mit den Daten umgehen kann, die der Debugger liefert. Dasselbe Verhalten gilt auch für ein *Union*.

Als Nächstes wird ein Record betrachtet, der mit dem gerade vorgestellten Literal definiert ist. Dies ist in Abbildung 10.6 zu sehen.

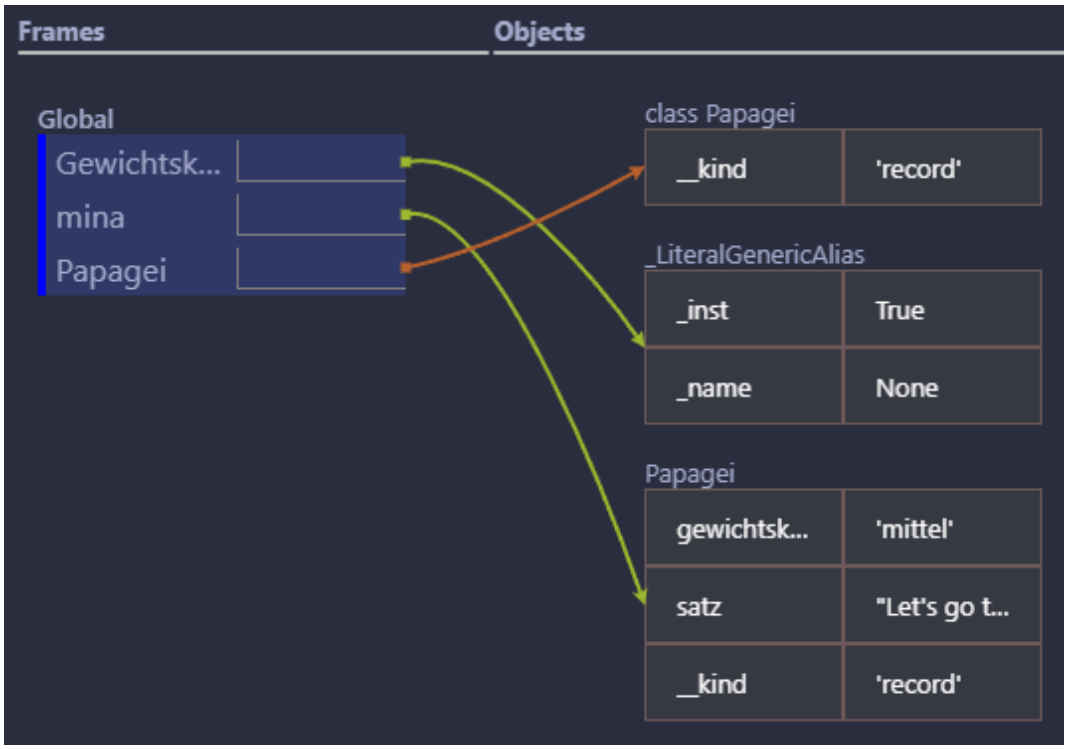


Abbildung 10.6: Darstellung Zeile 11 aus Listing 10.2

Hier ist zu erkennen, dass die Gewichtsklasse mit ihrem Wert korrekt dargestellt wird. Dies bedeutet, dass die Darstellung nur bei der Definition nicht wie erwartet ist.

Zuletzt wird noch Zeile 32 betrachtet, wenn die Funktion `wiegeTier` für *robin* aufgerufen wird. Hierbei handelt es sich um einen Funktionsaufruf innerhalb einer Funktion, was ebenfalls ein interessanter zu untersuchender Aspekt ist. Dies wird in Abbildung 10.7 dargestellt.

Zu erkennen ist, dass neben `wiegeTier` auch noch `gewichtFürGewichtsklasse` dargestellt wird. Die aktuell ausgeführte Funktion wird an oberster Stelle angezeigt. Ebenfalls wird der übergebene Parameter dargestellt.

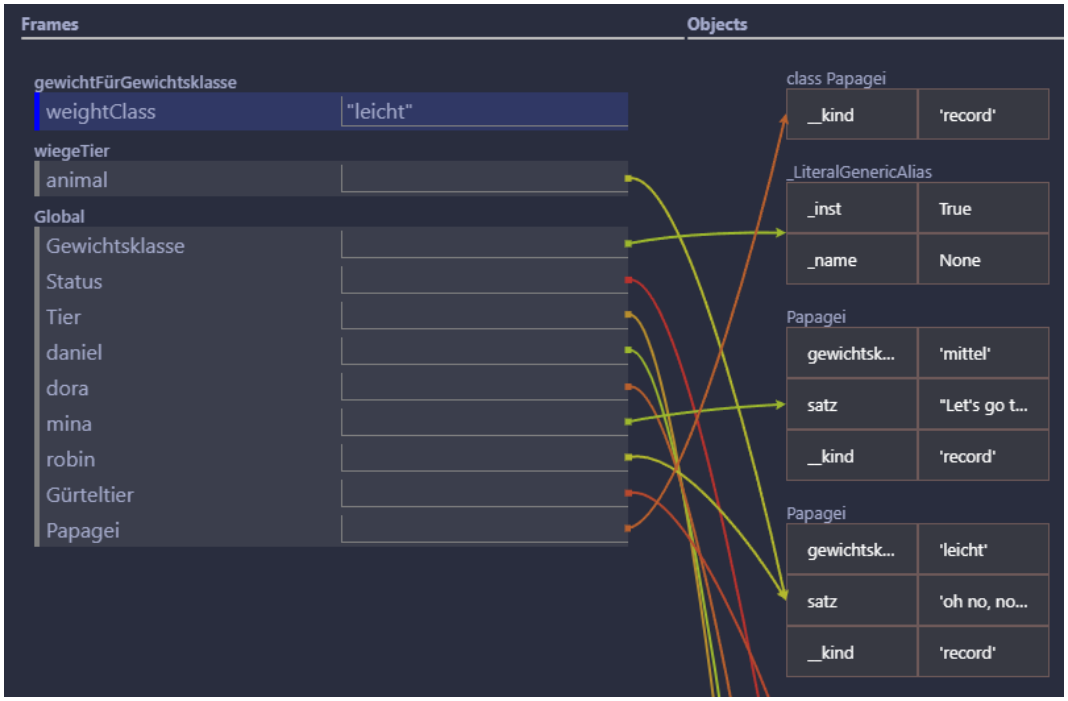


Abbildung 10.7: Darstellung Zeile 32 aus Listing 10.2

Auch dieses Beispiel verdeutlicht, dass komplexe Datentypen und mehrfache Referenzen zu einem Objekt korrekt und verständlich dargestellt werden können. Es zeigt auch, dass mit externen Datentypen umgegangen werden kann, obwohl die Darstellung derselben abhängig vom Debugger variieren kann. Die Tatsache, dass mehrere Funktionen ineinander verschachtelt aufgerufen werden können und dies sichtbar dargestellt wird, ist ebenfalls hervorzuheben. Das Hauptproblem bleibt jedoch die Effizienz. Um diese Visualisierung mit 41 Schritten zu erstellen, hat das Framework etwa 45 Sekunden benötigt.

10.2 Java Aufgaben

In diesem Abschnitt werden zwei Java-Aufgaben behandelt. Die erste Aufgabe beschäftigt sich mit Polynomen, während die zweite Aufgabe das Konzept von Klassen behandelt.

10.2.1 Polynome

Die erste Aufgabe befasst sich mit der Darstellung und Berechnung von Polynomen in Java. Das Ziel ist es, den Studierenden die Konzepte der Listen und Iterationen näherzubringen, während sie gleichzeitig die Repräsentation von Polynomen verstehen. Auch die Verwendung von Importanweisungen in Java wird näher betrachtet. Im untenstehenden Beispielcode Listing 10.3 ist die dazugehörige Aufgabe dargestellt. Dabei wurde diese auf die für die Evaluation relevanten Teile gekürzt.

```
1  package poly;
2
3  import hso.*;
4  import java.util.*;
5
6  class PolyMainOwn {
7
8      static double polyEval(List<Double> p, double sigma) {
9          double result = 0;
10         for (int i = 0; i < p.size(); i++) {
11             double a = p.get(i);
12             result = result + a * Math.pow(sigma, i);
13         }
14         return result;
15     }
16
17     public static void main(String[] args) {
18         List<Double> q = JSimple.makeList(1.0, 2.0, 3.0);
19
20         double eval = polyEval(q, 2);
21     }
22 }
```

Listing 10.3: Beispielprogramm für Polynome als Aufgabe aus Vorlesung

In der *main*-Methode, die von Zeile 17 bis 21 definiert ist, wird zuerst eine Liste von *Double*-Werten erstellt. Diese Liste wird mithilfe der Hilfsfunktion `JSimple.makeList()` erzeugt, die durch den Import in Zeile 3 zur Verfügung steht. Dies zeigt einen interessanten Aspekt des Codes: Die Verwendung von importierten Methoden. Abbildung 10.8 zeigt, wie dies während der Ausführung in Zeile 18 aussieht.

In der Abbildung ist zu erkennen, dass nicht mehr die Datei *PolyMainOwn.java* angezeigt wird, sondern *JSimple.class*. Das Framework öffnet zusätzlich zur aktuell auszuführenden Datei auch die Dateien, aus denen Methoden importiert wurden. Dabei bleibt das Hervorheben der aktuellen und nächsten auszuführenden Zeile erhalten. Zu sehen ist weiterhin, wie eine ArrayList, ein *Double*-Array sowie

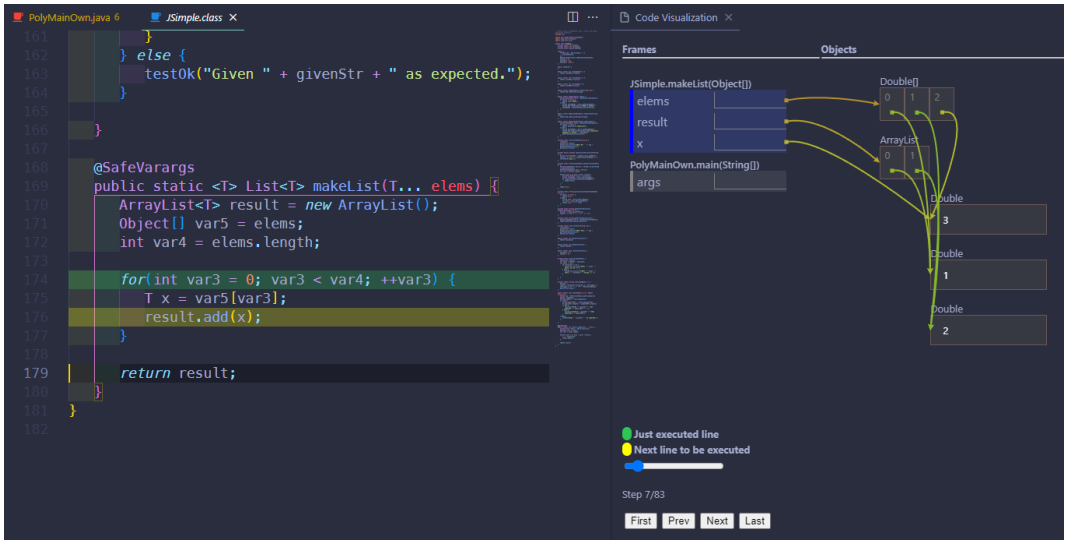


Abbildung 10.8: Darstellung Aufruf innerhalb Zeile 18 aus Listing 10.3

ein *Double* dargestellt werden. Der Übergabeparameter der aufgerufenen Methode aus *JSimple* besteht aus einem *Varargs* `T... elems`. Da in Zeile 18 *Double*-Werte übergeben werden, werden diese als *Double*-Array interpretiert und entsprechend visualisiert. Deutlich wird, dass ein *Double* ein Objekt ist, das nicht direkt in einem Array gespeichert wird, sondern zusätzliche Referenzen auf das jeweilige *Double* enthält. Die `makeList()`-Methode liefert eine ArrayList als Ergebnis zurück. Die Darstellung einer solchen ArrayList ähnelt der Darstellung einer normalen Liste oder eines Arrays. Auch hier werden die gespeicherten *Double*-Werte als Referenzen angezeigt. Es ist ebenfalls zu erkennen, dass alle auf dasselbe *Double*-Objekt verweisen. Damit wird verdeutlicht, dass in diesem Fall nicht mehrfach dasselbe *Double*, wie zum Beispiel die Zahl 2, im *Heap* gespeichert wird, sondern dasselbe *Double* mehrfach referenziert wird.

Das erzeugte Polynom wird in Zeile 20 mit einem Wert von 2 ausgewertet. Die dafür geschriebene Funktion ist in den Zeilen 8 bis 15 definiert. Als Beispiel wird der erste Durchlauf in Zeile 10 betrachtet. Die Visualisierung dieses Schritts ist in Abbildung 10.9 dargestellt.

Alle aktuell verwendeten Variablen, die zu diesem Zeitpunkt im Code relevant sind, werden in der Abbildung übersichtlich dargestellt. Hier wird erneut deutlich, dass beide ArrayLists auf dasselbe Element verweisen und nicht jede ArrayList jeweils eigene *Double*-Objekte enthält.

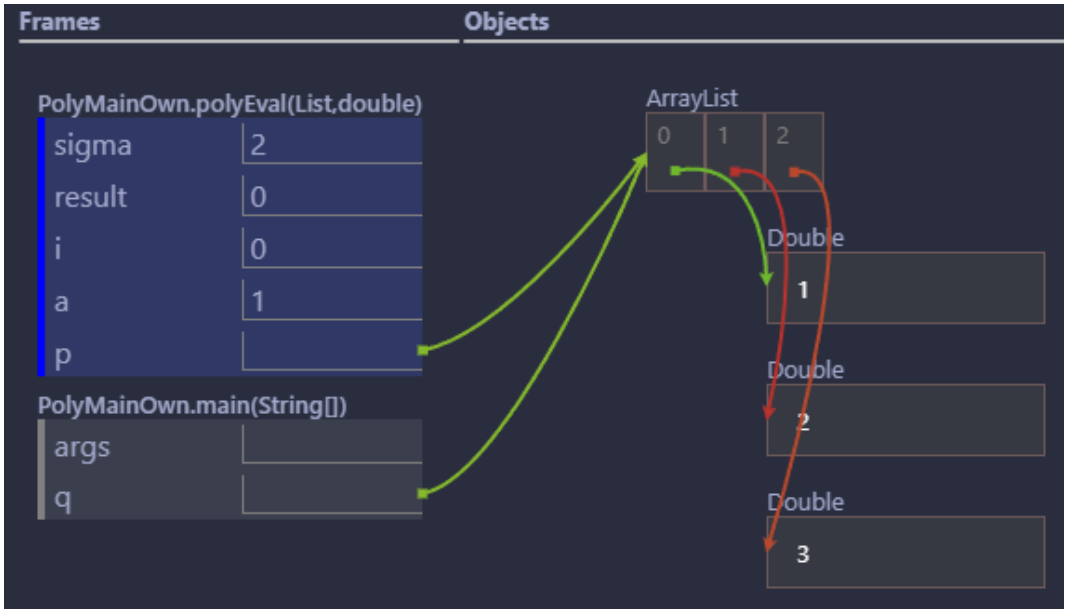


Abbildung 10.9: Darstellung Zeile 10 aus Listing 10.3

Im Verlauf der restlichen Ausführung werden keine neuen Konzepte oder Elemente eingeführt, die über das bisher Beschriebene hinausgehen. Daher wird hier nicht weiter darauf eingegangen.

Dieses Beispiel verdeutlicht, dass nicht nur Listen und Objekte visualisiert werden können, sondern auch das Datei-übergreifende Ausführen unterstützt wird. Die jeweils aktuell ausgeführte Datei wird geöffnet und die Visualisierung und Hervorhebung werden innerhalb dieser Datei fortgeführt. Ein Problem, das auch bei den Python-Dateien erwähnt wurde, ist die Effizienz. Für die Visualisierung dieser 25 Zeilen hat das Framework etwa 36 Sekunden benötigt.

10.2.2 Klassen

Das letzte Beispiel behandelt das Konzept von objektorientierten Klassen und deren Veränderung. Dabei wird erläutert, was Klassen sind, wie ein Konstruktor funktioniert, und was Kapselung bedeutet. Als Beispiel dient die Simulation eines Bankkontos, das einen Kontostand hat, der verändert werden kann. Der Beispielcode ist in Listing 10.4 dargestellt.

```
1  package bankingClassesMutable;
2
3  import hso.*;
4
5  class MutableBankAccount {
6      private double balance;
7      MutableBankAccount(double balance) {
8          if (balance < 0) {
9              throw JSimple.impossible("negativer Kontostand");
10         }
11         this.balance = balance;
12     }
13     double getBalance() {
14         return this.balance;
15     }
16     void withdraw(double amount) {
17         double newBalance = this.balance - amount;
18         if (newBalance < 0) {
19             throw JSimple.impossible("negativer Kontostand nach Abhebung");
20         }
21         this.balance = newBalance;
22     }
23 }
24
25 class BankingClassesMutableMain {
26     public static void main(String[] args) {
27         MutableBankAccount acc = new MutableBankAccount(100);
28         acc.withdraw(30);
29         System.out.println(acc.getBalance());
30     }
31 }
```

Listing 10.4: Beispielprogramm für Klassen als Aufgabe aus Vorlesung

In diesem Beispiel wird von Zeile 5 bis 23 die Klasse *MutableBankAccount* definiert und anschließend innerhalb der *main*-Methode von Zeile 26 bis 30 verwendet. Zuerst wird ein Bankkonto erstellt und mit einem Anfangsguthaben von 100 initialisiert. Wie der Zustand nach Zeile 27 aussieht, ist in Abbildung 10.10 dargestellt.

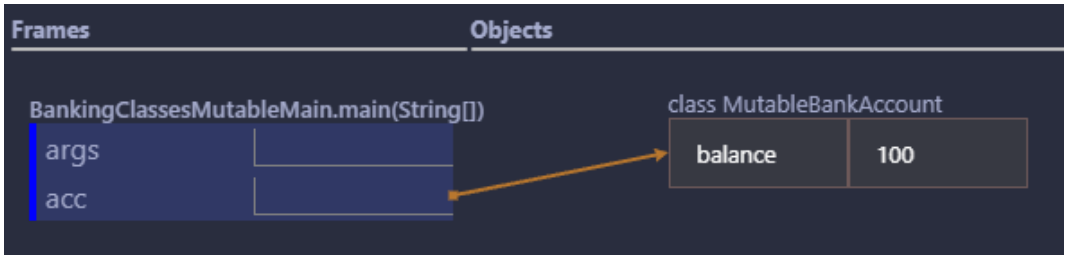


Abbildung 10.10: Darstellung Zeile 27 aus Listing 10.4

Die Abbildung zeigt, dass die Klasse mit den einzelnen gespeicherten Variablen dargestellt wird. Dabei wird der Typ der Variable nicht explizit angezeigt, sondern nur der Name *balance* zusammen mit dem Wert 100.

Da bereits gezeigt wurde, wie Methodenaufrufe dargestellt werden, wird in diesem Beispiel nicht weiter auf einzelne Aspekte eingegangen.

Zusammenfassend zeigt dieses Beispiel, dass Klassen und deren Zustand visualisiert werden können. Auch das Erzeugen von Klassenobjekten wird unterstützt und ist aufgrund der geringeren Komplexität dieses Beispiels effizienter. Die Erzeugung der Visualisierung dauert etwa 5 Sekunden.

11 Diskussion

In diesem Kapitel werden die Ergebnisse und Fortschritte der Arbeit bewertet. Anschließend wird ein Ausblick auf weitere Arbeiten zu diesem Thema gegeben.

11.1 Evaluation

Das Ziel dieser Arbeit war die Entwicklung eines generischen Frameworks zur Visualisierung von Programmabläufen, das Programmieranfängern das Verständnis und den Einstieg in die Programmierung erleichtern soll. Dazu wurden klare Anforderungen definiert, die als Leitfaden dienten. Diese Anforderungen erwiesen sich als sehr hilfreich bei der Bewertung bestehender Lösungen. Der so geschaffene Rahmen konnte erfolgreich genutzt werden, um herauszufiltern, welche bestehenden Lösungen wie gut in das Konzept passen. Die Erfüllung der Anforderungen kann in den Kategorien *nein*, *bedingt* und *ja* beschrieben werden, was eine übersichtliche Darstellung der Untersuchungsergebnisse ermöglichte. Eine Zusammenfassung dieser Untersuchung findet sich in Tabelle 11.1.

Anforderung	Debug Visualizer	Python Tutor
Handhabung und Bedienbarkeit	nein	bedingt
Darstellung	bedingt	bedingt
Effizienz	bedingt	ja
Unterstützung verschiedener Programmiersprachen	ja	ja
Unterstützung unterschiedlicher Versionen von Programmiersprachen	ja	nein

Tabelle 11.1: Erfüllung der Anforderung von existierender Lösungen

Da beide vorhandenen Lösungen mindestens eine *nein* oder *bedingt*-Bewertung aufweisen, können sie nicht als ausreichend betrachtet werden. Dies unterstreicht die Notwendigkeit einer Alternative, was in einer Vorarbeit in diesem Bereich resultier-

te. In dieser Vorarbeit werden vielversprechende Ansätze aufgezeigt, wie beispielsweise die Nutzung des DAP und dessen Integration als Erweiterung in VSC. Es hat sich gezeigt, dass dies äußerst sinnvoll ist und weiterverfolgt werden sollte.

Im Rahmen dieser Arbeit wurde die Architektur überarbeitet und die Code-Komponenten erweitert. Diese Maßnahmen erwiesen sich als notwendig und sinnvoll, da sie die weitere Entwicklung erheblich erleichterten. Die neue Architektur ermöglicht zudem eine einfache Erweiterung auf andere Programmiersprachen, was bereits bei der erfolgreich durchgeführten Integration von Java gezeigt wurde.

Die Überarbeitung und Erweiterung der vorherigen Arbeit zeigt, dass bedeutende Fortschritte erzielt wurden. In Tabelle 11.2 werden die Zustände vor und nach dieser Masterthesis verglichen.

Anforderung	Vorheriger Stand	Aktueller Stand
Handhabung und Bedienbarkeit	bedingt	ja
Darstellung	bedingt	ja
Effizienz	bedingt	bedingt
Unterstützung verschiedener Programmiersprachen	nein	ja
Unterstützung unterschiedlicher Versionen von Programmiersprachen	ja	ja

Tabelle 11.2: Gegenüberstellung der Anforderung vor und nach der Masterthesis

Es zeigt sich, dass die Handhabung, Bedienbarkeit und Darstellung von *bedingt* zu *ja* erfolgreich verbessert wurden. Bei der Darstellung ist jedoch aufgefallen, dass teilweise eine nicht vorteilhafte Visualisierung ermöglicht wird, wie zum Beispiel die HashMap in Abbildung 8.5 zeigt. Der Einsatz innerhalb einer Vorlesung ist daher nur sinnvoll, wenn solche Variablen mit dem aktuellen Stand des Frameworks vermieden werden. Zusätzlich konnte die Bewertung *nein* bezüglich der Unterstützung verschiedener Programmiersprachen auf *ja* geändert werden. Dabei ist zu beachten, dass sowohl Python als auch Java nun vollständig unterstützt werden und nicht nur bis zu einer bestimmten Komplexität, wie es bisher der Fall war. Außerdem wurde die Architektur neu strukturiert, um eine einfache Erweiterung zu ermöglichen. Die Einschränkung auf diese zwei Sprachen besteht jedoch.

Das Testen der Anwendung mittels CI gestaltete sich schwieriger als zunächst erwartet. Die aufgetretenen Probleme erforderten eine gewisse Zeit, jedoch wurden letztendlich Kompromisse gefunden, um eine solide Grundlage zu schaffen. Dadurch wird sichergestellt, dass die bestehende Logik auch bei zukünftigen Ände-

rungen weiterhin einwandfrei funktioniert. Das Testen während der Entwicklung hat sich als äußerst hilfreich erwiesen, da es klare Anforderungen definiert hat, die den Entwicklungsprozess unterstützten.

Zuletzt haben sich die ausgewählten Aufgaben, anhand derer das Framework evaluiert wurde, als sehr repräsentativ erwiesen. Sie ermöglichten sowohl die Überprüfung komplexer Verhaltensweisen in Java als auch in Python. Sie haben gezeigt, dass die Nutzung von Methoden aus anderen Dateien als der, die visualisiert wird, möglich ist und dass allgemeine Importe unterstützt werden. Diese Aufgaben illustrieren jedoch auch die Einschränkungen in Bezug auf die Effizienz. Es wurde deutlich, dass die Ausführungszeit erheblich zunimmt, je mehr Schritte durchlaufen werden. Insbesondere bei Java kann dies durch verschiedene Aspekte erklärt werden. Durch die generelle Durchführung eines *stepIn* tritt oft der Fall ein, dass ein *stepOut* erforderlich ist. Durch das *stepOut* wird erreicht, dass eine unrelevante Ebene verlassen werden kann. Das hat zur Folge, dass deutlich mehr Schritte ausgeführt werden, als später angezeigt werden. Auch der Wechsel auf eine andere Datei erfordert viel Zeit. Es ist jedoch auch auffällig, dass dies vom verwendeten Debugger abhängt. Der Vergleich des Python-Debuggers mit dem Java-Debugger zeigt deutliche Geschwindigkeitsunterschiede auf. Somit kann die Auswahl der Programmiersprache allein schon einen Einfluss auf die Effizienz haben.

11.2 Ausblick

Obwohl viele Aspekte des Frameworks erfolgreich implementiert und erweitert wurden, bleiben einige Anforderungen unerfüllt. Aus diesem Grund werden an dieser Stelle einige Anregungen und Vorschläge für die zukünftige Arbeit in diesem Bereich vorgestellt.

Es ist notwendig, die Unterstützung weiterer Programmiersprachen zu implementieren, um der Anforderung der *Unterstützung verschiedener Programmiersprachen* gerecht zu werden. Ein Schwerpunkt sollte auf den Programmiersprachen C und C++ liegen, wie bereits in Abschnitt 2.4 erläutert wurde.

Neben der Erweiterung von unterstützten Programmiersprachen könnte auch die Integration weiterer Darstellungsarten in Erwägung gezogen werden. Eine Möglichkeit wäre beispielsweise der Wechsel zwischen einer einfachen und einer komplexen Ansicht. Ein Beispiel hierfür sind Java-Wrapper. Anstatt jedes *Double* als

eigenes Objekt in der komplexen Ansicht darzustellen, könnte die einfache Ansicht diese als normales *double* darstellen, um eine übersichtlichere Darstellung zu ermöglichen. Gleichzeitig könnten sie passend markiert werden, um zu verdeutlichen, dass es sich nicht um einfache *Double*-Werte handelt.

Während der Darstellung sind einige Aspekte aufgefallen, die verbessert werden könnten. Zum Beispiel ist die Darstellung von HashMaps nicht ideal, wie in Abbildung 8.5 gezeigt wurde. Probleme traten auch bei der Darstellung von *Literal* und *Unions* auf, wie in Abschnitt 10.1.2 bei der Evaluation deutlich wurde. Um den Studierenden einen noch besseren Einblick zu ermöglichen, sollte die Darstellungen der Visualisierung überarbeitet werden.

Eine mögliche Erweiterung wäre die Option, erstellte *Traces* einfach zu importieren und exportieren. Dadurch könnte beispielsweise ein *Trace* eines langlaufenden Programms wie der Polynom-Aufgabe im Abschnitt 10.2.1 exportiert und an Dritte weitergegeben werden. Auf diese Weise könnte Zeit gespart werden, die ansonsten für die Visualisierung eines größeren Programms benötigt wird. Darüber hinaus könnte dies dazu verwendet werden, um systemabhängige Fehler zu testen. Dabei wird das *Trace*, welches auf einem System nicht funktioniert, auf einem anderen System getestet, um Framework-Fehler auszuschließen.

Ein weiterer wichtiger Aspekt betrifft die Entwicklungsumgebung. Derzeit unterstützt das Framework ausschließlich VSC als Entwicklungsumgebung. Da auch andere Entwicklungsumgebungen von Relevanz sein können, sollte die Integration dieses Frameworks in weitere Entwicklungsumgebungen in Betracht gezogen und angestrebt werden.

Zuletzt stellt auch die Performance des Frameworks ein Problem dar. Wie aus der *Evaluierung mit Aufgaben aus Vorlesungen* hervorgeht, dauert es länger als gewünscht, um den Programmcode zu visualisieren. Es ist daher notwendig, diesen Aspekt des Frameworks genauer zu untersuchen und eine mögliche Beschleunigung umzusetzen, um den Effizienzaspekt zu behandeln.

12 Fazit

In dieser Masterthesis mit dem Titel *Ein generisches Framework zur Visualisierung von Programmabläufen* wurde ein generisches Framework zur Visualisierung von Programmabläufen entwickelt. Das Ziel ist insbesondere, die Verständlichkeit und den Einstieg für Programmieranfänger zu vereinfachen.

Zu Beginn der Arbeit stand die klare Formulierung von Anforderungen, an denen sich im Verlauf der Arbeit orientiert wurde. Anhand dieser Kriterien wurden vorhandene Lösungen betrachtet. Als nennenswerte Lösungen stellten sich *Debug Visualizer* und der *Python Tutor* heraus. Es wurde festgestellt, dass keine dieser Lösungen den Anforderungen in vollem Umfang entspricht. Aus diesem Grund wurde bereits an einem Framework gearbeitet, das diese Anforderungen vollumfänglich erfüllen soll. Die Analyse dieser Vorarbeit zeigte jedoch, dass diese nur ein Grundgerüst ist, auf welchem aufgebaut werden kann.

Um ein solides Fundament für das Framework zu schaffen, wurden sowohl die Architektur als auch der Code umfassend überarbeitet. Dadurch wurde nicht nur eine einfachere Erweiterung auf verschiedene Programmiersprachen ermöglicht, sondern auch eine bessere Orientierung innerhalb des Codes. Die Implementierung der Erweiterung der Programmiersprachen Python und Java wurde erfolgreich umgesetzt und mit der Integration eines Test-Frameworks sowie einer CI ergänzt. Dadurch wurde eine solide Grundlage für zukünftige Entwicklungen geschaffen.

Die Evaluation des Frameworks anhand von Programmieraufgaben aus Lehrveranstaltungen zeigte, dass es in der Lage ist, die Aufgaben verständlich und korrekt zu präsentieren, ohne subjektive Bewertungen zu enthalten. Es wurde jedoch deutlich, dass die benötigte Zeit für die Ausführung von Programmen mit einer höheren Anzahl von Schritten noch verbessert werden muss.

Zusammenfassend kann diese Masterthesis als Erfolg und bedeutender Fortschritt betrachtet werden. Das Framework ist in der Lage, Aufgaben aus Java- und Python-

Vorlesungen zu bewältigen, was seinen Einsatz in Vorlesungen ermöglicht. Zunächst konzentriert sich der primäre Einsatzbereich auf die Nutzung durch Professoren, da die Generierung eines Traces noch zu zeitaufwendig ist. Dies ermöglicht jedoch bereits das Darstellen eines Programmablaufs während einer Vorlesung, um das Verständnis bereits in diesem Stadium zu vermitteln.

Abkürzungsverzeichnis

AI Angewandte Informatik 6
CI Continuous Integration 7
CSS Cascading Style Sheets 36
DAP Debugger Adapter Protocol 28
HTML Hypertext Markup Language 36
ID Identifikationsnummern 36
JSON JavaScript Object Notation 14
OS Betriebssystem 72
URL Uniform Resource Locator 20
UX User Experience 10
VSC Visual Studio Code 2

Tabellenverzeichnis

1.1 Beschreibung der Debugger-Steuerelemente in VSC 4

3.1 Auflistung der Programmiersprache und deren Debugger mit dem jeweiligen Supportlevel aus [Henning Dieterichs 2023] 17
3.2 Vergleich der Ausführungszeit in *Python Tutor* von Java und Python anhand einer einfachen float-Zuweisung 24
3.3 Unterstütze Sprachversionen von *Python Tutor*, herausgearbeitet aus [Python Tutor 2023a] 25
3.4 Erfüllung der Anforderung von existierender Lösungen 26

6.1 Kategorisierung der Datentypen von Java und Python 46

7.1 Erfüllung der Anforderung des Frameworks vor dieser Arbeit 52
7.2 Aktualisierung der Erfüllung der Anforderungen des Frameworks nach Verbesserung und Erweiterung für die Programmiersprache Python 56

8.1 Aktualisierung der Erfüllung der Anforderungen des Frameworks nach Erweiterung um die Programmiersprache Java 65

11.1 Erfüllung der Anforderung von existierender Lösungen 90
11.2 Gegenüberstellung der Anforderung vor und nach der Masterthesis . 91

Abbildungsverzeichnis

1.1 Debugger-Output für die Klasse Node 3
1.2 Debugger-Steuerelemente in VSC 3
1.3 Beispielhafte visuelle Darstellung eines Node aus [Python Tutor 2023a] 5

2.1 Bedienelemente der Visualisierung 8
2.2 Hervorheben der aktuellen und der nächst auszuführenden Zeile im Programmcode 9
2.3 Gegenüberstellung von textuellem und visuellem Debugger-Output . 10
2.4 Python Release Cycle aus [Python Software Foundation 2023] . . . 12

3.1 Nebeneinanderstellung von Programmcode und *Debug Visualizer* mit Visualisierung eines Arrays 15
3.2 Beispieldarstellungen von *Debug Visualizer* aus [Henning Dieterichs 2023] 16
3.3 Startseite von *Python Tutor* aus https://pythontutor.com/ 18
3.4 Editorfenster und Button zur Visualisierung in *Python Tutor* 19
3.5 Visualisierung und Bedienelemente in *Python Tutor* 20
3.6 Möglichkeiten der Veränderung der Visualisierung in *Python Tutor* . 22
3.7 Vereinfachte Darstellung eines *Doubles* 22
3.8 Korrekte Darstellung eines *Doubles* 23

4.1 Suchanfragen in Google für Visual Studio Code, Eclipse, IntelliJ IDEA, Visual Studio und Notepad++ im Vergleich aus [Google 2023] 28
4.2 Debug Adpater Protocol in Visual Studio Code 29
4.3 Ausführungsmöglichkeiten für Visualisierung des Frameworks . . . 31
4.4 Anzeige in Visual Studio Code nach Ausführung der Visualisierung von Python Visualization 32
4.5 Beispiel eines verschachtelten Tupel 33

5.1 Einfache Darstellung der Framework-Architektur 36
5.2 Detaillierte Darstellung der Architektur des Frameworks 37

6.1 Beispiel Debugger-Output eines Strings in Java 48
6.2 Beispiel Debugger-Output eines Tupel in Python 48

6.3 Beispiel Debugger-Output eines Wrappers in Java 49
6.4 Veranschaulichung von HashMap in Java 50
6.5 Beispiel Debugger-Output einer HashMap in Java 50

7.1 Korrekte und vollständige Darstellung von verschachtelten Tupel . . 55

8.1 Debugger-Output für s1 bis s4 61
8.2 Auflösungen der Variablen s1 - s4 62
8.3 Darstellung eines Wrapper in der Visualisierung 62
8.4 Darstellung eines String in der Visualisierung 63
8.5 Gegenüberstellung Java-Code einer HashMap und deren Visualisierung 64

9.1 Downloadfehler des Repository innerhalb der Container 74

10.1 Darstellung Zeile 16 aus Listing 10.1 78
10.2 Darstellung Zeile 23 und 22 aus Listing 10.1 79
10.3 Darstellung Zeile 30 aus Listing 10.1 80
10.4 Darstellung Zeile 9 aus Listing 10.2 82
10.5 Auswertung des Debugger-Outputs für ein Literal 82
10.6 Darstellung Zeile 11 aus Listing 10.2 83
10.7 Darstellung Zeile 32 aus Listing 10.2 84
10.8 Darstellung Aufruf innerhalb Zeile 18 aus Listing 10.3 86
10.9 Darstellung Zeile 10 aus Listing 10.3 87
10.10 Darstellung Zeile 27 aus Listing 10.4 88

Listings

1.1 Python-Code einer einfachen Nodestruktur 2

3.1 `serialize`-Methode für die Visualisierung eines Arrays in *Debug Visualizer* 14
3.2 Beispiel Code für die Visualisierung eines Arrays mit *Debug Visualizer* 14
3.3 Wrapper *Double* in Java 21
3.4 Float in Python 23
3.5 Float in Java 23

4.1 Beispiel des DAP für *int*-Array in Java 29
4.2 Auflösung des Beispiels für *int*-Array in Java 29
4.3 Beispielprogramm zur Untersuchung des Frameworks 31
4.4 Zwei verschachtelte Tupel 33

5.1 Beispiel eines Trace 39
5.2 Interface einer *DebugConfiguration* 40
5.3 Generische Konfiguration einer *DebugConfiguration* 40
5.4 Interface der programmiersprachenspezifischen *BackendSession* . . 41

6.1 Beispiel Debugger-Output eines primitiven Datentyps in Python . . 47
6.2 Beispiel Debugger-Output eines primitiven Datentyps in Java 47
6.3 Beispiel Debugger-Output eines Strings in Python 47
6.4 Erster Output des Debuggers von Byte in Java 49

7.1 Datenformat der aufzulösenden Variable 53
7.2 Auflösung der *variablesReference* 53
7.3 Pseudocode der Tiefensuche innerhalb des Frameworks 53

8.1 Ausschnitt Konfiguration der Extension 59
8.2 Java-Code für das Verständnis von Referenzen 60

9.1 Konfiguration der VSC-Testumgebung 66
9.2 Konfiguration eines Mocha-Tests 67
9.3 Mocha-Teststruktur 68
9.4 Mehrere Testfälle als Schleife definieren 68

9.5 After-Methode zum Löschen von temporären Files 69
9.6 Vorbereitung für explizites Testen von primitive Variablen 69
9.7 Expliziter Test für primitive Variablen 70
9.8 GitHub Action Workflow 73

10.1 Beispielprogramm für Listen und Dictionaries als Aufgabe aus Vorlesung 77
10.2 Beispiel für gemischte Daten als Aufgabe aus Vorlesung 81
10.3 Beispielprogramm für Polynome als Aufgabe aus Vorlesung 85
10.4 Beispielprogramm für Klassen als Aufgabe aus Vorlesung 88

Literaturverzeichnis

anseki (2023). *LeaderLine*. URL: https://anseki.github.io/leader-line/ (besucht am 17. 08. 2023).

Google (2023). *Google Trends*. Google Trends. URL: https://trends.google.de/trends/explore?date=2020-01-01%202023-08-01&q=%2Fm%2F0134xwrk,%2Fm%2F01fs1d,%2Fm%2F03v0mn,%2Fm%2F01r_y0,%2Fm%2F04t89l&hl=de (besucht am 01. 08. 2023).

Gretsch, Petra und Lars Holzäpfel (2016). *Lernen mit Visualisierungen: Erkenntnisse aus der Forschung und deren Implikationen für die Fachdidaktik*. Waxmann Verlag. 290 S. ISBN: 978-3-8309-8414-6.

Henning Dieterichs (2023). *Debug Visualizer - Visual Studio Marketplace*. URL: https://marketplace.visualstudio.com/items?itemName=hediet.debug-visualizer (besucht am 06. 08. 2023).

Hochschule Offenburg (2023). *Modulhandbuch*. URL: https://www.hs-offenburg.de/studium/studiengaenge/bachelor/angewandte-informatik/modulhandbuch?tx_modulhandbuch_fe%5Baction%5D=detailBySection&tx_modulhandbuch_fe%5Bcontroller%5D=Module&tx_modulhandbuch_fe%5BcourseOfStudy%5D=2&tx_modulhandbuch_fe%5Bsection%5D=1&tx_modulhandbuch_fe%5Bversion%5D=4&cHash=d1dac452f540a015e76c68591a6f634c (besucht am 31. 07. 2023).

Levente Polyák, Aaron Griffin und Judd Vinet (2023). *Xvfb(1) — Arch manual pages*. URL: https://man.archlinux.org/man/Xvfb.1.en (besucht am 30. 08. 2023).

Microsoft (2021). *Debug Adapters*. URL: https://microsoft.github.io/debug-adapter-protocol/implementors/adapters/ (besucht am 13. 08. 2023).

Microsoft (3. Aug. 2023a). *Continuous Integration*. URL: https://code.visualstudio.com/api/working-with-extensions/continuous-integration (besucht am 30. 08. 2023).

— (8. März 2023b). *Debugging in Visual Studio Code*. URL: https://code.visualstudio.com/docs/editor/debugging (besucht am 04. 08. 2023).

One Compiler (2023). *OneCompiler - Write, run and share code online | Free online compiler with 60+ languages and databases*. URL: https://onecompiler.com/ (besucht am 05. 08. 2023).

Python Software Foundation (2023). *Status of Python Versions*. Python Developer's Guide. URL: https://devguide.python.org/versions/ (besucht am 02. 08. 2023).

Python Tutor (2023a). *Python Tutor code visualizer: Visualize code in Python, JavaScript, C, C++, and Java*. URL: https://pythontutor.com/render.html#code=class%20Node%3A%0A%20%20%20%20def%20__init__%28self,%20data%29%3A%0A%20%20%20%20%20%20%20%20self.data%20%3D%20data%0A%20%20%20%20%20%20%20%20self.prev%20%3D%20None%0A%20%20%20%20%20%20%20%20self.next%20%3D%20None%0A%0Anode%20%3D%20Node%2810%29&cumulative=false&curInstr=7&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false (besucht am 04. 08. 2023).

— (2023b). *Python Tutor unsupported features*. Google Docs. URL: https://docs.google.com/document/d/13_Bc-l2FKMgwPx4dZb0sv7eMfYMHhRVgBRShha8kgbU/edit?usp=embed_facebook (besucht am 02. 08. 2023).

Reichenbach, Matthias (8. Sep. 2023). *ProgramFlow-Visualization*. Erstellungsdatum: 2023-03-15T09:10:44Z. URL: https://github.com/MatthReich/ProgramFlow-Visualization (besucht am 08. 09. 2023).

Velten, Marko (8. März 2023). *Visualisierung von Python Programmen in der Entwicklungsumgebung Visual Studio Code*. Offenburg.

Wehr, Stefan (11. Okt. 2022). *Write Your Python Program*. Erstellungsdatum: 2020-09-15T07:30:24Z. URL: https://github.com/skogsbaer/write-your-python-program (besucht am 01. 09. 2023).