Aufzeichnung und Visualisierung von Software-Laufzeitdaten mit SEE

oder

Recording and visualization of software runtime data with SEE

Bachelorarbeit

Sulan Abubakarov

Matrikelnummer: 4439505

06.12.2021

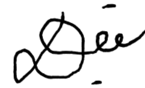
1. Gutachter: Prof. Dr. rer.nat. Rainer Koschke
2. Gutachter: Prof. Dr. Sebastian Maneth

Fachbereich Mathematik / Informatik
Studiengang Informatik

Erklärung

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

Bremen, den 06.12.2021



....

(Sulan Abubakarov)

Danksagung

Ich möchte an dieser Stelle einen Dank aussprechen an meine Familie, die mir durchweg durch das Studium die Kraft gegeben hat, nicht aufzugeben. Vor allem meinen Kindern, die selbst an den schlimmsten Tagen, mir ein Lächeln geschenkt und mich motiviert haben.

Abstract

Das Verständnis des Laufzeitverhaltens und Performanzeigenschaften von Software kann eine echte Herausforderung sein, gerade wenn man es mit einer großen Datenmenge zu tun hat. In dieser Arbeit wurde aus diesem Grund ein Profiler entwickelt, der zur Erfassung von Laufzeitdaten von `C#`-Programmen dient. Dessen Ergebnisse können im Anschluss mit einer selbstentworfenen Visualisierung in SEE dargestellt werden, anhand derer ein Programm auf Laufzeitflaschenhälse untersucht werden kann. Eine Benutzerstudie mit zehn Personen ergab, dass der entworfene Profiler und die Visualisierung der Daten eine sinnvolle Alternative zu herkömmlichen Profilern darstellt.

Vorwort

Aus Gründen der besseren Lesbarkeit wird in dieser Bachelorarbeit die Sprachform des generischen Maskulinums angewendet. Es wird an dieser Stelle darauf hingewiesen, dass die ausschließliche Verwendung der männlichen Form geschlechtsunabhängig verstanden werden soll. Zudem werden einige englische Fachausdrücke, die in der Informatik üblich sind, verwendet und nicht ins Deutsche übersetzt.

INHALTSVERZEICHNIS

1 Einleitung **1**

1.1 Motivation . . . 1

1.2 Zielsetzung . . . 1

1.3 Aufbau der Arbeit . . . 2

2 Grundlagen **3**

2.1 Softwarevisualisierung . . . 3

2.1.1 Bestandteile einer Software . . . 4

2.1.2 Visualisierungspipeline . . . 4

2.1.3 Forschungsstand . . . 5

2.1.3.1 Software Engineering Experience . . . 5

2.1.3.2 Software-Debugging mithilfe von Code Cities . . . 6

2.1.3.3 Performance-Profiling und Visualisierung in Software-Städten 7

2.1.3.4 Versionsunterschiede visualisiert durch UML-Klassendiagramme 8

2.1.3.5 Live-Visualisierung mit ExplorViz . . . 9

2.2 Software-Instrumentierung . . . 10

2.2.1 Executable-Instrumentierung . . . 11

2.2.2 Link-Time-Instrumentierung . . . 12

2.2.3 Quellcode-Instrumentierung . . . 12

2.3 Profiler . . . 13

2.3.1 Lazy-Allocation-Profiling . . . 13

2.3.2 Casual-Profiling . . . 13

2.3.3 Event-Based-Profiling . . . 14

2.4 Eingesetzte Software . . . 14

2.4.1 Unity . . . 14

2.4.2 Unity-Profiler . . . 15

2.4.3 Mono.Cecil . . . 16

2.4.4 ILSpy . . . 16

2.4.5 Sonstiges . . . 17

3 Entwurf **18**

3.1 Anforderungen . . . 18

3.1.1 Laufzeitdatenerfassung 18
3.1.2 Kompakte Speicherung der Laufzeitdaten 19
3.1.3 Visualisierung der Laufzeitdaten 19
3.2 Instrumentierung 20
3.2.1 Einzubettende Klasse 20
3.2.2 Grafische Benutzeroberfläche 21
3.3 JLG-Dateiformat 22
3.4 JLG-Parser 23
3.5 Visualisierung 23
3.5.1 Heatballoons 24
3.5.2 Profiler-Fenster 25
3.6 Bedienung 25

4 Implementation **28**
4.1 TraceEmbedder 28
4.2 EmbeddedTracer 33
4.3 Erweiterung der SEE-Visualisierung 34
4.3.1 Heatballoons 34
4.3.2 Profiler-Fenster 36
4.4 Bedienung 37
4.5 Tests 37
4.5.1 Automatisierte Tests 37
4.5.2 Tests zur Instrumentierung 38
4.5.3 Komplikationen 40

5 Benutzerstudie **42**
5.1 Forschungsfragen 42
5.2 Aufbau 42
5.2.1 Verwendete Software und Hardware 43
5.2.2 Aufgabenstellungen 43
5.2.3 Fragebögen 44
5.3 Durchführung 45
5.4 Auswertung 46
5.4.1 Einführungsfragebogen 46
5.4.2 Aufgabenstellungen 47
5.4.3 System-Usability-Scale 49
5.4.4 Offene Fragestellung 50
5.4.5 Threats-to-Validity 51

5.5 Beantwortung der Forschungsfragen 53

6 Fazit und Ausblick **54**

6.1 Fazit 54

6.2 Ausblick 55

6.2.1 Tool zur Abhängigkeitsauflösung 55

6.2.2 Tracing auf Statement-Ebene 55

6.2.3 Live-Visualisierung 55

6.2.4 Datenkompression 55

Abbildungsverzeichnis **56**

Listings **57**

Literaturverzeichnis **60**

Glossar **61**

Änderungsprotokoll **65**

KAPITEL 1

Einleitung

1.1 Motivation

Softwaresysteme sind aus der heutigen Zeit nicht mehr wegzudenken. Dennoch stellt die Softwarewartung einer der schwierigsten Aufgaben in der Softwareentwicklung dar. Nach mehreren Umfragen macht die Wartung im Lebenszyklus einer Software 60 bis 80% der Kosten aus [5]. Programmverständnis und Performanz-Optimierung stellen dabei Softwareentwickler im Wartungsprozess vor einer großen Herausforderung. Herkömmliche Ansätze wie Entwicklungsumgebungen, mit integrierten Debugger, Profiler und anderen Tools, die sowohl statische als auch dynamische Informationen ermitteln können, werden unter anderem für diese Herausforderung herangezogen. Jedoch mangelt es bei diesen Ansätzen oftmals an einer geeigneten Darstellung, um Verständnis sowie Übersicht zu schaffen.

Softwarevisualisierung, ein Teilgebiet der Informatik, beschäftigt sich mit Methodiken und Herangehensweisen, um die zuletzt genannten Punkte durch unterschiedliche Darstellungsmöglichkeiten anzugehen. Quellcode, Laufzeitdaten, Dokumentation, Evolution und der Entwurf einer Software werden dabei auf eine repräsentative Darstellung abgebildet. Sowohl im zweidimensionalen Raum mit unterschiedlichen Grafiken, als auch im dreidimensionalen Raum, die mehr Möglichkeiten zur Darstellung anbieten, wobei einzelne dreidimensionale Objekte dazu genutzt werden können, um Artefakte geeignet zu vertreten.

Bei der Softwarevisualisierung bleiben jedoch viele Fragen offen, die geklärt werden müssen. Die Frage ist, wie man an all die statischen und dynamischen Daten herankommt für die Visualisierung. Welche Tools dafür verwendet werden können und welche noch entwickelt werden müssen. Welche Darstellungen für die einzelnen Daten gut genug sind, um Verständnis und Übersicht zu schaffen.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist die Entwicklung eines Profilers zur Erfassung der Laufzeitdaten von `C#`-Programmen sowie die anschließende Visualisierung dieser Daten in dem *Software Engineering Experience* (SEE) Projekt, um Verständnis über den Ablauf einer Software auf Methoden-Ebene und seiner Performanz zu erlangen. Durch das Ergebnis dieser Arbeit sollen Entwickler, die sich mit Performanz-Analysen und Performanz-Optimierung von Programmen beschäftigen, durch eine neue Form der Performanz-Visualisierung profitieren und so leichter Laufzeitflaschenhälse ausfindig machen. Um sicherzustellen, dass der zu entwickelnde Profiler auch gebrauchstauglich ist, wird dieser mit den State-of-the-Art, dem *Unity-Profiler* [42] in einer Benutzerstudie zum Vergleich gesetzt. Im weiteren Verlauf dieser Arbeit, wird der zu entwickelnde Profiler, als *SEE-Profiler* aufgelistet.

1.3 Aufbau der Arbeit

Nach der Einleitung folgen im zweiten Kapitel die Grundlagen, die im Kontext dieser Arbeit nötig sind, um das weitere Geschehen nachvollziehen zu können. Dazu wird unter anderem der aktuelle Forschungsstand betrachtet, der implizit oder explizit die vorliegende Arbeit betrifft. Außerdem wird zum Ende des Kapitels auf die eingesetzte Software eingegangen. Im dritten Kapitel werden sämtliche Anforderungen an diese Arbeit aufgelistet sowie Entwürfe vorgestellt, wie die Anforderungen umgesetzt werden sollen. Des Weiteren wird abgegrenzt von Komponenten, die bereits vor dieser Arbeit entwickelt wurden. Im vierten Kapitel wird die Implementierung behandelt, hier wird letztendlich der Entwurf umgesetzt und wichtige Klassen und Methoden vorgestellt, die zur Umsetzung notwendig gewesen sind. Zudem werden die durchgeführten Tests, um die Funktionalität der zu entwickelnden Software zu überprüfen, vorgestellt und auf mögliche Komplikationen eingegangen. Im fünften Kapitel soll die durchgeführte Benutzerstudie im Vordergrund stehen, bei dem der zu entwickelnde SEE-Profiler mit den State-of-the-Art, dem Unity-Profiler im Vergleich gesetzt wird. Sowohl der Aufbau der Evaluation, die Durchführung, die Ergebnisse als auch die kritische Bewertung der Ergebnisse sollen hier zur Sprache kommen. Im siebten Kapitel finden sich Fazit und Ausblick.

KAPITEL 2

Grundlagen

In diesem Kapitel werden alle nötigen Grundlagen behandelt, von der Softwarevisualisierung bis zur Instrumentierung sowie zum Profiler. Dazu wird die Literatur beleuchtet und der aktuelle Forschungsstand näher betrachtet. Zudem werden die eingesetzten Ressourcen, um diese Arbeit abschließen zu können, vorgestellt.

2.1 Softwarevisualisierung

In der Softwarevisualisierung (SV) als Teilgebiet der Informatik, beschäftigte man sich in der Mitte der 80er-Jahre vorwiegend mit der Visualisierung von dem dynamischen Verhalten von Algorithmen [15]. Seitdem hat sich der Aufgabenbereich ausgeweitet, sodass Laufzeitverhalten, Evolution, Quellcode, Architektur oder andere Artefakte einer Software, im zweidimensionalen oder dreidimensionalen Raum visualisiert werden. Grade in der Softwareentwicklung, wo man es oft mit großen Systemen zu tun hat, die Millionen Zeilen von Code umfassen können, ist das Ziel der Softwarevisualisierung, die Komplexität dieser Systeme herunterzubrechen und eine Abstraktion zu schaffen, um diese verständlicher und übersichtlicher zu machen. Einige der Ansätze in der Softwarevisualisierung, versuchen aus diesem Grund Metaphern wie z. B. eine *Stadt-Metapher* zu verwenden oder andere Formen der Darstellung, um die zuletzt genannten zwei Punkte umzusetzen. So wird bei der Stadt-Metapher, Softwareartefakte und andere Teile einer Software, auf eine Stadt in einem dreidimensionalen Raum abgebildet [[22], [45], [25]]. Durch diese Abstraktion der Daten, soll eine bessere Übersicht geschaffen und die Mentallen-Modelle von Entwicklern angeregt werden, indem Teile der Realität, hier eine Stadt, in die virtuelle Welt transferiert und visualisiert wird. Um die Wichtigkeit von der Softwarevisualisierung weiter hervorzuheben, werden zwei Studien vorgestellt, die sich damit befasst haben, ob die Disziplin und deren Ergebnisse von Relevanz sind und welche Aspekte die meisten Entwickler schätzen bzw. wo sie Verbesserungspotential sehen.

Erste Studie In einer Studie mit 111 Teilnehmern, bei denen es sich um Forscher aus der Softwarewartung, Re-Engineering und Reverse-Engineering handelte, berichtete Koschke, dass 40% der Teilnehmer die Softwarevisualisierung absolut notwendig hielten für ihre Arbeit und dass 42% der Teilnehmer dies zwar wichtig fänden, aber den Verzicht nicht als kritisch ansehen. Damit sprachen sich insgesamt 82% für die Wichtigkeit der Softwarevisualisierung aus [21].

Zweite Studie In der Studie von Bassil und Keller, mit 107 Teilnehmern aus der Industrie, wurde durch eine offene Fragestellung, Vorteile ermittelt, die aus der Benutzung von unterschiedlichen Tools der Softwarevisualisierung hervorgehen. Die verwendeten SV-Tools konnten sich die Teilnehmer selbst aussuchen. Folgende Vorteile, haben die Teilnehmer bei

der Nutzung von diesen Tools gesehen [3]:

- Zeit- und Geldersparnis
- Besseres Softwareverständnis
- Erhöhte Produktivität
- Management von Softwarekomplexität
- Unterstützung zum Auffinden von Programmfehler

Des Weiteren wurden die Teilnehmer gefragt, was man besser machen könne bei den getesteten SV-Tools. Die zwei am höchsten bewerteten Antworten darauf waren, dass Funktionalität bereitgestellt werden müsse, um die Tools in anderen externen Tools integrieren zu können und dass eine Verbesserung des Imports und Exports von Daten und Visualisierungen vorhanden sein müsse.

2.1.1 Bestandteile einer Software

In der Softwarevisualisierung werden viele Aspekte einer Software betrachtet und wie diese angemessen visualisiert werden können. Nach Diehl befassen sich Forscher mit den Bestandteilen, Struktur, Verhalten und Evolution einer Software und dessen Visualisierung. Im Folgenden werden diese Bestandteile aus seiner Sicht näher erläutert [9].

Struktur Die Struktur bezieht sich auf die statischen Teile einer Software und deren Beziehungen untereinander, darunter fallen alle Teile, die ohne Ausführung einer Software erhoben werden können. Unter anderem fallen darunter der Quellcode, die verwendeten Datenstrukturen, der statische Aufrufgraph sowie die Organisation des Programms in einzelne Module.

Verhalten Das Verhalten bezieht sich wiederum auf die Ausführung eines Programms mit echten und abstrakten Daten. Dabei kann man die Zustandsänderungen, sowie den dynamischen Aufrufgraphen und das Zusammenspiel von diversen Objekten näher beleuchten [9]. Im Falle der vorliegenden Bachelorarbeit, wird das Verhalten betrachtet, indem `C#`-Programme instrumentiert und zur Laufzeit Informationen auf Methoden-Ebene erfasst werden. Dazu gehört unter anderem der dynamische Aufrufgraph, indem die aufgezeichneten Methoden in ihrer Aufrufreihenfolge in Beziehung gesetzt werden.

Evolution Die Evolution betrachtet den gesamten Entwicklungsprozess einer Software und hebt hervor, dass sich der Quellcode im Laufe der Zeit verändern kann, z. B. durch das Hinzukommen weiterer Funktionen oder aber durch die Beseitigung von vorhandenen Fehlern.

2.1.2 Visualisierungspipeline

Bis die Visualisierung zustande kommt, benötigt es einige weitere Schritte, die in Betracht gezogen werden müssen. Diehl unterscheidet zwischen Datenerfassung, Analyse und der finalen Visualisierung. Jedes dieser Teile stellt ihren Output, als Input der nachfolgenden Phase dar [9].

Datenerfassung In dieser Phase muss man sich ins Bewusstsein rufen, welche Artefakte visualisiert werden sollen. Bei einer Software kann sich die Quelle der Informationen variieren, je nachdem, ob man den Quellcode, Tests, die Dokumentation, den Softwareentwurf, die

Softwareevolution oder die einzelnen Laufzeitinformationen, bei der Ausführung der Software betrachtet. Die Methodik für die Erhebung dieser Daten kann sich unterscheiden, je nachdem welche Daten erhoben werden sollen [9]. Einige Vorgehensweisen zur Erhebung von Laufzeitdaten werden im Abschnitt 2.2 erläutert.

Analyse Die Menge an Daten, die man betrachten möchte, kann ein enormes Ausmaß annehmen. Deshalb wird in dieser Phase mit unterschiedlichen Methoden versucht, diese Datenmenge zu reduzieren. Die Reduktion kann dabei durch Filtern erfolgen, statische Programmanalysen oder durch statistische Methoden [9]. Eine mögliche Form der Reduktion wird im Abschnitt 2.1.3.4 behandelt, bei dem eine Transformation durchgeführt wird, um Klasseninformationen in einen Graphen zu überführen.

Visualisierung Die aus den vorherigen Phasen entnommenen Daten werden auf ein visuelles Modell abgebildet, z. B. könnten diese umgewandelt werden auf geometrische und grafische Informationen, die dann auf den Bildschirm oder einem anderen Medium als ein Bild oder eine Sammlung an Bildern projiziert werden [9]. Auf mögliche Darstellungsmöglichkeiten wird im nächsten Kapitel 2.1.3, durch unterschiedliche Projekte aus der Literatur eingegangen.

In dieser Bachelorarbeit, wie bereits in Abschnitt 2.1.1 erklärt, handelt es sich bei der Datenerfassung um Laufzeitinformationen, die bei der Ausführung von `C#`-Programmen erfasst werden. Die Analysephase, sieht eine optionale Begrenzung und die Visualisierung eine selbst entworfene Darstellung von den erhobenen Laufzeitdaten in der SEE-Umgebung vor. Näheres wird im Kapitel 3 zu der Datenerfassung, Analyse und Visualisierung behandelt.

2.1.3 Forschungsstand

In diesem Abschnitt werden einige Arbeiten und dessen Arbeitsergebnisse aus der Literatur betrachtet, die unmittelbar mit der vorliegenden Arbeit zusammenhängen bzw. teilweise die gleichen Themenbereiche behandeln.

2.1.3.1 Software Engineering Experience

Software Engineering Experience (SEE) stellt den Grundbaustein dieser Arbeit dar und ist ein Werkzeug zur Visualisierung von Software, welcher in Unity 3D [41] von der Softwaretechnik AG unter der Leitung von Koschke entwickelt wurde [22]. Die Grundidee hinter SEE ist, Techniken aus der Softwarevisualisierung zu nutzen, um Quellcode, Softwarearchitekturen und andere Artefakte zu verstehen und zu kommunizieren, indem eine Repräsentation in einem 3D Raum geschaffen wird, die sich einer Stadt-Metapher bedient, wie in vielen anderen Tools zur Visualisierung [[30], [45], [25]]. Dabei können 3D-Objekte bzw. Gebäude, wie sie im Kontext von SEE heißen, einzelne Dateien repräsentieren, einen Methodenaufruf zwischen zwei Klassen darstellen oder es werden Eigenschaften eines Objekts durch Metriken definiert, die auf eine bestimmte Darstellung abzielen. Ein Beispiel einer solchen Metrik wäre die *Lines-of-Code*, die durch die Größe eines Gebäudes widergespiegelt werden kann. Anschließend soll man die Möglichkeit bekommen in dieser Darstellung allein oder kooperativ mit anderen einzutauchen, sei es am Desktop mit der Maus oder mit einer VR-Brille.

Zum Zeitpunkt des Verfassens dieser Arbeit, gibt es in SEE diverse Darstellungsmöglichkeiten, die schon umgesetzt wurden. Aktuell ist es möglich, seiner Software bei der Ausführung zuzusehen und herauszufinden, an welchen Stellen es Performanzprobleme gibt, wie der Aufrufgraph aussieht und vieles mehr. Ebenfalls ist es möglich nachzuverfolgen, wie die Softwa-

reevolution verlief, indem man sieht, wie im Verlauf einer Software Dateien hinzukommen, gelöscht werden und wachsen. Neben den genannten Darstellungsmöglichkeiten werden zum Zeitpunkt des Verfassens dieser Arbeit weitere Möglichkeiten hinzuentwickelt.

2.1.3.2 Software-Debugging mithilfe von Code Cities

Die Bachelorarbeit von Lennart Kipka [20] dient als Basis für diese Arbeit. Neben ähnlichen theoretischen Grundlagen, werden einige Artefakte in dieser Arbeit wiederverwendet und erweitert, um Kompatibilität zwischen den Arbeitsergebnissen zu schaffen. Es handelt sich bei den Artefakten zum einen, um das *JLG-Dateiformat*, auf das im Abschnitt 3.3 noch näher eingegangen wird und zum anderen um den Quellcode, den Kipka geschrieben hat, bei dem es sich um einen Parser handelt, der für das Einlesen und Aufbereiten einer *JLG-Datei* zuständig ist sowie der Quellcode, der zur Erstellung bis zum Managen der Visualisierung verantwortlich ist. Das Tool zum Erheben der Laufzeitdaten wurde in Java geschrieben und ist deshalb ausschließlich auf Java-Anwendungen ausgelegt. Neben einigen Ähnlichkeiten unterscheidet sich die aktuelle Arbeit von Kipkas in der Hinsicht, dass die zu entwickelnde Instrumentation-Software auf `C#` ausgelegt ist und bei der Visualisierung sogenannte *Heatballoons* verwendet werden, um die Laufzeitdaten darzustellen. Zusätzlich wird ein *Profiler-Fenster* entwickelt, das tabellarisch ausgewählte Laufzeitdaten anzeigt. All diese genannten Themen werden im Laufe dieser Arbeit im Detail behandelt.

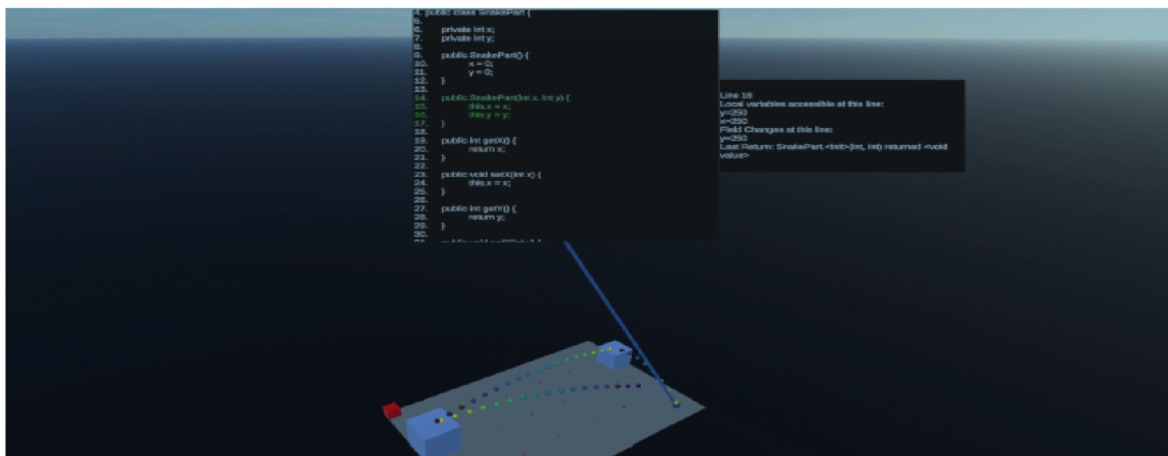


Abbildung 2.1: Ausschnitt eines visualisierten Debug-Prozesses in in der „JLGCity" von Kipka [20].

In der Abbildung 2.1 sieht man einen Ausschnitt der Visualisierung, die auch als „JLGCity" bezeichnet wird, die Kipka im Rahmen seiner Arbeit entwickelt hat. Die Visualisierung lässt sich dabei in drei Teile unterteilen [20]:

- Visualisierung der aufgezeichneten Codezeilen
- Visualisierung der Laufzeitdaten, in dem Fall die lokalen Variablen, Rückgabewerte und Felder.
- Visualisierung des Aufrufgraphen

Der erste Punkt wird umgesetzt, durch das größere Fenster in der Abbildung 2.1, wobei für die aktive verarbeitende Codezeile, die Quelldatei, in der sich die Zeile befindet, in das

Fenster geladen wird. Dabei werden die aktiven Codezeilen, in einem kräftigen Grünton gefärbt. Der Grünton verblasst im Laufe der Zeit, bis er, wie der restliche Codeabschnitt, die Farbe Weiß annimmt. Dies soll vor allem dazu dienen, die letzten ausgeführten Codezeilen deutlich von dem Rest abzugrenzen. Der zweite Punkt wird durch das kleinere Fenster neben dem Größeren umgesetzt, der lokale Variablen, Rückgabewerte und Felder anzeigt, die in der aktuellen verarbeitenden Codezeile manipuliert werden. Der letzte Punkt wird mithilfe der Funktionalität und Visualisierung, die Torben Groß im Rahmen seiner Bachelorarbeit entwickelt hat, umgesetzt [16]. Dabei wird ein Methodenaufruf damit verdeutlicht, dass eine Kante, die aus Kugeln besteht, wie in der Abbildung 2.1 zu sehen, zwischen der Aufrufer- und Aufgerufenen-Klasse entsteht. Alle Klassen, die sich im Aufrufgraphen befinden, werden dabei in einer hellblauen Farbe eingefärbt und nur die aufgerufene Klasse in einem dunkelblauen Farbton.

Das Ziel dieser Visualisierung ist es, einen Debugger-Prozess einer Java-Anwendung wie in einem Film in der JLGCity zu animieren. Die Animation selbst kann man dabei manuell oder automatisiert, sowohl vorwärts, als auch rückwärts abspielen. Zu jedem Zeitpunkt besteht die Möglichkeit, die Animation zu stoppen und sich den aktuellen Zustand in Detail anzuschauen. Neben den bereits genannten Features, gibt es eine Menge weitere, die erwähnenswert sind, die jedoch in dieser Arbeit nicht weiter betrachtet werden.

2.1.3.3 Performance-Profiling und Visualisierung in Software-Städten

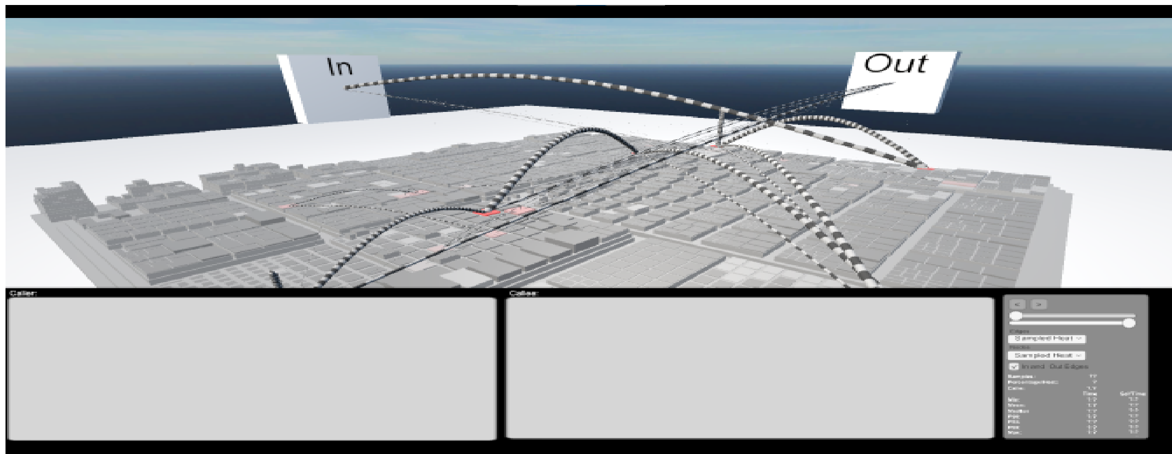


Abbildung 2.2: Ausschnitt der Visualisierung von Rohloff [34].

Parallel zur aktuellen Arbeit wurde eine Masterarbeit von Yannis Rohloff verfasst, bei dem zwei Profiler und eine Visualisierung in SEE entworfen wurden [34]. Bei den Profilern handelt es sich, zum einen den *Sampling-Profiler*, zum anderen einen *Instrumentation-Profiler*. Eine Besonderheit dieser Profiler ist, dass sie in der Lage sind, durch bestimmte Einstellungen, den Overhead zu verringern, der später in der Ausführung einer Anwendung entstehen kann. So kann bei dem Sampling-Profiler, der in einem vorbestimmten Intervall einen Thread einer Anwendung abtastet, um an den *Stack-Trace* zu kommen, die Abtastrate reduziert werden. Der Stack-Trace enthält dabei Informationen der Methoden-Ebene. Bei dem Instrumentation-Profiler hingegen können selektiv Methoden ausgewählt werden, die instrumentiert und damit aufgezeichnet werden sollen. Die Eigenheit des Sampling-Profilers ist unter anderem, dass bei dieser Methodik die Zielanwendung nicht instrumentiert werden muss, es werden ausschließlich die Threads einer Anwendung beobachtet, um an gewünschte

Informationen heranzukommen. Anders als in der vorliegenden Arbeit werden Anwendungen bei dem Instrumentation-Profiler in der Programmiersprache Java instrumentiert, um später bei der Ausführung Laufzeitdaten auf Statement-Ebene und Performanz-Daten zu erheben.

In der Abbildung 2.2 ist ein Ausschnitt der Visualisierung zu sehen, der im Rahmen der Arbeit von Yannis Rohloff entwickelt wurde. Die zwei Blöcke mit der Inschrift *In* und *Out* stellen dabei externe Schnittstellen dar, z. B. den Zugriff auf eine Datenbank. Die einzelnen Rohre zwischen den Blöcken bzw. Gebäuden deuten ein Aufrufverhältnis an und die Breite kann sich, je nachdem wie viel CPU-Zeit benötigt wird, variieren. Neben der eigentlichen Visualisierung und dem Aufrufverhältnis sind unten drei Fenster platziert. Das Fenster rechts in der Ecke, bietet neben Einstellungsmöglichkeiten, auch die Einsicht auf die Performanz-Daten. Bei den Fenstern links davon wird der Quellcode geladen und auf Statement-Ebene, die aktuelle behandelte Zeile Code hervorgehoben. Dabei kann man den Code im mittleren Fenster von den Aufgerufenen, im Fenster links davon den aktuellen Code des Aufrufers einsehen.

2.1.3.4 Versionsunterschiede visualisiert durch UML-Klassendiagramme

Der bisher betrachtete Forschungsstand, basierte auf SEE und die Darstellung der Visualisierung als eine Stadt-Metapher. Doch wie zuvor in der Einführung zu Softwarevisualisierung erklärt, gibt es unterschiedliche Darstellungsmöglichkeiten. So haben Seemann et al. ein Tool entwickelt, dass in der Lage ist, zwei objektorientierte Software-Versionen miteinander zu vergleichen, bei dem die gewählte Darstellung ein UML-Klassendiagramm repräsentiert [36].

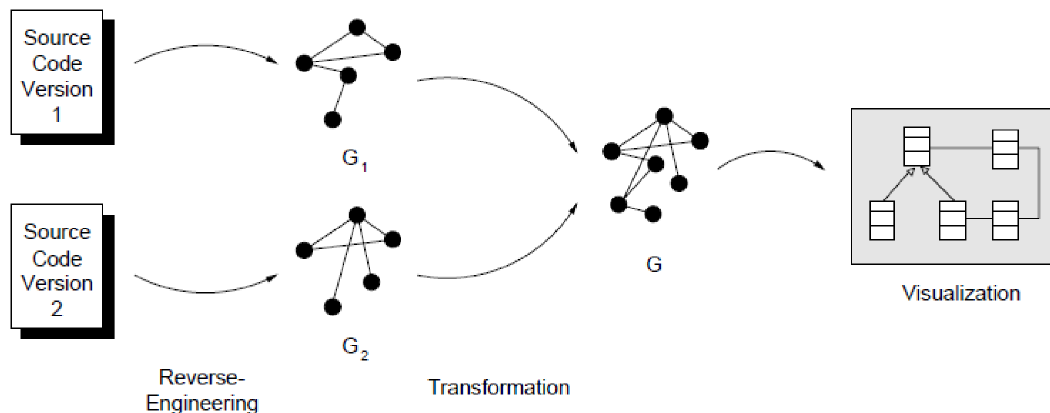


Abbildung 2.3: Schritte bis zur Konstruktion der Visualisierung von Seeman et al. [36].

Der grobe Ablauf bis zur Konstruktion der Visualisierung ist einmal in der Abbildung 2.3 zu sehen. Im ersten Schritt wird der Quellcode ausgewählt, der miteinander verglichen werden soll. Der nächste Schritt sieht Reverse-Engineering vor, bei dem ein Ganzes in seine Grundbausteine gebrochen wird. In diesem Fall werden aus dem Quellcode alle nötigen Informationen entnommen, die für die Gestaltung eines Graphen, um genauer zu sein, den „static structure graph", der im nächsten Schritt folgt, nötig ist. Dabei wird für jeden vorliegenden Quellcode ein individueller Graph erstellt, wobei die Knoten die Klassen repräsentieren und die Abhängigkeiten zwischen zwei Klassen durch eine Kante dargestellt werden. Bei der Abhängigkeit kann es sich um Vererbung oder jegliche andere Assoziation handeln. Die einzelnen Attribute und Methoden werden in den Knoten zwischengespeichert und einige andere

Informationen, welche die Abhängigkeit betreffen, in den Kanten. Die zwei entstandenen Graphen werden anschließend zusammengeführt und die Versionsunterschiede kenntlich gemacht, bis der finale Graph „static structure difference graph” visualisiert wird [36].

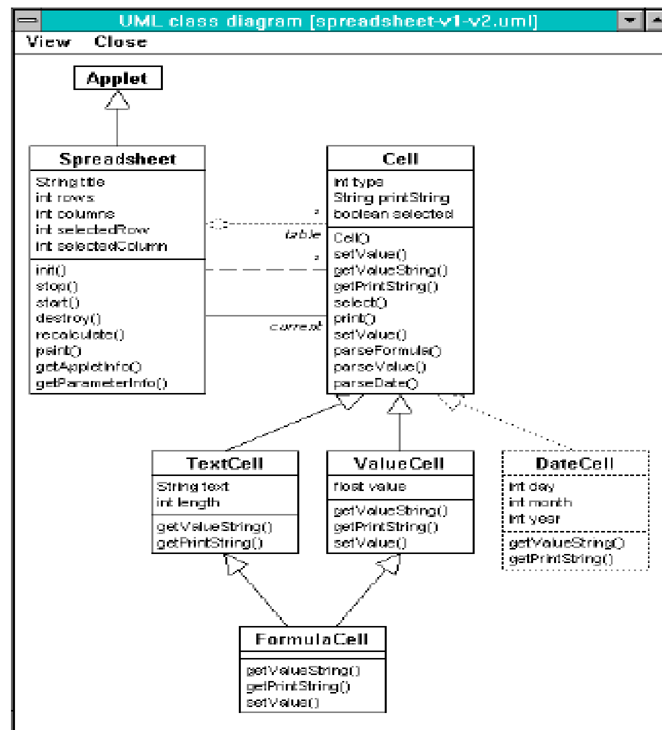


Abbildung 2.4: Ausschnitt eines Versionsunterschieds von Seeman et al. [36].

Die Abbildung 2.4 zeigt einen Ausschnitt des Ergebnisses, nachdem man zwei Graphen ineinander überführt und es einen Unterschied in der jeweiligen Version vorliegt. Die Klassen bzw. Interfaces, die mit Punkten umrandet sind, wie z. B. *DateCell*, stellen hierbei neu hinzugefügte Elemente dar. Auch die neu hinzugekommenen Abhängigkeiten werden durch eine gepunktete Linie dargestellt von *DateCell* nach *Cell* und von *Cell* wiederum nach *Spreadsheet*, in diesem Fall eine neue Aggregation und Vererbung. Die Abhängigkeiten, die wiederum entfernt werden, sind durch gestrichelte Linien dargestellt.

2.1.3.5 Live-Visualisierung mit ExplorViz

Die Anwendung *ExplorViz*, die von Fittkau et al. [12] entwickelt wurde, bietet eine Besonderheit im Gegensatz zu den bisher behandelten Anwendungen, dazu gehört, dass live, Laufzeitdaten parallel zur Ausführung der Zielanwendung visualisiert werden. Bei der Visualisierung selbst wird zwischen der „landscape level perspective” und der „system level perspective” unterschieden. In der „landscape level perspective” wird eine Visualisierung in Form eines UML-Verteilungsdiagramms geschaffen, in der die gesamte Softwarelandschaft repräsentiert wird. In der Abbildung 2.5 sieht man dazu ein Beispiel, bei dem die Kommunikation von Anwendungen in der Softwarelandschaft von PubFlow [33], durch ExplorViz in Form eines UML-Verteilungsdiagramm dargestellt wird [12].

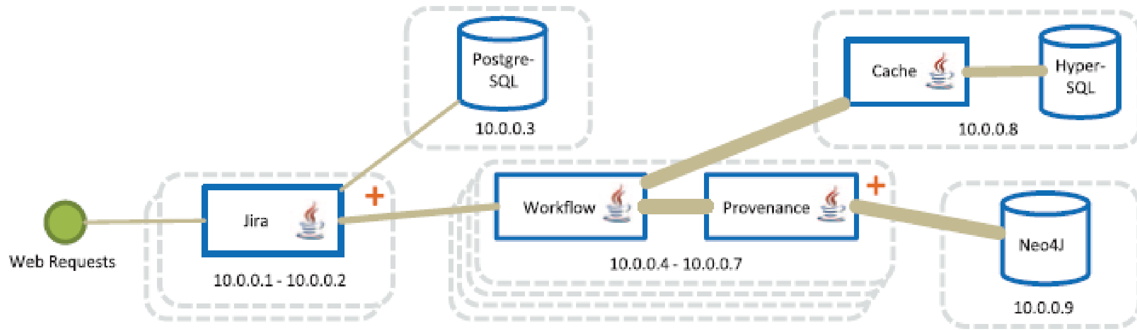


Abbildung 2.5: Visualisierung einer Softwarelandschaft in der „landscape level perspective" von Fittkau et al. [12].

Die „system level perspective" bedient sich der Stadt-Metapher wie in den vorher behandelten Anwendungen. Einen wesentlichen Unterschied, neben der Live-Visualisierung ist, dass inkrementell die Beziehungen zwischen Komponenten aufgelöst werden, anstatt alle auf einmal aufzulösen. Wie in der Abbildung 2.6 links zu sehen, werden z. B. alle Klassen in einem Paket bzw. in einer Gruppe gekapselt und erst auf Anfrage, wie rechts im Bild zu sehen, aufgedeckt. Dies ermöglicht ExplorViz eine höhere Skalierbarkeit und Performanz, als wenn z. B. alle Klassen und Beziehungen auf einmal präsentiert werden. Nach Fittkau et al. verfolgen sie damit das sogenannte „*Visual Information-Seeking Mantra*" von Shneiderman [38]. Bei diesem Prinzip von Shneiderman geht es grundlegend darum, dass bei einem visuellen Design, erst die Übersicht, dann das Zoomen bzw. Filtern und anschließend Details nach Anfrage folgen muss.

Abbildung 2.6: Visualisierung einer Software in der *system level perspective* von Fittkau et al. [12].

2.2 Software-Instrumentierung

Die Software-Instrumentierung stellt einen wesentlichen Teil der Softwarevisualisierung, Evaluation von Softwareperformanz und das Verständnis von parallelen Programmen dar, um einige zu nennen. Dazu werden Programme modifiziert, indem an unterschiedlichen Stellen bestimmter Code platziert wird, um Laufzeitdaten erfassen zu können. Zu den Laufzeitdaten selbst kann der Zustand der Anwendung, das Verhalten und die Performanz angesehen werden. Die erfassten Daten wiederum können für die unterschiedlichsten Zwecke verwendet werden. Zum einen für die Optimierung von Anwendungen, indem Stellen ausfindig gemacht

werden, die eine sehr hohe CPU-Last haben. Zum anderen können in der Softwarevisualisierung die Laufzeitdaten genutzt werden, um die Ausführung in einer abstrakteren Form darzustellen und dadurch Zusammenhänge zu verdeutlichen. Die Instrumentierung kann dabei an unterschiedlichen Zeitpunkten der Anwendung erfolgen, zum einen zur Kompilierzeit, zum anderen, wenn ein Linker sämtliche Binärdateien einer Anwendung zu einem Executable (EXE) zusammenführt, oder aber nachdem ein Executable vorliegt [31]. Eine Sache, die jedoch beachtet werden sollte ist, dass die eigentliche Instrumentierung das Verhalten der Anwendung nicht manipuliert. Des Weiteren ist eine Last durch die Instrumentierung nicht vermeidbar, da zusätzliche Logik für die Erfassung der Laufzeitdaten einen Einfluss auf die Performanz haben kann. Nachfolgend werden unterschiedliche Arten von Instrumentierungen vorgestellt von Pierce et al., die zum Teil aus dem Englischen übersetzt werden [31].

2.2.1 Executable-Instrumentierung

Die Instrumentierung direkt an Executables stellt zwar eine optimale Lösung für den Entwickler dar, ist jedoch sehr schwer umzusetzen, da Informationen über die Struktur des Codes fehlen. Ein Tool, der solch eine Instrumentierung durchführt, durchläuft drei Schritte. Bei dem ersten Schritt werden die Codeabschnitte zerlegt, in den einzelnen Instruktionen. Der zweite Schritt sieht die eigentliche Instrumentierung mit eigenem Code vor. Der letzte Schritt ist die Verlagerung des gesamten Codes an die richtigen Stellen und der Wiederaufbau des Executables. Schwierigkeiten bei diesen Schritten macht, wie schon vorher erwähnt, die fehlenden Strukturinformationen. So muss das Tool entweder auf Compiler-Informationen zurückgreifen oder aber Heuristiken erstellen. Kann dieses Problem nicht statisch gelöst werden, muss zur Laufzeit das Problem angegangen werden, was zusätzlichen Aufwand verursacht und daraus Performanzprobleme oder Zuverlässigkeitsprobleme resultieren. Ein weitaus größeres Problem kann bei der Instrumentierung entstehen, wenn nicht nur der Versuch der Instrumentierung fehlschlägt, sondern ein inkorrektes Programm am Ende daraus resultiert. Überwindet man jedoch diese Hürden, gibt es eine Menge Vorteile bei der Instrumentierung auf Executable-Ebene:

Unabhängigkeit vom Quellcode Der eigentliche Quellcode und damit auch das Projekt selbst muss nicht zur Instrumentierung vorliegen. Damit ist man in der Lage jede beliebige Anwendung zu instrumentieren.

Unabhängigkeit der Anwendungsgenerierung Binärdateien, die von einem Compiler einer beliebigen Programmiersprache produziert werden, können instrumentiert werden.

Automatische Instrumentierung der Bibliotheksmodule Das Aufzeichnen einer gesamten Anwendung wird dadurch ermöglicht, dass ein *Linker* statische Bibliotheken ebenfalls mit in den Executables einbindet, die ebenso instrumentiert werden können, wie der Rest der Anwendung.

Effizienz Der Entwickler wird durch die Instrumentierung auf Executable-Ebene dadurch entlastet, dass der instrumentierte Code nicht wieder compiliert werden muss und die statischen Bibliotheken keine Wartung unterzogen werden müssen.

Tools, die Instrumentierung auf Executable-Ebene durchführen, können unterschiedliche Anforderungen an den Executables selbst haben. Einige der Tools benötigen keine Symboltabelle, in der z. B. Informationen über Variablen, dessen Datentyp, Gültigkeitsbereich und viele andere Informationen vorhanden sind. Andere Tools jedoch benötigen weit aus mehr Informationen in der *Symboltabelle*, wie z. B. Profiler-Informationen oder Debugger-Informationen.

2.2.2 Link-Time-Instrumentierung

Eine weitere Möglichkeit der Instrumentierung besteht, bevor alle Objektdateien durch den Linker zusammengeführt werden zu einem Executable. Die Instrumentierung übernimmt in diesem Fall ein sogenannter „*object rewriter*" [31], der sich eine Objektdatei nimmt, diese mit eigenem Code modifiziert und weiter an den Linker gibt. Die Modifikation selbst wird an einem Relocation- und Symboltabelle vorgenommen, welche sich je nach Objektdatei unterscheiden können. In diesen Tabellen sind unterschiedliche Informationen zwischengespeichert, z. B. wie externe Funktionen und Variablen aufgelöst werden müssen und um welche Art von Daten es sich jeweils in der Objektdatei handelt. Es ist anzumerken, dass ohne diese Tabellen, eine Link-Time-Instrumentierung fast unmöglich ist. Ein weiterer Nachteil dieser Art der Instrumentierung ist, dass der beschriebene Prozess nicht automatisiert werden kann, wie bei der Executable-Instrumentierung. Der Anwender benötigt für diesen Prozess die Objektdateien, außerdem muss er Wissen über die Anforderungen des Linkers besitzen. Einher geht damit, dass man die einzelnen Quellcodedateien braucht, um die eigentlichen Objektdateien zu erzeugen.

2.2.3 Quellcode-Instrumentierung

Die früheste Instrumentierung kann zur Kompilierzeit erfolgen, dies stellt auch einer der einfachsten Varianten zur Instrumentierung dar. Es kann jedoch folgende Nachteile mit sich bringen:

- Quelldateien sind erforderlich, um die eigentliche Instrumentierung durchführen zu können. Dies kann nicht immer gewährleistet werden, gerade wenn es sich bei der Anwendung um kein eigenes oder Open-Source-Anwendung handelt, liegt in den meisten Fällen nur ein Executable vor.

- Einige der Tools zur Instrumentierung sind direkt im Compiler integriert und beschränken sich damit auf eine Menge von Anwendungen, je nachdem für welche Programmiersprache der Compiler geschrieben wurde.

- Die Effizienz der Instrumentierung wird dadurch beeinflusst, dass nach der Instrumentierung der Quellcode wieder compiliert werden muss.

- Die Instrumentierung ist dadurch limitiert, dass einige Bibliotheken nicht mit instrumentiert werden können, da sie durch den Linker erst im späteren Verlauf eingebunden werden. Eine Möglichkeit würde dennoch bestehen, und zwar indem man vorab die einzelnen Bibliotheksmodule instrumentiert und diese anschließend benutzt. Jedoch stellt dies einen viel zu großen Aufwand dar und eine Wartung der Bibliotheksmodule wäre unverzichtbar.

Neben den bereits genannten Nachteilen hat diese Art der Instrumentierung auch Vorteile. Zum einen hat man einen vereinfachten Prozess der Binärdatei-Erzeugung, bei dem man nicht weiter eingreifen muss. Zum anderen gibt es zur Zeit des Quellcodes unterschiedliche Möglichkeiten der Instrumentierung, die im späteren Verlauf verfallen. Der mit Abstand wichtigste Punkt, der zu nennen ist wäre, dass man zur Kompilierzeit genug Informationen besitzt, um den eigentlichen Code, der in eine Anwendung instrumentiert wird, minimal zu halten und damit die Performanz der Anwendung weniger beeinflusst.

2.3 Profiler

Performanz ist ein Thema, mit dem sich die meisten Entwickler immer wieder beschäftigen müssen. Wählt man die falschen Datenstrukturen, gibt man Ressourcen nicht wieder frei oder aus vielen anderen Gründen kann die Performanz einer Software grundlegend leiden. Zu diesem Zweck können Profiler verwendet werden, um die betroffenen Stellen ausfindig zu machen. Dabei sind Profiler in der Lage, die zur Laufzeit angefallenen Performanz-Daten in den unterschiedlichen Softwarekomponenten zu erfassen und dem Entwickler zur Verfügung zu stellen. Zu den Performanz-Daten gehört unter anderem die Speichernutzung, Methodenlaufzeiten, aber auch die Einsicht in die Nebenläufigkeit, um Verklemmungen aufzulösen.

Profiler lassen sich dadurch klassifizieren, wann sie die Daten, die sie erheben, präsentieren. Dabei unterscheidet man zwischen Profiler, die während der Laufzeit Daten dem Entwickler bereitstellen, wie der Unity-Profiler oder aber nach der Laufzeit, wie der in dieser Arbeit entstandene SEE-Profiler. Nachfolgend werden weitere Profiler aus der Literatur vorgestellt, die sich in ihrer Vorgehensweise und Methodik unterscheiden. Des Weiteren soll dadurch verdeutlicht werden, wie vielfältig ein Profiler genutzt werden kann.

2.3.1 Lazy-Allocation-Profiling

Eine Klasse von Profilern nutzen die Strategie „lazy allocation", die erstmals von Henry G. Baker vorgeschlagen wurde, bei dem die Zuordnung von Objekten auf den Heap nur dann geschieht, wenn diese auch langfristig benötigt werden. Objekte, die kurzfristig genutzt werden, sollen stattdessen auf dem Stack abgelegt werden. Dies ist ein Versuch Performanz zu sparen, indem man dem *Garbage-Collector* etwas die Arbeit abnimmt [2]. Ein Profiler im Java-Umfeld, der sich dieser Strategie in einer etwas anderen Art bedient, ist der von Shi et al. [37], bei dem generell keine Objekte in den Heap angelegt werden, wenn sie nicht in Gebrauch sind. So konnten Shi et al. mit dem Profiler ermitteln, dass auf die „SPECjvm98 benchmarks", einer Produktreihe, die zur Leistungsmessung von Computersystemen für JVM-Clientplattformen dient [40], im Durchschnitt auf 25% der Objekte zur Laufzeit, nicht zugegriffen wurde [37].

2.3.2 Casual-Profiling

Einige der verwendeten Profiler um die Performanz zu steigern, geben Stellen im Programm an, wo die meiste Zeit in der Laufzeitzeit verbracht wird. Nach Curtsinger und Berger bringt die Optimierung dieser Stellen keinen Mehrwert, weil das Problem meistens an einer ganz anderen Stelle liegt [8]. Aus diesem Grund führen sie in ihrem Paper das Konzept „cusual profiling" ein, bei dem im Vergleich zu anderen Ansätzen, exakt ermittelt wird, wo genau man bei dem Verbesserungsprozess ansetzen muss und wie viel Einfluss die einzelnen ermittelten Stellen haben. Um dies umzusetzen wurde ein Cusual-Profiler (COZ) entwickelt, der parallel zu der testenden Anwendung unterschiedliche „performance experiments" durchführt und so die Schwachstellen ermittelt. Bei diesen Experimenten wird der Einfluss einer Optimierung berechnet, indem einige Codeteile virtuell beschleunigt werden. In diesem Beschleunigungsprozess, der auch „virtually speeding" genannt wird, werden Pausen eingeführt in Codeteilen, die zeitgleich laufen, zu dem Codeabschnitt, der aktuell von Interesse ist. So konnte mit dem Einsatz von COZ unter anderem die Performanz von dem bekannten relationalen Datenbankmanagementsystem SQLite um 25% verbessert werden [8].

2.3.3 Event-Based-Profiling

Viele Anwendungen sind Event-basiert, das heißt sie nehmen einen Event als Input, ändern den Zustand dieses Events und geben ggf. selber einen Event wieder als Output. Klassische Event-basierte Software beruhen auf grafischen Benutzeroberflächen (GUIs), Webapplikationen oder Netzwerkprotokollen. Schaut man sich GUIs genauer an, gibt es unterschiedliche Elemente, die entweder statisch oder dynamisch sind, mit denen man interagieren kann. Solche Elemente stellen die Möglichkeit bereit, bestimmte *Listener* anzubringen, die bei einem Mausklick oder sonstigen Aktionen, eine bestimmte Funktionalität ausführen. Ein *Event-Based*-Profiler von Nagarajan und Memon klinkt sich in solchen Listener ein, um Informationen zu sammeln über die einzelnen Events. Das Ziel dabei ist eine Rekonstruktion einer Software im Sinne von Reverse-Engineering, durch die gesammelten Daten [28].

2.4 Eingesetzte Software

Nachfolgend wird die eingesetzte Software wie die *Mono.Cecil*-Bibliothek, der Dekompilierer ,,ILSpy", das verwendete Betriebssystem und einige weitere, die zur Umsetzung dieser Arbeit nötig waren, aufgelistet.

2.4.1 Unity

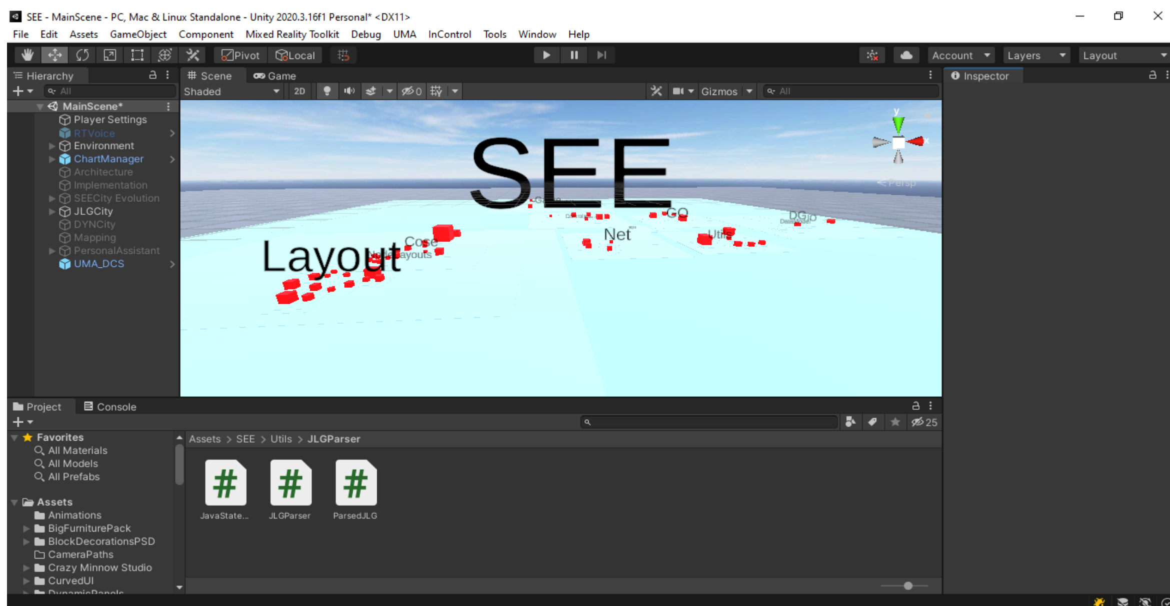


Abbildung 2.7: Ausschnitt der visualisierten Fassung von dem SEE Projekt in Unity [41].

Die Spiele-Engine Unity ist eine von *Unity Technologies* entwickelte Echtzeit Entwicklungsumgebung, mit der neben Spielen, auch Native-Applikationen, Webanwendungen, *Virtual-reality*-Anwendungen und vieles mehr entwickelt werden kann. Insgesamt sollen mehr als 25 Zielplattformen von der Engine angesteuert werden. Unity ist unter den gängigsten Betriebssystemen wie Windows, Linux und macOS nutzbar. Für die Entwicklung unterstützt die Engine sowohl den drei dimensionalen, als auch den zwei dimensionalen Raum [41].

Die Abbildung 2.7 zeigt einen kleinen Ausschnitt von der grafischen Benutzeroberfläche der

Unity-Engine. Wie man sehen kann ist die GUI in unterschiedliche Fenster aufgeteilt. Im Zentrum sieht man das ,,Scene"-Fenster, Objekte können hier auf die unterschiedlichsten Weisen modifiziert werden für die spätere Anwendung. Die Modifikation kann direkt in diesem Fenster geschehen oder aber mit der Hilfe des ,,Inspector"-Fensters. Eine solche Modifikation könnte z. B. sein, dass man einem *3D*-Objekt ein Skript übergibt und dessen Verhalten vorschreibt. Man unterscheidet die Objekte in unterschiedlichen Gesichtspunkten. Sei es die Form, die Position im Raum, Skalierung oder Rotation oder aber einfach nur die Aufgabe, die sie zu erfüllen hat. Es gibt unter den Objekten die Grundtypen wie Kugel und Zylinder oder aber Lichtquellen, Audiomechanismen, Partikelsysteme und viele mehr. Wie man schon erahnen kann, gibt es Objekte, die später bei der Anwendung sichtbar sein werden, aber auch welche, die der Nutzer der Anwendung wahrnimmt, aber nicht sieht. Alle Elemente, die in der Szene genutzt werden, sind einmal in dem Linken ,,Hierarchy"-Fenster neben dem ,,Scene"-Fenster aufgelistet. Einige dieser Elemente kann man direkt in der Hierarchy erzeugen, andere kommen aus dem ,,Assets"-Verzeichnis, was unten links angesiedelt ist. In dem ,,Assets"-Verzeichnis sind neben wiederverwendbaren Objekten, die im Kontext von Unity ,,Prefabs" genannt werden, sämtliche Ressourcen, die für die Entwicklung einer Anwendung nötig sind. Beispiele für solche Ressourcen wären Bilder, Audiodateien, Animationen und `C#`-Skripte.

In dieser Arbeit spielt Unity unter anderem deshalb eine wichtige Rolle, da SEE selbst innerhalb dieser Spiele-Engine entwickelt wird, sowie sämtliche Visualisierung, die im Rahmen dieser Arbeit entsteht.

2.4.2 Unity-Profiler

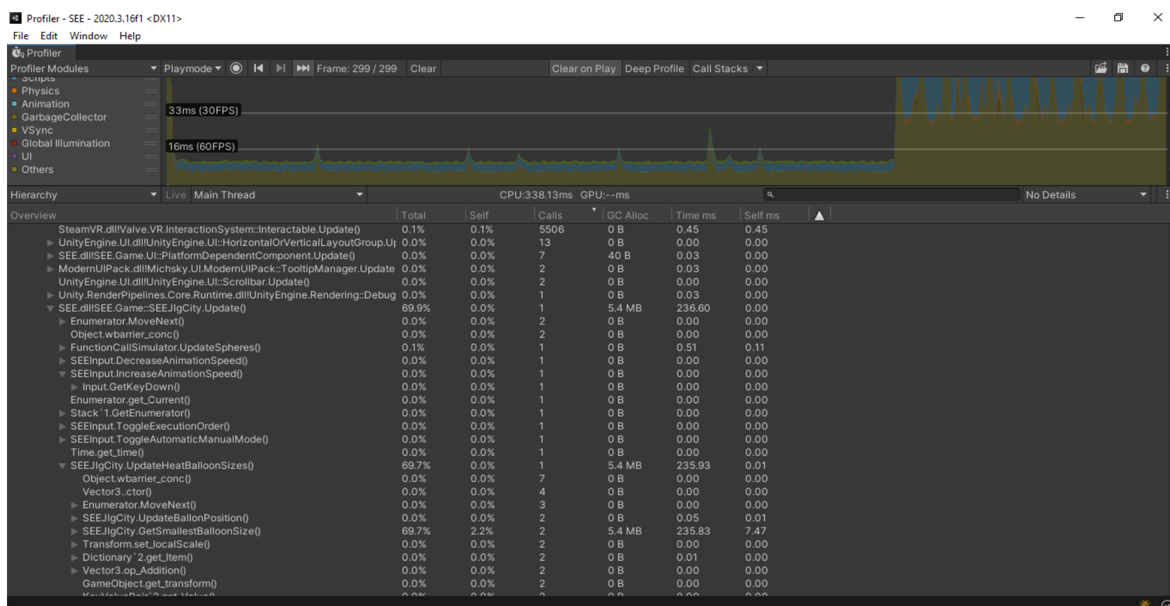


Abbildung 2.8: Die grafische Benutzeroberfläche von dem Unity-Profiler, wobei der Aufrufgraph dargestellt als eine Hierarchie, von den SEE-Laufzeitdaten zu sehen sind. [42].

Unter den vielen Werkzeugen, die Unity enthält, ist eines davon der Unity-Profiler, welcher in der Abbildung 2.8 zu sehen ist. Ein Werkzeug, um die Performanz-Daten zur Laufzeit einer Anwendung zu erfassen. Dabei können Daten über die CPU-Zeit, Speichernutzung, Rendering und Audio unter anderem erhoben und in unterschiedlichster Form repräsentiert werden. Mit diesen Informationen ist es möglich zu ermitteln, ob gewisse Skripte, Einstellungen oder

weitere Ressourcen die Performanz der betrachteten Anwendung beeinflussen [42].

2.4.3 Mono.Cecil

Eine bekannte Bibliothek in der C#-Szene, die Executable-Instrumentierung unterstützt ist Mono.Cecil. Diese Bibliothek wurde 2014 ins Leben gerufen von Evain und ist zu diesem Zeitpunkt als Open-Source-Projekt auf GitHub unter der MIT-Lizenz nutzbar [10]. Nach Evain hat die Bibliothek zwei Aufgaben:

- Es können *.NET*-Binärdateien analysiert werden, indem ein einfacher, dennoch effektiver *Object-Model* verwendet wird, ohne dass Binärdateien geladen und Reflection verwendet werden muss.
- Es können *.NET*-Binärdateien modifiziert werden, indem neue Metadaten-Strukturen hinzugefügt oder aber die Intermediate-Language angepasst wird.

Die *Mono.Cecil*-Bibliothek teilt sich in drei Paketstrukturen auf. Zum einen *Mono.Cecil*, wo sich die grundlegendsten Klassen befinden, um eine Dynamic-Linked-Library oder Executable zu laden und um diese modifizieren zu können. Außerdem Klassen, um bestimmte Basistypen aufzulösen, die man in C#-Programmen instrumentieren möchte. Zum anderen das Paket *Mono.Cecil.Cil*, dass Klassen enthält, die es ermöglichen auf Instruktionsebene unterschiedlichste Arten von Manipulation durchzuführen, sei es das Entfernen oder Abändern von bereits vorhandenen Instruktionen sowie das Hinzufügen neuer Instruktionen. Eine große Herausforderung an dieser Stelle ist es herauszufinden, wie der Compiler die einzelnen Instruktionen erzeugt, sie anordnet und wie die Abhängigkeiten zwischen den einzelnen Instruktionen sind. Das letzte Paket, welches für diese Arbeit von Relevanz ist, ist *Mono.Cecil.Rocks*, mit dem es möglich ist, die hinzugefügten Instruktionen mit den bereits vorhandenen zu synchronisieren. Dies ist deshalb wichtig, da jede Instruktion ein Label besitzt, nur neu hinzugefügte nicht. Durch die Synchronisation bekommen die neuen Instruktionen ebenfalls ein Label, um später Verweise durchführen zu können von einer Instruktion zur anderen.

Diese Bibliothek stellt die Basis von dem *TraceEmbedder* dar, der in dieser Arbeit entwickelt wird, zur Instrumentierung von C#-Anwendungen. Auf dem im Abschnitt 4.1 noch näher eingegangen wird.

2.4.4 ILSpy

Das Tool ILSpy, ist ein *.NET-assembly-browser* und Dekompilierer, dass in 2011 als Open-Source-Projekt auf GitHub unter der MIT-Licence seinen Anfang fand [29]. Nachfolgend werden einige Features der Software aufgelistet, die aus dem Englischen übersetzt und zum Teil umschrieben wurden [29]:

- Ein C#-Programm der als eine ausführbare Datei vorliegt, kann dekompiliert werden auf den zugrundeliegenden C#-Code, ähnlich wie der Code, der zur Kompilierzeit vorlag.
- Möglichkeiten zur Untersuchung von Metadaten einer Assembly
- Möglichkeiten zum Auffinden von Datentypen, Methoden und Eigenschaften in einem Assembly

Für diese Arbeit ist ein weiteres Feature von Interesse. Das Dekompilieren von Assemblies, um `C#`-Anwendungen auf die Ebene der Instruktionen herunterzubrechen. Damit können später leichter Fehler in der instrumentierten Anwendung entdeckt sowie Erkenntnis gewonnen werden, wie der Compiler bei der Instrumentierung vorgeht.

2.4.5 Sonstiges

Zusätzlich zu der bereits genannten Software, wird sowohl für die Entwicklung des Instrumentierung-Tools, als auch für die Visualisierung, die Programmiersprache „`C#`" verwendet. Die grundlegende Entwicklungsumgebung, die zum Programmieren verwendet wird, ist *Visual-Studio* von Microsoft [26]. Da die Entwicklungsumgebung, `C#` nur unter dem Betriebssystem Windows unterstützt, wurde sämtliche Entwicklung in *Windows-10-Home* vorgenommen.

KAPITEL 3
Entwurf

In diesem Kapitel wird auf die Anforderungen an diese Arbeit näher eingegangen und anhand von Entwürfen erläutert, wie diese umgesetzt werden. Zudem wird klar abgegrenzt von Komponenten, die bereits von Kipka [20], in Rahmen seiner Bachelorarbeit entwickelt wurden.

3.1 Anforderungen

Die Anforderungen für diese Arbeit wurden klar kommuniziert. Zum einen muss ein externes Tool entwickelt werden, der beliebige `C#`-Programme instrumentiert, um die Laufzeitdaten auf Methoden-Ebene erfassen zu können, die anschließend in ein geeignetes Dateiformat gespeichert werden. Zum anderen muss die Visualisierung in SEE so erweitert werden, dass die erhobenen Laufzeitdaten angemessen präsentiert werden, um anschließend die Daten auf Laufzeitflaschenhälse untersuchen zu können. Insgesamt entsteht durch das externe Tool und die Visualisierung ein Profiler, wie bereits erläutert, der SEE-Profiler. Bei der zu entwerfenden Visualisierung ist zudem auf die Kompatibilität mit der Visualisierung von Kipka zu achten. Auch die Datenzwischenspeicherung wurde aus diesem Grund identisch behandelt, um später eine einheitliche Schnittstelle zu gewährleisten.

3.1.1 Laufzeitdatenerfassung

An dem Tool, der für die Laufzeitdatenerfassung zuständig ist und in der Literatur oft als *Tracer* [[17], [24], [11]] bezeichnet wird, gibt es drei Anforderungen, auf die nun näher eingegangen wird. Wie schon im vorherigen Abschnitt erwähnt, muss dieser eine externe Anwendung darstellen, der in der Lage ist, jede beliebige `C#`-Anwendung zu instrumentieren. Bei der Erfassung selbst soll die Option bestehen, wie bei einem Debugger *breakpoints* festzulegen, ab wann Laufzeitdaten aufgezeichnet werden sollen bzw. wann dieser anhalten muss aufzuzeichnen. In jedem anderen Fall wird so lange aufgezeichnet bis das Zielprogramm geschlossen wird. Im letzteren Fall kann ein enormer Overhead entstehen, der die Performanz der Zielanwendung erheblich verringert. Die wichtigste Anforderung stellt jedoch die der Instrumentierung dar. Das zu entwickelnde Tool zur Instrumentierung soll nicht auf Quellcode-Instrumentierung basieren, sondern auf Executable-Instrumentierung. Dies soll zum einen den Fall abdecken, dass nicht immer eine Codebasis vorliegen kann, aber auch um das Zielprogramm nicht mit zusätzlichem Code auszudehnen. Es sei jedoch anzumerken, dass bereits Entwürfe erstellt und teilweise umgesetzt wurden zur Quellcode-Instrumentierung, die jedoch den Anforderungen nicht gerecht waren und aus diesem Grund verworfen wurden.

3.1.2 Kompakte Speicherung der Laufzeitdaten

Eine wichtige Überlegung bei der Erfassung von Laufzeitdaten ist, wie man diese in kompakter Form erfasst und zwischenspeichert. Da sich die Größe der aufgezeichneten Laufzeitdateien innerhalb von wenigen Minuten im Megabit und Gigabit bewegen und um Redundanzen zu vermeiden, muss ein geeignetes Dateiformat gewählt werden. Das JLG-Dateiformat welches noch im Abschnitt 3.3 näher erläutert wird, soll zu diesem Zweck genutzt und erweitert werden.

3.1.3 Visualisierung der Laufzeitdaten

An die Visualisierung in SEE wurden fünf Anforderungen gestellt, wie die erhobenen Daten präsentiert werden müssen:

1. Eine Kante soll entstehen zwischen Aufrufer und Aufgerufenem, wenn ein Methodenaufruf erfolgt. Dabei löst sich die Kante erst dann auf, wenn der Aufrufgraph einer Methode abgeschlossen ist.

2. Über die einzelnen Gebäude der Software-Stadt soll ersichtlich sein, wie oft ein Methodenaufruf erfolgt ist, in Form eines Strahls. Die Länge des Strahls ist abhängig von der Anzahl der Methodenaufrufe.

3. Klickt man auf eines der Gebäude, wird angezeigt, welche Methode wie oft aufgerufen wurde. Optional sollen bei jedem Methodenaufruf, die Methodenparameter und der Objektzustand einsehbar sein.

4. Die Aufzeichnung muss vorwärts und rückwärts, als eine Animation abspielbar sein. Entweder automatisch, mit einer bestimmten Geschwindigkeit oder manuell.

5. Es sollen Heatmaps entstehen, die andeuten, an welcher Komponente die CPU sich am meisten aufhält. Die CPU-Zeit, die benötigt wird zwischen Methodeneintritt und Methodenaustritt, soll den Farbgradienten eines Heatmaps bestimmen. Dabei könnte z. B. die Farbe Blau für eine minimale Auslastung und die Farbe Rot, als eine hohe Auslastung verstanden werden.

Im Laufe dieser Arbeit wurde Anforderung zwei und fünf zu einem zusammengeschlossen. Statt einem Strahl repräsentiert ein Luftballon über den jeweiligen Gebäuden sowohl die Anzahl an Methodenaufrufen als auch die CPU-Zeit. Grund hierfür ist, dass bereits ein Strahl existiert, der über einem Objekt erscheint, sobald auf das Objekt ein Mauszeiger bewegt wird, um den Namen des Objekts anzuzeigen. Auf den Luftballon, der auch später unter dem Namen ,,Heatballoon" behandelt wird, wird in Kapitel 3.4 noch detaillierter eingegangen. Zudem wurde auf die dritte Anforderung, optional den Objektzustand mitzuerfassen verzichtet, da im Rahmen dieser Arbeit keine Möglichkeit gefunden wurde dies umzusetzen, ohne das Rekursionsprobleme auftreten. Grund hierfür ist, dass der instrumentierte Code, Methoden der Zielanwendung ausführen und vice versa, bis ein *Stack-Overflow-Fehler* entsteht. Mit der Zielanwendung, ist eine beliebige instrumentierte `C#`-Anwendung zu verstehen. Des Weiteren wird auf den ersten Punkt nicht weiter eingegangen, da bereits die meiste Logik von Kipka für die Anforderung implementiert wurde in seiner Arbeit und deshalb nur wenige Änderungen vorgenommen werden mussten, um die Anforderung umzusetzen.

3.2 Instrumentierung

Das zu entwickelnde externe Tool, das im weiteren Verlauf als ,,TraceEmbedder” bezeichnet wird, soll wie in den Anforderungen angegeben auf Executable-Instrumentierung basieren. Der Entwurf in Form eines UML-Klassendiagramms dazu ist einmal in der Abbildung 3.1 zu sehen. Das Klassendiagramm zeigt alle Klassen und die Beziehungen untereinander, die den *TraceEmbedder* ausmachen. Die gleichnamige Klasse *TraceEmbedder* stellt dabei die Hauptklasse des zu entwickelnden Tools dar. Sie bündelt dazu die Hauptlogik, um `C#`-Anwendungen zu instrumentieren und bedient sich dazu der *Mono.Cecil*-Bibliothek. Die Klassen *FileManager*, *ReferenceResolver*, *ReferenceStruct* und *AssemblyManager* fungieren dabei als Hilfsklassen, die unterschiedliche Aufgaben übernehmen. Dazu gehört das Laden des Hauptassemblies der zu instrumentierenden `C#`-Anwendung, um die einzelnen Methoden mit eigenem Code zu versehen, damit die Laufzeitdaten erfasst werden können. Zum anderen das Importieren der wichtigsten Datentypen bzw. Klassen und Methoden, die im weiteren Verlauf zur Instrumentierung genutzt werden. Auf die Hilfsklassen und den Einsatz von den *TraceEmbedder* wird im Kapitel 4 noch näher eingegangen.

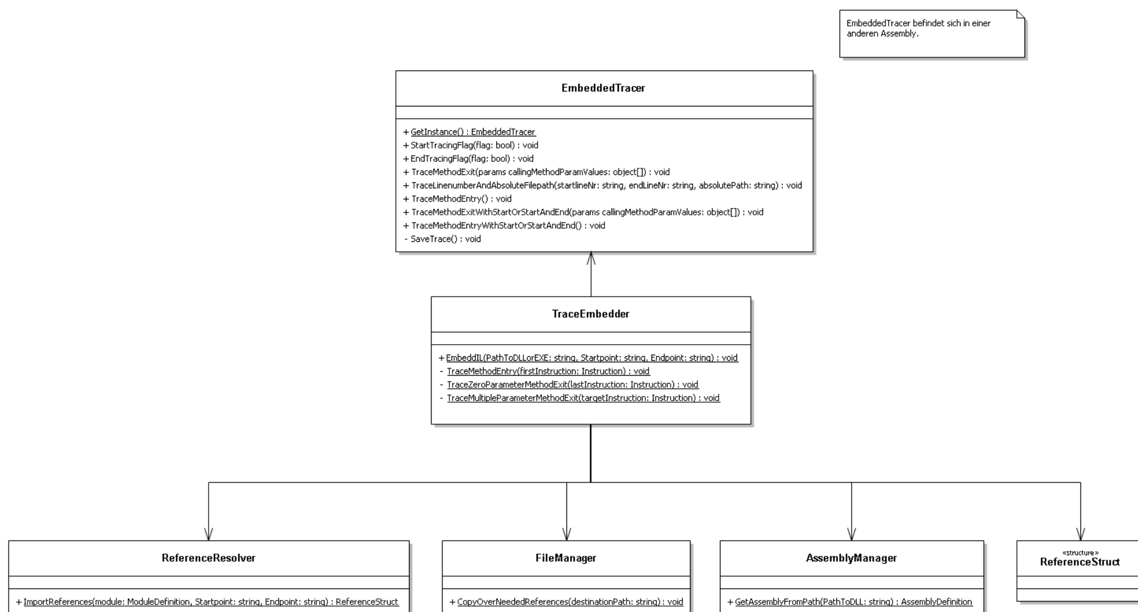


Abbildung 3.1: Sämtliche Klassen und Beziehungen, die für die Instrumentierung nötig sind, als UML-Klassendiagramm dargestellt.

3.2.1 Einzubettende Klasse

Die Hauptklasse, die eingebettet wird in sämtliche Methoden der `C#`-Anwendungen, ist die *EmbeddedTracer*-Klasse, die in der Abbildung 3.1 zu sehen ist. Sie ist für das Erfassen der Laufzeitdaten auf Methoden-Ebene zuständig. Die Laufzeitdaten umfassen Methodenparameter, die Ausführungszeit einer Methode, Metadaten sowie der Aufrufgraph, der die Aufrufbeziehungen zwischen Methoden darstellt sowie Informationen über den Methodeneintritt bzw. Methodenaustritt jeder einzelnen instrumentierten Methode enthält. Bei den Metadaten handelt es sich um die Zeilennummer von dem Methodenheader, als auch das Ende des Methodenkörpers sowie der absolute Pfad zu der Quelldatei, in der sich die jeweilige Methode befindet. Mit diesen Informationen sollen später in dem größeren Fenster, dass bereits in der

Abbildung 2.1 zu sehen war, die Quelldatei eingelesen und an der richtigen Stelle die aufgezeichnete Methode eingeblendet werden, zur Laufzeit der Visualisierung in der JLGCity. Für die Metadaten wird eine `PDB`-Datei vorausgesetzt, da durch das reine Kompilieren durch einen Compiler die genannten Daten wie Zeilennummern und absoluter Pfad in der finalen Assembly der Anwendungen entfallen. Der eigentliche Zweck einer solchen `PDB`-Datei, ist für das Debuggen gedacht, soll jedoch im Rahmen dieser Arbeit für die Erhebung der Metadaten benutzt werden, da keine Alternative gefunden wurde, um an die genannten Daten heranzukommen. Sollte keine *PDB*-Datei vorliegen, werden die Metadaten bei der Aufzeichnung ignoriert.

Da die EmbeddedTracer-Klasse sämtliche Daten aufnimmt und nur eine Instanz dieser Klasse Sinn ergibt, wird aufgrund dessen das Singleton-Entwurfsmuster angewandt. Weiterhin besitzt die Klasse einen *Finalizer*, um zum Zeitpunkt des Abräumens der einzig existierenden Instanz durch den *Garbage-Collector*, die erfassten Daten zwischenzuspeichern. Dies ist unter anderem der Fall, wenn eine Anwendung geschlossen wird.

Angesiedelt ist der EmbeddedTracer in einer isolierten Assembly ,,TraceUtil.dll". Die Assembly wird nach der Instrumentierung mit in das Verzeichnis der Zielanwendung platziert. Grund hierfür ist, dass durch die Instrumentierung Abhängigkeiten entstehen, bei der die Zielanwendung herausfinden muss, in welcher Assembly sich die EmbeddedTracer-Klasse befindet, um diese aufzulösen. Der Einsatz des EmbeddedTracers wird im Kapitel 4 noch näher besprochen.

3.2.2 Grafische Benutzeroberfläche

Als Schnittstelle für den Zugang des *TraceEmbedder*, ist eine grafische Benutzeroberfläche in *Windows-Forms* geplant, um den Umgang mit dem *TraceEmbedder* zu erleichtern. Bei *Windows-Forms* handelt es sich um einen *Framework*, welcher ein Teil von *Microsoft .Net* ist und zur Gestaltung von Desktopanwendungen dient [27]. Ein erstes Design der Anwendung ist in der nachfolgenden Abbildung zu sehen.

Abbildung 3.2: Entwurf der grafischen Benutzeroberfläche, die als Einstiegspunkt für den *TraceEmbedder* dient.

Die einzelnen Eingabefelder, stellen Möglichkeiten der Parametrisierung bereit, für die Hauptmethode des *TraceEmbedders*. Die Angabe des absoluten Pfads, soll verpflichtend sein, dies soll durch den Stern gleich rechts neben dem Label ,,Target path" zum Ausdruck gebracht werden. Die restlichen Felder sind optional, so wie in den Anforderungen vorgesehen.

3.3 JLG-Dateiformat

Bei dem verwendeten Dateiformat handelt es sich um das JLG-Dateiformat von Kipka, was abgekürzt für „Java-Log" steht [20]. Dieses Format wurde so ausgelegt, dass der Speicherbedarf mit zwei unterschiedlichen Methoden minimiert wird. Es gibt eine *Lookup-Tabelle* in der redundante Informationen zwischengespeichert werden und einen nummerischen Schlüssel, der diese Daten identifizieren kann und den man anstelle der redundanten Informationen verwendet. Bei den redundanten Daten handelt es sich in diesem Fall um relative Pfade zu Methodennamen und die Felder einer Klasse bzw. eines Objekts. Um einen Einblick zu bekommen, wie so ein JLG-Dateiformat aufgebaut ist, wird in dem nachfolgenden Listing, ein Beispiel dazu präsentiert:

```
$[C:\[...]\SEE\Assets\SEE\Utils\ColorPalette.cs,
    ↪ C:\[...]\SEE\Assets\SEE\Game\AbstractSEECity.cs,[...]]
-/0>39
colorIndex=0
§=1000,22
colorIndex=0
/-0>42
-/0>39
colorIndex=0,2
§=1043,05
[...]
*-0=ColorPalette.Viridis(System.Single);-1=AbstractSEECity.Hierarchical_Edge_Types();-2
=AttributeNamesExtensions.Name(SEE.DataModel.DG.NumericAttributeNames);-3=PlayerSettings.
Awake();-4=VRStatus.Enable(System.Boolean);-5=PlayerSettings.CreatePlayer(SEE.GO.
PlayerInputType);#0=constant1;[...]
```

Listing 3.1: Gekürzte Fassung des Inhalts einer JLG-Datei.

Nachstehend sind nach Kipka die einzelnen Symbole bzw. Identifizierer beschrieben, die in den Listing 3.1 vorkommen und noch weitere, die ebenfalls ein Teil des JLG-Dateiformats sind [20].

- „-...>...". Der - indiziert ein neues ausgeführtes Statement. Nach dem „-" kommt ein Wert aus der Lookup-Tabelle, der die Methode, in der die Zeile steht, repräsentiert. Nach dem „>" kommt dann die Zahl der Zeile.
  - Betritt ein Statement eine Methode, wird anstelle von „-" „-/" verwendet.
  - Verlässt ein Statement eine Methode, wird anstelle von „-" „-/" verwendet.
- „=>". Die Kombination der Zeichen „=>" am Anfang einer Zeile zeigen den Rückgabewert des letzten Statements, das eine Methode verlassen hat.
- „#". Ist dieses Zeichen am Anfang einer Zeile, steht in dieser eine Veränderung eines Feldwertes.
- „$". Nach diesem Zeichen kommen die Pfade zu allen Klassendateien des Zielprogramms.
- „*". Dieses Zeichen markiert den Start der Lookup-Tabellen, die immer am Ende der Log-Datei stehen.
- Ist kein Zeichen an der ersten Stelle einer Zeile, so steht in dieser Zeile der Wert einer lokalen Variable, die zum letzten ausgeführten Statement gehört. Für diese Information wurde kein Zeichen als Identfizierer gewählt, da es die meisten Zeilen der Log-Datei belegt und man so weiteren Speicher sparen kann.

Zusätzlich zu den vorhandenen Identifizierern wird bei der Verfassung dieser Arbeit ein weiterer hinzugefügt. Der Identifizierer dient zur Erfassung der *absoluten Zeit* einer Methode:

- „§". Wenn dieses Zeichen als Präfix einer Zeile vorkommt, dann gibt sie von der Methode, die noch nicht geschlossen wurde, mit den Zeichen „-/", die absolute Zeit in Millisekunden an.

Bei der *absoluten Zeit* einer Methode handelt es sich um die CPU-Zeit, die vom Beginn eines Methodenkörpers bis zum Ende der Methode benötigt wird, inklusive der Zeit der inneren Methoden. Aus dieser Zeit wird im späteren Verlauf die *eigene Zeit*, die eine Methode benötigt, errechnet, indem die Zeiten der inneren Methoden von der *absoluten Zeit* abgezogen werden.

3.4 JLG-Parser

Damit Laufzeitdaten, die durch den Einsatz des *EmbeddedTracer* erhoben wurden, in die SEE-Anwendung kommen, muss ein spezieller Parser entwickelt werden. Ein Parser ist ein Programm, das als eine Art Übersetzer fungiert und Daten aus einer Quelle in eine andere überführt. In diesem Fall wird die JLG-Datei, die das Resultat der erhobenen Laufzeitdaten darstellt, eingelesen und für die weitere Visualisierung bereitgestellt. Aus Kompatibilitätsgründen wird der JLG-Parser von Kipka verwendet, der die meiste Logik bereits besitzt. Der Parser wird zusätzlich erweitert, um die *absolute Zeit*, die eine Methode benötigt, auslesen zu können. Eine Besonderheit des Parsers ist, dass für die eingelesenen Daten sogenannte Datenklassen bereitstehen, um diese für die spätere Wiederverwendung strukturiert zwischenspeichern zu können in bestimmten Feldern. Bei den Datenklassen handelt sich um die Klassen *JavaStatement* und *ParsedJLG* welche nun näher erläutert werden [20]:

JavaStatement Die Datenklasse stellt, wie der Name schon sagt, ein Statement aus Java dar, welcher Lokalisationsdaten enthält. Diese Daten beinhalten, in welcher Zeile einer Datei sich ein Statement befindet, zu welcher Methode es angehört, welche Instanzvariablen in diesem Statement sich ändern und vieles mehr. Neben den genannten Lokalisationsdaten, die ursprünglich erhoben wurden, werden im Rahmen dieser Arbeit zwei weitere hinzugefügt. Ein Feld für die *absolute Zeit* und ein weiterer für die *eigene Zeit*.

ParsedJLG Die Datenklasse ist die Repräsentation aller erhobenen Daten aus der JLG-Datei. In dieser Datenklasse werden neben den *JavaStatement* Objekte, auch die einzelnen Dateipfade zu den jeweiligen Klassen, die aufgezeichnet werden in einer Liste zwischengespeichert. Die Lookup-Tabelle, die aus der JLG-Datei entnommen wird, wird zu diesem Zweck in zwei unterschiedliche Listen aufgeteilt. Eine Liste stellt eine Lookup-Tabelle dar, um die Position eines *JavaStatement*-Objekts zu ermitteln. Eine andere Liste stellt eine Lookup-Tabelle dar, um die Feldernamen aufzulösen.

3.5 Visualisierung

Die Visualisierung der Daten ist ein Zusammenschluss aus den dynamischen Aufrufgraphen von Groß [16], die bereits vorhandene Visualisierung von Kipka, die im Abschnitt 2.1.3.2 bereits behandelt wurde und den weiteren Darstellungen, die zusätzlich mit dieser Arbeit hinzukommen wie die Heatballoons und das Profiler-Fenster, die in den nächsten Abschnitten

näher erläutert werden. Da die Visualisierung von Kipka generisch entworfen wurde, wird auf die Animation zurückgegriffen, die das Vorwärts- bzw. Rückwärts-ablaufen der Laufzeitdaten erlaubt sowie weitere Funktionen, die noch im Abschnitt 3.6 besprochen werden. An dieser Stelle sei zu erwähnen, dass die meiste Logik der Visualisierung von Kipka in der Klasse *SEEJlgCityAnimation* angesiedelt ist und diese intensiv verwendet wird von den Heatballoons sowie von den Profiler-Fenstern.

3.5.1 Heatballoons

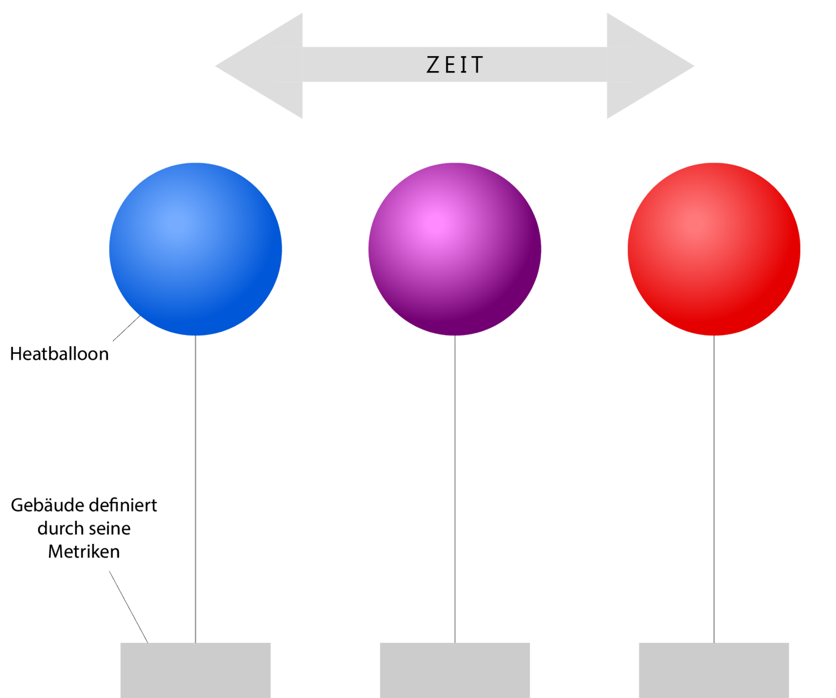


Abbildung 3.3: Entwurf der Heatballoons, wobei zusätzlich der Abkühlungsprozess dargestellt ist.

Neben den genannten Punkten im vorherigen Abschnitt, sind wie bereits angekündigt, weitere Visualisierungen Teil dieser Arbeit. Zum einen wird für die absolute und eigene Zeit einer Methode ein 3D-Objekt in Form eines Ballons vorgesehen, was über die jeweilige Klasse bzw. das jeweilige Gebäude schwebt. Wie in der Abbildung 3.3 zu sehen, ist es ein Kugel-Objekt. Die Größe der Kugeln sollen dabei anhand der Anzahl der Methodenaufrufe wachsen und die Farbe die aktuelle CPU-Zeit im Verlauf widerspiegeln. Es soll aber auch die Möglichkeit bestehen diese Funktionalität umzudrehen, sodass die CPU-Zeit im Verlauf die Größe und die Methodenanrufe wiederum die Farbe repräsentiert. Die Farbe soll unter anderem optional wieder abkühlen können, wenn eine bestimmte Zeit auf einen *Heatballoon* nicht zugegriffen wurde. Der Grundgedanke dafür stammt aus einer Metapher der realen Welt, bei dem eine Kugel, die immer wieder befeuert wird, erhitzt und einen roten Ton annimmt und wenn eine gewisse Zeit nichts passiert, diese sich wieder abkühlt.

3.5.2 Profiler-Fenster

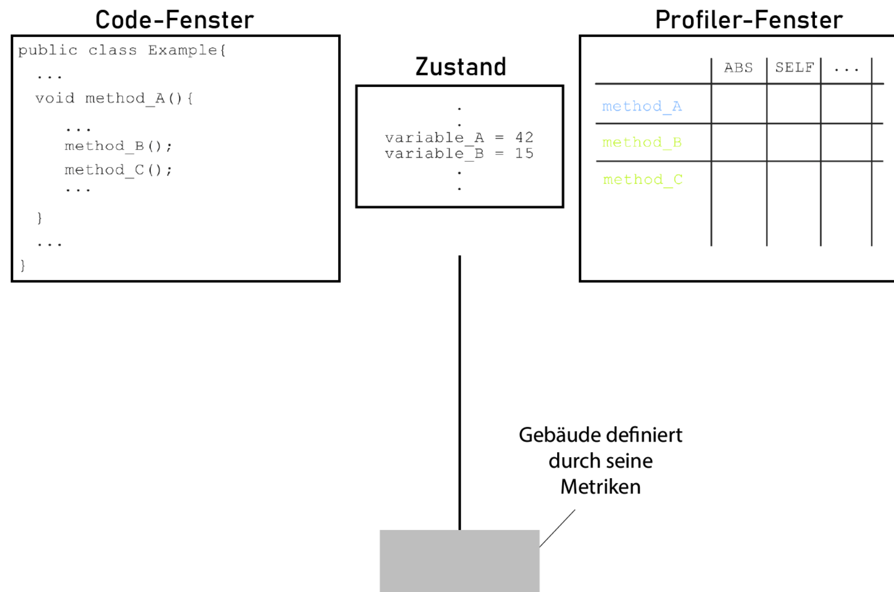


Abbildung 3.4: Entwurf des Profiler-Fenster.

Ein weiteres Textfenster „Profiler-Fenster" wie in der Abbildung 3.4 zu sehen, soll neben den bereits erläuterten Fenstern von Kipka platziert werden, da die Heatballoons allein noch viel zu abstrakt sind. Aus diesem Grund wird textuell bzw. tabellarisch für jede Methode folgende Informationen in dem Fenster eingeblendet:

- Die *absolute Zeit* „abs", die angibt, wie viel Zeit eine Methode zur Ausführung benötigt, inklusive der Zeit der inneren Methoden.
- Die *eigene Zeit* „self", die angibt, wie viel Zeit eine Methode zur Ausführung benötigt, unabhängig von der Zeit der inneren Methoden.
- Jeweils die durchschnittliche Zeit von der *absoluten* „avg(abs)" bzw. *eigenen Zeit* „avg-(self)". Es kann nämlich vorkommen, dass derselbe Methodenaufruf unterschiedliche Zeiten aufweist.
- Ein numerischer Wert „count", der angibt, wie oft bereits eine Methode aufgerufen wurde.
- Ein weiterer nummerischer Wert „calls", der angibt, wie viele andere inneren Methodenaufrufe aufgetreten sind.
- Alle inneren Methoden sollen ebenfalls aufgelistet werden, wenn ein *Methoden*-Objekt in der JLGCity selektiert wird, mit ihren eigenen Werten für die bereits genannten Punkte. Zur Unterscheidung, werden sie durch eine andere Farbe hervorgehoben.

3.6 Bedienung

SEE unterstützt unterschiedliche Bedienungsmöglichkeiten. Dazu gehört einmal die Bedienung am Desktop mit dem Keyboard und Maus, Gamepad und oder Touchscreen, bis hin

zu der Bedienung mit einem Head-Mounted-Display durch Controller oder Leap-Motion-Sensoren. In der folgenden Tabelle 3.1 sind einige Tastenkombinationen abgebildet für die Desktopumgebung, die vor dieser Arbeit schon implementiert wurden [23]:

Tastenkombination	Beschreibung
W/Pfeil nach oben	Vorwärts bewegen
S/Pfeil nach unten	Rückwärts bewegen
A/Pfeil nach links	Nach links (Seitenschritt) bewegen
D/Pfeil nach rechts	Nach rechts (Seitenschritt) bewegen
Q	Erheben
E	Sinken
Shift	Beschleunigen
+	Erhöht die Grundgeschwindigkeit der Bewegungen
-	Verringert die Grundgeschwindigkeit der Bewegungen

Tabelle 3.1: Standard Bedienungsmöglichkeiten in der Desktopumgebung

Neben der grundlegenden Bedienung existieren für unterschiedliche Teile von SEE, auch unterschiedliche Tastenkombinationen und Einstellungsmöglichkeiten. So auch für die JLGCity, die für diese Arbeit von größter Relevanz ist. Wie bereits in dem Abschnitt 2.1.3.2 erläutert, kann man die Animation in der JLGCity manuell oder automatisch durchlaufen. Im Folgenden sind in der Tabelle 3.2 alle Tasten aufgelistet, die für die Animation und damit auch für die Visualisierung, die in dieser Arbeit entworfen wird, von Relevanz sind, aber auch gleichzeitig einige der Funktionalitäten in der JLGCity widerspiegeln.

Tastenkombination	Beschreibung
I	Umschalten zwischen automatischer / manueller Ausführungsmodus
B	Ausführen bis der nächste Finalizer erreicht wurd
#	Ausführung auf die erste Anweisung setzen
<	Vorherige Anweisung ausführen
>	Nächste Anweisung ausführen
O	Umschalten zwischen der Ausführungsreihenfolge (vorwärts/rückwärts)
Pfeil nach unten	Halbiert die Ausführungsgeschwindigkeit
Pfeil nach oben	Verdoppelt die Ausführungsgeschwindigkeit

Tabelle 3.2: Manuelle Steuerung der Animation in JLGCity

Zusätzlich zu den bereits genannten Möglichkeiten der Bedienung, sind weitere Einstellungsmöglichkeiten Teil der zu entwickelnden Visualisierung, die in den ,,Inspector"-Fenster von Unity später eingestellt werden können. Die neu hinzukommenden Toggle-Funktionen und variable Eingabemöglichkeiten für die Heatballoons werden nachfolgend in der Tabelle 3.3 einmal aufgelistet und näher erläutert:

Einstellung	Beschreibung
Cooldown	Es lässt sich festlegen, ob der Heatballoon abgekühlt werden soll oder aber immer weiter erhitzt wird.
Color by CPU-Time	Es lässt sich festlegen, ob die Farbe des Heatballoons durch die CPU-Zeit oder durch die Anzahl der Methodenaufrufe bestimmt wird.
Absolute Time	Es lässt sich festlegen, ob die *absolute* oder *eigene Zeit* während der Animation betrachtet wird.
Upper bound time	Es wird die Obergrenze festgelegt, ab wann der Heatballoon maximal erhitzt ist, ausgehend von der CPU-Zeit.
Upper bound call	Es wird die Obergrenze festgelegt, ab wann der Heatballoon maximal erhitzt ist, ausgehend von der Anzahl der Methodenaufrufe.
Heat reset time	Es wird festgelegt in Sekunden, ab wann der Heatballoon abgekühlt wird, wenn keine weiteren Laufzeitdaten für diesen vorgesehen sind.
Scale factor	Es wird festgelegt, was auf die Größe des Heatballoons hinzuaddiert wird, sobald der Zähler für einen Methodenaufruf erhöht wird.
Initial distance	Legt die initiale Distanz zwischen einem Gebäude und dem korrespondierenden Heatballoon fest.

Tabelle 3.3: Toggle-Funktionen und variable Eingabemöglichkeiten für die Heatballoons zur Manipulation der Animation im ,,Inspector"-Fenster.

KAPITEL 4

Implementation

In diesem Kapitel wird auf die Implementierung von den TraceEmbedder, EmbeddedTracer, Hilfsklassen sowie Änderungen, die in SEE vorgenommen wurden, eingegangen. Abgeschlossen wird das Kapitel mit Tests, die durchgeführt wurden, um die Funktionalität der zu entwickelnden Software zu überprüfen. An dieser Stelle ist zu erwähnen, dass in diesem Abschnitt oftmals von „Zielanwendung" bzw. „Zielmethode" gesprochen wird. Damit sind die zu instrumentierenden `C#`-Anwendungen bzw. Methoden aus diesen Anwendungen gemeint.

4.1 TraceEmbedder

Die wichtigste Methode in dem TraceEmbedder ist *EmbeddIL()*. Sie bündelt die gesamte Logik, die nötig ist, um `C#`-Anwendungen zu instrumentieren. Dabei nimmt sie drei Argumente entgegen. Das erste Argument stellt den absoluten Pfad zu der Executable (EXE) bzw. Dynamic-Linked-Library (DLL) der zu instrumentierenden Anwendung dar, in der sich alle Module, Klassen und Methoden befinden. Mit dem zweiten Argument kann bestimmt werden, ab wann der EmbeddedTracer beginnen soll, mit der eigentlichen Aufzeichnung der Laufzeitdaten. Das letzte Argument kann optional bestimmen, an welchem Punkt die Aufzeichnung zum Halt kommen soll. Die Angabe der Methode ab wann aufgezeichnet bzw. die Aufzeichnung gestoppt werden soll, wird in der folgenden Form angegeben:

{*Paketname.*}**Klassenname.Methodenname*

Am Anfang wird die Paketstruktur aufgelistet, in der sich die jeweilige Klasse befindet, anschließend folgt der Klassenname und zum Schluss der Methodenname. Sollte sich eine Klasse in keinem Paket bzw. keiner Paketstruktur befinden, kann die Angabe des Pakets weggelassen werden. Eine Sache, die man berücksichtigen muss ist, dass die Angabe eines einfachen Methodennamen, zu Mehrdeutigkeiten führen kann, wenn in der gleichen Klasse mehrere Methoden denselben Namen haben durch eine Überladung. Die Mehrdeutigkeit wird in diesem Fall zugelassen, da durch die reiche `C#`-Syntax zu viele Sonderfälle existieren, die mitbetrachtet werden müssten.

Nachfolgend sind die wichtigsten Schritte aufgelistet, die der TraceEmbedder durchläuft, bis das Laufzeitverhalten einer `C#`-Anwendung aufgezeichnet werden kann:

Validierung Noch bevor die *EmbeddIL()*-Methode zur Ausführung kommt, wird die Eingabe des Benutzer validiert. Dabei wird überprüft, ob der angegebene absolute Pfad zu der jeweiligen EXE- bzw. DLL-Assembly der Zielanwendung gültig ist. Sollten keine Argumente übergeben werden, in dem Fall keines der Felder der grafischen Benutzeroberfläche, die in der Abbildung 4.1 zu sehen ist, ausgefüllt werden, wird dies durch eine Warnung kenntlich

gemacht. An dieser Stelle sei zu erwähnen, dass sich die Angabe eines Assemblies je nach `C#`-Anwendung variieren kann. Bei Unity-Anwendungen wie SEE, muss dazu die DLL-Assembly zur Instrumentierung ausgewählt werden. Bei einer normalen `C#`-Anwendung wird je nach Anwendung entweder eine EXE- oder DLL-Assembly erwartet. Bei falscher Angabe wird von der *Mono.Cecil*-Bibliothek die Assembly zurückgewiesen, mit der Begründung, dass sich die Assembly in einem falschen Format befindet. Die Ursache konnte nicht ermittelt werden, deshalb ist es unumgänglich beide Assemblies einmal auszuprobieren.

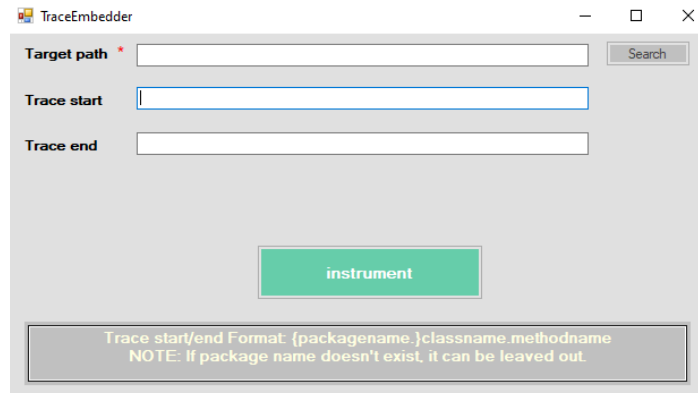


Abbildung 4.1: In *Windows-Forms* entwickelte grafische Benutzeroberfläche für den TraceEmbedder.

Ladeprozess Nach der Validierung der Eingabe, wird die zu instrumentierende Assembly mithilfe der *AssemblyManager.GetAssemblyFromPath()*-Methode in die *AssemblyDefinition*-Klasse, die aus dem *Mono.Cecil*-Paket stammt, geladen. Von der *AssemblyDefinition* aus kommt man an das Hauptmodul ran, wo sich sämtliche selbstdefinierte Klassen aus der Zielanwendung befinden. Das Hauptmodul selbst wird in die *ModuleDefinition*-Klasse geladen, welcher ebenfalls in dem *Mono.Cecil*-Paket definiert ist. Beide Klassen werden im Anschluss in dem Schritt „Iterationen" intensiv genutzt, um die Instrumentierung durchzuführen.

Abhängigkeitsauflösung Alle benötigten Datentypen und Methoden wie z. B. die Methoden von den EmbeddedTracer, die zur Instrumentierung nötig sind, werden importiert mit der *ImportReferences()*-Methode aus der *ReferenceResolver*-Klasse. Grund für den Import ist unter anderem, dass *Mono.Cecil* genauesten wissen muss, welche Datentypen bzw. Methoden aus welchen Assemblies instrumentiert werden sollen. Anschließend werden die importierten Datentypen und Methoden in dem *ReferenceStruct* aufbewahrt, für die spätere Wiederverwendung in diversen Methoden, um die Parameterzahl gering zu halten.

Iterationen Nachdem die vorherigen Schritte abgearbeitet wurden, wird in dieser Phase über jede Klasse aus dem Hauptmodul iteriert und die einzelnen Methoden instrumentiert. Die *ILProcessor*-Klasse aus dem *Mono.Cecil.Cil*-Paket wird zu diesem Zweck verwendet. Sie bietet die Möglichkeit, *Mono.Cecil*-Instruktionen vor den bereits bestehenden Instruktionen einer Zielmethode hinzuzufügen, mit der *InsertBefore()*-Methode oder aber nach einer Instruktion mithilfe der *AfterBefore()*-Methode. Diese Funktionalität wird dazu genutzt, um die EmbeddedTracer-Klasse und dessen Methoden in die Methoden der Zielanwendung einzubetten, um die bereits genannten Laufzeitdaten erfassen zu können. Eine Besonderheit, die mitbetrachtet wird ist, dass der bereits existierende Code einer Methode mit einem *try-finally*-Block umschlossen wird, sodass der instrumentierte Code ausgeführt wird, selbst

wenn ein Fehler geworfen werden sollte. Dies soll vor allem dazu dienen, Inkonsistenzen bei den erhobenen Laufzeitdaten zu vermeiden. In den Listings 4.1 und 4.2 wird einmal beispielhaft gezeigt, wie eine Instrumentierung einer Methode aussehen kann. Die nähere Erläuterung der einzelnen eingebetteten Methoden wird im Abschnitt 4.2 vorgenommen. Die nachfolgenden Schritte, beschreiben, was innerhalb der Iteration genauer passiert.

Aufzeichnungsbegrenzung Wird der Start- bzw. Endpunkt angegeben, wann die Aufzeichnung starten oder aber stoppen soll, wird bei der Iteration, die korrelierende Zielmethode, mit den dafür bestimmten Methoden aus EmbeddedTracer instrumentiert. Im Falle dessen, dass nur der Startpunkt angegeben ist oder sowohl Startpunkt als auch der Endpunkt, werden die Methoden *TraceMethodEntryWithStartOrStartAndEnd()*, *TraceMethodExitWithStartOrStartAndEnd()*, *StartTracingFlag()* und *EndTracingFlag()* aus EmbeddedTracer instrumentiert. Die ersten genannten beiden Methoden stellen damit lediglich eine extra Schicht der Methoden *TraceMethodEntry()* und *TraceMethodExit()* dar und dienen dazu Grenzen setzen zu können. Der letzte Fall, wenn nur der Endpunkt betrachtet werden soll, kann die extra Schicht entfallen. Eine einfache Fallunterscheidung in den zuletzt genannten zwei Methoden, sorgt dafür, dass diese nicht mehr betreten werden, wenn der Endpunkt erreicht wurde.

```
public static int ExampleMethod(int param)
{

       return InnerCall(param);

}
```

Listing 4.1: Vor der Instrumentierung der *ExampleMethod*-Methode.

```
public static int ExampleMethod(int param)
{
       TracerUtil.EmbeddedTracer.GetInstance().TraceLinenumberAndAbsoluteFilepath("20", "23",
           ↪ "C:\\[...]\\Program.cs");
       TracerUtil.EmbeddedTracer.GetInstance().TraceMethodEntry();
       try
       {
              return InnerCall(param); // Der vorherige Code
       }
       finally
       {
              TracerUtil.EmbeddedTracer.GetInstance().TraceMethodExit(param);
       }
}
```

Listing 4.2: Nach der Instrumentierung der *ExampleMethod*-Methode.

In Listing 4.3 sieht man einmal die einzelnen Instruktionen für die *ExampleMethod()*-Methode aus dem Listing 4.1, die vor der Instrumentierung galten. Das Listing 4.4 zeigt die modifizierte Version der Instruktionen, nachdem die Instrumentierung durchgeführt wurde. Was dabei auffällt ist, dass neue Instruktionen hinzugekommen sind, bestehende entfernt oder ersetzt wurden. Das Ersetzen und Entfernen ist deshalb wichtig, da der bestehende Code angepasst werden muss, vor allem auf den neu hinzukommenden *try-finally*-Block.

```
IL_0000: nop
IL_0001: ldarg.0
IL_0002: call int32 [...]
IL_0007: stloc.0
IL_0008: br.s IL_000a

IL_000a: ldloc.0
IL_000b: ret
```

Listing 4.3: Instruktionen von der *ExampleMethod*-Methode, vor der Instrumentierung.

```
IL_0000: call class [...]
    ↪ TracerUtil.EmbeddedTracer::GetInstance()
IL_0005: ldstr "20"
IL_000a: ldstr "23"
IL_000f: ldstr "C:\\[...]\\Program.cs"
IL_0014: callvirt instance void [...]
    ↪ EmbeddedTracer::TraceLinenumberAndAbsolute-
      Filepath(string, string, string)
IL_0019: call class [...] EmbeddedTracer::GetInstance()
IL_001e: call instance void [...]
EmbeddedTracer::TraceMethodEntry()
.try
{
        IL_0023: nop
        IL_0024: ldarg.0
        IL_0025: call int32 [...]
        IL_002a: stloc.0
        IL_002b: br.s IL_002d

        IL_002d: ldloc.0
        IL_002e: stloc.1
        IL_002f: ldloc.1
        IL_0030: stloc.2
        IL_0031: leave.s IL_0050
} // end .try
finally
{
        IL_0033: call class [...]
            ↪ TracerUtil.EmbeddedTracer::GetInstance()
        IL_0038: ldc.i4.s 1
        IL_003a: newarr [mscorlib]System.Object
        IL_003f: dup
        IL_0040: ldc.i4.s 0
        IL_0042: ldarg.s param
        IL_0044: box [mscorlib]System.Int32
        IL_0049: stelem.ref
        IL_004a: call instance void [...]
            ↪ EmbeddedTracer::TraceMethodExit(object[])
        IL_004f: endfinally
} // end handler

IL_0050: ldloc.2
IL_0051: ret
```

Listing 4.4: Instruktionen von der *ExampleMethod*-Methode, nach der Instrumentierung.

Metadaten Wie bereits im Abschnitt 3.2.1 erläutert, kann durch das Vorhandensein einer PDB-Datei, Metadaten miterfasst werden. Dabei wird der angegebene absolute Pfad, mit dem gleichnamigen Assembly-Namen, jedoch mit der Dateiendung „.pdb", durchsucht. Sollte eine PDB-Datei vorliegen, so ist es möglich aus den Zielmethoden unterschiedliche Debugin-Informationen zu entziehen. Die *SequencePoint*-Klasse aus dem *Mono.Cecil.Cil*-Paket enthält dabei die Zeilennummern sowie den absoluten Pfad zu der Quelldatei, in der sich die jeweilige Zielmethode zur Kompilierzeit befand. Die entzogenen Daten werden im Anschluss bei der Instrumentierung der *TraceLinenumberAndAbsoluteFilepath()*-Methode übergeben. Die zuletzt genannte Methode kommt aus dem EmbeddedTracer und wird im Abschnitt 4.2 näher beschrieben.

Methodenparameter-Erfassung Es wird zwischen drei Methoden aus dem TraceEmbedder unterschieden, die zur Einbettung von Code zuständig sind, um später mit dem EmbeddedTracer Methodenparameter erfassen zu können. Es gibt die *TraceParameterMethodEntry()*-Methode, der Code am Anfang des Methodenkörpers einer Zielmethode platziert und

zwei weitere Methoden *TraceZeroParameterMethodExit()* und *TraceMultipleParameterMethodExit()*, die am Ende eines Methodenkörpers Code platzieren. Speziell bei der *TraceMultipleParameterMethodExit()*-Methode wird über alle Methodenparameter der Zielmethode iteriert und in einem Array zwischengespeichert, der dann der *TraceMethodExit()*-Methode übergeben wird, um die Parameterwerte erfassen zu können. Die Entscheidung, der Trennung der Logik in drei Methoden, wurde deshalb getroffen, da die verwendeten Instruktionen und die Anordnung dieser, sich je nach Szenario grundlegend unterscheiden. Während die Instrumentierung am Anfang einer Methode einfacher ausfällt, stellt die Instrumentierung einer Methode am Ende des Methodenkörpers eine Herausforderung dar. Um welche Herausforderung es sich genau handelt, wird im Schritt „Umlenkung der Intruktionssprünge” geschildert.

Methodenzeit-Erfassung Die Erfassung der *absoluten Zeit* wurde ursprünglich direkt durch den TraceEmbedder vorgenommen, in dem die *Stopwatch*-Klasse aus der Standardbibliothek *System.Diagnostics* geladen und in die Zielmethode eingebettet wurde. Dabei besitzt Stopwatch die Funktionalität, die Zeit zwischen zwei Codebereichen zu messen. Das eigentliche Erfassen der Zeit durch den Stopwatch erfüllte seinen Zweck, jedoch forderten die instrumentierten Programme, die *System.Diagnostics*-Standardbibliothek in einer anderen Version als auf den Rechner installiert, um die Stopwatch-Klasse auflösen zu können. Da dies eine Wartung der Standardbibliothek mit einer bestimmten Version mit sich ziehen würde, wurde um das Problem zu lösen, die Stopwatch-Klasse direkt in den Methoden von dem EmbeddedTracer integriert. Näheres wird im Abschnitt 4.2 erklärt.

Umlenkung der Intruktionssprünge In der Instrumentierungsphase gab es einige Schwierigkeiten, da Anweisungen, Schleifen und *try-catch-finally*-Blöcke, Instruktionssprünge besitzen, die umgelenkt werden müssen. Diese Sprünge geben an, an welcher Stelle wieder angesetzt werden soll, wenn z. B. ein Fehler vorliegt, je nachdem in welchem Zweig einer Anweisung das Programm gelangt nach einer Bedingung oder wenn eine Schleife verlassen wird. All diese Sprünge müssen angepasst werden, wenn man versucht Code am Ende einer Methode einzubetten, sodass die erste Instruktion angesprungen wird von dem eingebetteten *try-finally*-Block. Beachtet man diesen Schritt nicht, wird der am Ende des Methodenkörpers instrumentierte Code in die einzelnen Blöcke hineingezogen oder aber es treten Fehler durch die Instrumentierung auf.

Speicherungsprozess Nachdem über alle Klassen iteriert und jede Zielmethode instrumentiert wurde, besteht der letzte Schritt darin, die Änderungen zwischenzuspeichern. Die *AssemblyDefinition*-Klasse besitzt dazu die *write()*-Methode, die genau diese Funktionalität bereitstellt. Neben den eigentlichen Zwischenspeichern, ist es zudem möglich einen neuen Assembly-Namen mitanzugeben, um eine klare Trennung zu schaffen zwischen der ursprünglichen EXE- bzw. DLL-Assembly und der modifizierten Fassung. Auf dieses Feature wurde zurückgegriffen, um die modifizierte Version der Assembly kenntlich zu machen, indem ein Infix „.original” in den originalen Assembly-Namen hinzugefügt wird.

Übertragung der Abhängigkeiten Damit die Zielanwendung die eingebettete Klasse EmbeddedTracer aus dem *TraceUtil.dll*-Assembly auflösen bzw. finden kann, muss dieser in den Zielordner der Anwendung platziert werden. Diese Aufgabe übernimmt die Methode *CopyOverNeededReferences()* aus der *FileManager*-Klasse. Im Laufe dieser Arbeit kam es jedoch zu Komplikationen, die das Auffinden des *TraceUtil.dll*-Assembly durch die Zielanwendung betrifft. Im Rahmen dieser Arbeit konnte dieses Problem, nach langer Suche nicht gelöst werden. Eine Vermutung, die aufgestellt wurde, besteht darin, dass einige der getesteten `C#`-Anwendungen, die bereitgestellte Abhängigkeit nicht auflösen können, da sie

bei der Suche der Abhängigkeiten nur die eigenen internen statisch festgelegten Assemblies beachten oder aber gegen Tools zur Erfassung von Laufzeitdaten abgesichert wurden.

Nachfolgend werden einige Ansätze erläutert, die unternommen wurden, um dieses Problem zu lösen, die jedoch fehlgeschlagen sind:

Erster Lösungsansatz Ein Lösungsansatz war es, das Tool *il-repack* [44] zu verwenden, um die Assembly der Zielanwendung mit der *TraceUtil.dll*-Assembly zu verschmelzen, sodass die Abhängigkeiten intern aufgelöst werden. Dieser Ansatz hat zwar bei einigen Anwendungen funktioniert, bei denen sich alle Abhängigkeiten direkt in den Verzeichnissen der Zielanwendung befanden. Bei vielen anderen funktionierte dies nicht. Eine Vermutung, die aufgestellt wurde ist, dass Abhängigkeiten in der Zielanwendung intern festgelegt wurden durch statische Pfade, sodass man beim Verschmelzen nicht alle Abhängigkeiten zusammenbekommt, damit die Zielanwendung ordnungsgemäß arbeiten kann.

Zweiter Lösungsansatz Aus dem Wissen, dass einige Anwendungen ihre Abhängigkeiten auch in den Umgebungsvariablen von dem Windows-Betriebssystem durchsuchen, war es ein Ansatz, den absoluten Pfad zu der *TraceUtil.dll*-Assembly, in die PATH-Variable zwischenzuspeichern. Keines der getesteten `C#`-Anwendungen konnte so die Assembly auflösen. Weder eine Ursache, noch eine Vermutung wurde aufgestellt, da dieses Verhalten nicht erklärt werden konnte.

Dritter Lösungsansatz Jeder Computer, auf dem die virtuelle Maschine Common-Language-Runtime (CLR) installiert ist, die unter anderem für die Ausführung von `C#`-Anwendungen zuständig ist, ist ein so genannter *Global-Assembly-Cache* (GAC) vorhanden. In diesem Cache werden Assemblies zwischengespeichert, die von Anwendungen geteilt werden können [14]. Aus diesem Grund wurde die *TraceUtil.dll*-Assembly in den Cache platziert. Dazu hat sich das Tool *Global-Assembly-Cache-tool (Gacutil.exe)* angeboten, mit dem man unter anderem Assemblies in den Cache hinein laden kann [13]. Dieser Ansatz hat wie bei den vorherigen Ansätzen bei einigen Anwendungen Wirkung gezeigt, bei vielen anderen besteht das beschriebene Problem weiterhin.

Bei den getesteten `C#`-Programmen handelte es sich um SEE, ILSpy und dnSpy [39] auf den im Abschnitt 4.5.2 noch näher eingegangen wird, sowie *Dimmer* [32], eine vom Umfang her kleinere grafische Benutzeroberfläche. Neben den bereits genannten Anwendungen wurden selbstentworfene Webanwendungen getestet. Nach dem Ausschlussverfahren wurde auf das Kopieren von der *TraceUtil.dll*-Assembly in das Verzeichnis der Zielanwendung zurückgegriffen, weil dieser Ansatz bei den meisten getesteten Anwendungen funktioniert hat.

4.2 EmbeddedTracer

Der Ablauf des EmbeddedTracers beginnt damit, dass die *GetInstance()*-Methode aufgerufen wird, um an die einzig existierende Instanz zu kommen. Anschließend werden über diese Instanz, sämtliche Methoden aufgerufen die für die Erfassung der Laufzeitdaten zuständig sind. Ein Beispiel wie der EmbeddedTracer benutzt werden kann, ist im Listing 4.2 zu sehen. Nachfolgend ist der wichtigste Funktionsumfang zur Erfassung der Laufzeitdaten, näher erläutert:

Start- und Endzeitpunkt Mit der optionalen *StartTracingFlag()*- bzw. *EndTracingFlag()*-Methode sowie den Methoden *TraceMethodEntryWithStartOrStartAndEnd()* und *TraceMethodExitWithStartOrStartAndEnd()* kann festgelegt werden, wann die Laufzeitdaten aufgezeichnet werden sollen bzw. wann die Aufzeichnung zum Halt kommt. Dazu wird ein

einfacher *Flag* übergeben und gesetzt, um die Laufzeitdatenerfassung zu unterbinden oder aber die Erfassung von Daten zu gestatten. Das Listing 4.5 und 4.7 zeigt einmal, an welcher Stelle der Zielmethode, die genannten Methoden platziert werden würden nach der Instrumentierung.

Aufrufgraph Nachdem die *TraceMethodEntry()*-Methode betreten wird, wird nach dem JLG-Dateiformat das Betreten der Zielmethode aufgezeichnet. Bevor der Aufruf der gleichen Methode zu Ende geht, wird durch den Aufruf der *TraceMethodExit()*-Methode, der Austritt festgehalten. Insgesamt entsteht so der finale Aufrufgraph.

Zeilennummer und absoluter Pfad Wie bereits im Abschnitt 4.1 erläutert, wird mit der *TraceLinenumberAndAbsoluteFilepath()*-Methode, die Metadaten für jede Zielmethode erfasst. Für die finale JLG-Datei wird zu diesem Zweck in einer Datenstruktur die Metadaten mit der jeweiligen Zielmethode in Beziehung gesetzt.

Methodenparameter und absolute Zeit Die Methodenparameter werden mit der *TraceMethodExit()*-Methode erfasst, der einen Array übergeben bekommt, mit den einzelnen Parameterwerten. Intern in der Methode werden anschließend die Werte, mit den dazugehörigen Parameternamen zugeordnet und als eine Zeichenkette zwischengespeichert. Zudem wird am Ende der *TraceMethodEntry()*-Methode, für die jeweilige Zielmethode eine Stopwatch gestartet, die am Anfang von *TraceMethodExit()* zum Halt kommt. Die Zeit, die gemessen wurde, wird zu den Laufzeitdaten, die für die Zielmethode erfasst wurden, mit aufgenommen.

Zwischenspeicherung Der letzte Schritt erfolgt, bevor die Anwendung vollständig geschlossen wird. Dazu wird die *SaveTrace()*-Methode im *Finalizer*-Konstruktor aufgerufen, um die bis dahin erfassten Laufzeitdaten, dem JLG-Dateiformat entsprechend zu einer Zeichenkette zusammenzufassen und zwischenzuspeichern. Der Ort der Zwischenspeicherung erfolgt im Hauptordner der ausgeführten Anwendung.

4.3 Erweiterung der SEE-Visualisierung

In diesem Abschnitt wird auf die Implementierung der Heatballoons, Profiler-Fenster und Bedienung näher eingegangen. Wie bereits erwähnt sind alle Implementierungsdetails eng verknüpft mit den Quelldateien von Kipka, da sie eine Aufgabe erfüllen und eine Anforderung die Kompatibilität beider Visualisierungen ist.

4.3.1 Heatballoons

Das Erzeugen der Heatballoons läuft über die *HeatballoonBuilder*-Klasse. Dabei wird bei der Erzeugung zwischen den Pflichtfeldern Name, Größe, Position und dem optionalen Feld Farbe unterschieden, wobei die Farbe mit der Hilfsklasse „ColorUtil" bereitgestellt wird. Bei dem zu erzeugenden Heatballoon handelt es sich um einen *Prefab*, also ein wiederverwendbares Objekt, das mit dem `C#`-Skript „HeatballoonManipulator" versehen ist. Das Skript sorgt dafür, dass wenn zwei Heatballoons sich überlagern, der größere über den kleineren positioniert wird, um die vollständige Verdeckung zu vermeiden. Nach Erzeugung des Heatballoons, wird dieser erst einmal in Relation zu dem zugehörigen 3D-Objekt bzw. Gebäude gesetzt, z. B. einer Klasse, dessen CPU-Zeit und Anzahl der Methodenaufrufe widergespiegelt werden soll. Die Relation wird in einer Datenstruktur zwischengespeichert. Zusätzlich werden diese

beiden Komponenten mit einem *LineRenderer*-Objekt verknüpft, um visuell die Zugehörigkeit darzustellen. Das LineRenderer-Objekt ist Teil von Unity selbst, womit man zwischen zwei Punkten eine Linie zeichnen lassen kann. Der beschriebene Prozess wird dann angestoßen, wenn ein Heatballoon angefordert wird. Dies ist immer dann der Fall, wenn bei den verarbeitenden Laufzeitdaten, in den Aufrufgraph, ein Methodeneintritt verzeichnet wurde.

Im Verlauf der Visualisierung bzw. Animation in der JLGCity wird zwischen zwei Aktualisierungsstellen unterschieden, welche den Zustand der Heatballoons betreffen:

Erste Aktualisierungsstelle In der von Unity angebotenen *Update()*-Methode aus der *SEEJlgCityAnimation*-Klasse wird per Frame, unabhängig von der Animation geprüft, ob der Benutzer in die Visualisierung hineingezoomt hat oder aber ob innerhalb der voreingestellten Rücksetzzeit ,,Heat reset time", der Zustand eines Heatballoons durch die verarbeiteten Laufzeitdaten unverändert geblieben ist. Im ersten Fall, wird in Relation zu den Gebäuden und den bereits verarbeiteten Laufzeitdaten, die Größe und Position der Heatballons angepasst. Es werden zusätzlich die verlinkten Positionen des LineRenderer-Objekts angeglichen. Für die Anpassung der Größe ist die *UpdateHeatBalloonSizes()*-Methode zuständig und die *UpdateBallonPosition()*-Methode übernimmt die Anpassung der Position. Im zweiten Fall wird, wenn lange nicht mehr auf einen Heatballoon zugegriffen wurde und die *Rücksetzzeit* erreicht ist, die Farbe in ihren Ursprung wieder zurückgesetzt, dies wird von der *ResetHeatBalloonsColor()*-Methode geregelt. Die Option, dass sich die Heatballoons abkühlen, kann wie in den Bedienungsmöglichkeiten vorgesehen, ausgeschaltet und die Rücksetzzeit variable gewählt werden.

Zweite Aktualisierungsstelle Die zweite Aktualisierungsstelle ist abhängig von den verarbeiteten Laufzeitdaten innerhalb von der *SEEJlgCityAnimation*-Klasse von Kipka. Wann immer in diesen Daten ein Methodeneintritt verzeichnet wurde im Aufrufgraph, wird von der *UpdateVisualization()*-Methode aus der *SEEJlgCityAnimation*-Klasse, die *UpdateHeatballoons()*-Methode aus der *HeatballoonManager*-Klasse aufgerufen. Bei dem Aufruf werden alle nötigen Parameterwerte mit übergeben, um einen Heatballon aktualisieren zu können anhand der aktuell verarbeitenden Laufzeitdaten. Ein solch wichtiger Parameterwert, ist einmal die *ParsedJLG*-Datenklasse, die im Abschnitt 3.4 bereits behandelt wurde. Aus diesem kommt man an die *JavaStatement*-Datenklasse, welche die absolute und eigene Zeit der Methode, bei dem ein Methodeneintritt vorliegt, enthält. Basierend auf den Voreinstellungen werden mit diesen Zeiten entweder die Farbe oder die Größe des korrelierenden Heatballoons upgedatet. Des Weiteren wird, nachdem sich die Größe geändert hat, auch die Position angepasst, sodass der Heatballoon immer weiter aufsteigt, desto größer er wird. Dies soll vor allem Überdeckung vermeiden. Neben der Anpassung der Position, wird das korrelierende LineRenderer-Objekt upgedatet, um die Linie zwischen den Heatballoons und den Gebäuden neu zu zeichnen. Zudem wird die Zugriffszeit für jeden Heatballoon zwischengespeichert, für die erste Aktualisierungsstelle, wenn es darum geht die Heatballoons abzukühlen. Die bisher genannten Möglichkeiten stellen nur einen Teil der Aktualisierung der Heatballoons dar. Je nachdem welche Einstellungsmöglichkeiten vorgenommen wurden für die Heatballoons, kann die Farbe bzw. Größe ebenfalls durch die Anzahl an Methodeneintritten bestimmt werden. Da die Beschreibungen ähnlich diskutiert werden würden, wird die Redundanz an dieser Stelle erspart.

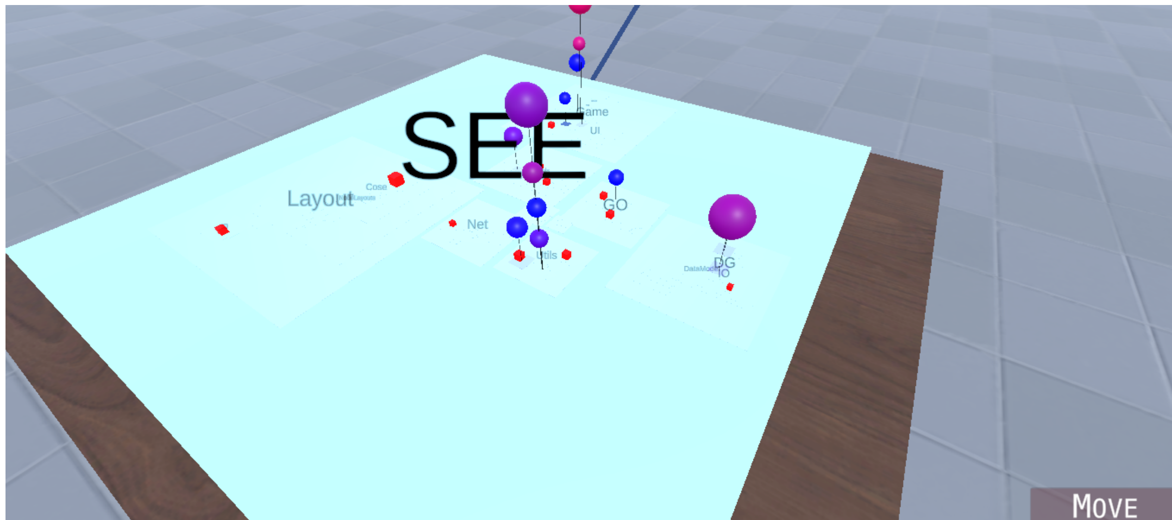


Abbildung 4.2: Einsatz von Heatballoons, bei dem Laufzeitdaten von dem SEE-Projekt in SEE bzw. JLGCity selbst dargestellt werden.

In der Abbildung 4.2 sieht man die implementierten Heatballoons im Einsatz, sowie die LineRenderer-Objekte. Der vorliegende Zustand, zeigt einmal die aufgezeichneten Laufzeitdaten von SEE, die durch den Einsatz von den EmbeddedTracer ermittelt werden konnten, die wiederum in der JLGCity innerhalb von SEE visualisiert werden.

4.3.2 Profiler-Fenster

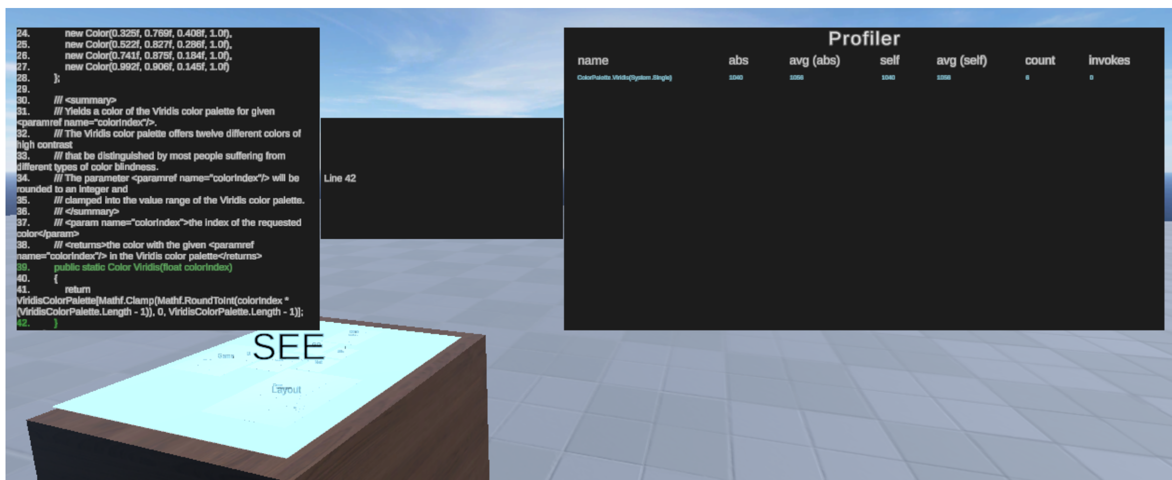


Abbildung 4.3: Präsentation der Laufzeitdaten im Profiler-Fenster.

Für die Umsetzung des Profiler-Fensters wurde das bereits vorhandene Canvas-Objekt von Kipka erweitert. Das Objekt besitzt zum Zeitpunkt dieser Arbeit bereits zwei Fenster-Objekte, die in Abschnitt 2.1.3.2 erläutert wurden. Für die Wiederverwendung wurde dieses Objekt als ein *Prefab* abgespeichert. Um die Laufzeitdaten darzustellen, wird neben den bereits vorhandenen Fenster-Objekten, ein weiteres Fenster hinzugefügt, wie im Entwurf vorgesehen. Mit Textelementen wurden die einzelnen Spalten hervorgehoben und ein *Scroll-Rect*-Element sorgt dafür, dass man über eine beliebig lange Liste von Datensätzen scrollen kann. In der

Abbildung 4.3 sieht man einmal links die bereits vorhandenen Fenster und rechts das Profiler-Fenster. Die Verarbeitung der Laufzeitdaten und die anschließende Darstellung im Profiler-Fenster übernimmt die *SEEJlgCityAnimation*-Klasse von Kipka, die im Rahmen dieser Arbeit zu diesem Zweck erweitert wurde. Nachfolgend wird der Prozess einmal näher erläutert.

Zur Laufzeit der Animation wird mit Hilfe der *UpdateProfilerInfos()*-Methode alle verarbeiteten Informationen aus der eingelesenen JLG-Datei in unterschiedlichen Datenstrukturen zwischengespeichert. Die Datenstrukturen werden in der *ActivateNodeTextWindow()*-Methode wiederverwendet, um die einzelnen Einträge im Profiler-Fenster darzustellen, wann immer man auf ein Gebäude klickt, das entweder ein Methodenobjekt oder Klassenobjekt darstellt. Handelt es sich um ein Methodenobjekt, werden alle Informationen, die bis zur Selektion erhoben wurden für die Methode und der inneren Methoden, im Profiler-Fenster eingeblendet. Bei einem Klassenobjekt werden alle enthaltenen Methoden der selektierten Klasse angezeigt. Optional besteht zudem die Möglichkeit, dass jeweilige Heatballoon-Objekt zu einem Klassenobjekt anzuklicken, um sich die Informationen aller Methoden dieser Klasse anzeigen zu lassen.

4.4 Bedienung

Für die Umsetzung der Bedienungsmöglichkeiten bzw. Voreinstellungen, wurde die Klasse *HeatballoonEditor* implementiert, die von der Klasse ,,Editor" aus dem *UnityEditor*-Paket erbt. Durch die Vererbung wird die *OnInspectorGUI()*-Methode aus der *Editor*-Klasse überschrieben, mit der man individuelle Layout-Elemente erstellen kann, die unterschiedliche Eingaben entgegennehmen. Die Liste an Eingabemöglichkeiten wurden bereits in der Tabelle 3.3 behandelt. Die erhobenen Eingaben werden anschließend der *HeatballoonManager*-Klasse übergeben, für die ordnungsgemäße Darstellung der Heatballoons zur Laufzeit der Animation.

4.5 Tests

In diesem Abschnitt werden alle Tests genannt die durchgeführt wurden. Dazu werden sowohl automatisierte, als auch händisch durchgeführte Tests behandelt und diskutiert sowie auf Komplikationen eingegangen.

4.5.1 Automatisierte Tests

Neben der eigentlichen Implementierung wurden unter anderem Unit- und Integrationstests geschrieben, mit dem Ziel, das Zusammenspiel von Klassen bzw. einzelne Klassen isoliert auf mögliche Eingaben bzw. Abläufe zu überprüfen. Bei den Tests selbst handelt es sich vorwiegend um White-Box-Tests. Um genauer zu sein um Pfadtests, die bei den getesteten Methoden eine hundertprozentige Anweisungsüberdeckung aufweisen. Das heißt, jede Anweisung wurde mit Kenntnis der Implementierung mindestens einmal durchlaufen, bei jeder getesteten Methode. In all den verfassten Tests, wurde nach dem Arrange-Act-Assert Muster vorgegangen. Bei diesem Muster wird im *Arrange* alles vorbereitet für das, was man testen möchte, in *Act* wird der eigentliche Teil ausgeführt, der bei dem Test von Interesse ist und in *Assert* vergleicht man das Erwartete mit dem Ergebnis aus *Act*. Mit diesem Vorgehen wurden die Klassen *HeatballoonBuilder*, *HeatballoonManager* und *ColorUtil*, die von Interesse sind und direkt mit den Heatballoons zusammenhängen, getestet. Methoden, die als trivial angesehen werden können, wie z. B. Getter- bzw. Setter-Methoden, wurden nicht behandelt.

Nachfolgend werden einige Tests vorgestellt, eine ausführliche Beschreibung und Behandlung aller erstellten Tests ist nicht Hauptgegenstand dieser Arbeit:

Erster Test Die *HeatballoonBuilder*-Klasse die für die Zusammensetzung eines Heatballoons zuständig ist, wurde drauf getestet, ob die übergebenen Parameter richtig übernommen und zur Erzeugung des Heatballoons genutzt wurden. Um dies einzuleiten wurden in *Arrange*, alle nötigen Attribute die der Heatballoon besitzen soll angelegt und in *Act* der Prozess zur Erzeugung des Heatballoons zuständig ist eingeleitet. In *Assert* wurden die angelegten Attribute mit den Attributen des Heatballoons verglichen. Das Resultat zeigte einen negativen Test.

Zweiter Test Die *HeatballoonManager*-Klasse wurde drauf getestet, ob ein Heatballoon für eine Klasse „X" erstellt wird, wenn in den Laufzeitdaten ein Aufruf einer Methode der Klasse „X", aufgezeichnet wurde. „X" steht in diesem Fall für eine variable Klasse und soll nicht als eine bestimmte Klasse verstanden werden. Auch dieser Test verlief negativ.

Dritter Test Einige weitere Tests, die für den *HeatballoonManager* gedacht sind, überprüfen, ob die Heatballoons im Zusammenhang mit der Animation auch richtig upgedatet werden. Dazu wurde überprüft, ob sich die Größe eines Heatballoons ändert, wenn ein weiterer Aufruf mit der korrelierenden Methode verzeichnet wurde. Diese Tests verliefen ebenso negativ.

Die genannten Tests basieren auf dem NUnit-Framework [6] bzw. dem Unity-Test-Framework [43]. Es sei zudem an dieser Stelle ausdrücklich zu erwähnen, dass durch die Tests nicht gewährleistet ist, dass Fehler auszuschließen sind oder die Korrektheit der Software bewiesen ist.

4.5.2 Tests zur Instrumentierung

Automatisierte Tests kommen für die Instrumentierung nicht infrage, da der Aufwand der Umsetzung zu komplex und aufwendig wäre. Aus diesem Grund wurden größere Programme genommen, instrumentiert und manuell auf mögliche Anwendungsfälle überprüft. Bei den Programmen handelt es sich um SEE einer Unity-Anwendung und zwei grafische Benutzeroberflächen ILSpy und dnSpy. Die ersten beiden Programme wurden bereits erläutert, bei dnSpy handelt es sich um einen Debugger, der zusätzliche Funktionalität besitzt, Assemblies zu editieren [39].

Die Überprüfung, ob die Instrumentierung auch korrekt verlaufen ist, wird durch ILSpy vorgenommen, indem die instrumentierte Assembly dekompiliert und der eingebettete Code betrachtet wird. Nachfolgend werden einige der manuell getätigten Testfälle einmal für SEE aufgelistet und diskutiert. Die Testfälle für den *ILSpy* und *dnSpy* sehen absolut identisch aus, deshalb wird die weitere Redundanz erspart.

Test 1: Startpunkt bei der Aufzeichnung

Bei diesem Test geht es darum, dass die Aufzeichnung beschränkt werden kann durch die Angabe eines Startpunkts. In diesem Fall würde nur die Aufzeichnung betrachtet werden können ab einer bestimmten Methode. Nachfolgend wurden die einzelnen Schritte inszeniert, um den Test durchzuführen:

Erster Schritt Aus dem Zielprogramm wurde ein Pfad zu einer Methode ausgesucht, ab dem die Aufzeichnung starten soll. Der Pfad wird optional aus einem Paketnamen oder

Anordnung von Paketen und den Klassennamen sowie den Methodennamen gebildet. Da bereits bei allen zu betrachtenden Programme der gesamte Quellcode vorliegt, wurde sich dessen bedient. Ansonsten ist es auch möglich mit ILSpy die zu testende Anwendung zu dekompilieren, um einen Pfad zu einer Methode herauszufinden.

Zweiter Schritt Nach der Instrumentierung mit der Angabe des Startpunkts, wurde die Zielanwendung dekompiliert mit ILSpy und nachgeschaut, ob der Punkt tatsächlich gesetzt wurde. Dies macht sich dadurch erkenntlich, dass die *StartTracing()*-Methode aus der *EmbeddedTracer*-Klasse am Anfang der Zielmethode aufgeführt ist. Das Listing 4.5 zeigt dazu den instrumentierten Code, bei dem die *HierarchicalEdgeTypes*-Methode als Startpunkt dient.

Dritter Schritt Als nächstes wird nach Ausführung der Zielanwendung geschaut, ob in der JLG-Datei tatsächlich ab dem gesetzten Startpunkt angefangen wurde aufzuzeichnen. Listing 4.6 zeigt einen Ausschnitt der erzeugten JLG-Datei, in dem der Startpunkt richtig gesetzt wurde. Alle händisch durchgeführten Tests, um den Startpunkt zu setzen, verliefen negativ.

```
public static HashSet<string> HierarchicalEdgeTypes()
{
	EmbeddedTracer.GetInstance().StartTracingFlag(flag: true);
	EmbeddedTracer.GetInstance().TraceLinenumberAndAbsoluteFilepath("144", "154",
	    ↪ "C:\\[...]\\AbstractSEECity.cs");
	EmbeddedTracer.GetInstance().TraceMethodEntryWithStartOrStartAndEnd();
	try
	{
		return new HashSet<string> { "Enclosing", "Belongs_To", "Part_Of", "Defined_In" };
	}
	finally
	{
		EmbeddedTracer.GetInstance().TraceMethodExitWithStartOrStartAndEnd(null);
	}
}
```

Listing 4.5: Eine Methode die instrumentiert wurde, um die Laufzeitdaten zu erfassen.

```
$[C:\Pfad zur Datei [...]\AbstractSEECity.cs, ...]
-/0>144
§=1,6248
/-0>154
[...]
*-0=AbstractSEECity.HierarchicalEdgeTypes();[...]
```

Listing 4.6: Auszug aus der erzeugten JLG-Datei, dass zeigt das der Startpunkt richtig gesetzt wurde.

Test 2: Endpunkt bei der Aufzeichnung

Mit diesem Test soll geprüft werden, ob die Aufzeichnung gestoppt werden kann, durch Angabe eines Endpunkts. Nachfolgend wurden die einzelnen Schritte durchgeführt, um den Test durchzuführen:

Erster Schritt Wie bei dem ersten Test wurde aus dem Zielprogramm ein Pfad zu einer Methode ausgesucht, ab dem die Aufzeichnung gestoppt werden soll.

Zweiter Schritt Anschließend wurde nach der Instrumentierung, mit der Angabe des Endpunkts, die Zielanwendung dekompiliert mit ILSpy und nachgeschaut, ob der Punkt tatsächlich gesetzt wurde. Dies macht sich dadurch erkennbar, dass die *StartTracing()*-Methode aus der

EmbeddedTracer-Klasse am Ende der Zielmethode mit in dem *finally*-Block aufgeführt wird. Das Listing 4.7 zeigt dazu den instrumentierten Code, bei dem die *HierarchicalEdgeTypes()*-Methode als Endpunkt dient.

Dritter Schritt Als nächstes wurde nach Ausführung der Zielanwendung geschaut, ob in der JLG-Datei bei der Angabe des Endpunkts die Aufzeichnung gestoppt wurde. Listing 4.8 zeigt einen Ausschnitt der erzeugten JLG-Datei, in dem der Endpunkt richtig gesetzt wurde. Alle händisch durchgeführten Tests, um den Endpunkt zu setzen, verliefen negativ.

```
public static HashSet<string> HierarchicalEdgeTypes()
{
	EmbeddedTracer.GetInstance().TraceLinenumberAndAbsoluteFilepath("144", "154",
	    ↪ "C:\\[...]\\AbstractSEECity.cs");
	EmbeddedTracer.GetInstance().TraceMethodEntry();
	try
	{
		return new HashSet<string> { "Enclosing", "Belongs_To", "Part_Of", "Defined_In" };
	}
	finally
	{
		EmbeddedTracer.GetInstance().TraceMethodExit(null);
		EmbeddedTracer.GetInstance().EndTracingFlag(flag: false);
	}
}
```

Listing 4.7: Eine Methode die instrumentiert wurde, um die Laufzeitdaten zu erfassen.

```
$[C:Pfad zur Datei [...]\ColorPalette.cs, C:Pfad zur Datei \[...\]\AbstractSEECity.cs]
[...]
-/0>39
§=0,002
colorIndex=0
/-0>42
-/1>144
§=0,9015
/-1>154
*-0=ColorPalette.Viridis(System.Single);-1=AbstractSEECity.HierarchicalEdgeTypes();
```

Listing 4.8: Auszug aus der erzeugten JLG-Datei, dass zeigt das ab dem eingefügten Endpunkt, nicht weiter aufgezeichnet wurde.

4.5.3 Komplikationen

Nachfolgend sind unterschiedliche Komplikationen aufgeführt, die implizit oder explizit mit den durchgeführten Tests zusammenhängen:

Instruktionsfehler Viele der manuell durchgeführten Testfälle sind negativ ausgefallen. Es gibt jedoch einige Fälle, in denen die Instrumentierung teilweise nicht gelungen ist und auch nicht gelöst werden konnte. Dies hängt damit zusammen, dass die manuell hinzugefügten Instruktionen fehlerhaft platziert, redundant hinzugefügt oder aber grundlegend falsch sind im jeweiligen Kontext der instrumentierten Methoden. Überprüft wurde dies mit der Hilfe von ILSpy bei dem Fehler in den Instruktionen durch einen Kommentar hervorgehoben werden. Ein solcher Fehler ist in dem Listing 4.9 zu sehen. Weitere Fehler sind nicht auszuschließen und es können andere auftreten, je nachdem wie die einzelnen Anweisungen in `C#` angeordnet sind. Diese Art der Fehler führen dazu, dass sobald die Stelle zur Laufzeit ausgeführt wird, die Anwendung sich schließt, mit der Fehlermeldung, dass etwas mit den Instruktionen nicht stimme. Eine allgemeine Lösung, ohne eine Kettenreaktion von Fehlern auszulösen, wurde im Rahmen dieser Arbeit nicht gefunden.

```
public override HashSet<string> GetChangedObjects()
{
	//IL_0055: Incompatible stack heights: 1 vs 0
	EmbeddedTracer.GetInstance().TraceLinenumberAndAbsoluteFilepath("187", "187",
	    ↪ "C:\\[...]\\EditNodeAction.cs");
	EmbeddedTracer.GetInstance().TraceMethodEntry();
	try
	{
		if (memento.node != null)
		{
			return new HashSet<string> { memento.node.ID };
		}
		return new HashSet<string>();
	}
	finally
	{
		EmbeddedTracer.GetInstance().TraceMethodExit(null);
	}
}
```

Listing 4.9: Eine Methode die instrumentiert wurde, um die Laufzeitdaten zu erfassen.

Auflösungsprobleme Des Weiteren konnten zwar einige Anwendungen instrumentiert werden, jedoch stellte die Ausführung der Anwendungen im Anschluss ein Problem dar. Wie bereits im Abschnitt 4.1 erläutert, gibt es einige Programme, die *TraceUtil.dll*-Assembly nicht auflösen können und dazu gehört dnSpy sowie die selbstentworfenen Webanwendungen. Damit ist die Anforderung sämtliche C#-Programme aufzuzeichnen, im Rahmen dieser Arbeit nicht gelungen.

Last- und Peformancetests Zudem sei an dieser Stelle zu erwähnen, dass keine geeignete Anwendung gefunden wurde, um Last- und Performancetests automatisiert durchzuführen. Aus diesem Grund wurden nur manuelle Tests die aus der reinen Beobachtung hervorgehen durchgeführt. Wie bereits im Abschnitt 3.1.1 erläutert, entsteht durch die Instrumentierung ein Overhead und kann nur minimiert werden, indem die Aufzeichnung begrenzt wird. Die Beobachtung von der instrumentierten SEE-Instanz, hat gezeigt, dass durch die Aufzeichnung, der Ladeprozess von der Instanz ungemein erschwert wurde. Dies liegt unter anderem daran, dass bei Unity-Anwendung so genannte „Start"- bzw. „Awake"-Methoden aufgerufen werden können, wenn Skript-Instanzen geladen werden. Die zusätzliche Last, die durch die Aufzeichnung entsteht, sorgt letztendlich dafür, dass sich der Ladeprozess der SEE-Instanz um einige Sekunden hinauszögert. Bei normalen C#-Anwendungen [[29], [32]] sind es wiederum wenige Sekunden, da das Verhalten zum größten Teil durch Events getriggert wird und nicht wie bei Unity-Anwendungen pro Frame durch die internen Unity-Methoden wie „Update".

KAPITEL 5

Benutzerstudie

In diesem Kapitel wird auf die Benutzerstudie eingegangen, die aufgrund der aktuellen *COVID-19*-Pandemie online durchgeführt wurde. Dabei wird der SEE-Profiler in der Unity-Version „2020.3.16f1", im Vergleich zu den State-of-the-Art, dem Unity-Profiler gesetzt, wobei nur der selbe Funktionsumfang betrachtet wird. Zudem wird auf die verwendeten Fragebögen, Aufgabenstellungen sowie Aufbau und die Durchführung der Benutzerstudie näher eingegangen. Abgeschlossen wird das Kapitel durch die Ergebnisse und Bewertung der Benutzerstudie, sowie die Beantwortung der Forschungsfragen, die mit der Studie einhergehen.

5.1 Forschungsfragen

Die nachfolgenden Forschungsfragen, sollen mit der Benutzerstudie beantwortet werden.

Erste Forschungsfrage Ist der SEE-Profiler mindestens so effektiv, beim Auffinden von Performanz-Informationen, im Vergleich zum klassischen Ansatz, dem Unity-Profiler?

Zweite Forschungsfrage Ist der SEE-Profiler mindestens so effizient, bei gegebener Aufgabenstellung, die möglichen Lösungen zu ermitteln, im Vergleich zum klassischen Ansatz, dem Unity-Profiler?

Dritte Forschungsfrage Ist der SEE-Profiler gebrauchstauglich?

Die ersten zwei Forschungsfragen, behandeln die Usability-Aspekte, Effektivität und Effizienz des SEE-Profilers. Bei der Effektivität geht es darum, ob die Anwender, hier die Probanden in der Lage sind, die gestellten Aufgaben erfolgreich zu lösen. Dazu wurden drei Aufgabenstellungen entworfen, jeweils für den SEE-Profiler und den Unity-Profiler, die von der Aufgabenstellung her absolut identisch sind, um einen angemessenen Vergleich sicherzustellen. Anders als die Effektivität, versucht man bei der Effizienz zu ermitteln, wie schnell die gestellten Aufgaben gelöst werden. Zu diesem Zweck wurde die Bearbeitungszeit von beiden Profilern, für die Bearbeitung aller Aufgaben zum Vergleich gesetzt. Zur Beantwortung der letzten Forschungsfrage, wird der *System-Usability-Scale* (SUS) [4] verwendet, um die Gebrauchstauglichkeit zu bestimmen. Auf den SUS, wird im Abschnitt 5.2.3 noch näher eingegangen.

5.2 Aufbau

In diesem Abschnitt soll der Aufbau der Benutzerstudie erläutert werden, der unter anderem angepasst wurde, nach einem Durchlauf und Feedback, eines Testpiloten. Daraus resultierend

wurden die betrachteten Laufzeitdaten, um die Aufgaben zu lösen, minimiert. Der Grund dafür ist, dass die Bearbeitungszeit in einem angemessenen Rahmen gehalten werden soll, damit die Probanden fokussiert an der eigentlichen Problematik arbeiten können, ohne von der Masse der Laufzeitdaten überfordert zu sein.

5.2.1 Verwendete Software und Hardware

Durchweg durch die gesamte Studie wurde ein Rechner verwendet, der mit dem Betriebssystem *Windows-10-Home* versehen ist. Der Rechner besitzt ein *Intel-Core-i5-8250U*-Prozessor und 8 GB Arbeitsspeicher. Bei der verwendeten Software handelt es sich um die Kommunikationssoftware *Discord* [18]. *Discord* erlaubt es private Kanäle zu erstellen, um sich mit mehreren Personen auszutauschen. Dieses Feature wurde für die Studie verwendet, um die Probanden einzeln einzuladen und während der Studie kommunizieren zu können. Alle Links zu den Einführungsvideos und Fragebogen wurden in dem erstellten Kanal den Probanden bereitgestellt. Für die Durchführung der Aufgaben wurde ein Fernzugriff auf den vorher genannten Rechner ermöglicht, mit der bekannten Software *TeamViewer* [1]. Zuletzt wurde die Webanwendung *KoBoToolbox* [19] verwendet, um die Studie mit den Fragebögen und Aufgabenstellungen durchzuführen. Es sei an dieser Stelle zu erwähnen, dass mögliche Latenz-Probleme durch den Einsatz von TeamViewer entstehen können, was sich negativ auf den zu prüfenden Faktor Effizienz auswirken kann.

5.2.2 Aufgabenstellungen

Um einen fairen Vergleich zwischen den SEE-Profiler und den Unity-Profiler zu gewährleisten, wurden identische Aufgabenstellungen gestellt. Die betrachteten Laufzeitdaten für den SEE-Profiler entstammen der erfassten Daten, einer instrumentierten SEE-Instanz. Während es sich bei den Laufzeitdaten für den Unity-Profiler eine Aufzeichnung des Abspielens von SEE, in Unity handelt. Nachfolgend werden die Aufgaben generisch wiedergegeben, in der Studie selbst beziehen sie sich explizit auf den betrachteten Profiler. Es sei zudem erwähnt, dass die Zeit für die Bearbeitung festgehalten wurde, um die Effizienz zu messen.

Erste Aufgabe Bei dieser Aufgabe geht es darum, fünf Methoden zu ermitteln aus dem SEE-Projekt, die sehr viel CPU-Zeit benötigen. Dazu wird nur die absolute Zeit einer Methode betrachtet. Die ermittelten Methoden sollen anschließend absteigend nach ihrer absoluten Zeit in eine Tabelle eingetragen werden.

Zweite Aufgabe Bei dieser Aufgabe geht es darum, so schnell wie möglich drei bestimmte Methoden aus dem SEE-Projekt zu finden und dessen absolute Zeit, eigene Zeit oder die Anzahl der Methodenaufrufe anzugeben.

Dritte Aufgabe Bei dieser Aufgabe geht es darum, fünf Methoden zu ermitteln aus dem SEE-Projekt, die am meisten aufgerufen wurden. Die ermittelten Methoden sollen anschließend, absteigend nach ihrer Aufrufanzahl in eine Tabelle eingetragen werden.

Bei der ersten Aufgabe ist zu erwähnen, dass die Betrachtung der eigenen Zeit einer Methode keinen Einfluss auf die Studie hat und einfachheitshalber nur die absolute Zeit betrachtet wird. Zu beiden Profilern gibt es zudem ein Einführungs-Video und eine kleine Aufgabe, in der sich die Probanden erst einmal an das jeweilige Tool gewöhnen können. Zwischen den einzelnen Aufgaben, wurden Warteräume platziert, bei dem der Proband Fragen stellen kann und darauf hingewiesen wird, die einzelnen Aufgaben sorgfältig durchzulesen und auf die Hinweise

zu achten. Um mögliche Lerneffekte bei der Reihenfolge der Bearbeitung der Aufgaben zu vermeiden, wurde zwischen Gruppe „A” und „B” unterschieden. Die Gruppe „A” fängt dabei mit dem Unity-Profiler an und arbeitet sich in Richtung SEE-Profiler, während die Gruppe „B” genau anders herum verfährt. Die Zuteilung zu den einzelnen Gruppen ist abwechselnd geschehen.

5.2.3 Fragebögen

Die Benutzerstudie startet damit, dass ein simpler Fragebogen auszufüllen ist, der den Erfahrungsstand des Probanden ermitteln soll. Bei den Fragen handelt es sich unter anderem darum, wie viele Jahre die Probanden schon programmieren, ob sie Erfahrungen mit Profilern haben oder Kenntnisse über SEE. Alle gestellten Fragen in der Einführung werden in Detail im Abschnitt 5.4.1 behandelt. Neben den Einführungsfragebogen, folgt nach Bearbeitung aller Aufgaben zum jeweiligen Profiler, der System-Usability-Scale [4]. Bei SUS handelt es sich um einen Fragebogen, bei dem zehn Fragen gestellt werden, die in einer Skala von 1 bis 5 beantwortet werden. Anschließend lässt sich ein Score errechnen, der zwischen 0 und 100 liegt, der angibt, ob die vorliegende Anwendung gebrauchstauglich ist. Nachfolgend wird die Formel eingeblendet, die man zum Berechnen des Scores verwenden kann:

$$
\begin{aligned}
X &= Summe(Ungerade\ nummerierte\ Fragen) - 5 \\
Y &= 25 - Summe(Gerade\ nummerierte\ Fragen) \\
Score &= Summe(X, Y) * 2,5
\end{aligned}
$$

Dabei gilt jede Anwendung, die einen Score von über 68 Punkten erreicht, als gebrauchstauglich. Vorteil dieses Fragebogens ist, dass dieser einfach durchzuführen und ein gängiges Mittel ist, um die Usability einer Anwendung zu ermitteln. Nachfolgend werden die gestellten Fragen aufgelistet, in diesem Fall generisch, wobei anstelle von „System”, einfach nur SEE-Profiler oder Unity-Profiler ersetzt werden kann.

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

Neben den Einführungsfragebogen und dem System-Usability-Scale, gibt es am Ende noch eine offene Fragestellung für den SEE-Profiler, bei dem die Probanden angeben können, was Ihnen an dem Profiler gefallen hat und was nicht. Damit sollen qualitative Daten erhoben werden, um das Verbesserungspotential des Profilers auszuschöpfen.

5.3 Durchführung

Insgesamt zehn Probanden haben an der Benutzerstudie teilgenommen, die allesamt einen Hintergrund in der Informatik haben. Von den zehn Probanden befinden sich acht davon mindestens im fünften Semester. Ein Proband ist momentan im ersten Semester, bringt jedoch Erfahrung in der Programmierung mit sich und ein weiterer der seinen Master abgeschlossen und eine Festanstellung hat. Nachfolgend sind die einzelnen Schritte aufgelistet, zur Durchführung der Benutzerstudie:

1. Der SEE- und Unity-Profiler werden erst einmal aufgesetzt und auf den initialen Zustand gebracht. Alle weiteren Hintergrundprozesse werden geschlossen, um weitere Last zu vermeiden.

2. Der Proband wird in den für die Benutzerstudie erstellten Discord-Kanal eingeladen und über die einzelnen Links, die zu den Einführungsvideos und Fragebogen führen, aufgeklärt. Nach Einteilung des Probanden in eine Gruppe, wird der Ablauf der Studie im groben erläutert und darauf hingewiesen, dass er zu jeder Zeit Fragen stellen kann und auf die einzelnen Hinweise achten soll, um die Aufgaben lösen zu können. Die Zuteilung des Probanden zu einer Gruppe wird abwechselnd je Proband vorgenommen. Dementsprechend sind die zu benutzenden Links auch beschriftet, sodass der Proband den richtigen Fragebogen ausfüllt. Zudem wird der Proband darauf aufmerksam gemacht, dass die Bearbeitungszeit aller Aufgaben gestoppt wird. Zuletzt wird eine Zustimmung eingeholt, um die erhobenen Daten weiterzuverarbeiten und zu veröffentlichen.

3. Die Bearbeitung der Fragebögen, geschieht auf dem Rechner des Probanden selbst, sodass kein Einfluss auf das Verhalten des Probanden bei der Beantwortung der Fragen und Aufgaben genommen wird. Dabei startet der Proband erst einmal mit dem Einführungsfragebogen und nach Bearbeitung im Warteraum, erfolgt der Fernzugriff durch TeamViewer auf den Rechner, auf den die Profiler zur Bearbeitung der Aufgaben sich befinden.

4. Je nachdem in welche Gruppe der Proband zugeteilt wurde, fängt er mit der Bearbeitung der Aufgaben mit dem SEE- bzw. Unity-Profiler an. Zuerst folgt eine Einführung in dem jeweiligen Profiler, durch ein Einführungsvideo. Darin werden einige Grundlagen erklärt, die für die Bearbeitung der Aufgaben nötig sind. Anschließend soll der Proband eine leichte Aufgabe lösen, um sich an die Steuerung und den Profiler zu gewöhnen.

5. Nachdem die Einführung vorüber ist, folgen wie im Abschnitt 5.2.2 erklärt, drei Aufgaben. Zwischen den Aufgaben gibt es Warteräume, bis die nächste Aufgabe angetreten wird.

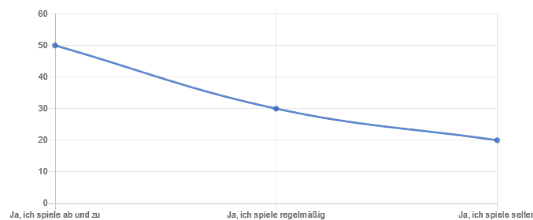
6. Nach Bearbeitung der Aufgaben, folgt direkt der SUS der im Abschnitt 5.2.3 besprochen wurde. Optional hat man bei dem SEE-Profiler die Möglichkeit nach dem SUS, eine subjektive Meinung abzugeben, wie er die Darstellung und Umsetzung empfunden hat.

7. Nachdem die vorherigen Schritte durchgeführt wurden, befindet sich der Proband am Ende der Studie und ein Dank sowie eine kurze Diskussion schließt die Benutzerstudie ab.

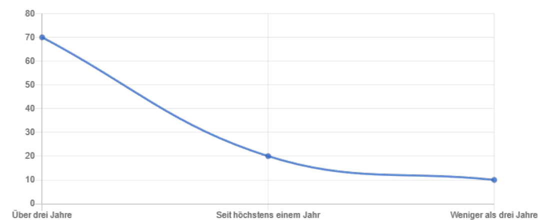
5.4 Auswertung

In den nachfolgenden Unterabschnitten wird die gesamte Auswertung im Detail betrachtet, besprochen und bewertet.

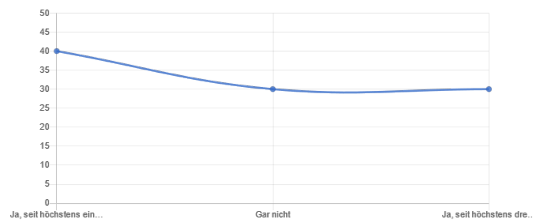
5.4.1 Einführungsfragebogen



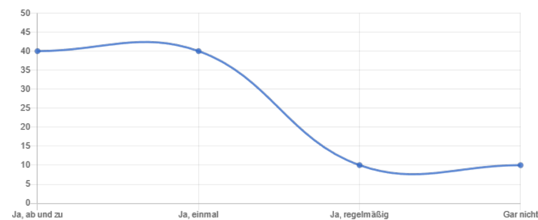
(a) Hast du Erfahrung mit 3D-Videospielen auf dem Desktop-PC?



(b) Wie viele Jahre programmierst du schon?

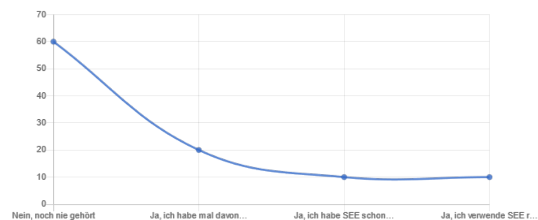


(c) Hast du Erfahrung in der Programmiersprache C#?

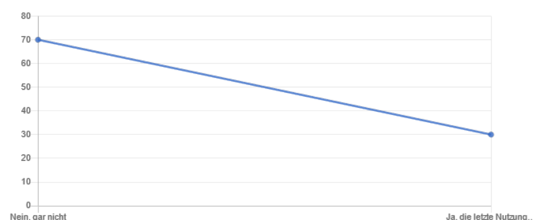


(d) Hast du schon in Softwareprojekten mitgearbeitet?

(e) Kennst du dich in dem Bereich der Softwarevisualisierung aus?



(f) Hast du Erfahrung mit SEE (Software Engineering Experience)?



(g) Kennst du dich mit Profiler aus?

(h) Hast du Erfahrung mit dem Unity-Profiler?

Abbildung 5.1: Auswertung der Ergebnisse des Einführungsfragebogens.

Aus der Auswertung des Einführungsfragebogens, kommen unterschiedliche Ergebnisse zusammen, die aus der Abbildung 5.1 zu entnehmen sind. Dabei stellt sich heraus, dass jeder Teilnehmer mindestens einmal Erfahrung mit 3D-Videospielen gemacht hat und dass damit impliziert wird, dass mit der Visualisierung von dem SEE-Profiler im dreidimensionalen

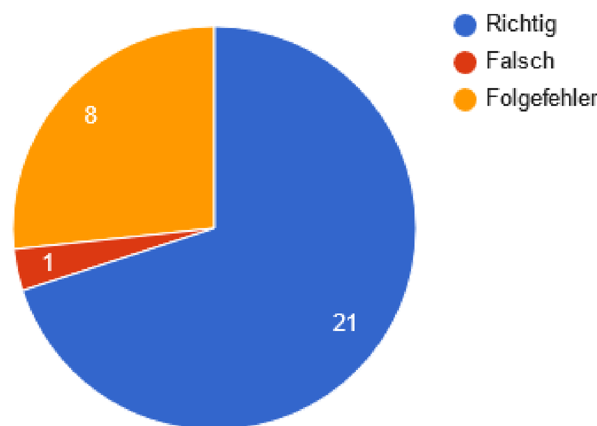
Raum der Teilnehmer nicht allzu sehr überfordert war. Mit der Frage ,,b" konnte der Kenntnisstand der Probanden über die Programmierung abgefragt werden. Dabei programmieren sieben Teilnehmer, schon weit über drei Jahre, während die restlichen drei Probanden, unter drei Jahren programmieren. Für die Teilnahme selbst, reichen Kenntnisse über Pakete, Klassen und Methoden völlig aus. Diese Anforderung wird somit von jeden Teilnehmer erfüllt und eine isolierte Betrachtung wird somit ausgeschlossen. Eine interessante Fragestellung stellt die Frage ,,c" dar. Da sämtlicher Code, der betrachtet wird, in `C#` vorliegt. Dabei gaben sieben Probanden an mindestens die Programmiersprache zu kennen, die restlichen drei Probanden hingegen haben keine Erfahrung in dieser Programmiersprache. Bei Frage ,,d", ob die Probanden schon einmal an einem Softwareprojekt teilgenommen haben, gaben neun an, dass sie mindestens an einem Projekt teilgenommen haben, nur ein Teilnehmer hat an keinem Projekt teilgenommen. Die nachfolgenden Fragen sind etwas spezifischer und sollen den Stand der Probanden über ihr Wissen über Softwarevisualisierung abfragen. Etwas mehr als die Hälfte haben entweder von der Disziplin gehört bzw. sich in die Thematik eingelesen. Nur zwei der Probanden haben keine Kenntnisse über die Disziplin. Die Frage ,,f", sollte ermitteln, ob die Probanden mit SEE vertraut sind. Dabei gaben sechs Probanden an, von SEE noch nie gehört zu haben, die restlichen vier hingegen haben mindestens von SEE gehört und zwei Probanden haben sogar schon einmal mit SEE gearbeitet. Diese zwei Probanden werden ebenfalls nicht isoliert betrachtet, da sie im Durchschnitt die Aufgabenstellungen und Fragebögen wie die restlichen Teilnehmer beantwortet haben sowie ähnliche Bearbeitungszeiten aufweisen. Die restlichen zwei Fragen zielen auf das Wissen der Probanden über Profiler ab. Dabei gaben sieben Probanden an, sich mit Profilern nicht auszukennen, während der Rest sich zwar mit Profilern auskennt, die letzte Nutzung eines solchen Tools jedoch etwas lange her ist. Nicht überraschend, wurde die Frage, ob die Probanden Erfahrungen mit dem Unity-Profiler haben, ähnlich beantwortet. Mit der Ausnahme, dass niemand den Unity-Profiler nutzt, sondern diejenigen, die sich mit Profilern auskennen, schon einmal von dem Tool gehört haben.

5.4.2 Aufgabenstellungen

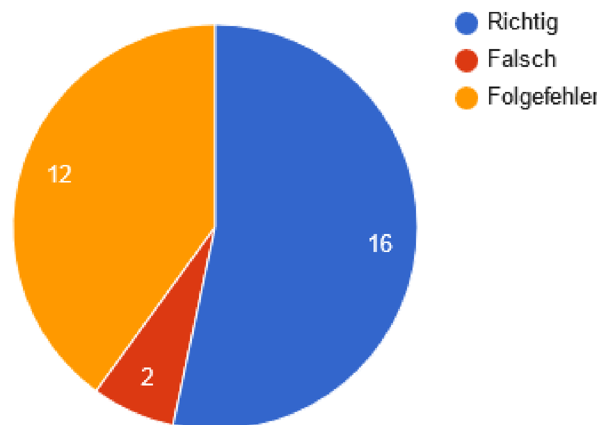
Proband	Gruppe	U-Aufgaben	S-Aufgaben	U-Zeit (m)	S-Zeit (m)
P01	A	O R O	O R R	9,26	9,01
P02	A	R R O	O R R	8,45	7,50
P03	A	R R R	O R R	9,43	7,56
P04	A	R R F	O R R	9,58	9,12
P05	A	O R O	O R R	9,01	7,02
P06	B	R R O	O R R	9,10	7,25
P07	B	R R O	O R R	8,17	8,32
P08	B	O R F	F R R	7,27	8,42
P09	B	O R O	O R R	8,12	6,55
P10	B	O R O	R R R	7,28	10,39

Tabelle 5.2: Alle Aufgaben ausgewertet, für alle zehn Probanden.

Aus der Tabelle 5.2 kann man für jeden Probanden entnehmen, wie viele Aufgaben richtig (R), falsch (F) oder teilweise richtig, durch Folgefehler (O) bearbeitet wurden. Das Präfix „S" bei den Spaltenbezeichner, steht für den SEE-Profiler und „U" für den Unity-Profiler. Neben den Antworten für die Aufgaben steht zu jeden Proband die insgesamt für alle Aufgaben benötigte Zeit. Die Angabe der Zeit wird deutlichkeitshalber in Minuten angegeben. Errechnet man die durchschnittliche Zeit für den SEE-Profiler, liegt die Zeit abgerundet auf zwei Nachkommastellen bei 8,11 Minuten, während die benötigte durchschnittliche Zeit des Unity-Profilers bei 8,56 Minuten liegt. Festzuhalten ist jedoch, dass es bei allen Teilnehmern zu Verzögerungen durch schlechtes Internet und des Umwegs durch den Fernzugriff kam. Damit spiegeln die Zeiten einen Versatz dar. Es sei an dieser Stelle zudem zu erwähnen, dass die Reihenfolge der aufgelisteten Teilnehmer, nicht der Reihenfolge der Teilnahme entspricht. Dies soll verhindern, dass auf die einzelnen Teilnehmer die erhobenen Daten wieder zurückzuführen sind.



(a) Auswertung der Aufgabenstellung für den SEE-Profiler.



(b) Auswertung der Aufgabenstellung für den Unity-Profiler.

Abbildung 5.2: Auswertung aller Aufgabenstellungen dargestellt mit einem Pie-Chart.

In der Abbildung 5.2 ist noch einmal der Übersicht halber als Pie-Chart dargestellt, wie viele richtige und falsche Antworten, sowie Folgefehler bei der Bearbeitung der Aufgaben aufgetreten sind. Die Anzahl der Fehler unterscheiden sich kaum, hingegen aber die Anzahl der

richtigen Antworten und Folgefehler. Unter den Folgefehler fällt alles, bei dem ein falscher Wert zu den nachfolgenden Antworten führt. Bei der ersten Aufgabe, bei der es darum geht fünf Methoden zu ermitteln, die sehr viel absolute Zeit benötigen und diese anschließend absteigend in eine Tabelle einzutragen, wurde oftmals bei beiden Profilern eine Methode übersehen. Grund hierfür ist, dass bei dem SEE-Profiler nicht alle Heatballoons betrachtet wurden nach ihrer absoluten Zeit und bei dem Unity-Profiler nicht jeder Knoten im Aufrufgraph beleuchtet wurde, der zum SEE-Projekt angehört. Dies geht aus der Beobachtung der Teilnehmer, bei der Bearbeitung der Aufgaben hervor. Des Weiteren wurde, aus denselben Gründen, grade bei dem Unity-Profiler die dritte Aufgabe, bei der es darum geht, die Anzahl der Methodenaufrufe mit den Methodennamen, absteigend in eine Tabelle einzutragen, mit Folgefehlern oder grundlegend falsch bearbeitet.

Es lässt sich durch die Auswertung der Aufgaben unter der Berücksichtigung der Umwelteinflüsse folgern, dass der SEE-Profiler bei der Bearbeitung von der gleichen Problemstellung unter minimierten Datensätzen, vergleichbare bzw. bessere Effektivität und Effizienz besitzt, wie der Unity-Profiler. Diese Folgerung spiegelt lediglich die Ergebnisse von zehn Probanden dar, eine weitere Studie, mit mehr Teilnehmern und Betrachtung einer größeren Datenmenge ist unumgänglich.

5.4.3 System-Usability-Scale

Proband	Unity-Antworten	SEE-Antworten	Unity-Score	SEE-Score
P01	5 1 4 2 4 1 4 1 3 1	4 2 3 2 3 2 4 2 2 1	85	67,5
P02	4 1 4 1 3 1 4 1 4 1	3 1 4 2 3 1 3 3 3 1	85	70
P03	4 1 5 1 3 1 4 1 4 1	4 1 4 1 3 1 3 2 3 1	87,5	77,5
P04	3 1 4 1 3 1 4 2 4 1	3 1 4 1 3 1 3 2 3 1	80	75
P05	3 1 4 1 3 1 4 1 3 1	3 1 3 1 2 2 3 2 3 1	80	67,5
P06	4 1 4 1 4 1 5 1 4 1	2 2 3 1 3 2 4 2 5 1	90	72,5
P07	2 4 2 1 3 2 4 5 2 2	4 3 3 1 4 1 4 4 3 3	47,5	65
P08	5 1 5 2 5 1 5 1 4 2	5 1 5 1 5 1 4 1 5 2	92,5	95
P09	4 2 4 2 4 2 4 2 4 1	2 2 3 1 3 2 4 5 5 1	77,5	65
P10	4 1 4 1 4 1 4 1 4 1	3 2 4 1 2 1 3 2 3 1	87,5	70

Tabelle 5.3: Der System-Usability-Scale ausgewertet, für alle 10 Probanden.

In der Tabelle 5.3 sieht man einmal die Auswertung des System-Usability-Scales für jeden Probanden. Die Reihenfolge, in der die Probanden gelistet werden, entspricht nicht der Reihenfolge der Teilnahme, damit soll verhindert werden, dass die Ergebnisse auf einen Teilnehmer wieder zurückführt werden. Die Antworten für jede Frage wurden von links nach rechts für den jeweiligen Profiler in den Spalten ,,Unity-Antworten" und ,,SEE-Antworten" eingetragen. Gleich rechts daneben steht der Score, für den jeweiligen Profiler, der sich durch die Antworten ergab. Rechnet man für den Unity- und SEE-Profiler den Score zusammen, ergibt sich, nachdem dieser Score mit der Anzahl der Teilnehmenden teilt, ein durchschnittlicher Wert von 81,25 für den Unity-Profiler und 72,5 im Durchschnitt für den SEE-Profiler. Damit wären beide Profiler zumindest nachdem SUS, gebrauchstauglich. Was jedoch auffällt, ist

das Rauschen in den erhobenen Daten vorhanden ist. Der Proband P07 hat im Vergleich zu den restlichen Teilnehmern einen deutlich geringeren Score für den Unity-Profiler gegeben und Teilnehmer P08 hat für beide Profiler einen deutlich hohen Score abgegeben. Durch die Verstreuung, wenn auch minimal, wird der Zentralwert betrachtet, der unter anderem durch die Abbildung 5.3 dargestellt wird. Der Zentralwert für den Unity-Profiler beträgt 85 und für den SEE-Profiler 70.

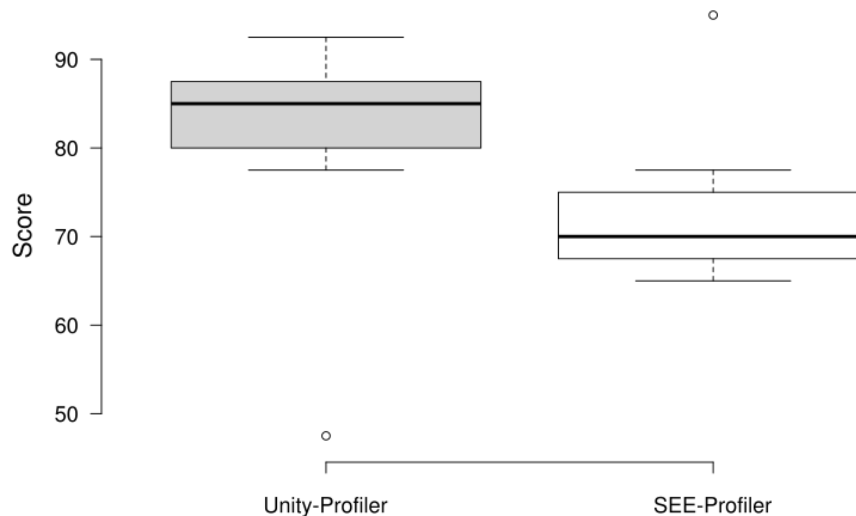


Abbildung 5.3: Den Score für den Unity-Profiler und SEE-Profiler dargestellt durch ein Box-Plot-Diagramm.

Eine Voreingenommenheit durch den abweichenden Score ist nicht auszuschließen. Eine weitere Erkenntnis aus den erhobenen Daten ist, dass auf die Frage, ob die Teilnehmer die Bedienung von dem SEE-Profiler als sehr umständlich empfinden, eine erkennbare Bewertung abgegeben wurde, der diesen Punkt zustimmt. Dies ist darauf zurückzuführen, dass die Navigation vom angeklickten Gebäude bis zum Profiler-Fenster viel zu umständlich ist. Dies ergab sich durch die abschließende Diskussion der Studie. Ein Vorschlag eines Teilnehmers, das zur Kenntnis genommen wird für die spätere Weiterentwicklung ist, dass anstelle des Profiler-Fensters neben den restlichen Fenstern, das Profiler-Fenster zentral in der Sicht des Benutzers erscheint, sodass eine Navigation zu dem Fenster nicht mehr nötig ist.

5.4.4 Offene Fragestellung

Wie bereits erläutert, hatten die Teilnehmer die Chance ihre Meinung abzugeben, über den SEE-Profiler. Im letzten Kapitel wurde bereits auf eine Antwort eingegangen, nachfolgend sind weitere gelistet, wobei bei der Übernahme, Rechtschreibfehler korrigiert wurden:

- ,,Verbessert werden sollte die Steuerung. Ansonsten hat mir die Darstellung vom SEE-Profiler gut gefallen."

- ,,Man könnte die Überschaubarkeit verbessern, vor allem, wenn man nach einer bestimmten Klasse oder Methode suchen möchte. Ansonsten hat die Bedienung schon Spaß gemacht."

- ,,Die Überschaubarkeit allgemein könnte besser werden, um den Überblick der Klassen und den Paketen zu verbessern. Dass die Größe die Anzahl der Aufrufe und die Farbe die CPU-Zeit anzeigt, finde ich eine echt passende Idee, mit der man das verdeutlichen kann, vor allem im Vergleich zu den anderen Methoden."

Neben der Bedienung ist die Überschaubarkeit oftmals bemängelt worden. Dies ist unter anderem darauf zurückzuführen, dass durch eine Fehldarstellung der Gebäude in der JLGCity, keine sichtbare Höhe oder Tiefe gegeben war. Dieser Fehler wurde zwar am Ende der Studie von den Entwicklern des SEE-Projekts behoben, jedoch wurde diese Änderung für die Studie nicht übernommen. Der Grund für diese Entscheidung war, dass die restlichen vier Teilnehmer kurz vor dem Ende der Benutzerstudie, einen Vorteil gegenüber den vorherigen Probanden hätten und damit die erfassten Daten leicht verfälscht wären.

5.4.5 Threats-to-Validity

Threats to Validity oder auch ,,Bedrohung der Gültigkeit", befasst sich damit, ob die Ergebnisse der Benutzerstudie tatsächlich valide sind. Man unterscheidet bei diesem Ansatz zwischen der internen Gültigkeit, also Faktoren, die einen direkten Einfluss auf die Gültigkeit des Ergebnisses haben und externer Gültigkeit, bei dem die Frage geklärt wird, ob die erfassten Ergebnisse verallgemeinert werden können auf andere Personen, Situationen, Orte oder Zeiten [46]. Nachfolgend werden einige Faktoren der internen und externen Gültigkeit erläutert, bevor sie einzeln beantwortet werden, mit dem Bezug zu der durchgeführten Benutzerstudie.

Interne Gültigkeit

History Abhängig davon, an welchem Tag und Uhrzeit die Benutzerstudie von den Probanden durchgeführt wird, kann dies Einfluss auf die Ergebnisse der Studie haben.

Maturation Die Zeit, welche für die Bearbeitung der Benutzerstudie benötigt wird, sollte angemessen gewählt werden. Ansonsten kann sich der Teilnehmer schnell langweilen, den Fokus verlieren oder aber einen Lernerfolg haben, welcher sich negativ oder positiv auf die Ergebnisse auswirkt.

Testing Die gleichen Aufgaben, unter den gleichen Umständen sollten nicht von demselben Teilnehmer bearbeitet werden, da sonst die Ergebnisse sich unterscheiden können, als wenn die Aufgaben zum ersten Mal bearbeitet werden.

Instrumentation Es soll darauf geachtet werden, wie die einzelnen Daten erhoben werden. Je nachdem welche Methodik angewandt wird und wie diese umgesetzt wird, kann dies einen Einfluss auf die Ergebnisse haben.

Selection Je nachdem, wie die Auswahl der Teilnehmer für die Studie ausfällt, kann dies einen Einfluss auf die Ergebnisse haben. Dabei kann eine Gruppe von freiwilligen Teilnehmern oder aber selektiv ausgewählte Personen aus einer Gruppe, unterschiedlichen Einfluss auf die Ergebnisse haben.

Mortality In Benutzerstudien ist es keine Seltenheit, dass Probanden die Studie abbrechen. Es muss an dieser Stelle genau geschaut werden, zu welcher Gruppe man den Probanden einordnen kann. Je nachdem in welche Gruppe man den Probanden eingliedert, kann dies weniger oder mehr die Gültigkeit der Ergebnisse infrage stellen.

Externe Gültigkeit

Interaction of selection and treatment Es sollte immer darauf geachtet werden, dass die ausgewählte Gruppe von Teilnehmern repräsentativ sind und keine Teilnehmer ausgeschlossen werden, die ebenfalls infrage kommen würden.

Interaction of setting and treatment Die Umgebung und Problemstellung für die Benutzerstudie sollte entsprechend des zu erreichenden Ziels gewählt werden. Dies kann dann der Fall sein, wenn z. B. die Studie eines Systems, das in der Industrie eingesetzt werden soll, mit veralteter Software durchgeführt wird, anstatt mit neuerer Software, wie es in der industriellen Praxis üblich ist.

Interaction of history and treatment Führt man die Benutzerstudie an einem bestimmten Tag oder Uhrzeit durch, bei dem es zu Interneteinbrüchen oder anderen technischen Problemen kommen kann, kann dies die Gültigkeit der Ergebnisse beeinflussen.

Auf diese genannten Faktoren wird nun detaillierter eingegangen und die ausgewerteten Ergebnisse kritisch hinterfragt.

Interne Gültigkeit

History Der Tag und die Uhrzeit, an dem die Probanden teilgenommen haben, wurde sich nach den Probanden gerichtet, an dem es Ihnen am besten passt. Trotzdem ist nicht auszuschließen, dass einige Teilnehmer dennoch einen ungünstigen Termin gewählt haben, sodass sich das auf die Ergebnisse negativ ausgewirkt hat. Dies stellt nur eine Vermutung dar, da einige der Teilnehmer nach wenigen Minuten nach der Benutzerstudie sich schon wieder verabschieden mussten.

Maturation Wie im Abschnitt 5.2 erläutert, wurde die Menge an Laufzeitdaten reduziert, nachdem ein Durchlauf mit einem Testpiloten über 60 Minuten Bearbeitungszeit betragen hatte. Dadurch wurde erhofft, dass eine höhere Bereitschaft zur Teilnahme und eine angenehmere Bearbeitung der Aufgaben zur Folge hat. Die durchschnittliche Zeit, der Bearbeitung betrug 48 Minuten. Da jedoch nicht aktiv nach gefragt wurde, ob die Teilnehmer sich bei der Bearbeitung der Aufgaben gelangweilt haben oder ihren Fokus verloren haben, kann dennoch nicht davon ausgegangen werden, dass die Ergebnisse dadurch beeinflusst wurden.

Testing Da sämtliche Aufgaben z. B. für den SEE-Profiler mit der gleichen Steuerung und Datensätzen durchgeführt wurde, kann nicht ausgeschlossen werden, dass die Bearbeitung der ersten Aufgabe, bereits positive Effekte auf die Bearbeitung der zweiten Aufgabe hat, nachdem die Steuerung und einige der Datensätze bereits bekannt sind. Das Gleiche gilt für die Bearbeitung der Aufgaben, bei dem Unity-Profiler.

Instrumentation Mehrdeutigkeiten bei der Fragestellung, in den Fragebögen bis auf den System-Usability-Scale, könnten die Ergebnisse beeinflusst haben. Dies würde sich dadurch bestätigen, dass einige wenige Probanden bei der absteigender Auflistung von Methoden und ihrer absoluten Zeit, dieser Aufgabe nicht nachgekommen sind, wie in der dritten Aufgabe, bei den Unity-Aufgaben 5.2 zu entnehmen ist.

Selection Einen negativen Einfluss auf die Ergebnisse durch selektive Teilnehmer, aufgrund der aktuellen Lage ist nicht auszuschließen. Da es schwer war, die Menge an Probanden zu finden für die Studie, wurden explizit Mitstudienerde angeschrieben und befragt, ob sie an der Studie teilnehmen würden. Einige wenige haben sich persönlich gemeldet, in diesem Fall kann die Motivation dieser Teilnahme einen negativen Einfluss auf die Ergebnisse haben. Ein Extremfall wurde bereits ausgemacht bei der Auswertung des SUS 5.3, der im Vergleich zu den restlichen Teilnehmern, einen viel zu hohen Score vergeben hat.

Mortality Dieser Faktor wird nicht weiter beachtet, da keiner der Teilnehmer die Studie abgebrochen hat.

Externe Gültigkeit

Interaction of selection and treatment Alle Teilnehmer der Studie haben einen Hintergrund in der Informatik und einige wenige kennen sich bereits mit Profilern aus, dadurch wird davon ausgegangen, dass die Teilnehmer repräsentativ genug sind. Jedoch ist die Teilnehmerzahl gering, weshalb die Ergebnisse, ob sie aussagekräftig genug sind, hinterfragt werden kann.

Interaction of setting and treatment Die Problemstellung entspricht nicht ganz des Einsatzzwecks, da nur eine minimierte Datenmenge betrachtet wurde, während in der Praxis tausende, wenn nicht Millionen von Code betrachtet werden muss. In diesem Gesichtspunkt sind die Ergebnisse nicht ausdrucksstark genug.

Interaction of history and treatment Es kam während der Bearbeitung der Aufgaben zu Verzögerungen durch schlechtes Internet sowie durch den Fernzugriff, dass sich negativ auf die Ergebnisse ausgewirkt hat. Die Effizienz und der reibungslose Ablauf litt darunter. Aus diesem Grund sind grade Werte, welche Effektivität und Effizienz betreffen, nicht aussagekräftig genug.

5.5 Beantwortung der Forschungsfragen

Nachfolgend werden die Forschungsfragen beantwortet, in Bezug zur durchgeführten Benutzerstudie. Einfachheitshalber werden die Fragen nochmal aufgelistet.

Erste Forschungsfrage Ist der SEE-Profiler mindestens so effektiv, beim Auffinden von Performanz-Informationen, im Vergleich zum klassischen Ansatz, dem Unity-Profiler?

Zweite Forschungsfrage Ist der SEE-Profiler mindestens so effizient, bei gegebener Aufgabenstellung, die möglichen Lösungen zu ermitteln, im Vergleich zum klassischen Ansatz, dem Unity-Profiler?

Dritte Forschungsfrage Ist der SEE-Profiler gebrauchstauglich?

Wie in Abschnitt 5.2 bereits erwähnt, muss zur Beantwortung der ersten zwei Forschungsfragen klargestellt werden, dass es Umwelteinflüsse durch den Fernzugriff und der Internetgeschwindigkeit gab, die Einfluss auf die Effektivität und die Effizienz genommen hat, sodass nicht eindeutig Aussagen über die Usability-Aspekte gemacht werden können. Werden die Umwelteinflüsse hingenommen, da sie für beide Profiler gleichermaßen galten, so könnte man die ersten zwei Forschungsfragen mit einem „Ja" beantworten, ansonsten ohne eine weitere Studie mit einem „Nein". Ganz so streng wird das nicht bei dem SUS gesehen, da hier eine subjektive Meinung abgegeben wurde, wie sie den jeweiligen Profiler empfunden haben. Da der SEE-Profiler einen Score von mindestens 70 Punkten erreicht hat, auch ohne die Betrachtung vom Rauschen in den Datensätzen, kann diese Forschungsfrage mit einem „Ja" beantwortet werden. Damit wird impliziert, dass der SEE-Profiler eine sinnvolle Alternative zu einem klassischen Ansatz, wie dem Unity-Profiler darstellt.

KAPITEL 6

Fazit und Ausblick

6.1 Fazit

Im Rahmen dieser Arbeit wurde der *SEE-Profiler* entwickelt, der sich aus dem TraceEmebedder, einer eigenen Darstellungsform, den Heatballoons, Profiler-Fenster sowie bereits vorhandenen Komponenten in SEE zusammensetzt. Der TraceEmbedder stellt dabei ein Programm dar, das entwickelt wurde, um `C#`-Anwendungen zu instrumentieren, damit Laufzeitdaten auf Methoden-Ebene erhoben und auf Laufzeitflaschenhälse untersucht werden können. Bei den Laufzeitdaten selbst handelt es sich um die absolute und eigene Zeit einer Methode, den Methodenparametern sowie den Aufrufgraph. Nach erfolgreicher Instrumentierung und Aufzeichnung der Laufzeitdaten, werden diese im Anschluss in einer JLG-Datei, dem JLG-Dateiformat entsprechend zwischengespeichert. Das Aufzeichnen der Laufzeitdaten hat gezeigt, dass eine hohe Last entsteht, bei den instrumentierten `C#`-Programmen. Es wurden jedoch Möglichkeiten zur Begrenzung der aufzuzeichnenden Daten integriert, sodass der Overhead höchstens für eine kurze Zeitspanne gilt.

Von einem JLG-Parser in SEE, kann die erzeugte JLG-Datei eingelesen und in der JLGCity innerhalb von SEE visualisiert werden. Dabei werden die erhobenen Laufzeitdaten, mit einer selbstentworfenen Darstellung, den Heatballoons wiedergegeben. Diese stellen dreidimensionale Kugel-Objekte dar, die über korrelierende Gebäuden in der JLGCity schweben und mit einer Linie mit diesen verknüpft sind, um die Zugehörigkeit auszudrücken. Die von den Heatballoons repräsentierenden Laufzeitdaten, können in einem selbstentworfenen Profiler-Fenster tabellarisch angesehen werden. Insgesamt kann die Visualisierung, mithilfe der bereits vorhandenen Komponenten in der JLGCity, vor- und rückwärts durchlaufen werden, um eine Performanz-Analyse durchzuführen oder aber um Verständnis über `C#`-Anwendungen auf Methoden-Ebene zu gewinnen.

In einer Benutzerstudie von zehn Probanden wurde der SEE-Profiler im Vergleich zu dem Unity-Profiler gesetzt, bei dem identische Aufgabenstellungen jedoch mit unterschiedlichen Laufzeitdaten bearbeitet wurden. Das Ziel der Benutzerstudie war es herauszufinden, ob der SEE-Profiler effektiv, effizient und gebrauchstauglich ist. Die Effektivität und Effizienz des Profilers wurden durch die richtige Bearbeitung der Aufgaben und durch die Bearbeitungszeit bestimmt. Dabei schneidet der SEE-Profiler im Vergleich zu den State-of-the-Art, dem Unity-Profiler unter den gegebenen Umständen der Studie, minimal besser ab. Die Gebrauchstauglichkeit wurde mithilfe des System-Usability-Scales bestimmt, wobei der Zentralwert des errechneten Scores bei dem SEE-Profiler 70 und des Unity-Profilers 85 beträgt. Der Zentralwert musste herangezogen werden, da bei einigen wenigen Probanden, ein Bias bei der Vergabe der Punktzahl in dem SUS zu vermuten waren. Trotz allem sind beide Profiler, nachdem SUS gebrauchstauglich und der SEE-Profiler stellt damit eine sinnvolle Alternative zu herkömmlichen Profilern dar.

6.2 Ausblick

Im Folgenden werden Möglichkeiten und Ideen der Erweiterung und Verbesserung vorgestellt, die mit den Ergebnissen dieser Bachelorarbeit angegangen werden können.

6.2.1 Tool zur Abhängigkeitsauflösung

Wie bereits erklärt, gab es einige Schwierigkeiten mit `C#`-Anwendungen, die *TraceUtil.dll* nicht auflösen konnten, weshalb das Aufzeichnen nicht funktionierte. Im Rahmen dieser Arbeit konnte kein Tool gefunden werden, der dieses Problem lösen kann. Eine mögliche Erweiterung dieser Arbeit wäre es, ein solches Tool zu entwickeln und damit den TraceEmbedder universal für jede `C#`-Anwendung zu machen. Eine mögliche Idee wäre es, die *EmbeddedTracer*-Klasse vollständig aus Instruktionen zusammenzusetzen, sodass keine Assembly benötigt wird zum Auflösen. Damit wäre es möglich eine volle Integration der Klasse, in jede beliebige `C#`-Anwendung durchzuführen.

6.2.2 Tracing auf Statement-Ebene

In dieser Arbeit wurde die Möglichkeit geschaffen, `C#`-Anwendungen auf Methoden-Ebene aufzuzeichnen. Eine mögliche Erweiterung würde vorsehen, zusätzlich auf Statement-Ebene aufzeichnen zu können. Dazu müsste ein Tool entwickelt werden, dass einem Debugger gleicht. Eine Idee wäre es, das ganze mit der Laufzeitumgebung Common-Language-Runtime von `C#` umzusetzen, ähnlich wie es Kipka für die Programmiersprache Java gemacht hat, in dem man sich dafür in den Java-Virtual-Machine-Prozess einklinkt.

6.2.3 Live-Visualisierung

Eine zukunftsorientierter Gedanke, wäre es die Visualisierung parallel zum ausgeführten Programm zu erzeugen. Da jedoch viele Operationen nacheinander mit einer enormen Geschwindigkeit bearbeitet werden, müsste man ein Gedächtnisfeature einbauen, sodass man die Visualisierung stoppt, die bisher behandelten Operationen sich wieder anschaut und wenn man mit der Analyse der bisherigen verarbeiteten Laufzeitdaten fertig ist, dort ansetzt, wo man gestoppt hat ober eben auf den aktuellsten Stand springt.

6.2.4 Datenkompression

Eine Sache, die man beim Erheben der Laufzeitdaten mitgenommen hat ist, dass in kurzer Zeit eine hohe Datenmenge entsteht, die teilweise nach wenigen Minuten im Megabit Bereich liegt. Schaut man sich einen Teil der Aufzeichnung an, merkt man, dass an vielen Stellen Redundanzen vorhanden sind. Ein formaler Algorithmus, der dieses Problem angeht und Redundanzen minimiert, kann eine ganze Bachelorarbeit umfassen.

ABBILDUNGSVERZEICHNIS

2.1 Ausschnitt eines visualisierten Debug-Prozesses in in der ,,JLGCity” von Kipka [20]. 6
2.2 Ausschnitt der Visualisierung von Rohloff [34]. 7
2.3 Schritte bis zur Konstruktion der Visualisierung von Seeman et al. [36]. . . . 8
2.4 Ausschnitt eines Versionsunterschieds von Seeman et al. [36]. 9
2.5 Visualisierung einer Softwarelandschaft in der ,,landscape level perspective” von Fittkau et al. [12]. 10
2.6 Visualisierung einer Software in der *system level perspective* von Fittkau et al. [12]. 10
2.7 Ausschnitt der visualisierten Fassung von dem SEE Projekt in Unity [41]. . . 14
2.8 Die grafische Benutzeroberfläche von dem Unity-Profiler, wobei der Aufrufgraph dargestellt als eine Hierarchie, von den SEE-Laufzeitdaten zu sehen sind. [42]. 15

3.1 Sämtliche Klassen und Beziehungen, die für die Instrumentierung nötig sind, als UML-Klassendiagramm dargestellt. 20
3.2 Entwurf der grafischen Benutzeroberfläche, die als Einstiegspunkt für den *TraceEmbedder* dient. 21
3.3 Entwurf der Heatballoons, wobei zusätzlich der Abkühlungsprozess dargestellt ist. 24
3.4 Entwurf des Profiler-Fenster. 25

4.1 In *Windows-Forms* entwickelte grafische Benutzeroberfläche für den TraceEmbedder. 29
4.2 Einsatz von Heatballoons, bei dem Laufzeitdaten von dem SEE-Projekt in SEE bzw. JLGCity selbst dargestellt werden. 36
4.3 Präsentation der Laufzeitdaten im Profiler-Fenster. 36

5.1 Auswertung der Ergebnisse des Einführungsfragebogens. 46
5.2 Auswertung aller Aufgabenstellungen dargestellt mit einem Pie-Chart. 48
5.3 Den Score für den Unity-Profiler und SEE-Profiler dargestellt durch ein Box-Plot-Diagramm. 50

LISTINGS

3.1 Gekürzte Fassung des Inhalts einer JLG-Datei. 22
4.1 Vor der Instrumentierung der *ExampleMethod*-Methode. 30
4.2 Nach der Instrumentierung der *ExampleMethod*-Methode. 30
4.3 Instruktionen von der *ExampleMethod*-Methode, vor der Instrumentierung. . 31
4.4 Instruktionen von der *ExampleMethod*-Methode, nach der Instrumentierung. . 31
4.5 Eine Methode die instrumentiert wurde, um die Laufzeitdaten zu erfassen. . . 39
4.6 Auszug aus der erzeugten JLG-Datei, dass zeigt das der Startpunkt richtig gesetzt wurde. 39
4.7 Eine Methode die instrumentiert wurde, um die Laufzeitdaten zu erfassen. . . 40
4.8 Auszug aus der erzeugten JLG-Datei, dass zeigt das ab dem eingefügten Endpunkt, nicht weiter aufgezeichnet wurde. 40
4.9 Eine Methode die instrumentiert wurde, um die Laufzeitdaten zu erfassen. . . 41

LITERATURVERZEICHNIS

[1] AG, T. [2005], 'Teamviewer', `https://www.teamviewer.com/de/`. (zugegriffen am: 30.10.2021).

[2] Baker, H. G. [1992], 'Cons should not cons its arguments, or, a lazy alloc is a smart alloc', *ACM Sigplan Notices* **27**(3), 24–34.

[3] Bassil, S. and Keller, R. K. [2001], Software visualization tools: Survey and analysis, *in* 'Proceedings 9th International Workshop on Program Comprehension. IWPC 2001', IEEE, pp. 7–17.

[4] Brooke, J. [1986], 'System usability scale (sus): a quick-and-dirty method of system evaluation user information', *Reading, UK: Digital Equipment Co Ltd* **43**, 1–7.

[5] Canfora, G. and Cimitile, A. [2001], Software maintenance, *in* 'Handbook of Software Engineering and Knowledge Engineering: Volume I: Fundamentals', World Scientific, pp. 91–120.

[6] Charlie Poole, Rob Prouse, S. B. N. C. et al. [2017], 'Nunit', `https://nunit.org/`. (zugegriffen am: 25.10.2021).

[7] Corporation, O. [1995], 'Java', `https://www.java.com/de/`. (zugegriffen am: 28.10.2021).

[8] Curtsinger, C. and Berger, E. D. [2015], Coz: Finding code that counts with causal profiling, *in* 'Proceedings of the 25th Symposium on Operating Systems Principles', pp. 184–197.

[9] Diehl, S. [2007], 'Visualizing the structure, behaviour, and evolution of software'.

[10] Evain, J. et al. [2004], 'Mono.cecil', `https://github.com/jbevain/cecil`. (zugegriffen am: 28.08.2021).

[11] Fasolino, A. and Visaggio, G. [1999], Improving software comprehension through an automated dependency tracer, *in* 'Proceedings Seventh International Workshop on Program Comprehension', pp. 58–65.

[12] Fittkau, F., Waller, J., Wulf, C. and Hasselbring, W. [2013], Live trace visualization for comprehending large software landscapes: The explorviz approach, *in* '2013 First IEEE Working Conference on Software Visualization (VISSOFT)', IEEE, pp. 1–4.

[13] *Gacutil.exe (Global Assembly Cache tool)* [o. D.], `https://docs.microsoft.com/en-us/dotnet/framework/tools/gacutil-exe-gac-tool`. (zugegriffen am: 29.07.2021).

[14] *Global Assembly Cache* [n. D.], `https://docs.microsoft.com/en-us/dotnet/framework/app-domains/gac`. (zugegriffen am: 29.07.2021).

[15] Gómez-Henríquez, L. M. [2001], 'Software visualization: An overview'.

[16] Gross, T. [2020], Visualisierung von dynamischen aufrufgraphen mit vr-basierten software-staedten, Bachelor's thesis, Universität Bremen, Bremen, Germany.

[17] Hebig, R. [2018], 'Ui-tracer', *Software Engineering und Software Management 2018* .

[18] Inc., D. [2015], 'Discord', `https://discord.com/`. (zugegriffen am: 30.10.2021).

[19] Initiative, H. H. [o. D.], 'Kobotoolbox', `https://www.kobotoolbox.org/`. (zugegriffen am: 30.10.2021).

[20] Kipka, L. [2020], Bachelorarbeit: Software-debugging mithilfe von code cities oder: Software-debugging with code cities, Bachelor's thesis, Universität Bremen, Bremen, Germany.

[21] Koschke, R. [2002], Software visualization for reverse engineering, *in* 'Software Visualization', Springer, pp. 138–150.

[22] Koschke, R. [n.d.], 'See', `https://see.uni-bremen.de/`. (zugegriffen am: 24.07.2021).

[23] Koschke, R. et al. [o. D.], 'Interactions', `https://github.com/uni-bremen-agst/SEE/wiki/Interactions`. (zugegriffen am: 24.11.2021).

[24] Lahl, O. and Pietrowsky, R. [2008], 'Tracer: A general-purpose software library for logging events in computerized experiments', *Behavior research methods* **40**(4), 1163–1169.

[25] Lange, C. F. and Chaudron, M. R. [2007], Interactive views to improve the comprehension of uml models-an experimental validation, *in* '15th IEEE International Conference on Program Comprehension (ICPC'07)', IEEE, pp. 221–230.

[26] Microsoft [1997], 'Visual studio', `https://visualstudio.microsoft.com/de/`. (zugegriffen am: 30.08.2021).

[27] Microsoft [2002], 'Windows forms', `https://dotnet.microsoft.com/apps/desktop`. (zugegriffen am: 30.08.2021).

[28] Nagarajan, A. and Memon, A. [2003], Refactoring using event-based profiling, *in* 'Proceedings of the 1st International Workshop on Refactoring: Achievements, Challenges, Effects (REFACE)', Citeseer.

[29] Pammer, S. et al. [2011], 'Ilspy', `https://github.com/icsharpcode/ILSpy`. (zugegriffen am: 28.08.2021).

[30] Panas, T., Epperly, T., Quinlan, D., Saebjornsen, A. and Vuduc, R. [2007], Communicating software architecture using a unified single-view visualization, *in* '12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)', IEEE, pp. 217–228.

[31] Pierce, J., Smith, M. D. and Mudge, T. [1995], Instrumentation tools, *in* 'Fast Simulation of Computer Architectures', Springer, pp. 47–86.

[32] Pires, N. [2004], 'Dimmer - adjust the brightness of all monitors, screens and displays', `https://www.nelsonpires.com/software/dimmer`. (zugegriffen am: 05.11.2021).

[33] *PubFlow: Publication Workflows for Scientific Data - From acquisition and processing toward archival and publication* [o. D.], `https://www.pubflow.uni-kiel.de`. (zugegriffen am: 29.09.2021).

[34] Rohloff, Y. [2021], Performance-profiling und visualisierung in software-staedten oder performance profiling and visualization in software cities, Master's thesis, Universität Bremen, Bremen, Germany.

[35] Rupp, C. and Queins, S. [2003], 'Testen mit use-cases'.

[36] Seemann, J. and von Gudenberg, J. W. [1998], Visualization of differences between versions of object-oriented software, *in* 'Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering', IEEE, pp. 201–204.

[37] Shi, J., Ji, W., Zhang, L., Gao, Y., Zhang, H. and Qing, D. [2016], Profiling and analysis of object lazy allocation in java programs, *in* '2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/-Distributed Computing (SNPD)', pp. 591–596.

[38] Shneiderman, B. [2003], The eyes have it: A task by data type taxonomy for information visualizations, *in* 'The craft of information visualization', Elsevier, pp. 364–371.

[39] Simmonas, G. et al. [o. D.], 'dnspy', `https://github.com/dnSpy/dnSpy`. (zugegriffen am: 25.09.2021).

[40] *Standard Performance Evaluation Corporation* [o. D.], `https://www.spec.org/jvm98/`. SPECjvm98.

[41] Unity Technologies [2005*a*], 'Unity', `https://unity.com/`. (zugegriffen am: 24.08.2021).

[42] Unity Technologies [2005*b*], 'Unity profiler', `https://docs.unity3d.com/Manual/Profiler.html`. (zugegriffen am: 24.08.2021).

[43] Unity Technologies [2019], 'Unity test framework', `https://docs.unity3d.com/Packages/com.unity.test-framework@1.1/manual/index.html`. (zugegriffen am: 24.08.2021).

[44] Valdy, F. et al. [2011], 'il-repack', `https://github.com/gluck/il-repack`. (zugegriffen am: 24.08.2021).

[45] Wettel, R. and Lanza, M. [2007], Visualizing software systems as cities, *in* '2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis', IEEE, pp. 92–99.

[46] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B. and Wesslén, A. [2012], *Experimentation in software engineering*, Springer Science & Business Media.

GLOSSAR

Begriff	Beschreibung	Seiten
.NET	Stellt ein Open-Source-Framework welches unter der Leitung von Microsoft, zur Entwicklung und Ausführung von Anwendungsprogrammen verwendet wird.	16
Assembly	Ein Assembly stellt ein Bauteil dar, das unterschiedliche Typen und andere Ressourcen enthalten kann. Im C#-Umfeld, gibt es im Kontext dieser Arbeit zwei Arten von Assemblies wie die Executable (EXE) und Dynamic-Linked-Library (DLL), die betrachtet werden.	16, 17, 21, 29, 32, 33, 38
Aufrufgraph	Ein Aufrufgraph, soll im Kontext dieser Arbeit, als ein Aufrufverhältnis zwischen Methoden verstanden werden. So wird bei der Aufzeichnung, immer erfasst, wann eine Methode betreten und wann sie wieder verlassen wurde. Sollte innerhalb einer Methode, weitere inneren Methoden aufgerufen werden, so wird die Aufzeichnung die durch die inneren Methoden enstanden ist gekapselt, von der äußeren Methode.	4, 19, 20, 34, 49, 54
Binärdatei	Diese Datei enthält Bitmuster und ist in den meisten Fällen nicht für Menschen lesbar. Produziert werden kann diese Datei, durch den Kompelierungsprozess.	11, 16
Common-Language-Runtime	*Common-Language-Runtime* (CLR) stellt eine Laufzeitumgebung für *.Net*-Programme dar. Unter anderem ist die CLR, zur Ausführung von C#-Programmen zuständig.	33, 55
Compiler	Ein Programm zur Überführung von Programmcode, von einer Quelle zur anderen. So kann z. B. Quellcode direkt zu Maschinencode umgewandelt werden, der von einem Computer ausgeführt werden kann.	11, 12, 16, 17, 21
Debugger	Ein Debugger ist ein Programmierwerkzeug, dass zur Untersuchung und Auffinden von Fehlern, in Programmen verwendet werden kann.	1

Begriff	Beschreibung	Seiten
Dekompilierer	Dies stellt ein Programm dar, der versucht den Prozess eines Compilers wieder umzudrehen. In dem z. B. eine ausführbare Datei, zurückgeführt wird auf den Quellcode, der jedoch nicht exakt dem vorherigen Zustand entspricht.	14, 16
Dynamic-Linked-Library	Die Dynamic-Linked-Libarary (DLL), stellt eine dynamische Programmbibliothek dar. Diese Bibliotheken können später von unterschiedlichen Programmen geteilt und benutzt werden.	16, 28
Executable	Executables (EXE) oder auch in Deutsch ,,ausführbare Dateien", bezeichnet eine Datei die als Programm ausgeführt werden kann.	11, 12, 16, 28
Finalizer	Ein Punkt in einem Programm, der die Ausführung einer Anwendung stoppt. Wird klassisch im Zusammenhang mit einem Debugger verwendet.	18, 26
Finalizer	Eine C#-Konstrukt, das zum Abschluss eines Objekts aufgerufen wird, bevor das Objekt vollständig von dem vom Heap geholt wird.	21, 34, 62
Garbage-Collector	Eine Software, die automatisiert in bestimmten Intervallen, Objekte aus dem Heap abräumt. Im Kontext dieser Arbeit, wird auf dieses Feature zurückgegriffen durch *Finalizer* oder *Destructoren*.	13, 21
Intermediate-Language	Eine Menge an binären Instruktionen, die von der *Common-Language-Runtime* ausgeführt werden können. Im Kontext von *.NET*, bezieht sich der Instruction-Language auf Executables oder Dynamic-Linked-Libraries.	16
Java	Eine objektorientierte Programmiersprache, die zum Zeitpunkt des Verfassens dieser Arbeit unter der Leitung von dem Unternehmen *Oracle Corporation* weiterentwickelt wird [7].	6, 8, 23, 55
JLG-Datei	Eine Datei, die nach dem JLG-Dateiformat, Laufzeitdaten zwischenspeichert [20].	6, 23, 34, 37, 39, 40, 54
JLG-Dateiformat	Eine von Lennart Kipka entwickeltes Dateiformat, der im Rahmen seiner Bachelorarbeit, zur Zwischenspeicherung von Laufzeitdaten entstanden ist [20].	6, 19, 22, 34, 54, 62
JLG-Parser	Eine von Lennart Kipka entwickelter JLG-Parser, der im Rahmen seiner Bachelorarbeit, zum Einlesen von JLG-Dateien verwendet wird um die erhobenen Laufzeitdaten in der JLGCity visualisieren zu können [20].	23, 54

Begriff	Beschreibung	Seiten
JLGCity	Eine von Lennart Kipka entwickelte Visualiserung innerhalb von SEE, dass eine Animation wie in einem Film, mit Laufzeitdaten schafft [20].	6, 7, 25, 26, 36, 51, 54, 56
JVM	Die Java-Virtual-Mashine (JVM) ist Teil der Java-Runtime-Environment und unter anderem zuständig für die Ausführung von Java-Bytecode.	13
Linker	Stellt einzelne Programmmodule zu einem ausführbaren Programm zusammen.	11, 12
Mentalle-Modelle	Ein Modell, dass über das Verhalten und Vorgehen eines Menschen entscheidet, je nachdem welche Situation vorliegt. Ein Mensch kann viele Mentalle Modelle besitzen, wie etwas funktioniert. Oft tendiert er zu einfacheren Modellen und wenn ein solcher versagt, wird zu komplexeren Modellen zurückgegriffen.	3
Objektdatei	Eine Objektdatei kann der Output eines Compilers sein, wobei mehrere solcher Dateien zu einer ausführbaren Datei zusammengeführt werden können.	12
Profiler	Ein Profiler ist ein Programmierwerkzeug, die das Laufzeitverhalten von Programmen auf Laufzeitflaschenhälse untersuchen kann.	1, 3, 7, 13, 18, 37, 44–47, 49, 50, 53, 54
Re-Engineering	Stellt eine Disziplin dar, zur Anpassung bestehender Softwaresysteme sowie eine Neuschöpfung eines alten Softwareprodukts.	3
Reverse-Engineering	In dieser Disziplin geht es darum, ein Ganzes in seine Einzelteile herunterzubrechen, oftmals um dadurch Modifikationen durchzuführen und eine weitere Version einer bestehenden Software zu konstruieren.	3, 8, 14
Singleton	Ein Entwurfsmuster, bei dem man sicher geht, dass es von einer Instanz exakt eine Kopie existiert.	21
Softwarewartung	Die Wartung von Software umfasst jede Änderung, die vorgenommen wird, nach Auslieferung eines Softwareprodukts.	3
State-of-the-Art	Bezeichnet in diesem Kontext, den aktuellsten Entwicklungsstand der Technik.	1, 2, 42, 54
System-Usability-Scale	Stellt einen Fragebogen dar, der zur Bewertung von Usability-Aspekten einer Software aussagen macht. Im Kontext dieser Arbeit wird dieser verwendet, um die Gebrauchstauglichkeit des zu entwickelenden Profiler, zu erfassen.	42, 44, 49, 52, 54

Begriff	**Beschreibung**	**Seiten**
Thread	In Deutsch auch als „Ausführungsstrang” bezeichnet, stellt einen Teil eines Prozesses dar.	7

Änderungsprotokoll

Nachfolgend wird eine Tabelle aufgeführt, mit Bezug zu den Probanden, in der alle Änderungen festgehalten sind, die während der Benutzerstudie vorgenommen wurden.

Testperson	Änderung
TP3	Ab der dritten Testperson wurden Rechtschreibfehler in den Fragebögen korrigiert.
TP4	Ab der vierten Testperson kam eine neue Anzeige dazu, dass immer oben links in der Ecke des Bildschirms erscheint, wenn man auf ein Methoden- oder Klassen-Objekt sowie Heatballoon klickt. In dieser Anzeige wird ein Text eingeblendet, dass den Probanden darauf hinweist, für welche Methode oder Klasse das Profiler-Fenster geöffnet wurde. Vor dieser Anzeige, wurde den Teilnehmern immer mündlich mitgeteilt, auf welches Objekt sie geklickt haben. Des weiteren wurde bei dem Profiler-Fenster zur Abtrennung der Spalten von den Zeilen eine neue Linie platziert. Dies dient lediglich zur Dekoration und hat keinen Einfluss auf die Bearbeitung der Aufgaben. Diese beiden Änderungen wurden zudem, den Teilnehmern mitgeteilt, nachdem sie das Einführungsvideo gesehen haben.

Tabelle 6.2: Änderungen die während der Benutzerstudie vorgenommen wurden.