Performance-Profiling und Visualisierung in Software-Städten

oder
Performance Profiling and Visualization in Software Cities

Masterarbeit

Yannis Rohloff
Matrikelnummer: 411676

Erster Gutachter: Prof. Dr. rer.nat. Rainer Koschke
Zweiter Gutachter: Prof. Dr.-Ing. Gabriel Zachmann

23.08.2021

Fachbereich Mathematik / Informatik
Studiengang Informatik

E R K L Ä R U N G

Ich versichere, die Masterarbeit oder den von mir zu verantwortenden Teil einer Gruppenarbeit*) 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.

*) Bei einer Gruppenarbeit muss die individuelle Leistung deutlich abgrenzbar und bewertbar sein und den Anforderungen entsprechen.

Bremen, den _____

_____
(Unterschrift)

Abstract

Die Messung und die Visualisierung von Performance-Informationen ist für die Optimierung einer Anwendung eine wertvolle Tätigkeit. In dieser Arbeit wurden ein Profiler und eine Visualisierung entwickelt, welche neben Laufzeitanteilen, Aufrufanzahl und Mittelwerten einzelner Methodenlaufzeiten auch Perzentile ermitteln und darstellen kann. Die Visualisierung ist interaktiv und kann, durch die Darstellung des Aufrufkontextes mit dreidimensionalen Verbindungskanten, die Untersuchung der Programm-Performance erleichtern. In einer Benutzerstudie mit 16 Teilnehmern bestätigte sich die entworfene Visualisierung, als eine sinnvolle Alternative zu der Darstellung eines klassischen Profilers.

INHALTSVERZEICHNIS

KAPITEL 1

Einleitung

1.1 Motivation

In der Softwareentwicklung ist der Test von Software eine wichtige Tätigkeit, um Fehler frühzeitig zu erkennen. Wird diese Tätigkeit vernachlässigt, kann es zu erheblichem Mehraufwand in der Wartung kommen. Ein wichtiger Teil, welcher jedoch leicht in den Hintergrund rücken kann, ist die Performance der Anwendung, welche durch den klassischen Test von Anwendungen nur bedingt erfasst wird. Mit Last- und Performancetests und Software-Profiling können Anwendungen zur Laufzeit untersucht werden und es kann sichergestellt werden, dass die entwickelten Projekte auch nichtfunktionale Anforderungen erfüllen.

Neben der guten Benutzbarkeit einer Anwendung, beispielsweise durch intuitive Oberflächen, und weiteren Kriterien der Softwarequalität, könnte für die Akzeptanz einer Anwendung durch Benutzer potentiell auch die Reaktionszeit einer Anwendung ein sehr wichtiger Faktor sein, denn Anwendungen mit langen Wartezeiten können schnell frustrierend werden. Für die Analyse der Softwareperformance werden *Profiler* eingesetzt. Dies sind Programme, welche den Rechenzeitbedarf von einzelnen Programmkomponenten messen. Insbesondere mit der Verbreitung *Software-as-a-Service* Web-Anwendungen, welche von vielen Menschen gleichzeitig genutzt werden, wird zudem eine Betrachtung der Performanz in unterschiedlichen Belastungssituationen immer wichtiger. Diese wird in Last- und Performancetests untersucht [1]. Neben der Veränderung der Reaktionszeiten des Systems spielen hier auch Robustheit und Zuverlässigkeit der Anwendung eine wichtige Rolle.

Um Software besser zu verstehen, werden grafische Visualisierungen verwendet. Insbesondere im Bereich der *Program Comprehension* wird die Visualisierung verwendet, um die Navigation in Softwareprojekten zu erleichtern und das Laufzeitverhalten greifbarer zu machen [2]. Software-Städte sind eine beliebte Visualisierungsform, um die hierarchischen Eigenschaften und Relationen von Programmen darzustellen.

Ziel dieser Arbeit ist es, Performancedaten innerhalb einer dreidimensionalen Software-Stadt zu visualisieren, um so ein leichteres Verständnis der Performance und des Laufzeitverhaltens zu ermöglichen und so langsame oder häufig ausgeführte Abschnitte der Anwendung hervorzuheben, welche potentiell optimiert werden können. Die Verwendung von Perzentilen, welche in Last- und Performancetests häufig verwendet werden, soll ebenfalls integriert werden.

Anmerkung In dieser Arbeit wird im Bezug auf Personen nie auf ein einziges Geschlecht bezogen. Die Formulierungen, sind deshalb als generisches Maskulinum zu lesen. Innerhalb dieser Arbeit werden einige englische Fachbegriffe aus der Literatur übernommen und nicht ins Deutsche übersetzt. Obwohl diese Abschlussarbeit auf Deutsch verfasst wurde, werden in dieser Arbeit die englische Dezimaltrennzeichen, also ein Punkt statt einem Komma verwendet. Grund hierfür ist, dass diese Schreibweise in der Informatik eher geläufig ist.

1.2 Aufgabenstellung

Die bisher primär, für die Darstellung statischer Informationen, verwendete Umgebung für Software-Visualisierung *SEE* soll durch Funktionen für die Untersuchung von Performance-Informationen von Software erweitert werden. Es soll möglich sein, in der 3D-Anwendung die Performance einzelner Methoden darzustellen, um mögliche Optimierungspotentiale zu erkennen.

Hierfür sollen geeignete Metriken gemessen und dargestellt werden. Für die Messung der Daten soll in dieser Arbeit hierfür ein existierender Profiler für Java-Anwendungen angebunden oder neu entwickelt werden. Hier wurde sich früh für die Entwicklung eines neuen Profilers entschieden. Für die Visualisierung in *SEE* sollen neue Funktionen in die bestehende Anwendung integriert werden, welche eine Untersuchung dieser Metriken ermöglichen. Ziel ist insbesondere die Untersuchung von webbasierten Anwendungen, weshalb zudem der Bezug zu Last- und Performancetests erhalten bleiben soll. Der Profiler soll neben *Hot-Spots* und mittleren Laufzeitgeschwindigkeiten die, in Last- und Performancetests häufig verwendeten, Perzentile messen können.

Hauptziel ist es, herauszufinden, ob sich die 3D-Darstellung in SEE für die Untersuchung von Performancedaten für das Aufdecken von Performanceproblemen eignet und ob sie von Entwicklern angenommen wird. Die entwickelte Umgebung soll deshalb in einer Benutzerstudie mit praktizierenden Entwicklern evaluiert werden.

KAPITEL 2

Grundlagen

Das Grundlagenkapitel gibt zunächst einen Überblick über die Analyse und Visualisierung von Software im Allgemeinen. Schwerpunkt dieser Arbeit ist die Erhebung und Darstellung von Performance-Informationen, weshalb der Fokus in den weiteren Abschnitten hierauf liegt. Es werden verwandte Visualisierungen für dynamische und statische Informationen aufgezeigt und die zu erweiternde Anwendung *SEE*, der Arbeitsgruppe, in welcher diese Masterarbeit geschrieben wird, vorgestellt.

2.1 Visualisierung von Software

Software kann aus großen Mengen Programmcode bestehen. Sie besteht teilweise aus so viel Quelltext, dass sie für einzelne Personen nur schwer im Gesamten zu verstehen sind. Der Linux-Kernel besteht zum Beispiel aus mehreren Millionen Zeilen Quelltext [3]. Ziel der Softwarevisualisierung ist es, große Mengen an Informationen über Software so zusammenzufassen und darzustellen, dass sie ein leichteres Verständnis über die betrachteten Software oder ganze Software-Systeme ermöglicht. Visualisierungen werden für viele unterschiedliche Tätigkeiten, wie zum Beispiel die Wartung oder Reverse-Engineering verwendet [4]. Für die Visualisierung werden bekannte Techniken wie klassische Diagramme, unterschiedlichste Graph-Darstellungen, aber auch spezielle Metaphern, wie die Darstellung von Software als virtuelle Städte verwendet, um die Zusammenhänge greifbarer zu machen. Sowohl textbasierte [5] als auch dreidimensionale [6] Ansätze für die Visualisierung wurden mit dem Ziel der Vereinfachung des Verstehens von Software entwickelt. Eine 3D-Visualisierung kann helfen, Sachverhalte über die Software leichter verständlich zu machen und es ermöglicht die Struktur der Code-Basis besser zu verinnerlichen. In einer Untersuchung von Khaloo et al. [6] konnten vor allem Testpersonen mit geringerer Programmiererfahrung Aufgaben zum Softwareverständnis besser lösen, als in einer klassischen Darstellung in der Entwicklungsumgebung Visual Studio [7].

2.1.1 Statische Informationen

Statische Informationen von Software beinhalten jegliche Information, die über ein Programm gewonnen werden, ohne, dass dieses Programm ausgeführt wird. Caserta et al. [8] geben einen Überblick über unterschiedliche, darstellbare statische Informationen und auf welchen Ebenen diese betrachtet werden können. Auf Zeilen-Ebene sind die Anzahl der Zeilen, aber auch der Inhalt dieser Zeilen interessant. Die Anzahl der Zeilen, welche unterschiedliche Kontrollstrukturen wie `if`-Verzweigungen oder `while`-Schleifen enthalten kann, lassen sich beispielsweise mit *SEESoft* [9] und *sv3D* [10] darstellen.

Auf der Ebene von Methoden und Klassen sind Metriken, wie die Verbundsstärke des Auf-

rufgraphen oder Aufrufsequenzen interessant. Die Darstellung des Aufrufgraphens kann das Verständnis über die Funktionalität von einzelnen Klassen vereinfachen.

Die höchste Ebene bildet die Architekturebene. Sie beinhaltet die hierarchischen Anordnungen von objektorientierten Softwareprojekten. In dem Überblick von Caserta et al. [8] wird deutlich, dass die meisten Visualisierungstools ihren Fokus auf diese Ebene legen.

Analysen bezüglich der Anwendungsperformance können mit einer statischen Untersuchung nur eingeschränkt gemacht werden, da sich das Programm nicht ausgeführt wird. Es können also keine echten Laufzeiten gemessen werden, sondern höchstens die theoretische Komplexität der Algorithmen untersucht werden. So können theoretische Grenzen der Algorithmenlaufzeit bestimmt und bewiesen werden. Diese Arbeit beschäftigt sich jedoch explizit nicht mit der theoretischen Analyse von Algorithmen, sondern mit dem Messen von tatsächlichen Laufzeiten einzelner Programmdurchläufe.

2.1.2 Dynamische Informationen

In der statischen Analyse und bei der Erhebung von statischen Informationen über eine Anwendung, befindet sich die Anwendung selbst nicht in Ausführung. Aussagen über das tatsächliche Laufzeitverhalten können also nur bedingt getroffen werden. In der dynamischen Analyse wird die Software während der Laufzeit untersucht [2]. Analysen der Softwareperformance können auf unterschiedlichen Ebenen durchgeführt werden. Last- und Performancetests untersuchen eher die tatsächlichen Antwortzeiten, an den nach außen gerichteten Schnittstellen einer Anwendung während Profiling-Tools wiederum die Laufzeiten von einzelnen Methoden, teils sogar auf Zeilenebene [11] untersuchen. In Kapitel 2.2.2 und 2.2.3 werden die Last- und Performancetests, sowie die Profiling-Tools näher vorgestellt.

2.1.3 Visualisierung

Der folgende Abschnitt basiert auf einem von Rainer Koschke gegebenen Überblick über Software Visualisierung [4]. Es gibt eine Vielzahl an Möglichkeiten mit Software zu arbeiten. Ein großer Teil dieser Arbeitszeit wird damit verbracht Software zu verstehen [2, 12, 13]. Die reine textuelle Betrachtung von Quellcode oder Systemen kann sehr umständlich sein und viel Zeit einnehmen. Ziel der Visualisierung von Software ist, mit alternativen Darstellungsarten für ein schnelleres Verständnis der Software, oder bestimmter Eigenschaften dieser, zu sorgen. Nach einer Definition von Roman and Cox [14] kann Softwarevisualisierung als eine Abbildung von Softwarebestandteilen oder Eigenschaften auf visuelle Eigenschaften grafischer Darstellungen verstanden werden. Diese visuellen Eigenschaften können Aspekte wie Höhe, Breite, Farbe, Form, die Nähe von Elementen oder sogar animierte Eigenschaften wie Animationsgeschwindigkeit oder Animationsrichtung sein.

Für die Darstellung von Informationen, welche zunächst nicht greifbar sind, werden Metaphern verwendet, um sie auf leicht verständliche Konzepte abzubilden. Es ist jedoch nicht immer klar, ob und inwiefern grafische Darstellungen den textuellen Repräsentationen dieser Informationen überlegen sind [4]. Grund hierfür sind die vielen unterschiedlichen Darstellungen, welche verwendet werden könnten. Beispielsweise könnten sich dreidimensionale Darstellungen in speziellen Fällen besser eignen als zweidimensionale. Ebenso kann je nach Datenstruktur eine unterschiedliche Anordnung von grafisch dargestellten Graphstrukturen ausschlaggebend für die Gebrauchstauglichkeit der Visualisierung sein. Empirische Studien zeigen jedoch, dass in speziellen Fällen die grafischen Repräsentationen den textuellen überlegen sind [15].

Dies ähnelt dem No-Free-Lunch-Theorem, welches aus der Optimierung [16] bekannt ist.

Im Wesentlichen besagt es, dass immer spezielle Probleminstanzen existieren, welche von einem anderem Algorithmus besser gelöst werden können. Demnach gibt es keine Algorithmen, die für alle möglichen Eingaben am besten geeignet sind[1]. So hängt auch, nach der *match-mismatch* Hypothese von Gilmore und Green [17] die Lösbarkeit davon ab, ob das gestellte Problem zu der verwendeten Notation passt [4]. Jede Visualisierung hebt bestimmte Informationen hervor, während andere Informationen versteckt werden.

In der Softwarevisualisierung ist diese versteckte Information häufig der Quell-Code an sich, welcher durch beliebige Darstellungen versteckt oder abstrahiert wird. Jede Darstellung hat also sowohl Vorteile, als auch Nachteile.

Wichtig für die Entwicklung neuer Visualisierungen ist es also, diese in möglichst realitätsnahen Experimenten zu evaluieren.

Für die zielführende Nutzung von Visualisierungen kann eine Nutzung vieler unterschiedlicher Ansichten deshalb sinnvoll sein. Bassil und Keller [18] identifizieren hierzu folgende Punkte als essentiell für das Arbeiten mir mehreren Ansichten[2]:

- Suchwerkzeuge für grafische/textuelle Elemente,
- textuelle Darstellung des Quell-Codes,
- hierarchische Repräsentation,
- die Nutzung von Farben,
- Quellcode-Suchen,
- Navigierbare Hierarchien und
- einfacher Zugriff zu den entsprechenden Abschnitten im Quell-Code.

Hierdurch wird deutlich, dass es trotz einer guten Visualisierungen wichtig es ist, sie möglichst eng mit dem zugrundeliegenden Quellcode zu verknüpfen. In Abschnitt 2.2.1.6 wird eine Darstellungsform vorgestellt, welche diese Verbindung hervorragend umsetzt.

2.1.4 Beispiele unterschiedlicher Visualisierungen

Abhängig von den darzustellen Informationen eignen sich unterschiedliche Darstellungsformen besser, in der Softwarevisualisierung werden häufig hierarchische Darstellungen verwendet, weshalb die meisten der folgenden Beispiele hierarchische Strukturen darstellen. Neben hierarchischen Visualisierungen existieren jedoch noch weitere Formen. Aufgrund der großen Mengen an Darstellungen, welche bereits für unterschiedliche Metriken vorgestellt wurden, kan in dieser Arbeit kein vollumfänglicher Überblick gegeben werden. Für weitere Darstellungsformen wird deshalb auf die Überblicke von Caserta et al. [8], Carpendable [19], Hammad et al. [20, 21] und Koschke [4] verwiesen.

Es folgen nun einige ausgewählte Anwendungen und Formen der Softwarevisualisierung.

[1] Die Feststellung der Ähnlichkeit zu dem No-Free-Lunch Theorem ist nicht aus der Hauptquelle dieses Abschnittes, sondern wurde eigenständig festgestellt.

[2] Aufzählung übersetzt und übernommen aus dem Überblick von Rainer Koschke [4]

2.1.4.1 Diagramme

Die wohl bekanntesten Visualisierungsformen sind klassische Diagramme. Balkendiagramme, Spalten, Linien- und auch Kreisdiagramme sind offensichtliche Hilfsmittel um größere Mengen von Informationen kompakt zu visualisieren und verständlich zu machen. Sie bilden Zahlenwerte auf Aspekte wie Breite, Höhe, Positionen oder Farbe innerhalb der Darstellung ab. Insbesondere für schnellen Informationsaustausch eignen sie sich besonders gut, da die meisten Personen sie bereits kennen und verstehen. Hammad et al. [20] verweisen ihrem Überblick über Visualisierungen von Software-Klonen auf mehrere Arbeiten [22, 23, 24], welche in diesem Kontext unterschiedliche Metriken mit diesen Diagrammarten darstellen.

Klassische Diagramme auf zweidimensionalen Koordinatensystemen stellen in der Regel nur einzelne Attribute und keine Relationen oder Hierarchien zwischen Elementen dar. Sie eignen sich deshalb für die Darstellung von Relationen innerhalb der Software, insbesondere bei einer großen Informationsmenge, nur bedingt.

2.1.4.2 UML-Diagramme

Eine sehr verbreitete Darstellungsform von Softwarearchitekturen objektorientierter Projekte sind UML-Diagramme [25], da diese Verbindungen zwischen einzelnen Komponenten darstellen zu können. Diese Sammlung an vereinheitlichen Definitionen bietet unterschiedliche Darstellungen für verschiedene Sachverhalte. Klassendiagramme werden verwendet, um Beziehungen zwischen Klassen darzustellen. Sequenzdiagramme stellen Transaktionssequenzen zwischen Objekten einer Anwendung zur Laufzeit dar. *Activity Diagrams* wiederum ähneln *Flow Charts* und können für die Darstellung von Algorithmen oder Prozeduren verwendet werden.

Diese und weitere Diagramme bieten einen mächtigen Baukasten für die Planung einer Softwarearchitektur. In der Regel werden diese Diagramme vor oder während der Entwicklung manuell angelegt. Theoretisch können sie jedoch auch automatisch erzeugt werden. Hierbei treten Probleme auf, sobald zu große Informationsmengen dargestellt werden sollen, da eine automatische Anordnung nur schwer möglich ist oder der Platz für die Darstellung schlicht nicht ausreicht [8].

2.1.4.3 Graph-Visualisierung

Für die Darstellung von Relationen innerhalb von Software können häufig Visualisierungen von Graphen verwendet werden. Einzelne Elemente werden als Kreise oder Quadrate repräsentiert und mit Linien verbunden. So lassen sich insbesondere Aufrufe zwischen einzelnen Methoden oder Klassen gut darstellen. Zwischen dem *Caller*, der aufrufenden Methode oder Klasse und dem *Callee*, der Klasse oder Methode, welche aufgerufen wurde, werden Linien gezogen. Die Gesamtheit aller Aufrufkanten bildet einen vollständigen Aufrufgraphen. Die Begriffe *Caller* und *Callee* wurden aus verwandten Arbeiten [5, 26] für diese Masterarbeit in ihrer englischen Form übernommen.

Mit Anwendungen wie *GraphVIZ* [27] können einfach zweidimensionale Graphen erzeugt werden. Problematisch an dieser Darstellung ist, dass sie im Kontext der Softwarevisualisierung,

wo häufig große Strukturen dargestellt werden, schnell unübersichtlich wird [8]. Ein vollständiger Aufrufgraph zwischen allen Klassen oder Methoden einer objektorientierten Anwendung ist in vielen Fällen kaum noch erkennbar. Auch unabhängig der Softwarevisualisierung beschäftigen sich viele Bereiche mit alternativen Graphdarstellungen, welche kompaktere Darstellungen ermöglichen oder spezielle Arten von Graphen besser darstellen können [28, 29, 30].

2.1.4.4 Treemaps

Für die Darstellung von hierarchischen Graphstrukturen, wie beispielsweise einem Dateiverzeichnis oder einer Klassenhierarchie innerhalb einer Anwendung eignen sich Treemaps sehr gut. Johnson et al. [31] stellten diese Art der Darstellung 1999 vor. Treemaps bestehen aus Rechtecken oder vergleichbaren primitiven Formen unterschiedlicher Größen, welche jeweils ihre Kindelemente beinhalten. Die Dimensionen der Kindelemente entsprechen wählbaren Metriken wie der Anzahl der Code-Zeilen. Die, aus der Veröffentlichung von Johnson et al., übernommene Abbildung 2.1 zeigt eine Beispieldarstellung. Die Kindelemente entsprechen Dateien und ihre Fläche der Dateigröße. Als zweite Informationsquelle wird die Farbe der Elemente genutzt. Sie stellt hier den Dateityp dar. Die Treemaps bieten eine angenehme planare Darstellungsform, welche in dieser oder vergleichbaren Formen auch in den, in dieser Arbeit zu sehenden, Software-Städten immer wieder auftaucht.

Die Treemap ist eine gut geeignete Grundlage für die 3D-Darstellung von Software in Software-Städten, wie sie in Kapitel 2.1.7 beschrieben wird, indem sie als Grundriss der Stadt verwendet wird und in die Höhe erweitert wird. Zudem lässt sie sich gut erweitern, indem die einzelnen Elemente der Treemap mit Linien verbunden werden, um weitere Relationen der dargestellten Programmkomponenten darzustellen [28]. Diese Kombination wird insbesondere auch in dieser Masterarbeit verwendet, um Aufrufkanten zwischen Methoden darzustellen.

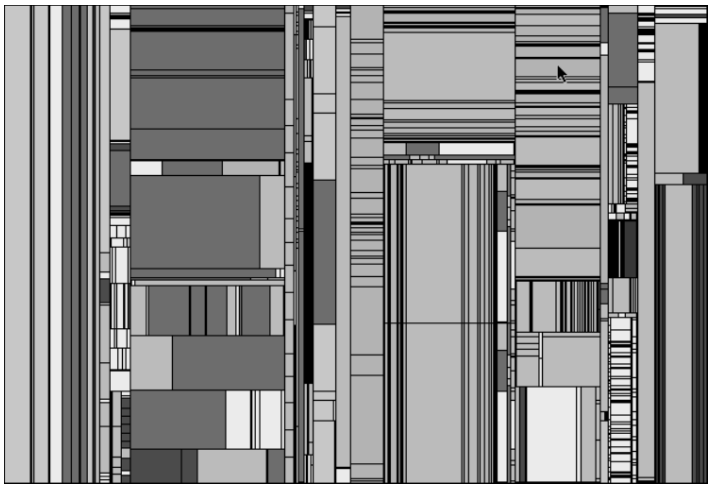


Abbildung 2.1: Beispielabbildung einer Treemap, vorgestellt von Johnson et al. [31].

2.1.4.5 Hierachical Edge Bundles

Für Graphen mit einer großen Menge an Kanten zwischen den Knoten schlägt Holten [30] *Hierachical Edge Bundles* vor. In dieser Darstellung sind die Knoten als ein Ring angeordnet, welcher zusätzlich über mehrere Ebenen hierarchisch gruppiert ist. Die Kanten verlaufen innerhalb des Rings und können auf unterschiedlichen hierarchischen Ebenen gebündelt werden. Diese Bündelung reduziert die Menge an darzustellenden Kanten und erhöht die Übersicht, versteckt hierdurch jedoch auch Detailinformationen. Der Parameter β kontrolliert, wie in Abbildung 2.2 sichtbar, die Bündelungsstärke.

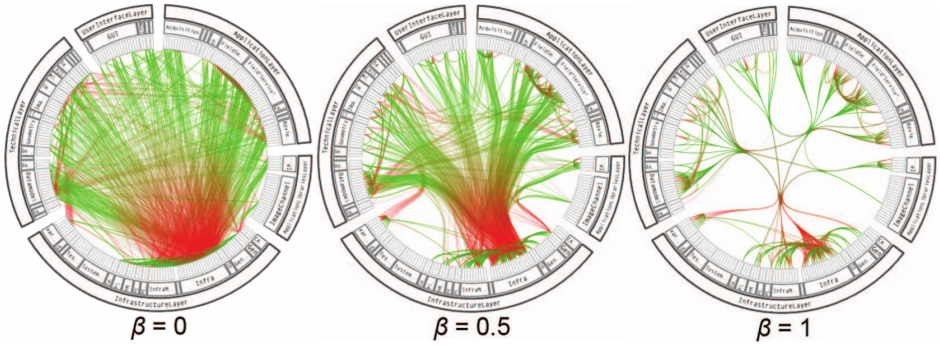


Abbildung 2.2: Hierachical Edge Bundles von Holten [30].

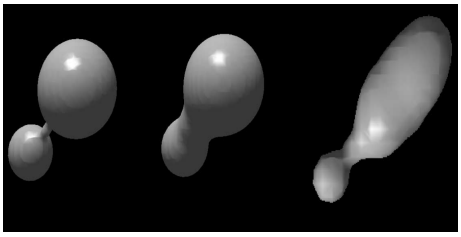
2.1.5 Verwendung von Animationen

Nach einer Umfrage von Rainer Koschke [4] von 2003, wurden Animationen früher recht selten in der Visualisierung von Software verwendet. Ein größerer Teil der befragten Wissenschaftler gab jedoch an, dass sie denken, dass Animationen insbesondere für die Darstellung von dynamischen Informationen hilfreich sein könnten. In jüngeren Arbeiten kommen Animationen mittlerweile häufiger vor. In Abschnitt 2.2.1.5 wird beispielsweise die animierte Darstellung von Ogami et al. [26] für dynamische Performance-Informationen vorgestellt.

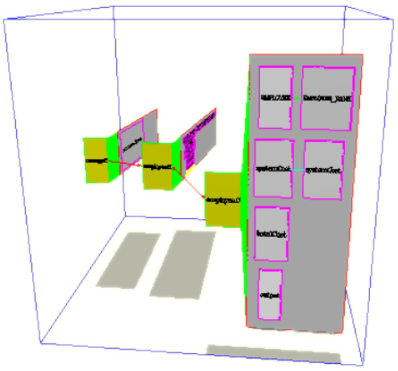
2.1.6 Verwendung von 3D

Die Verwendung von drei Dimensionen, anstelle von zweidimensionalen Darstellungen verspricht durch die dritte Dimension weitere und potentiell übersichtlichere Darstellungsformen. 3D-Visualisierungen von Software gab es bereits recht früh. Rilling und Mudur [32] zeigten 2002 die Abbildung von Software Strukturen und Metriken auf mit *3D-Metaballs*. Mit diesen Metaballs können parametrisierbare 3D-Strukturen erschaffen werden, welche unterschiedliche Aspekte von Software darstellen. Reiss [33] stellte bereits 1995 Software Strukturen mit Hilfe von geschachtelten Boxen dar, welche zweidimensionale Diagramme, um eine dritte Achse im Raum ergänzt. Diese ähneln einer dreidimensionalen Darstellung von UML-Diagrammen. Eine vergleichbare 3D-Darstellung UML-ähnlicher Diagramme wurde später auch von Savidis et al. [34] vorgestellt.

In dieser Arbeit liegt der Fokus auf einer 3D-Darstellung von Software mit Hilfe von Softwarestädten. Diese werden in Kapitel 2.1.7 näher vorgestellt.



(a) Metaballs von Rilling und Mudur [32]



(b) 3D-Diagramm von Reiss [33]

Abbildung 2.3: 3D-Darstellungen für die Softwarevisualisierung.

2.1.7 Software-Städte

In der Literatur taucht häufig die Metapher auf, Informationen in Form einer virtuellen Stadt darzustellen [35, 36, 37, 38]. Innerhalb dieser Städte repräsentieren unterschiedliche visuelle Merkmale der Häuser einzelne Metriken der dargestellten Software mit dem Ziel, sie leichter verständlich zu machen. Die Grundidee, eine Stadt als Metapher zu verwenden gehe laut Jeffery [39] auf Science-Fiction-Werke der 80iger-Jahre wie den Film *TRON* [40] zurück, in welchem bereits sehr frühe stilisierte Visualisierungen einer virtuellen Welt zu sehen waren.

Frühe Implementierungen dieser Metapher für die Darstellung von Software gehen auf Knight und Murno [36] zurück. Sie stellen mit realistisch gestalteten Gebäuden und Stadtbezirken die Struktur der Software dar. So stellt die Höhe der Häuser die Anzahl der Zeilen dar und die Farbe gibt die Sichtbarkeit (public/private/etc.) von Java-Methoden an.

Abbildung 2.4: Interaktive Software-Stadt mit Produktions-Informationen.

Panas et al. [41] schlugen 2003 ebenfalls eine 3D-Visualisierung mit realistischen Gebäuden vor. Ihre Stadt soll das aktuelle Entwicklungsgeschehen widerspiegeln. Die Visualisierung in Abbildung 2.4 markiert Häuser, deren assoziierte Komponenten aktuell in Bearbeitung sind mit gelber Farbe. Braun markiert werden Komponenten, welche lange nicht verwendet werden und möglicherweise nicht mehr benötigt werden. Die *Hot-Spots*, Komponenten, welche häufig verwendet werden, werden mit Feuer-Animationen markiert. Abbildung 2.5 zeigt eine

schlichtere Darstellung von Lanza et al. [42], welche den visuellen Stil vieler anderen Arbeiten besser repräsentiert, da diese in der Regel mit einfachen Formen und Farben arbeiten.

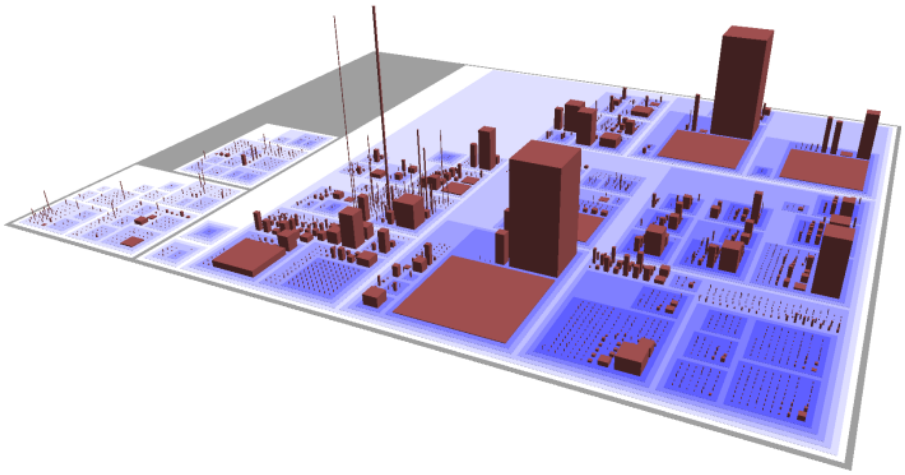


Abbildung 2.5: Typische Visualisierung einer Software-Stadt von Lanza et al. [42]

Auch in jüngerer Vergangenheit erfreut sich die Nutzung der City-Metapher immer wieder großer Beliebtheit in der Softwarevisualisierung. Balogh et al. [43] verfolgen einen spielerischen Ansatz, indem sie ihre Software-Stadt, innerhalb des Computerspiels *Minecraft*, automatisch generieren lassen. Abbildung 2.6 zeigt die Stadt einer Beispiel-Anwendung. Fittkau et al. nutzen die City-Metapher für die Darstellung von Performance-Informationen [44, 45] und verwenden moderne *Virtual Reality*-Endgeräte [46] um besser in die 3D-Darstellung eintauchen zu können. Für die Nutzung mit VR-Brillen eignet sich die 3D-Visualisierung mit der City-Metapher besonders gut und wurde deshalb den vergangenen Jahren häufig dort erprobt [47, 48, 46, 49].

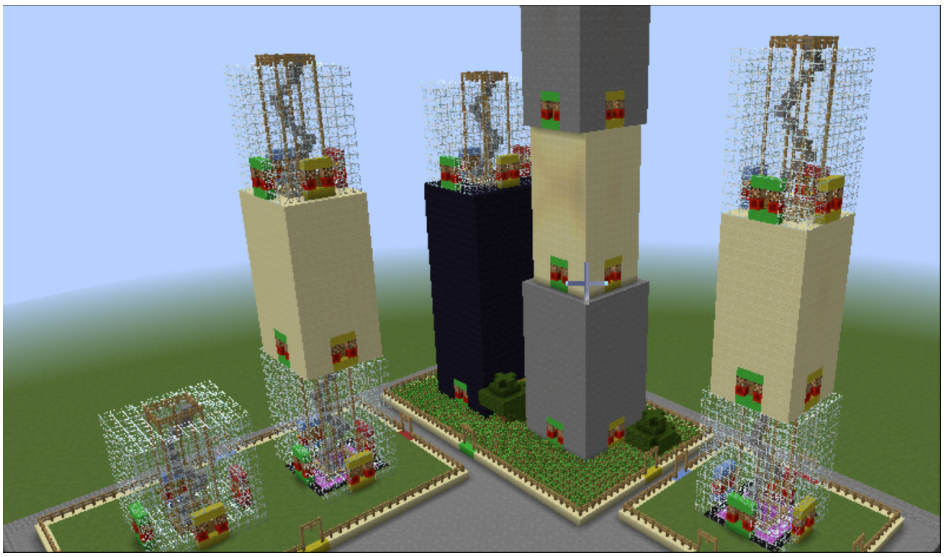


Abbildung 2.6: Visualisierung einer Software-Stadt in *Minecraft* von Balogh et al. [43]

Hierarchische Netze Während viele Software-Städte für die Darstellung der Projekthierarchie auf die dritte Dimension erweiterte Treemaps oder ähnliche planare Layouts verwenden,

stellen Balzer et al. [50, 51] eine Darstellung vor, welche ähnlich wie mit der die City-Metapher, Blöcke für die Repräsentation von Methoden verwendet. Sie verwenden bei der Darstellung der Projekthierarchie einen von oben herabhängenden, netzähnlichen Baum für die Visualisierung der Paketstruktur und Halbkugeln, welche die Methoden einer Klasse umschließenden. Die von den Halbkugeln umschlossenen Blockansammlungen können als Städte interpretiert werden.

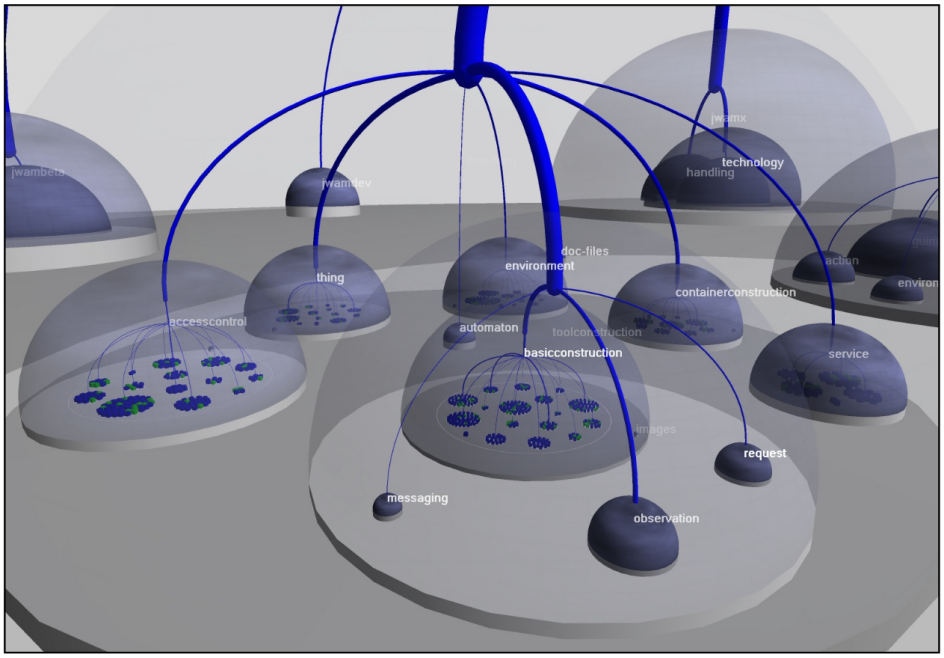


Abbildung 2.7: Hierarchisches Netz von Balzer et al. [50, 51] stellt die Software Hierarchie dar.

2.1.8 Software Engineering Experience

Die „Software Engineering Experience“, kurz SEE [52], ist eine 3D-Anwendung zur Darstellung und Analyse von Software. Die folgende Zusammenfassung basiert auf einem Video von Rainer Koschke [53]. Sie wird von der „Arbeitsgruppe Softwaretechnik“ entwickelt und wird durch studentische Arbeiten erweitert und evaluiert. Ein Teil dieser Masterarbeit ist es auch, die SEE um neue Funktionalitäten zu erweitern.

Die mit SEE analysierten Anwendungen, werden in SEE ebenfalls mit Hilfe der Stadt-Metapher dargestellt. Einzelne semantische Elemente aus der untersuchten Software können so hierarchisch dargestellt werden. Wie in den anderen Software-Städte entsprechen einzelne Häuser dieser Softwarestadt einer Klasse oder einzelnen Methoden. Für die Anordnung der Stadt bietet SEE unterschiedliche Layouts an. Das innerhalb dieser Arbeit verwendete Layout ist das Treemap-Layout. SEE kann verwendet werden, um beliebige statische Metriken wie Zeilenanzahl, Code-Komplexität oder Anzahl von Code-Duplikaten darzustellen. Um diese Metriken darzustellen, können Breite, Tiefe, Höhe und Farbe der Häuser manipuliert werden. SEE ist primär ein Visualisierungstool für beliebige Eingabewerte. Die Metriken werden durch externe Tools berechnet und über spezielle Dateiformate in SEE übertragen.

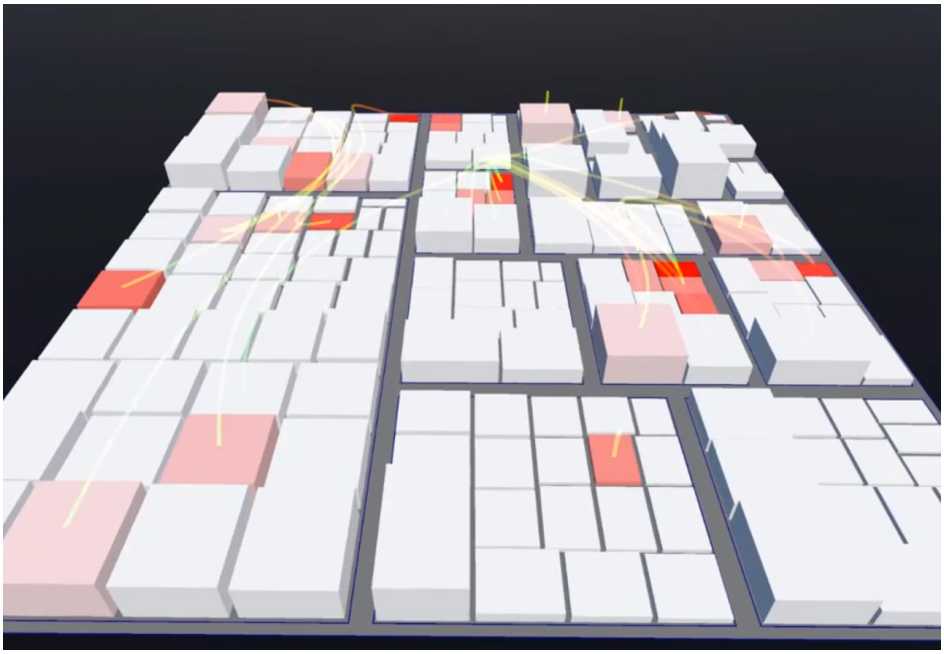


Abbildung 2.8: Darstellung des Linux Net Subsystems mit einem Treemap-Layout in SEE [54]

Hierfür nutzt SEE die *Graph Exchange Language* (GXL) [55, 56], welche ein auf XML-basierendes Format zum Austausch von Graphen ist. Neben der hierarchischen Struktur des Softwareprojektes können in diesem Format auch der Aufrufgraph und entsprechende Metriken angegeben werden.

Hierdurch ist SEE flexibel und kann beliebige Metriken auf unterschiedliche visuelle Aspekte der Softwarestadt abbilden. Dies soll vereinfachen, sich schnell zurecht zu finden, auch wenn zwischen vielen unterschiedlichen Metriken gewechselt wird. Dadurch, dass mehrere unterschiedliche Aspekte gleichzeitig belegt werden können, lassen sich unterschiedlichste Metriken in einer Darstellung kombinieren. Diese Abbildung von Werten auf visuelle Aspekte entspricht ganz der im Abschnitt 2.1.3 besprochenen Definition von Roman und Cox [14].

Eine weitere wichtige Anwendung von SEE ist die Architekturanalyse. Es ist möglich entworfene Architekturplanungen mit der tatsächlichen Implementierung zu vergleichen. Hierbei werden Methodenaufrufe innerhalb der einzelnen Softwareteile verglichen und es werden Konvergenzen, Absenzen und Divergenzen identifiziert. Konvergenzen sind Aufrufe, welche der geplanten Architektur entsprechen während Divergenzen ihr widersprechen. Absenzen sind Aufrufe, welche in der Architektur geplant sind, jedoch nicht in der Implementierung auftauchen [53].

Durch die Einbindung von einem Versionskontrollsystem lässt sich die Entwicklung der Software, über mehrere Versionen hinweg, in einer Animation darstellen [54]. Geplant sind tiefergreifende Einbindungen von Versionskontrollsystemen [53]. Abbildung 2.8 zeigt einen Ausschnitt aus einem Beispielvideo [54] zu dieser Animation.

Die Grundlage von SEE ist zwar die Darstellung statischer Metriken, jedoch sind auch dynamische Analysen sehr gut in den 3D-Städten abbildbar. *Programm-Traces*, also aufgezeichnete Programmabläufe, können für Debugging-Zwecke innerhalb von SEE dargestellt werden. Hierfür lassen sich einzelne Programmabläufe für Debugging-Zwecke sequentiell abspielen. Hierbei sind sowohl Vorwärts-, als auch Rückwärtsschritte möglich [57]. Auch an der Einblendung und der Editierbarkeit des Quellcodes direkt innerhalb von SEE wird gearbeitet. SEE entwickelt

sich also von einer Visualisierungsplattform zu einer Entwicklungsplattform weiter.

In Abbildung 2.8 zeigt ebenfalls Kanten, welche einzelne Blöcke miteinander verbinden. Diese Kanten basieren auf *TinySpline* [58] und können in SEE für die Darstellung des Aufrufgraphen verwendet werden. Sie unterstützt ebenfalls das Bündeln mehrerer Kanten. Innerhalb dieser Masterarbeit wurde diese Kantenrepräsentation jedoch nicht verwendet, sondern eine alternative Darstellung implementiert. Weiteres hierzu in Abschnitt 3.5.7.3.

Die Anwendung unterstützt zudem die kollaborative Nutzung. So ist es möglich mit mehreren Personen eines Softwareenticklungungsprojektes gemeinsam die Visualisierungen zu betrachten, ohne sich an einem physischen Ort zu treffen. Die Personen können in der Anwendung durch 3D-Avatare dargestellt werden. Zukünftig wird der Forschungsschwerpunkt in großen Teilen in der kollaborativen Nutzung des Systems liegen.

Die Nutzung von unterschiedlichsten Eingabegeräten bietet hierfür größtmögliche Flexibilität. Die Anwendung kann klassich an einem Desktop-Computer mit Maus und Tastatur bedient werden. Neue Technologien wie Virtual-Reality-Brillen, Augmented-Reality-Brillen, Gestensteuerung und Touch-Bildschirme werden jedoch ebenso unterstützt.

SEE wurde in der 3D-Game-Engine *Unity* [59] implementiert. Unity bietet in seiner Entwicklungsumgebung sehr viele Funktionen, um Einstellungen für die entwickelte *Szene* zu verwalten. SEE wird deshalb zum aktuellen Stand noch in großen Teilen direkt aus der Entwicklungsumgebung von *Unity* heraus verwendet und Einstellungen für die dargestellten Softwarestädte werden direkt in der Unity-Oberfläche vorgenommen.

2.2 Softwareperformance

In dieser Arbeit soll SEE um Visualisierungen von Softwareperformance erweitert werden. Hierfür werden in den folgenden Abschnitten zunächst verwandte Arbeiten für die Visualisierung von Performancedaten, insbesondere in Softwarestädten, betrachtet und danach die Grundlagen für die Erhebung dieser Performancedaten beschrieben.

Unter Performance-Informationen können neben der Laufzeitperformance, also der zeitlichen Dauer, die einzelne Komponenten für ihre Ausführung benötigen auch Metriken wie Speicherverbrauch, Dead-Locks, Benutzererfahrungen, Bildwiederholfrequenz in Computerspielen, Energieffizienz oder ähnliches verstanden werden. In dieser Arbeit wird sich ausschließlich mit der Laufzeit-Dauer einzelner Komponenten beschäftigt.

2.2.1 Visualisierung von Software-Performance

Bei der Visualisierung von Softwareperformance oder generellem Softwareverhalten kann unterschieden werden zwischen mehreren Herangehensweisen. Mit kompletten *Execution Traces* ist es möglich, den gesamten aufgezeichneten Ablauf einer Anwendung darzustellen. Für eine kompaktere Darstellung können die großen Mengen zusammengefasst werden, indem Durchschnittwerte oder die Gesamtaufrufzahl einzelner Methoden ermittelt und dargestellt wird.

2.2.1.1 Textuelle Darstellung in Profilern

Profiler sind Werkzeuge zum Erheben von Performance-Informationen, sie untersuchen Anwendungen zur Laufzeit und ermitteln Informationen, welche Methoden am häufigsten aufgerufen werden, oder insbesondere wie hoch die Methodenlaufzeiten sind. Sie funktionieren üblicherweise mit Sampling-Methoden oder durch Instrumentierung des Programmcodes. Abschnitt 2.2.2 beschreibt die Technologien näher. In klassischen Profilern finden sich häufig textuelle Darstellungen in Tabellen mit Sortier- und Filterfunktionen. Die Tabellen listen alle untersuchten Methoden und ihre durchschnittliche Laufzeit oder Laufzeitanteile. Die hierarchische Struktur des Aufrufgraphen wird zudem durch aufklappbare Zeilen dargestellt. VisualVM [60] stellt zudem in Form von kleinen Balken-Grafiken das Verhältnis zwischen den einzelnen Methoden grafisch dar. Klassische Profiler werden dafür kritisiert, dass sie im Vergleich zu grafischen Darstellungen, gerade in komplexeren Szenarien, unterlegen seien [26].

VisualVM taucht im Verlauf dieser Masterarbeit in Abschnitt 4.3 erneut auf, da die Anwendung dort als klassische Darstellungsform mit der Implementierung in dieser Arbeit verglichen wird.

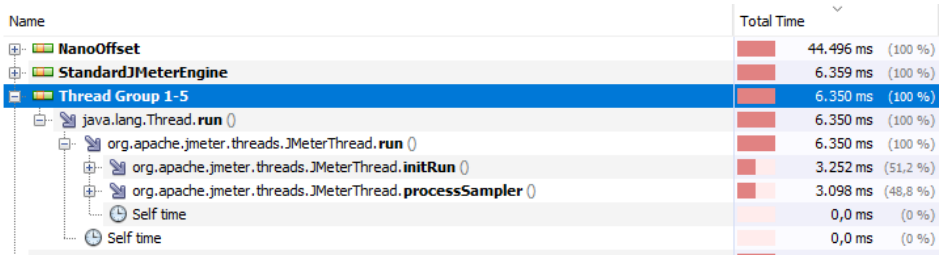


Abbildung 2.9: Darstellung der gemessenen Profiling-Informationen in *VisualVM* [60]

2.2.1.2 Flame-Graphs

Flame-Graphen sind eine, für die Untersuchung der Performance, beliebte Darstellungsform [61, 62]. Sie geben für einzelne Methoden an, wie hoch ihr Laufzeitanteil während der Laufzeit einer Anwendung war. Die in Abbildung 2.10 gezeigte vereinfachte Darstellung eines Flame-Graphens von Bezemer et al. [61] zeigt für die Methoden `a`, `b`, `c`, `c`, in welchen Methoden die Anwendung sich während des Untersuchungszeitraums aufgehalten hat. Die dargestellte Hierarchie entspricht dem jeweiligen Aufruf-Stack, über welchen die Methode aufgerufen wurde.

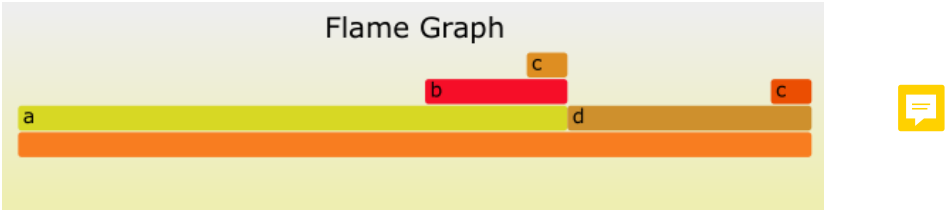


Abbildung 2.10: Flame Graph Darstellung von Bezemer et al. [61]

2.2.1.3 Extravis

UML-Sequenzdiagramme oder Baumstrukturen werden schnell zu komplex, wenn sie verwendet werden um komplette *Execution Traces* visuell darzustellen. Cornelissen et al. [29, 63] schlagen mit *Extravis* zwei interaktive Visualisierungen vor, welche eine kompaktere Darstellung versprechen: Die *Massive Sequence View* in Abbildung 2.11 stellt die gesamte Aufrufsequenz, nach einzelnen Paketen gruppiert an. Die *Circular Bundle View* in Abbildung 2.12 fasst die Aufrufkanten zwischen einzelnen Methoden zu einzelnen Kanten zusammen, welche je nach Aufrufhäufigkeit auf einer Grün-Rot-Skala eingefärbt werden. Sie verwenden für diese Darstellung die in Abschnitt 2.1.4.5 vorgestellten *Hierachical Edge Bundles* von Holten et al. [30]. Hier ist jedoch nicht die Laufzeit der einzelnen Methoden der Fokus, auch wenn sich die Darstellung ebenso dafür eignen würde. Es sind in diesen Ansichten dennoch *Hot-Spots* der Anwendung erkennbar.

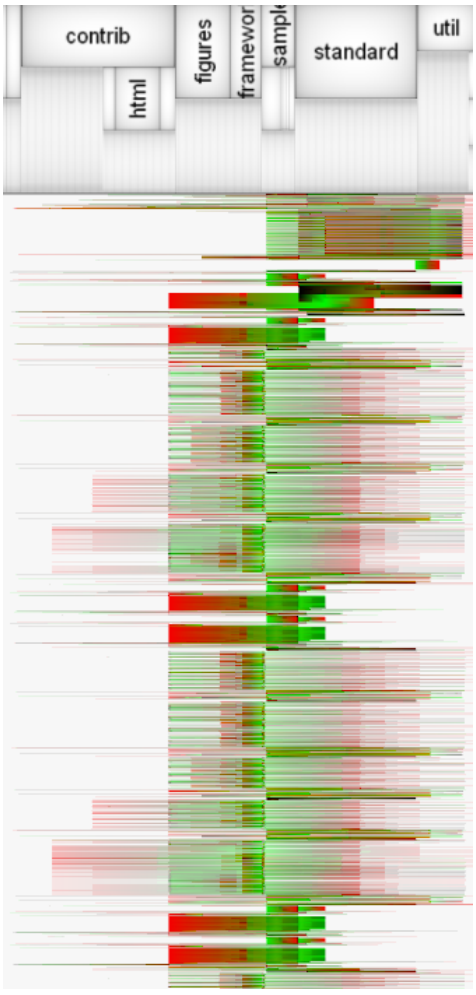


Abbildung 2.11: *Massive Sequence View* aus *Extravis* [29]

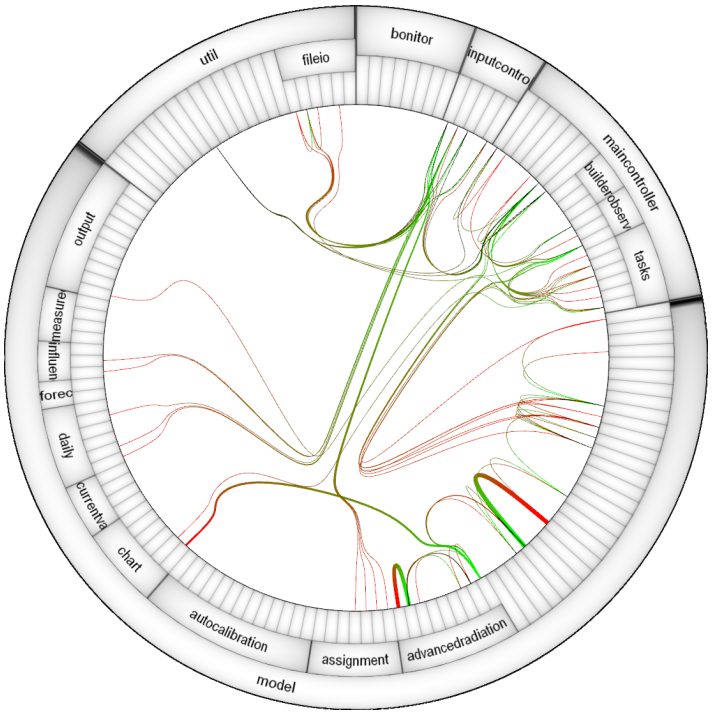


Abbildung 2.12: *Circular Bundle View* aus *Extravis* [29]

2.2.1.4 Laufzeitverhalten in Software-Städten

Fittkau et al. [44, 45] beschäftigen sich ebenfalls mit der Darstellung von *execution traces*, jedoch mit Hilfe der City-Metapher. In ihrer Software *ExplorViz* stellen einzelne Blöcke jeweils einzelne Komponenten der Anwendung dar (Klassen, Pakete, Methoden). Diese werden durch orangefarbene Linien miteinander verbunden, welche die Aufruffrequenz oder die Aufruflaufzeiten einzelner Methoden repräsentieren. Die Höhe der Klassen-Blöcke, stellt zudem die Anzahl der Objekt-Instanzen dar. Die verwendete Darstellung ist eine modifizierte Version des Layouts der *CodeCity* von Wettel [64]. ExplorViz bietet zudem Funktionalitäten wie eine periodisch aktualisierte Live-Ansicht und eine *Time-Shift*-Funktion, mit welcher über eine Zeitachse bestimmte Zeitpunkte ausgewählt werden können.

Für einen Vergleich mit einer zweidimensionalen Darstellung verglichen sie *Extravis* mit ihrer eigenen Software *ExplorViz*. In kontrollierten Experimenten stellten sie Aufgaben zum Systemverständnis. Die Experimente beinhalteten Aufgaben wie:

- Benennen von Klassen mit vielen eingehenden Kommunikationen.
- Finden bestimmter Konstruktur-Aufrufe.
- Verstehen und Auflisten von Hauptschritten, welche von der untersuchten Anwendung durchlaufen werden.

Sie fanden keine erheblichen Unterschiede in den Bearbeitungszeiten der Aufgaben. Die Korrektheit der gelösten Aufgaben innerhalb ihrer Anwendung *ExplorViz* sei jedoch signifikant höher als im Vergleichsobjekt *Extravis* gewesen.

Die Studie hatte zwar den Bezug zum Systemverständnis, *Explorviz* unterstützt jedoch auch die Darstellung von Methodenlaufzeiten für die Performanceanalyse [65]. Für die Live-Ansicht innerhalb ihrer Anwendung haben sie zudem eine Methode zum Erheben großer Mengen an Messdaten mit hohem Durchsatz entwickelt [66, 67].

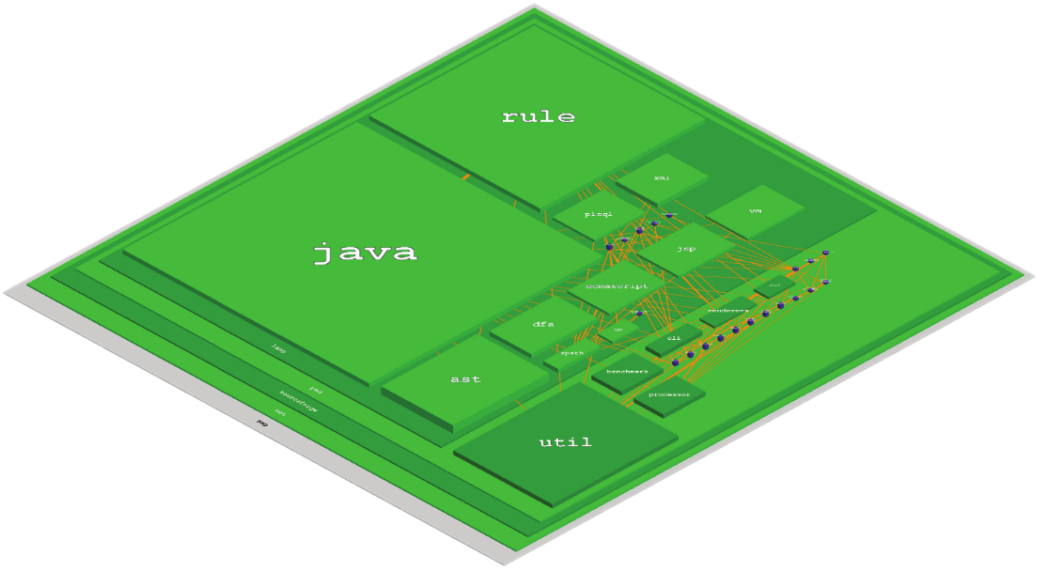


Abbildung 2.13: Darstellung eines *execution trace* in *ExplorViz*. Die Abbildung zeigt die Aktivitäten zwischen einzelnen Java-Paketen. (Markierungen im Originalbild von Fittkau [44] wurden retuschiert.)

2.2.1.5 3D Darstellung dynamischer Performancedaten

Ogami et. al [26] stellen in ihrer Arbeit eine Visualisierung vor, welche mit der in dieser Masterarbeit entwickelten Darstellung vergleichbar ist. Sie erheben mit Hilfe von Bytecode-Instrumentierung Performance-Informationen und stellen diese in einer Softwarestadt dar. Ihre Softwarestadt besteht aus Häusern, welche Methoden repräsentieren. Sie legen großen Fokus darauf, dass die profilierten Daten in einer Live-Darstellung bereits während des Profiling-Zeitraums dargestellt werden. Abbildung 2.14 zeigt diese Darstellung. Die Höhe der einzelnen Häuser verändert sich also kontinuierlich, abhängig davon, welche Methoden zum aktuellen Zeitpunkt den höchsten Einfluss auf die Programm-Performanz haben. Die Höhe entspricht der Summe aller Aufrufe der entsprechenden Methode im Verhältnis zu dem betrachteten aktuellen Zeitintervall, welcher in der Live-Ansicht der unmittelbar vergangener Zeitraum ist. Die Darstellung der Methoden, geordnet nach Klassenzugehörigkeit soll laut den Autoren der in dieser Arbeit eine leichtere Verständlichkeit des Kontextes ermöglichen, was in klassischen textuellen Darstellungen wie in VisualVM weniger gegeben sei.

Auf der Idee, den Kontext deutlicher zu machen, soll in dieser Masterarbeit ebenfalls aufgebaut werden und der Aufrufkontext durch das Darstellen von Aufrufkanten verstärkt werden, was in der Arbeit von Ogami et al. nicht getan wurde. Mehr hierzu in Kapitel 3.

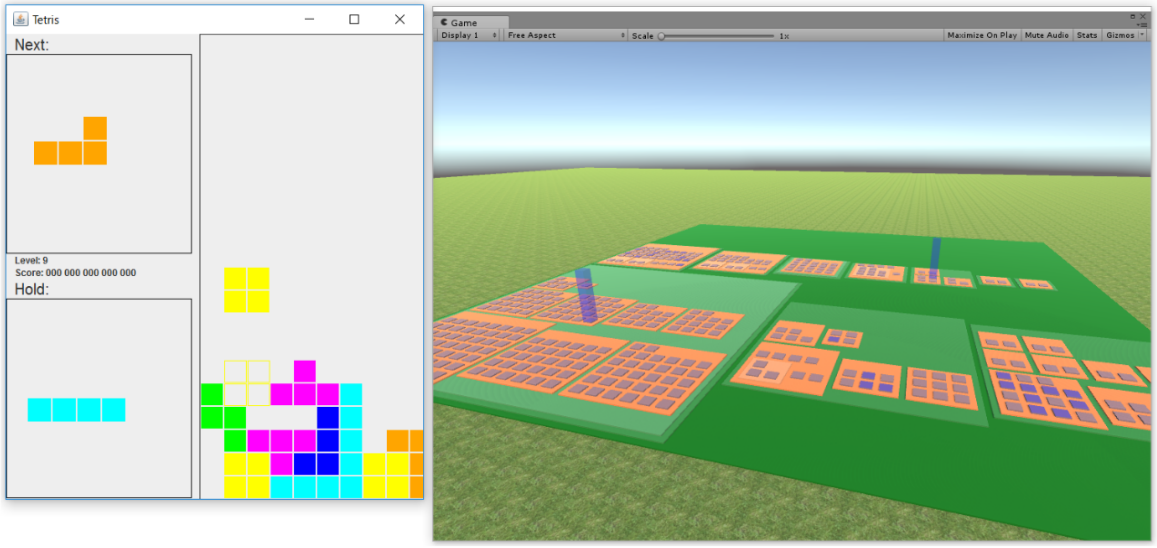


Abbildung 2.14: Live-Ansicht der Profiling-Ergebnisse während der Analyse einer Java-Implementierung von *Tetris* von Ogami et al. [26]

2.2.1.6 Visualisierung im Quell-Text

Beck et al. [5] stellen ein Konzept und ein Plugin für die etablierte Entwicklungsumgebung *Eclipse* vor, welches Performance-relevante Informationen direkt im Quellcode darstellt. Hierfür werden kleine Diagramme in einzelne Programmzeilen integriert. Sie versuchen hiermit zu vermeiden, dass an mehreren Orten zugleich Informationen präsentiert werden, was zu einer Zerstreuung der Aufmerksamkeit führen könne. Diese stark integrierte Darstellungsform nennen sie *in-situ vizualization* [5, 68]. Sie erheben unterschiedliche Metriken: Die Methodenlaufzeit eines einzelnen Methodenaufrufes und die *method time*, welche die Summe mehrerer Methodenaufrufe der selben Methode sind. Zudem unterscheiden sie zwischen der *self time* einer Methode und der *invocation time*, welche die Aufrufzeit von externen Methoden aus der be-

trachteten Methode heraus beschreibt. Diese Metriken werden, wie in Abbildung 2.15 sichtbar, als kleine Grafiken dargestellt. In einer kleinen Benutzerstudie überprüften sie die Benutzbarkeit und Effektivität dieser Darstellungsart. Vier Testpersonen haben das Plugin verwendet, um ihre eigene Software zu optimieren. Hierbei wurden Performance-Verbesserungen von 1,5% bis 40.6% erzielt. Die Interviews mit den Testpersonen ergaben zudem, dass *Hot-Spot*-Listen zwar auch alle relevanten Informationen bieten, jedoch der Kontext nicht ersichtlich sei. Der Quelltext müsse in traditionellen Vorgehen manuell mit den Profiling-Ergebnissen verknüpft werden.

Diese Erkenntnis wirkt sich insofern auf diese Masterarbeit aus, dass in der Visualisierung dieser Arbeit ebenfalls eine enge Koppelung an den Quellcode vorhanden ist und Visualisierung der *Hot-Spots* im Quell-Code implementiert wurde.

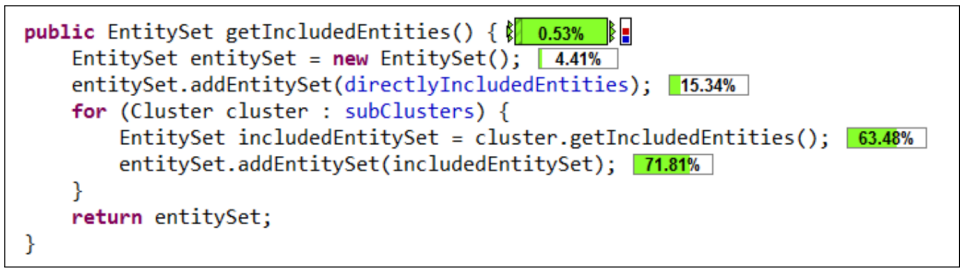


Abbildung 2.15: Miniatur-Diagramme integriert in den Quelltext [5]

2.2.1.7 Application Performance Monitoring

„Application Performance Monitoring“ (APM) Anwendungen kombinieren unterschiedliche Darstellungen von verschiedenen in Produktion relevanten Metriken und stellen diese für eine Dauerüberwachung von Systemen in Übersichten zusammen [69]. Dargestellte Informationen reichen vom Antwortzeitverhalten von Webseitenaufrufen, Profilinginformationen und Speicherüberwachung bis zu operativen Informationen wie beispielsweise Übersichten über die Anzahl angemeldeter Benutzer. In diesem Bereich gibt es eine Vielzahl kommerzieller Anwendungen wie *New Relic* [70] oder *Dynatrace* [71].

2.2.2 Profiling

Für ein tieferes Verständnis des Anwendungsverhaltens gibt eine Vielzahl an Tools und Darstellungsformen. *Profiler* werden für viele Arte von Software verwendet, um während der Laufzeit Analysen zu erstellen und *Hot-Spots* (die am häufigsten aktiven Stellen des Programms) oder die tatsächliche Laufzeit von Methodenaufrufen ermitteln [72].

Profiler können unterschiedliche Metriken erfassen. Die häufigsten sind das Messen von Laufzeiten von einzelnen Methoden oder Funktionen. Analysen der Speichernutzung sind jedoch in einigen Profilern ebenfalls möglich [60]. Der Fokus dieser Arbeit liegt auf der Messung von Laufzeiten, weshalb nur Profiler betrachtet werden, welche die Zeit messen, welche den Laufzeitverbrauch messen.

Der *observer effect* [2, 72], beschreibt das Phänomen, dass beobachtete Software sich anders Verhalten kann, als würde sie normal betrieben werden [2]. Profiling kann beispielsweise zu Timing-Problemen in Anwendungen mit mehreren Threads führen [2, 73]. Der Grund hierfür ist, dass die Tools, welche für die Untersuchungen genutzt werden auf unterschiedliche Weisen in das Laufzeitverhalten eingreifen. Diese Seiteneffekte müssen beachtet und minimiert werden.

Es existieren im Wesentlichen zwei Haupttypen von Profilern [72, 74]. Sampling-Profiler tasten die untersuchte Anwendung in Intervallen ab und sammeln Stichproben des Systemzustands, ohne die Anwendung zu verändern. Instrumentation-Profiler instrumentieren die Anwendung, sodass zusätzliche Daten gemessen werden können. Der Profiler *VisualVM* ist Bestandteil des Java Developement Kit und unterstützt sowohl instrumentiertes Profiling, als auch Sampling-Profiling. Es werden in dieser Arbeit die englischen Bezeichnungen für Sampling, Profiler verwendet.

2.2.2.1 Instrumentation-Profiler

Instrumentation-Profiler, verändern aktiv das untersuchte Programm und ergänzen es um Logging-Prozeduren. So können Zähler eingebaut werden, welche die Anzahl von Methodenaufrufe mitzählen. Zum Messen der Laufzeit einzelner Methoden werden an Ein- und Austrittspunkten der Methoden Messpunkte eingebaut. Vergleichbare Instrumentierungen gibt es auch für weitere Analysen wie Speichernutzung, oder die Anzahl instanziierter Objekte. Insbesondere die Java Virtual Machine bietet hier entsprechende Schnittstellen, um zur Laufzeit der Anwendung Daten abzufragen [75]. Das tiefe Eingreifen in den Programmcode kann zu einem starken *Observer-Effect* [4] führen. Das Hinzufügen von zusätzlichen Routinen verlängert die gemessenen Zeiten wodurch die Messungen von der Art der Instrumentierung abhängen. Die tatsächlichen Laufzeiten können deshalb nur annähernd bestimmt werden [72]. Weitere Seiteneffekte können durch die veränderte Speichernutzung des Gesamtsystems auftreten [72]. Auch Kompileroptimierungen oder die dynamische Optimierung der JVM während der Laufzeit können stark negativ beeinflusst werden, dadurch, dass zusätzlicher Profiling-Code ausgeführt wird [72].

Für die Instrumentierung von Java-Programmen existiert die statische und die dynamische Instrumentierung [74]. Die statische Instrumentierung instrumentiert das Programm vor der Kompilierung und dem Programmstart. Die einfachste Form dieser Instrumentierung wäre das manuelle Einbauen von Funktionen zum Messen der Laufzeit. Üblicherweise wird dies jedoch durch den Profiler automatisiert. Dynamische Instrumentierung benötigt keinen Zugriff auf den Originalquellcode und führt vor der Kompilierung keine Instrumentierung durch. Mit der *Java Virtual Machine* ist eine nachträgliche Instrumentierung des kompilierten Bytecodes

möglich, wodurch ein dynamischer Instrumentation-Profiler möglich ist [75]. Dies ermöglicht eine Instrumentierung der Anwendung während ihrer Laufzeit, ohne, dass die Anwendung vor dem Start verändert werden muss. Eine Untersuchung eines lange laufenden Programms, ohne es anhalten oder neu starten zu müssen, wird so möglich.

2.2.2.2 Sampling Profiler

Die weniger eingreifende Methode ist das *Sampling*. Hierbei wird in kurzen Intervallen die aktuelle *Stack-Trace* der beobachteten Threads abgetastet. So lassen sich für jeden einzelnen Thread die Stellen identifizieren, an welchen der Prozess sich aktuell befindet. Im Falle von Java lassen sich Stack-Traces leicht über *jstack* [76] ausgeben.

Sampling-Profiler nutzen diese dicht beieinander liegenden Stichproben (Samples), um eine Approximation der Auslastung von einzelnen Programmkomponenten zu berechnen, ohne die Anwendung instrumentieren zu müssen. Auf diese Weise können *Hotspots* berechnet werden. Dies sind die Stellen des Programmes, an deren Positionen sich das Programm während dem Messzeitraum am häufigsten befindet. Da die Stack-Traces auch die Informationen zu den Methoden enthalten, welche die aktuelle Methode aufgerufen haben ist jeweils der Aufrufkontext verfügbar. Dieser kann genutzt werden um Darstellungen die Flame-Graphs aus Abschnitt 2.2.1.2 zu erstellen [61].

Je dichter die Sampler abtasten, desto präziser werden die erhobenen Daten. Bei einem Messintervall von 200ms existiert beispielsweise, für schnell laufende Methoden, das Potential vom Sampling-Profiler nie gesehen zu werden. Für die Identifizierung von Hotspots ist diese Unzulänglichkeit jedoch vernachlässigbar, denn das Programm hält sich zeitlich nur sehr kurz innerhalb dieser Methode auf, weshalb sie nur geringes Potential zur Performance-Optimierung bieten. Eine zu schnelle Abtastrate führt, dadurch dass das Programm immer wieder kurz pausiert werden muss, ähnlich wie instrumentierende Profiler, zu einem Overhead. Mytkowicz et al. [72] zeigen für mehrere Profiler, dass bei einer Abtastrate von 10ms ein Overhead von circa 20% zu erwarten ist. Dieser Intervall ist üblicherweise jedoch konfigurierbar. Über einen längeren Beobachtungszeitraum reichen durch die höhere Anzahl an Samples möglicherweise größere Abtastintervalle aus, sodass der Overhead gering gehalten werden kann. Auch das Laufzeitverhalten der Anwendung kann, durch die Abtastung beeinflusst werden.

Mytkowicz et al. [72] stellen zudem fest, dass viele Java-Sampling-Profiler sich gegenseitig in ihren Ergebnissen widersprechen. Als Gründe benennen sie, dass der Samplinintervall nicht zufällig, sondern fix ist. Sie schlagen deshalb vor eine zufällig variierende Samplingrate innerhalb eines festgelegten Intervalls zu verwenden um Synchronisierungen mit dem *Thread-Scheduler* zu vermeiden. Ein Weiteres von ihnen identifiziertes Problem sind *yield-points*. Sie werden in anderen Quellen auch als *Safepoints* bezeichnet [77] und sind Positionen im Programm, an denen es „sicher“ ist, den *Garbage Collector* auszuführen. Die JVM kann *yield-points* unter Umständen überspringen. Hierdurch würde der Sampling-Profiler erst zu einem späteren Zeitpunkt pausieren und ein Sample an der falschen Stelle sammeln.

Der Nachteil von *Sampling* ist, dass die Anzahl der Methodenaufrufe und ebenso die genaue Laufzeit der Methoden nicht bekannt ist. Es kann lediglich die gesamte CPU-Zeit, welche über mehrere Ausführungen der Methode hinweg benötigt wurde approximiert werden. Für eine exakte Messung der Laufzeit einzelner Methodendurchläufe wird eine andere Messmethode benötigt.

Der Profiler *YourKIT* [78] ergänzt das herkömmlichen Sampling-Profiling um eine reduzierte Form der Instrumentierung, bei welcher lediglich die Anzahl Funktionsaufrufe gezählt werden, jedoch nicht die Laufzeit gemessen wird [72].

Da im Rahmen dieser Arbeit der Fokus der Performance-Analysen im Bereich Last- und Performancetests liegt, welche üblicherweise auf gegen langlaufende Multi-User Systemen durchgeführt werden, erscheinen Sampling-Profiler zunächst als die sinnvollere Wahl, da sie einen geringeren Overhead haben. Die Messung der exakten Laufzeiten ist jedoch ebenso Wertvoll, wenn diesbezüglich nicht-funktionale Spezifikationen nachgewiesen werden sollen. Diese könnten beispielsweise wie folgt lauten: „99% der Anfragen müssen schneller als in Sekunde beantwortet werden.“ Für den Nachweis dieser Anforderung wäre ein Messen der tatsächlichen Laufzeiten notwendig.

2.2.2.3 Self-Times

Beim Messen von Methodenlaufzeiten kann unterschiedlich vorgegangen werden. Zum einen kann die echte CPU-Zeit ermittelt werden, welche die Methoden insgesamt verbrauchen. Das heißt die Laufzeit, abzüglich der Zeit in welcher der CPU-Scheduler den Thread der Methode pausiert oder während der Thread im Wartezustand ist. Nicht alle Profiler können diese Zustände herausfiltern. Zudem wäre dies auch nicht zwingend zielführend für eine Analyse der Performance, da Probleme gerade durch blockierte Threads oder ähnliches auftreten können. Die *total time* einer Methode kann also echte verbrauchte CPU-Zeit oder die echte vergangene Zeit von Anfang bis Ende der Methode gemessen werden.

Eine weitere Unterscheidung kann zwischen *total time* (oder *method time*) und *self time* gemacht werden. Von Profilern, wie beispielsweise *VisualVM* und *Rizel* [79] und in der Arbeit von Beck et al. [5] wird diese Unterscheidung gemacht. Die *self time* beinhaltet nur die Zeit, welche tatsächlich innerhalb der Methode verbracht wird, abzüglich der Zeit, welche für Methodenaufrufe innerhalb der Methode benötigt wird. Beck et al. [5] definieren die *self time* wie in Abbildung 2.16 illustriert. Wenn die Laufzeit von nicht-instrumentierten Methoden nicht ermittelt werden kann, dann wird sie üblicherweise zu der *method time* der aufrufenden Methode gezählt.

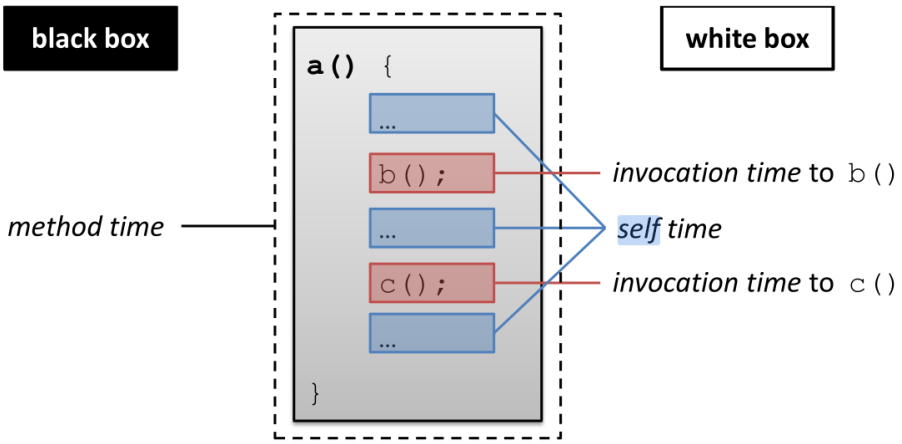


Abbildung 2.16: Veranschaulichung der *self time* von Beck et al. [5]

2.2.2.4 Performance Evolution Blueprint

Bergel et al. [79] stellen in ihrer Arbeit fest, dass klassische Profiler wie *JProfiler* [80] und *YourKIT* [78] nur bedingt geeignet sind um unterschiedliche Profiling-Ergebnisse miteinander zu vergleichen. Mit den genannten Profilern können die Ergebnisse von zwei untersuchten

Zeiträumen dargestellt werden. Es werden dann die Differenzen zwischen einzelnen Methoden dargestellt. Dies kann verwendet werden, um zwei Softwareversionen miteinander zu vergleichen. Alternativ können auch zwei unterschiedliche Benchmarks oder Benutzungsszenarien auf der selben Softwareversion verglichen werden. Als Nachteil dieser Darstellung benennen sie, dass die Profiler nicht in Betracht ziehen, welche Stellen im Quellcode verändert wurden und welche nicht. Durch das Fehlen dieser Information könnten unveränderte Methoden zu starken Fokus auf sich ziehen, da diese eine hohe Varianz in ihren Laufzeiten haben. Weiterhin kritisieren sie die reine textuelle Darstellung, da sie für die Erkennung von Mustern laut Tufte [81] nur bedingt geeignet seien.

Sie schlagen deshalb eine visuelle Darstellung vor, in welcher für jede profilierte Methode mit einer von sechs unterschiedlichen Farben markiert wird, je nachdem ob die Methode verändert wurde (dunklerer Farbton) oder nicht (hellerer Farbton) und ob sie schneller (grün) oder langsamer (rot) als die Referenz ist. Methoden, welche nur in einem der beiden verglichenen Profile auftauchen, werden gelb oder schwarz dargestellt. In Abbildung 2.17 ist ihre visuelle Baum-Darstellung mit diesen Farben dargestellt. Der Baum entspricht dem Aufrufbaum einzelner Methoden, wie er auch in den textuellen Profilern dargestellt wird. Wie in der Grafik angegeben entspricht die Höhe der Laufzeit und die Breite der Elemente entspricht der Anzahl der Aufrufe.

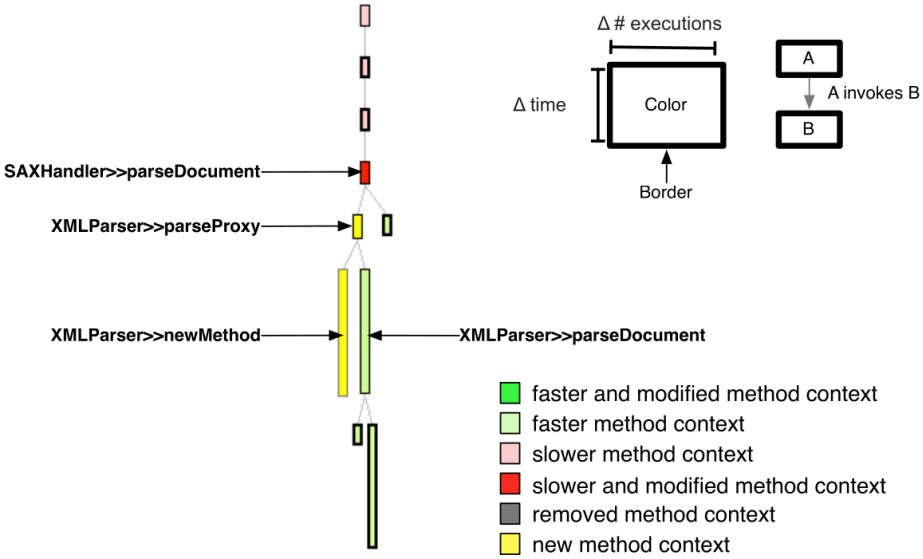


Abbildung 2.17: Darstellung eines *Performance Evolution Blueprints* von Bergel et al. [79].

2.2.3 Last- und Performancetests

Der folgende Text basiert primär auf einem Überblick über Last- und Performancetests von Jiang et al. [1] und eigenen Erfahrungen in meinem aktuellen Beschäftigungsverhältnis als Softwaretester bei *Dataport AöR* [82].

Last- und Performancetests werden durchgeführt, um unterschiedliche Performanceeigenschaften von Anwendungen zu messen [1]. Häufig werden sie im Bereich von Web-Anwendungen durchgeführt. Hauptmesswert ist in der Regel die Antwortzeit von unterschiedlichen HTTP-Aufrufen. Weitere untersuchte Systemwerte können jedoch auch Speicherverbrauch, CPU-Auslastung oder Netzwerkauslastung sein. Da diese Anwendungen nicht nur von einem einzigen Benutzer, sondern parallel von mehreren gleichzeitigen Benutzern genutzt werden, genügen händisch durchgeführte Tests von einzelnen Funktionen und manuelles Messen der Antwortzeiten nicht mehr. In solchen System müssen nicht nur Antwortzeitkriterien einzelner Funktionalitäten erfüllt werden, sondern es muss geprüft werden, wie sich die Antwortzeiten der Anwendung in unterschiedlichen Szenarien verhalten. Hierfür werden Last-Profile [1] für unterschiedliche Benutzerverhalten erstellt und automatisiert, welche dann für einen Test in unterschiedlich großen Mengen parallel durchgeführt werden. Es gibt eine Vielzahl von Test-Tools und Frameworks, welche diese Art von Test ermöglichen. In dieser Arbeit wurde *JMeter* [83] verwendet. Bei meinem Arbeitgeber wird aktuell ein, in *Visual Studio* integriertes, Framework [84] und das sehr junge Tool *K6* [85] verwendet.

Es gibt drei Kategorien von Tests in diesem Bereich. Die Grenzen zwischen diesen drei Kategorien können verschwimmen und in der Regel tauchen sie als Kombinationen auf. Jiang et al. [1] unterscheiden zwischen den folgenden Kategorien.

In einem *Performancetest* geht es hauptsächlich darum, zu Messen welche Antwortzeiten das System bei unterschiedlichen Anfragen hat. Ohne die Kombination mit dem *Lasttest* kann diese Kategorie sehr viele Anwendungstypen und Arten von Performancemessungen umfassen. Insbesondere *Single-User*-Systeme können so ebenfalls getestet werden. Ein trivialer Performance-Test wäre zum Beispiel ein einfacher Unit-Test, welcher die Laufzeit einer Funktion misst und fehlschlägt, wenn sie zu lang ist [1].

Als *Lasttest* wird ein Test bezeichnet, bei dem eine spezielle Auslastungssituation angestrebt wird, um beispielsweise zu überprüfen, ob ein System den in Zukunft erwarteten Benutzerzahlen standhalten kann und trotz dieser Auslastung korrekt funktioniert.

An die Grenzen wird das System mit einem *Stresstest* getrieben. Diese Tests erhöhen beispielsweise sukzessive die Anzahl der parallelen Benutzer oder der Request-Rate um den die Grenzen des Systems zu identifizieren. Ziel dieser Tests ist die Untersuchung unter Extremsituationen.

Bei allen Tests ist es ratsam, möglichst viele Samples für einzelne Anfragetypen zu sammeln um aussagekräftige Statistiken über die tatsächlichen Laufzeiten einzelner Anfragen zu erhalten, denn durch die parallelen Prozesse und die eventuell schwankende Netzwerkzeit kommt es in der Regel immer zu einem Grundrauschen. Ebenso kann es Anfragen geben, welche nur in seltenen Fällen langsame Antwortzeiten aufweisen. Durch zeitgesteuerte Hintergrundaktionen auf den Serversystemen kann es zu kurzzeitigen negativen Auswirkungen auf die Antwortzeiten kommen. Ebenso können auch bestimmte Aufrufparameter innerhalb der gestellten Anfrage zu einer Veränderung des Antwortzeitverhaltens führen. Der gezielte Entwurf von Test-Szenarien und den, in ihnen verwendeten, Anwendungsszenarien ist also von großer Bedeutung. Es gibt viele unterschiedliche Methoden realistische Szenarien zu generieren. Häufig werden für den Entwurf dieser Szenarien Experteninterviews oder Log-Daten verwendet [1].

Die Antwortzeiten werden im Bereich der Last- und Performancetests häufig als Durch-

schnittswerte oder Quantile zusammengefasst [1, 86, 87, 83, 84]. Obwohl das arithmetische Mittel über mehrere Testläufe recht gut vergleichbar ist, hat es den Nachteil, dass seltene Ausreißer nur schwer sichtbar werden. Diese starken Ausreißer werden im Zweifelsfall jedoch von einzelnen Benutzern der Anwendung überproportional stark wahrgenommen und sollten deshalb frühzeitig identifiziert werden. Quantile werden häufig auch als Perzentile bezeichnet, welche die Quantilgrenze in Prozent, statt im Intervall von 0 bis 1 angegeben und deshalb leichter lesbar sind. Das 0.5-Quantil entspricht dem oder 50%-Quantil, dem 50. Perzentil und dem Median. Um Antwortzeiten zu erkennen, welche in seltenen Fällen langsamer sind als im Großteil der Antwortzeiten, jedoch in den meisten Fällen recht schnell sind, eignen sich neben dem Median und dem Maximum, die höherwertigen Perzentile im Bereich des 80. bis zum 99. Perzentils sehr gut. Sie filtern zwar extreme Ausreißer heraus, werden jedoch nicht so stark durch die große Mengen an schnellen Antwortzeiten beeinflusst, wie das arithmetische Mittel. Maximal- und Minimalwerte verlieren beispielsweise durch einen einzigen Messwert ihre Aussagekraft, wenn dieser Seitenaufruf in seiner Laufzeit ein extremer Ausreißer ist. Diese Statistiken werden von den meisten Last- und Performancetest-Tools nach Testende und teilweise bereits während der Testlaufzeit angezeigt [83, 85].

Es gibt eine Vielzahl möglicher Performanceprobleme und Bottlenecks. Es gibt allerdings auch Prozeduren, welche durch ihre Komplexität einen größeren Laufzeitbedarf erfordern und deshalb nicht weiter optimierbar sind. Es ist auf Grund der Menge nicht möglich alle möglichen Fehlerbilder aufzuzählen. Insbesondere wird es schwierig alle potentiellen Performanceprobleme mit nur einer Methode erkennen zu können.

Probleme, die ich auch persönlich schon mit meinem derzeitigen Arbeitgeber *Dataport AöR* [82] aufdecken konnte, sind unter anderem langsame Datenbankzugriffe, da diese durch die erhöhte Systemlast durch mehrere gleichzeitige Benutzer überlastet sind und unerwartet ressourcenhungrige Funktionen aus Fremdbibliotheken, welche erst durch eine Untersuchung der Performance aufgefallen. Hinzu kommen auch häufig limitierende oder ungünstig konfigurierte Netzwerkkomponenten. Ein Problem, welches wir in jüngerer Vergangenheit häufig beobachten sind Job-Warteschlangen, welche nicht schnell genug abgearbeitet werden und dann im Nachhinein für eine hohe Hintergrundbelastung des System sorgen. Weniger mit der Seitenantwortzeit zusammenhängend sind *Memory Leaks*, welche zu Stabilitätsproblemen führen können.

Was bei den Last- und Performancetests jedoch häufig fehlt, ist die engere Koppelung mit dem Quellcode, da lediglich die Aufrufdauer der *HTTP*-Aufrufe gemessen wird. Ein tieferer Einblick in die Auslastung innerhalb der Anwendung wäre wünschenswert.

Die Erfahrungen, welche im Kontext der Last- und Performancetests gemacht wurden, sollen mit dem *Profiling* kombiniert werden. In Kapitel 3 wird darauf eingegangen, wie dies in dieser Masterarbeit umgesetzt wurde.

2.3 Verwendete Software

2.3.1 libAwesome

libAwesome [88] ist eine Java Web-Anwendung für die Verwaltung von Buchbeständen für kleine bis mittelgroße Büchereien. Die Anwendung wurde von Studenten im Rahmen des Moduls „Softwareprojekt 2“ entwickelt. Sie befindet sich aktuell für zwei Büchereien im produktivem Betrieb und wird von Rainer Koschke gepflegt. Diese Anwendung wurde in dieser Arbeit während der Entwicklung der Profiler und der Visualisierung als Hauptuntersuchungsobjekt verwendet. Zudem wurde anhand dieses Testobjektes die Benutzerstudie und die weiteren Evaluationen in Kapitel 4 durchgeführt. Die meisten, in dieser Arbeit gezeigten Bilder der entwickelten Anwendungen, nutzen als Grundlage die Software-Stadt von *libAwesome*.

Die Anwendung wurde in einen Glassfisch 5.0.0 Applikationsserver mit der integrierter Derby Datenbank installiert. Abbildung 2.18 zeigt die Übersicht über die ausleihbaren Bücher und Medien.

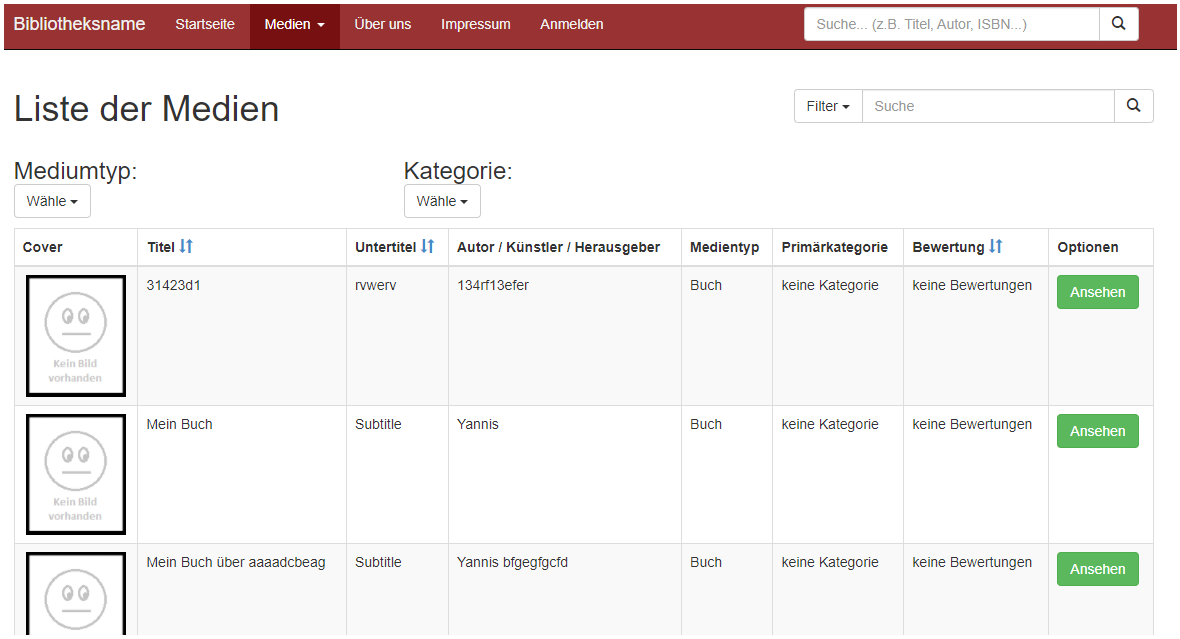


Abbildung 2.18: Liste der Medien in der Web-Anwendung *libAwesome* [88]

2.3.2 Unity

Unity ist eine Game-Engine für die Entwicklung von 3D Spielen. Zusätzlich bietet sie einen mächtigen Editor, welcher das Bearbeiten von einzelnen *Spielszenen* und das Zusammenstellen von virtuellen Welten erleichtert. Diese Entwicklungsumgebung ist kompatibel mit *Visual Studio*, sodass der Quell-Code der entwickelten Spiele in einem klassischen Editor entwickelt werden kann. Programmiert wird in den neuen Versionen von Unity üblicherweise mit C#. Da 3D-Visualisierungen einem Computerspiel stark ähneln, eignen sich Game-Engines häufig auch in wissenschaftlichen Kontexten sehr gut.

Diese Masterarbeit basiert auf der Anwendung SEE, welche in Abschnitt 2.1.8 bereits näher beschrieben wurde. Da SEE in der Game-Engine Unity [59] entwickelt ist, wurde die, in dieser Arbeit entwickelte Visualisierung in SEE, ebenfalls mit C# und in Unity entwickelt.

2.3.3 Programmiersprachen

Neben dem, in C# entwickelten Quell-Code in Unity wurde zusätzlich wurde ein Profiler in Java entwickelt, welcher in Kapitel 3 näher beschrieben wird. Hierfür wurde Java verwendet. Für viele Teile der Arbeit wurde zudem, die für statistische Berechnungen optimierte, Programmiersprache *R* [89] verwendet. Diese wurde zum einen in den Profiler integriert, jedoch auch für die Evaluierungen und die Darstellungen der Diagramme in Kapitel 4 verwendet. Für die Arbeit mit *R* wurde die Paketsammlung *tidyverse* [90] verwendet, mit welcher viele Aufgaben, insbesondere die Visualisierung von 2D-Grafiken, stark vereinfacht werden.

2.3.4 Entwicklungsumgebungen

Die Entwicklung des Quell-Codes dieser wurde in den Entwicklungsumgebungen *Visual Studio* [7], *IntellJ IDEA* [91] und *RStudio* [92] durchgeführt.

2.3.5 Betriebssysteme

Der Inhalt der Arbeit wurde auf einem Rechner mit Windows 10 entwickelt und konnte nur bedingt unter Linux getestet werden. Grundsätzlich sind jedoch alle entwickelten Komponenten auch mit Linux kompatibel. Dies kann jedoch nicht garantiert werden.

2.3.6 Sonstiges

Der Profiler [60] und die Entwicklungsumgebung Netbeans [93] wurden für Vergleichszwecke in der Evaluation in Abschnitt 4.3 genutzt. Für die Evaluation von Profiler und Visualisierung wurden neben *libAwesome* zusätzlich noch die Anwendungen *Gitblit* [94] und *Minecraft* [95] verwendet. An einigen Stellen wurde für die Belastung der Anwendung *libAwesome* das Lasttest-Tool *JMeter* [83] verwendet.

KAPITEL 3

Konzept und Implementierung

3.1 Anforderungen

Auf Basis der im Grundlagenkapitel diskutierten, anderen Arbeiten im Bereich Profiling, Visualisierung von Software und der Last- und Performancetests ergeben sich einige Anforderungen, deren Umsetzung Ziel dieser Arbeit war. Die Implementierungen der Anforderungen werden im Nachhinein detaillierter beschrieben.

A1 Innerhalb von SEE sollen folgende Performancedaten dargestellt werden können.

- Sampleanzahl
- Aufrufanzahl
- Methodenlaufzeit
  - Mittelwert
  - Perzentile
  - Median
  - Maximum
  - Minimum

A2 Die Metriken sollen als *self time* und *total time* darstellbar sein.

A3 Die Metriken sollen in der 3D-Ansicht ohne Neustart der Anwendung wechselbar sein.

A4 Die Statistiken sollen für Aufrufkanten und Methoden separat erhoben werden und darstellbar sein.

A5 Die Aufrufrichtung soll durch eine Animation visualisiert werden.

A6 Der Quellcode einzelner Elemente soll in SEE einsehbar sein.

A7 Es soll eine Live-Ansicht implementiert werden.

A8 Aus Anforderung A1 und A7 folgen:

1. Der Profiler soll für Java-Anwendungen funktionieren.
2. Es soll ein Sampling-Profiler implementiert werden.
3. Es soll ein Instrumentation-Profiler implementiert werden.
4. Die Profiler sollen für eine lange Profiling-Zeiträume geeignet sein.
5. Die Profiler sollen an laufende Java-Prozesse anbindbar sein.

Anforderung A1 Die häufige Nutzung der Perzentile im Bereich Last- und Performancetests zeigt, dass ein Mehrwert in der zusätzlichen Auswertung der höheren Quantile liegt. Die alleinige Betrachtung der Hot-Spots, welche mit einem Sampling-Profiler erhoben werden können oder die Mittelwerte der Laufzeiten, welche Instrumentation-Profiler wie VisualVM erheben, reichen möglicherweise nicht immer aus, um selten auftretende Performanz-Probleme zu erkennen. Da auch Methoden erkannt werden sollen, welche nur bei einem geringen Prozentsatz ihrer Aufrufe langsam sind, werden neben dem Median auch das 90., 95. und 99. Perzentil zu erhoben. Maximum und Minimum folgen den bekannten Definitionen aus der Mathematik. Mit dem Mittelwert ist das arithmetische Mittel gemeint.Für Quantile und Perzentile gibt es unterschiedliche Definitionen dafür, wie es berechnet wird [96, 97], wenn es zwischen zwei Werten der Menge liegt und nicht ein einziges Element der Menge als Mitte gewählt werden kann. Eine Möglichkeit ist es zum Beispiel, entweder den oberen oder unteren Wert zu wählen. In der Implementierung soll der Mittelwert zwischen den beiden Werten errechnet werden.

Anforderung A2 Die Trennung nach *self time* und *total time*, welche aus anderen Profilern bekannt ist, soll übernommen werden, da die *self times* wertvolle Informationen darüber geben können, in welcher Methode sich ein Performance-Problem tatsächlich befindet.

Anforderung A3 Damit die Visualisierung einfach nutzbar ist, sollen keine Neustarts notwendig sein, nur um zwischen den dargestellten Metriken zu wechseln.

Anforderung A4 und A5 Eine der wichtigsten Anforderungen ist A4. Durch die Darstellung der Aufrufkanten soll der Aufrufgraph visuell sichtbar werden und zu einem besseren Verständnis des Kontextes führen. In der Live-Ansicht von Ogami et al. [26] in Abbildung 2.14 (Seite 18), ist dieser Kontext nicht sichtbar, wodurch nur die aktiven Module sichtbar werden, jedoch nicht ersichtlich ist, wie diese miteinander interagieren. Zusätzlich fordert Anforderung A5, dass die Aufrufrichtung durch eine Animation visualisiert werden soll. Ein *Methodenaufruf* oder *Aufrufkante* meint im Folgenden immer die Kombination aus Caller (aufrufende Methode) und Callee (aufgerufende Methode) bezeichnet. Zu einer einzigen Methode können also mehrere Aufurfkanten gehören. Die Gesamtstatistik zu einer *Methode*, ohne Beachtung des Callers, muss separat, ebenfalls erhoben werden, damit diese Werte auf den Methoden-Blöcken dargestellt werden können.

Anforderung A6 In Abschnitt 2.1.3 wurde bereits erwähnt, dass für ein Verständnis der Anwendung die Verbindung zu dem zugrundeliegenden Quell-Code wichtig ist. Hierfür fordert Anforderung A6, dass in der Visualisierung der Quell-Code der einzelnen Aufrufkanten und Methoden leicht abrufbar sind.

Anforderung A7 Weiterhin ist eine Live-Ansicht geplant, wie sie in den Arbeiten von Ogami et al. [26] und Fittkau et al. [98] umgesetzt wurde. Aus diesen beiden Gründen eignet sich für das Vorhaben die Verwendung von etablierten Profilern leider nicht, da diese nicht ohne Weiteres einen direkten Kommunikationskanal zu der Visualisierung in SEE aufbauen können und die Perzentile nicht berechnen können. Die Live-Ansicht wurde zu einem frühen Zeitpunkt dieser Arbeit geplant, wurde im Rahmen dieser Arbeit letztendlich jedoch nicht mehr implementiert.

Anforderung A8 Für die Profiler bestehen weitere Anforderungen. Da sie für den Anwendungskontext von Last- und Performancetests und die Überwachung von länger laufenden Systemen, wie Web-Anwendungen vorgesehen sind, sollen sie über einen längeren Zeitraum laufen können und nach Start der Anwendung gestartet werden können. Anforderung A8 fordert deshalb, dass der Profiler dynamisch sein soll, damit auch langlaufende Web-Anwendungen unterbrechungsfrei untersucht werden können.

Durch den „Observer Effect“ [2, 73] könnte es vor allem bei instrumentiertem Profiling, zu starken Beeinflussungen des Laufzeitverhaltens der Anwendung kommen, da die Instrumentierung deutlich invasiver ist, als ein Sampling-Profiler. Die unterschiedlichen Eigenschaften der beiden Profiling-Methoden sind der Grund dafür, dass beide Verfahren eingebaut werden sollen. Die Profiler sollen für Java-Anwendungen implementiert werden, weil ein universeller Profiler zu aufwändig wäre und mit *libAwesome* eine gut testbare Java-Anwendung zur Verfügung steht.

Nicht Implementierte Anforderungen Anforderung A7 konnte leider im vorhandenen zeitlichen Rahmen nicht mehr implementiert werden. Im Ausblick wird darauf eingegangen, wie der Profiler um eine Live-Ansicht erweitert werden könnte. Die Fähigkeit den Profiler über Zeiträume von mehreren Stunden laufen zu lassen konnte nicht evaluliert werden. Es wurde jedoch ein Modus im Profiler implementiert, welcher für die Berechnung der Quantile nicht alle Daten vorhalten muss, wodurch ein wachsender Speicherverbrauch im Profiler, sowie ein rechenaufwändiges Berechnen der Quantile mit riesigen Datenmengen vermieden wird. Hierdurch wurde eine Grundlage geschaffen, wodurch der Profiler zukünftig auch in längeren Tests verwendet werden könnte.

3.2 Proof of Concept

Während der initialen Recherchephase dieser Masterarbeit wurde ein Proof-Of-Concept entwickelt. Dieser dient zur Überprüfung der Machbarkeit der geplanten Messungen und Darstellungen, aber auch um das Konzept der Visualisierung frühzeitig am Objekt zu sehen um weitere wichtige Funktionalitäten zu erkennen. Für den Proof-of-Concept wurde die Software *libAwesome* [1] mit einem Profiler untersucht. Es wurde ein rudimentärer Stack-Sampler für Java implementiert, ein Python-Skript zum aufbereiten der Stack-Traces und eine erste Implementierung für SEE, um die berechneten Statistik darzustellen. *JMeter* wurde als Lasttreiber verwendet um eine Lastsituation zu erzeugen. Der Test wurde so programmiert, dass eine Seite einer Web-Anwendung parallel von mehreren virtuellen Benutzern wiederholt aufgerufen wurde. Vergleichsweise wurde die tabellarische Darstellung der gleiche Informationen des CPU-Profilers „VisualVM“ betrachtet. Es wurden mit JStack [76] in unbestimmten Intervallen Stack-Traces gesammelt. Diese wurden danach mit einem Python-Tool gruppiert und die Sampleanzahl ausgegeben. Die genaue Funktionsweise des Proof-of-Concepts ähnelt der in Abschnitt 3.6.2 beschriebenen Funktionsweise der endgültigen Implementierung in Java. Die wichtigste Erkenntnis war, dass die Visualisierung direkt an den Quell-Code gekoppelt werden sollte, da es ansonsten zu umständlich ist, anhand der Methoden und Dateinamen die Verbindung zwischen der Visualisierung und dem Quell-Code herzustellen.

[1] *libAwesome wurde in Abschnitt 2.3.1 näher beschrieben.*

3.3 Beispielvideos

Da es sich bei der in dieser Arbeit entwickelten Anwendung um ein sehr visuelles Produkt handelt, welches sich durch seine interaktiven Eigenschaften und Animationen definiert, wurden Videos erstellt um die texttuelle Erklärung zu ergänzen. Im folgenden Kapitel gibt es einige Abbildungen, welche einzelne Teile illustrieren, die Videos helfen jedoch, die Interaktion mit der Anwendung besser zu verstehen.

Es wurden zwei Beispielvideos angefertigt, welche auf *YouTube* über den, der Arbeit beiliegenden USB-Datenträger verfügbar sind. Die Beispielvideos haben keine Vertonung, es sind jedoch beschreibende Untertitel verfügbar. Diese können über die Einstellungen des *YouTube*-Videoplayers aktiviert werden. Den Videos auf dem USB-Datenträger liegen die Untertitel im `.srt`-Format bei. Diese können in gängigen Videoplayern einfach dem Video angefügt werden.

Demovideo Profiler	`https://youtu.be/Bq5Ef31JUWo`
Demovideo Visualisierung	`https://youtu.be/Qgb8azxCW_k`

Im *Demovideo Profiler* wird gezeigt, wie die grafische Oberfläche des implementierten Profilers verwendet wird. Zunächst wird die Anwendung *libAwesome* mit dem Sampling-Profiler *gesampled* und danach mit dem Instrumentation-Profiler instrumentiert und profiliert.

Das *Demovideo Visualisierung* zeigt nacheinander alle Funktionen der Visualisierung anhand erhobener Performance-Daten der Anwendung *libAwesome*. Die 3D-Kanten stellen durch ihre Breite einzelne Metriken zu Methodenaufrufen dar, während die Gesamtheit der Kanten den Aufrufgraphen zwischen den einzelnen Methoden aufspannt.

Ein weiteres Video zu der Darstellung in SEE wurde für die Benutzerstudie in Abschnitt 4.3.5.2 aufgenommen. Dieses hat eine zusätzliche Audiospur, welche die Aufgabe für die Testpersonen der Studie erklärt. Das Video zeigt jedoch nicht den vollen Funktionsumfang, da in der Studie nur die Sampling-Informationen dargestellt wurden.

3.4 Aufbau

Abbildung 3.1 zeigt den Informationsfluss von der gestesteten Anwendung bis zur Visualisierung. Aus dem *System under Test*, der Anwendung, welche untersucht wird, werden die Profiling Informationen durch den *yProfiler* erhoben. Diese Daten werden in Dateien herausgeschrieben, sodass sie danach an die Visualisierung in SEE weitergereicht werden können.

In Abschnitt 3.5 werden zunächst die Funktionen der Visualisierung in SEE beschrieben. Dies beinhaltet Abbildungen, welche zeigen, wie die Visualisierung funktioniert. Praxisbeispiele werden in der Evaluation in Abschnitt 4.1 gezeigt. Daraufhin werden in Abschnitt 3.6 die zwei implementierten Profiler beschrieben. Dies beinhaltet die grafische Oberfläche, sowie technische Details zur Funktionsweise der Profiler. Der Profiler unterstützt zudem einen Modus, in welchem die Quantile approximiert werden, auf welchen, in Abschnitt 3.6.3.5 ebenfalls eingegangen wird.

Abbildung 3.1: Datenfluss von getesteter Anwendung bis zur Visualisierung in SEE

3.5 Visualisierung in SEE

Die Videos in Abschnitt 3.3 zeigen die grundlegenden Funktionen der Visualisierung. Wie die Visualisierung effektiv genutzt werden kann, wird in Abschnitt 4 im Evaluationskapitel und in der Beschreibung der Aufgaben, welche in der Benutzerstudie in Abschnitt 4.3.2 verwendet wurden, deutlicher. Für ein besseres Verständnis davon, wie die Visualisierung in der Praxis genutzt werden kann, wird den Lesenden deshalb empfohlen sich diese Beispiele anzuschauen.

„PerformanceEdges“ ist der Name des implementierten Moduls für SEE, welches die mit den Profilern gesammelten Daten grafisch darstellt und untersuchbar macht. Dieses Modul besteht im Wesentlichen aus zwei Teilen: Der 3D-Darstellung und der zweidimensionalen Benutzeroberfläche für Quell-Code und Einstellungen.

Die, am Boden der 3D-Ansicht angeordneten Blöcke, sind die Software-City der untersuchten Anwendung. Die einzelnen Blöcke entsprechen einzelnen Methoden der Anwendung, welche hierarchisch nach Paket- und Klassenzugehörigkeit in einer *TreeMap* angeordnet sind. Die Darstellung der Software als Software-Stadt und die Anordnung als *Treemap* wurde nicht im Rahmen dieser Masterarbeit entwickelt. Sie war, wie in Abschnitt 2.1.8 erläutert, bereits vorhandener Bestandteil von SEE.
Neu implementiert sind die verbindenden Kanten, welche die Unterschiedlichen Performance-Metriken darstellen können. Die darzustellenden Metriken sind über ein Drop-Down-Menü wechselbar. Abbildung 3.2 zeigt eine exemplarische Darstellung, in welcher die Kanten, abhängig von ihrer *Sampled Heat*, unterschiedlich breit skaliert sind. Zu sehen sind die aus dem

„In“-Block eingehenden Kanten, welche sich dann durch die Software-Stadt der untersuchten Anwendung von Methode zu Methode fortsetzen und zuletzt in den „Out“-Block zurückgehen. Die In- und Out-Blöcke stehen repräsentativ für alle Methoden, welche nicht in der dargestellten Software-Stadt vorhanden sind, da sie aus externen Bibliotheken sind. Abbildung 3.3 zeigt zusätzlich die 2D-Oberfläche, in welcher der Quell-Code von der aktuell ausgewählten Kante oder Methode zu sehen ist.

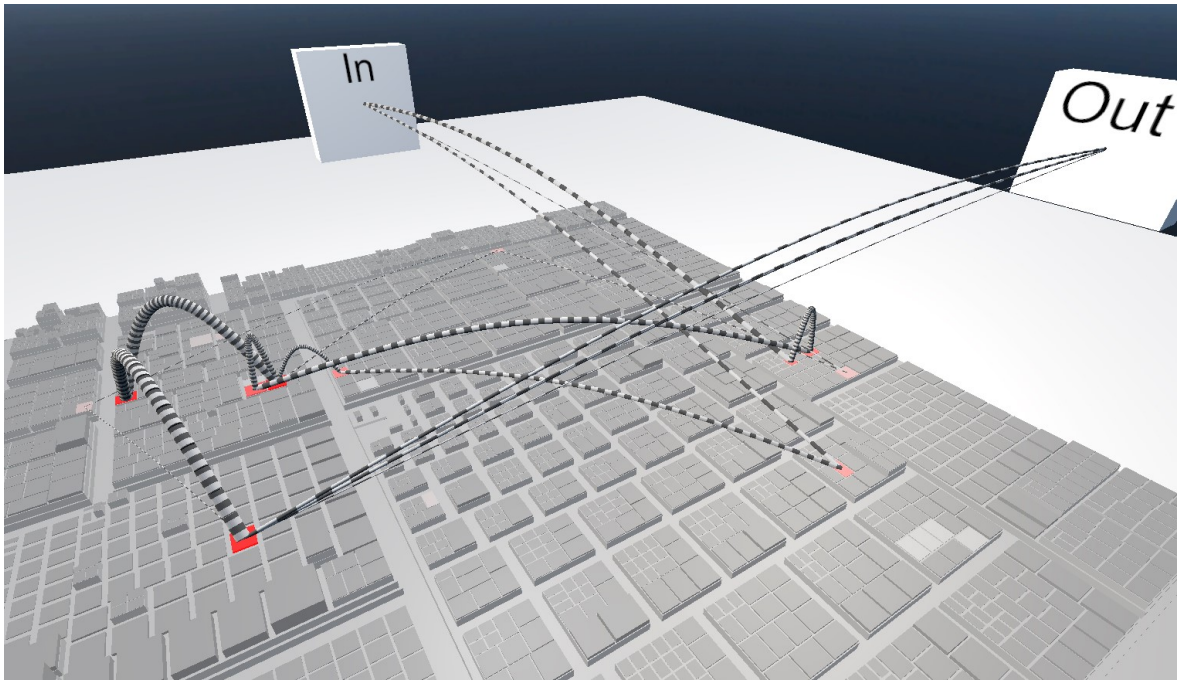


Abbildung 3.2: Darstellung der mit dem Profiler erhobenen Daten in SEE

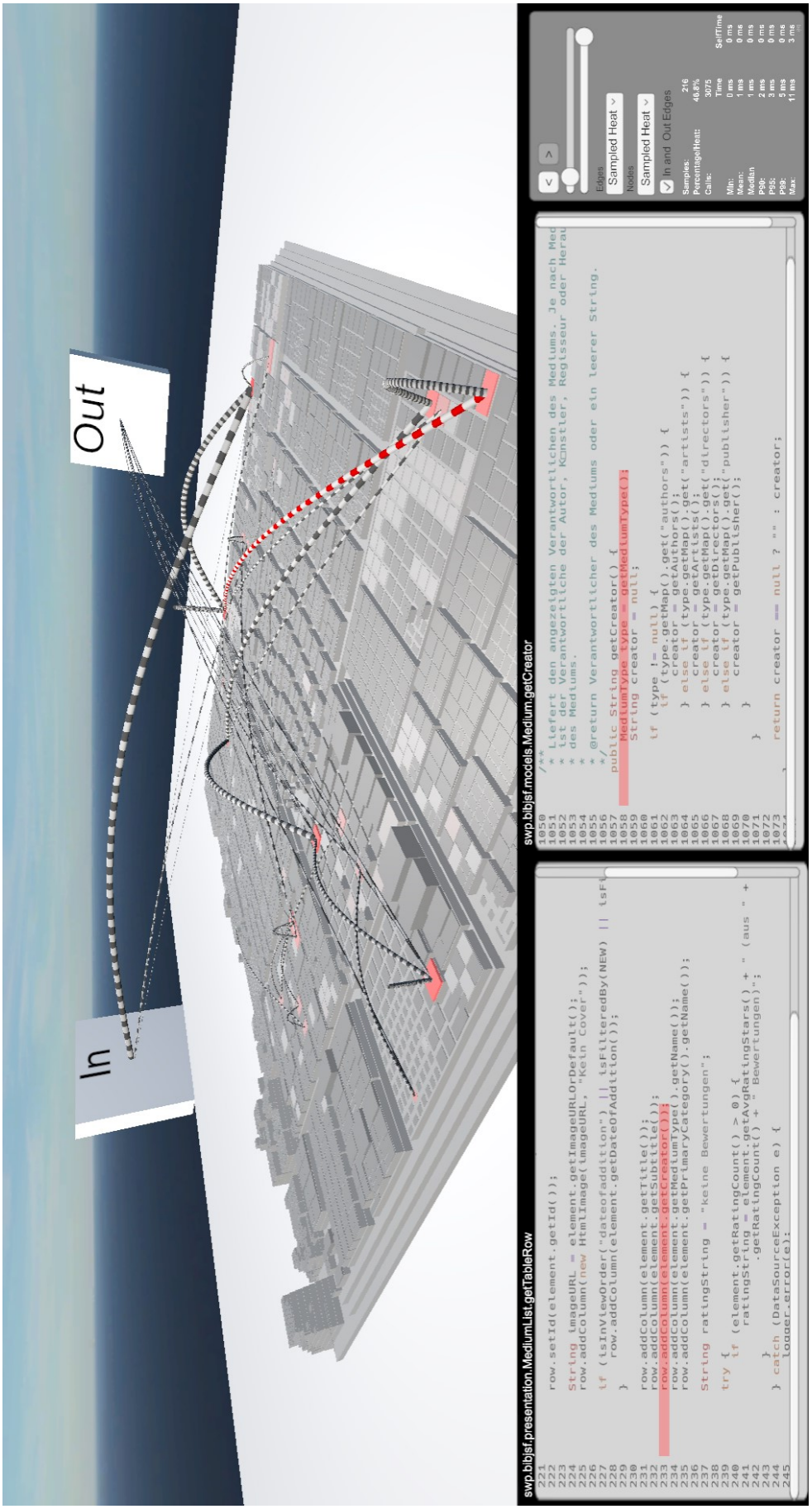


Abbildung 3.3: 3D-Darstellung und 2D-Oberfläche in SEE

3.5.1 Kanten und Methoden

Die in Abbildung 3.2 dargestellten 3D-Rohre entsprechen den Methodenaufrufen. Sie werden im Folgenden als „Aufrufkanten“ oder „Kanten“ bezeichnet. Die Aufrufrichtung einer Kante ist in den Abbildungen nicht ersichtlich, da sie durch eine Animation dargestellt wird. In der animierten Visualisierung bewegen die grau-weißen Segmente einer Kante sich in Richtung des Aufrufes. Sie verlaufen von der aufrufenden Methode, dem *Caller*, zu der aufgerufenen Methode, dem *Callee*. Es ist direkt sichtbar, dass durch die Darstellung der Aufrufkanten der Aufrufgraph aller Methodenaufrufe erkennbar gemacht wird, wodurch ein Gesamtkontext sichtbar wird. Es sind jedoch nicht alle, theoretisch existierenden Aufrufkanten, abgebildet, da Methodenaufrufe, welche vom Profiler nicht erfasst wurden, in der Darstellung nicht auftauchen. Der Graph entspricht ebenso auch keinem einzelnen *Stack-Trace*, sondern er ist die Vereinigung aus allen beobachteten *Stack-Traces* oder Methodenaufrufen.

Die, mit dem Profiler erhobenen Metriken zu den einzelnen Methoden und Aufrufkanten werden visuell auf die 3D-Objekte abgebildet. Bei den Methoden wird die Farbtiefe der „Häuser“ (Methoden-Blöcke) der Stadt verändert, während bei den Kanten die Breite verändert wird. Je breiter die Kanten oder je röter die Blöcke sind, desto höher der Wert der dargestellten Metrik. Die Breite und die Farbtiefe sind nach oben begrenzt, weshalb alle Werte relativ zum größten vorhandenen Wert skaliert werden. So wird verhindert, dass unbegrenzt breite Kanten dargestellt werden. Für die Methodenblöcke, welche in einem dunklerem Grauton eingefärbt sind, liegen keine entsprechenden Performance-Metriken vor. Diese Methoden wurden im getesteten Szenario nicht aufgerufen oder konnten nicht instrumentiert werden. Die darstellbaren Metriken sind die mit dem Sampling-Profiler erhobene *Sampled Heat*, welcher der Anzahl der Samples entspricht und die Statistiken des Instrumentation-Profilers: die Aufrufanzahl, die mittlere Methodenlaufzeit, die minimale und maximale Laufzeit, der Median und oberen Perzentile (90., 95., 99.). Weitere Perzentile wären durch minimale Anpassungen in der Implementierung ebenfalls auswählbar.

Während Abbildung 3.3 Ergebnisse des Sampling-Profilers zeigt, wird in Abbildung 3.4 der Instrumentation-Profiler verwendet. Die Breite der Kanten entspricht dem 95. Perzentil der jeweiligen Aufruflaufzeiten. Die breiteren Kanten sind also Methodenaufrufe, welche in 5% ihrer Aufrufe hohe Laufzeiten im Vergleich zu den anderen Laufzeiten haben. Sehr dünne Kanten können ausgeblendet werden, indem der obere Schieberegler verschoben wird. Hierdurch können Kanten mit geringerer Relevanz ausgeblendet werden. Auch die größten Kanten können ausgeblendet werden, um einen besseren Überblick über die kleinen Kanten zu bekommen, nachdem die auffälligsten Kanten bereits untersucht wurden. Eine vorstellbare Situation ist, dass eine rechenintensive Methode andere Methoden überschattet, jedoch schnell deutlich wird, dass diese Methode für die aktuelle Untersuchung nicht weiter interessant ist. Durch das Ausblenden der Kanten mit den höchsten Werten, werden kleinere Kanten automatisch neu skaliert und werden besser sichtbar. Diese Neuskalierung wird in Abbildung 3.4 illustriert. Die unteren und oberen Grenzen lassen sich über den Schieberegler im Informationspanel ohne Wartezeiten umstellen. In Abbildung 3.4 ist nur der Schieberegler aus dem Informationspanel eingeblendet.

Anmerkung: Die Darstellung der Software-Stadt, bestehend aus den Methodenblöcken, wurde nicht im Rahmen der Arbeit implementiert. In dieser Arbeit wurde das Verbinden der neuen Kantenrepräsentation zwischen den statischen Blöcken und die Anbindung an die Performancedaten implementiert. Zudem wurde die zugehörigen, zweidimensionale Oberfläche implementiert, welche in Abbildung 3.3 zu sehen ist.

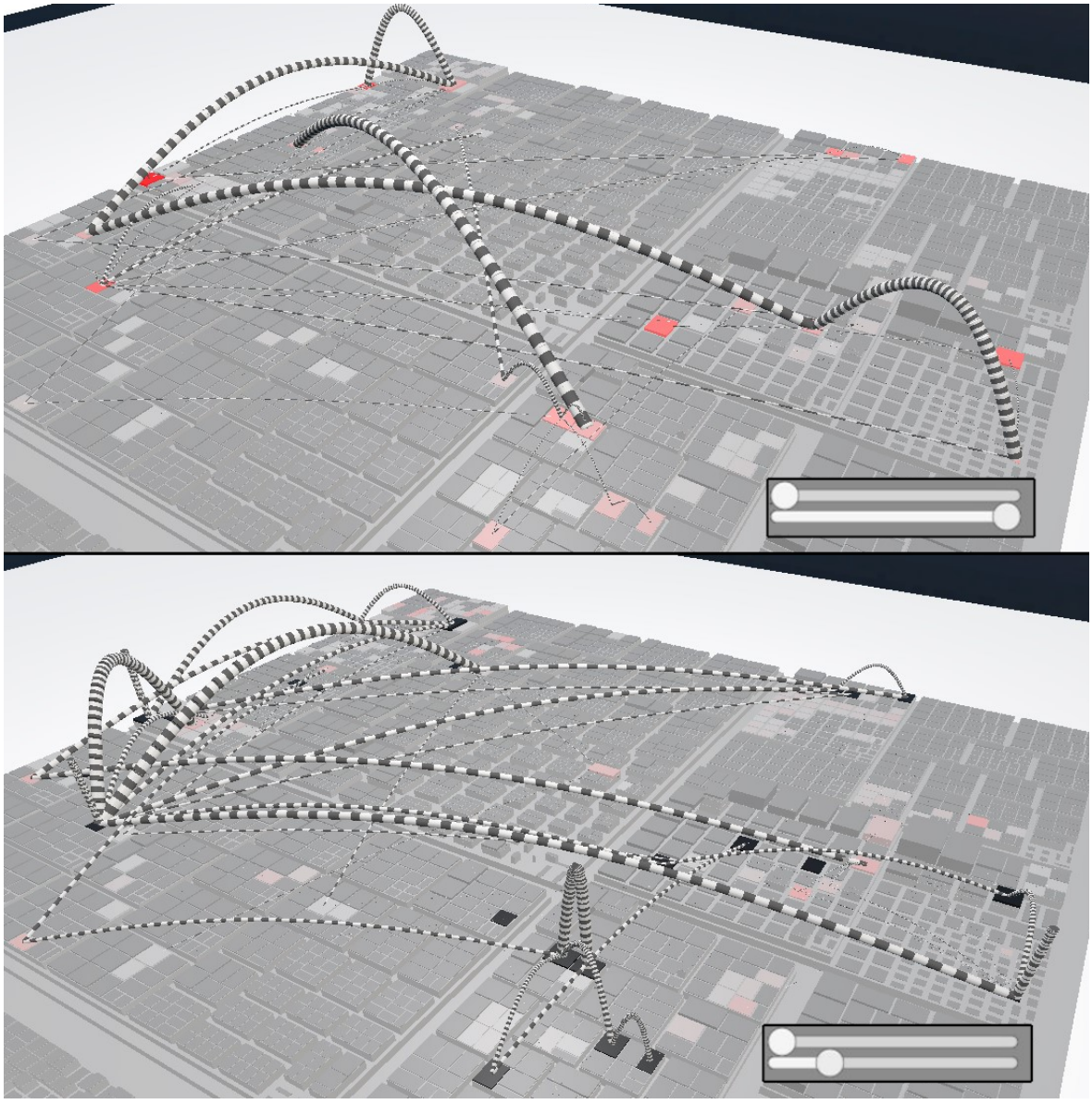


Abbildung 3.4: Verstecken der breitesten Kanten durch den Schieberegler. Durch das herausfiltern der größten Kanten werden dünnere Kanten breiter und sichtbarer.

Für alle Metriken, für welche *self times* vorliegen, kann ebenfalls die *self time*-Darstellung aktiviert werden. Das sind die Metriken zur Methodenlaufzeit, welche mit dem Instrumentation-Profiler erhoben wurden, abgesehen von der Aufrufanzahl. Abbildung 3.5 zeigt den Unterschied zwischen der Darstellung von der *total time* und der *self time* am Beispiel des 95. Perzentils. Hierduch werden die Methoden deutlicher sichtbar, in welchen tatsächlich Laufzeit verbraucht wurde. Bei dargestellter *self time* ist jedoch der Aufgrufgraph gar nicht, oder nur schwach erkennbar, da in den meisten Fällen die letzte Methode in einer Aufrufkette die höchste *self time* hat. Die *self times* lassen sich auch auf den Methoden-Blöcken darstellen.

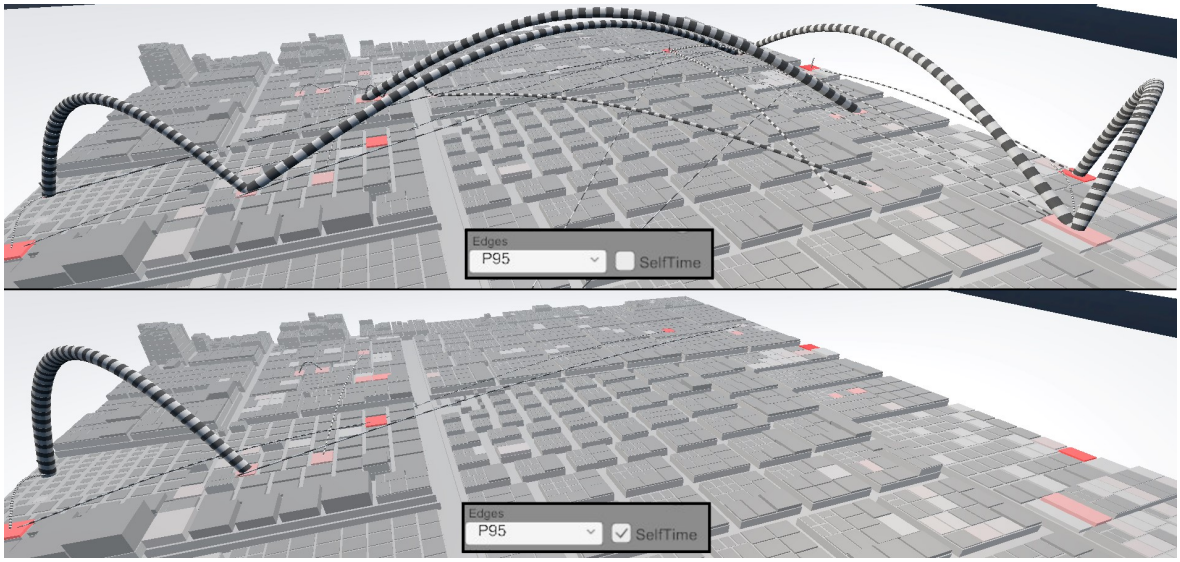


Abbildung 3.5: Unterschied der Darstellung des 95. Perzentils (0.95 Quantil) als *total time* im Vergleich zur *self time*

3.5.2 Einstellungen und Informationspanel

Abbildung 3.6 zeigt das Einstellungs- und Informationspanel. Hier können für die 3D-Ansicht Darstellungseinstellungen vorgenommen werden und die Statistiken der aktuell ausgewählten Aufrufkante oder Methode werden angezeigt. In zwei Drop-Down-Menüs sind die Statistiken auswählbar, welche für grafische Darstellung verwendet werden sollen. Falls für die ausgewählte Statistik *self times* vorhanden sind, wird zudem eine *Checkbox* eingeblendet, mit welcher zwischen *self time* und *total time* gewechselt werden kann. Die im Drop-Down-Menü auswählbaren Einträge sind „Sampled Heat“, „Calls“, „Heat“, „Mean“, „Min“, „Median“, „P90“, „P95“, „P99“ und „Max“. Dies sind Kürzel für die erhobenen Metriken. Eine Erweiterung um weitere Perzentile ist durch geringe Anpassungen in der Implementierung möglich. Abbildung 3.5 zeigt links eine Darstellung mit dargestelltem Median und rechts den Median der *self time*. Der Unterschied ist, dass bei der Betrachtung der *self time* der Aufrufkontext in vielen Fällen nicht mehr deutlich erkennbar ist, da nur die tiefste Methode im Aufrufgraphen tatsächlich selbst Laufzeit in Anspruch nimmt oder mit dem Aufruf externer Methoden beschäftigt ist.

Zusätzlich sind in diesem Panel Knöpfe zum Vor- und Zurückspringen verfügbar. Sie ermöglichen das Vor- und Zurückspringen zwischen Kanten, welche zuvor ausgewählt wurden.

Die mit „In and Out Edges“ beschriftete Checkbox blendet alle eingehenden und ausgehenden Kanten und die „In/Out“-Blöcke ein oder aus. Diese Kanten können für die, mit dem Sampling-Profiler erhobenen Daten, dargestellt werden. Der Instrumentation-Profiler kann keine eingehenden und ausgehenden Kanten erkennen.

Unter den Einstellungen befindet sich eine tabellarische Darstellung der ermittelten Metriken. Sie sind jeweils mit *total time* und *self time* gelistet.

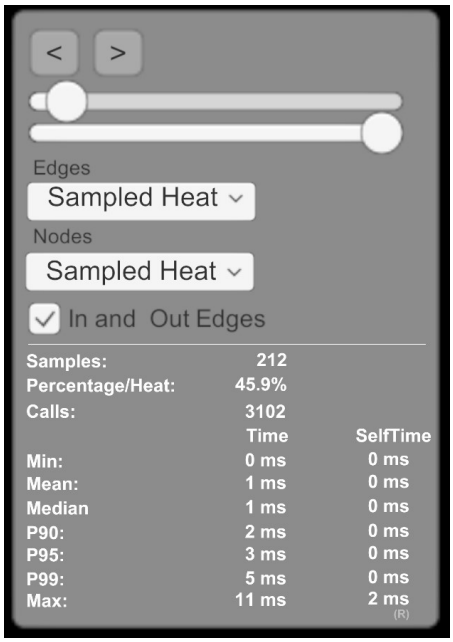


Abbildung 3.6: Informationspanel

3.5.3 Quell-Code

Die Kanten und Methoden-Blöcke können mit der Maus ausgewählt werden, wodurch in der Oberfläche der Quell-Code und die Performance-Informationen angezeigt werden. Wird eine Aufrufkante ausgewählt, öffnet sich im linken Code-Fenster die Methode des Callers. Dort werden ebenfalls die Zeilen markiert, von welchen der ausgewählte Aufruf ausgeht. Im rechten Code-Fenster wird die aufgerufene Methode angezeigt, der Callee. In Abbildung 3.3 auf Seite 35 sind diese Code-Fenster nebeneinander zu sehen. In den Code-Fenstern werden ebenfalls die Hot-Spots auf Zeilenebene hervorgehoben. Die Hervorhebungen in Abbildung 3.7 zeigen unterschiedlich stark markierte Zeilen, abhängig davon, wie häufig sie vom Sampling-Profiler gesehen wurden. Diese Hervorhebung könnte in 3.7 darauf schließen lassen, dass die Aufrufe der `get`-Methode mit den Parametern `BACKUP_WEDNESDAY` und `BACKUP_THURSDAY` weniger performant sind, als die anderen Methoden. An dieser Stelle ist dies vermutlich jedoch ein Trugschluss, welcher durch einen zu kurzen Profiling-Zeitraum entstanden ist. Die Parameter werden vermutlich in der Anwendung nicht wirklich unterschiedlich rechenintensiv sein. Wird der Mauszeiger über eine der hervorgehobenen Zeilen im Callee gehalten, werden die

entsprechenden, ausgehenden Kanten in der 3D-Ansicht schwarz hervorgehoben. Die Hervorgehobenen Zeilen sind nur verfügbar, wenn der Sampling-Profiler verwendet wurde. Der Instrumentation-Profiler kann die hierfür notwendigen Informationen nicht erheben.

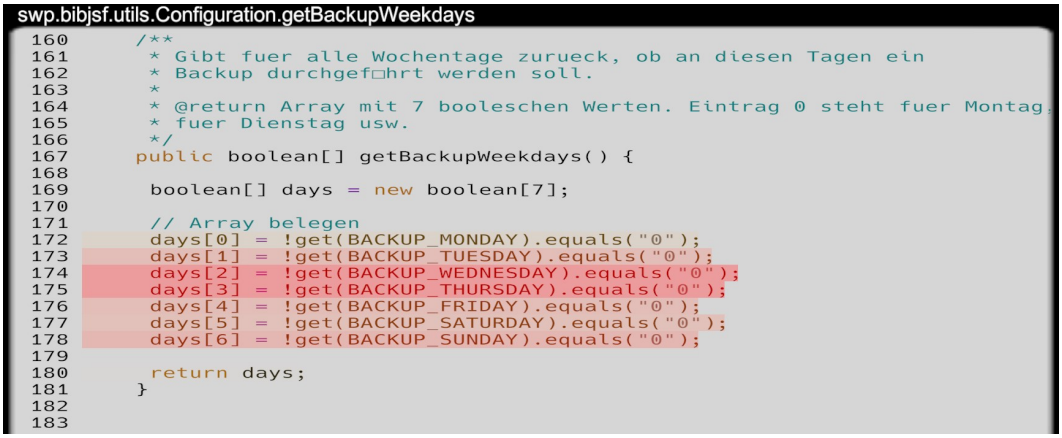


Abbildung 3.7: Darstellung des Quellcodes

Die hervorgehobenen Zeilen im rechten Code-Fenster sind anklickbar. Dies hat den selben Effekt, wie das Anklicken der zugehörigen Kante: Es öffnet die entsprechende Kante und zeigt den nächsten Methodenaufruf an. Da von einer Zeile mehrere unterschiedliche ausgehende Kanten existieren können, ist es möglich, dass sich stattdessen ein Dialogfenster öffnet, welches alle Zielkanten anzeigt, welche gemeint sein könnten. Auch dieses Dialogfenster hebt die 3D-Kanten automatisch farblich hervor, wenn die Maus über eine der Auswahlmöglichkeiten bewegt wird. Abbildung 3.8 zeigt den Auswahldialog.

Die hervorgehobenen Zeilen sind nur dann verfügbar, wenn der Sampling-Profiler verwendet wurde, da der Instrumentation-Profiler die Zeilen der Aufrufe nicht kennt.

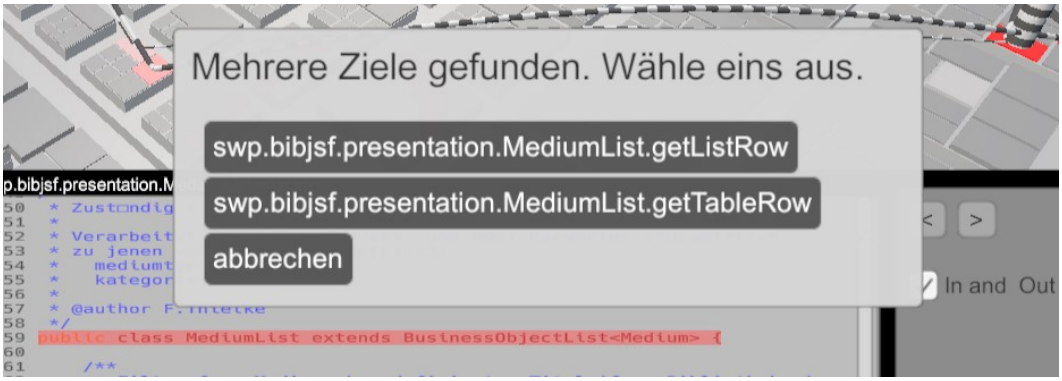


Abbildung 3.8: Auswahldialog bei mehreren Zielkanten auf einer Zeile.

3.5.4 Steuerung

Die Anwendung wird mit Maus und Tastatur gesteuert. Mit Hilfe der Pfeiltasten wird die Kamera durch den virtuellen Raum bewegt. Mit der Maus wird die Kamera gedreht. Sie fixiert hierbei einen Fokuspunkt, welcher sich auf Höhe der Software-Stadt befindet. Durch halten der rechten Maustaste und gleichzeitiges Bewegen mit der Maus lässt sich so die Kamera um diesen Fokuspunkt herum rotieren. Hierdurch wird nicht die Kamera um ihre eigene Achse, sondern

die Kameraposition um die betrachtete Stadt rotiert. Wird die Kamera auf der vertikalen Achse nach oben und unten bewegt, wird seitliche Perspektive zu einer Top-Down-Ansicht auf die Stadt.

Anmerkung: Die Steuerung wurde nicht im Rahmen Arbeit implementiert. Sie wurde lediglich in Teilen nach Bedarf angepasst. Insbesondere wurde jedoch das Input-System so angepasst, dass die Maussteuerung für die Benutzerstudie auch mit einer Remote-Sitzung funktioniert, da das Halten der rechten Maustaste und das Drücken der Tasten W, A, S und D via Remote-Control-Software in Unity standardmäßig nicht wie gewünscht erkannt wurde.

3.5.5 Sonstige Eingabegeräte

Die Weiteren von SEE unterstützten Eingabegeräte wurden in dieser Arbeit nicht weiter betrachtet, sind prinzipiell jedoch nicht grundsätzlich inkompatibel. Für diese Arbeit wurde ausschließlich die Desktop-Umgebung mit Steuerung durch Maus und Tastatur betrachtet. Für eine Nutzung mit einer VR-Brille müsste, vor allem die zweidimensionale Oberfläche angepasst werden.

3.5.6 Start der Visualisierung

Nach der Ausführung des Profilers liegen die CSV-Datein mit den Daten gesammelt in einem Ordner bereit. Im Unity-Editor wird der Dateipfad zu diesem Ordner angegeben. Dieser Schritt ist zu Beginn des Beispielvideos in Abschnitt 3.3 zu sehen. Die Metriken werden aus der CSV-Datei eingelesen und mit den Elementen der Software-Stadt verknüpft, sodass sie jeder Kante und jedem Methodenblock zugeordnet sind. Zusätzlich muss die zugrundeliegende Software-Stadt aus einer GXL-Datei erzeugt werden. GXL-Dateien sind eine standardisierte XML-Struktur, welche unter anderem die hierarchische Struktur von Softwareprojekten enthalten [55, 56]. Diese GXL-Datei wird mit Tools von *Axivion* [99] generiert, welche zur Verfügung standen. Das GXL-Format und die Generierung der Stadt an sich, wurden nicht im Rahmen dieser Arbeit entwickelt, sondern waren bereits vorher fester Bestandteil von SEE.

Eine Version der Anwendung, bei welcher die GXL der Softwarestadt und die Profiling-Daten bei Anwendungsstart ausgewählt werden können, ohne den Unity-Editor zu nutzen, wurde nicht implementiert, da die entwickelte Komponente zukünftig in die gesamte SEE-Umgebung integriert werden soll. Die Untersuchung der Profiling-Informationen in dieser Arbeit wurden also immer über den Unity-Editor gestartet. Es steht im Anhang jedoch eine ausführbare Demoversion mit Profiling-Daten für die Anwendung *libAwesome* bereit, welche ohne den Unity-Editor gestartet werden kann. Zusätzlich sind die Aufgaben aus der Benutzerstudie aus Abschnitt 4.3 ebenfalls als ausführbare Dateien verfügbar. Siehe hierzu Anhang A.1.

3.5.7 Implementierungsdetails

3.5.7.1 PerformanceAnalysis-Szene

Es wurde eine „Unity-Szene“ für die Performance-Analysen erstellt. In einer Unity-Szene befinden sich hierarchisch angeordnete *GameObjects*. In Unity sind GameObjects hierarchisch anordenbare Objekte, welche 2D oder 3D-Objekte innerhalb der Szene beinhalten. An diese Objekte sind C#-Skripte gebunden, welche die Logik der einzelnen Objekte beinhalten. In der vorbereiteten Szene befindet sich bereits ein `SEECity`-GameObject befindet. Die `SEECity` ist das Objekt, welches die Methoden-Blöcke aus der GXL Datei generiert und anordnet. Dieses Objekt wurde nicht im Rahmen der Arbeit implementiert und bildert die Basis von SEE. Losgelöst von der `SEECity` ist in der Szene ein `PerformanceEdges`-GameObject vorhanden, welches zentral, als Singleton, die von mir implementierten Funktionalitäten beinhaltet. Über dieses `PerformanceEdges`-GameObject können, in der Unity-Entwicklungsumgebung, Einstellungen konfiguriert werden. Hier wird über einen Dateipfad angegeben, an welcher Stelle sicher der Quell-Code der untersuchten Anwendung befindet und die aufgezeichneten Performance-Daten liegen. Das `PerformanceEdges`-Objekt beinhaltet zudem alle, für die Oberfläche benötigten Elemente, welche während der Laufzeit generiert werden. Ein weiteres Teilmodul ist die `PerformanceUI`, welches die einzelnen UI-Elemente und die zugehörigen Skripte enthält.

Das Hauptskript der `PerformanceEdges` ist so aufgebaut, dass es beim Start der Anwendung im angegebenen Ordner nach CSV-Dateien sucht, welche die Performance-Informationen der Profiler enthalten und diese dann iterativ einliest. Sie enthalten pro Zeile eine Methode oder Aufrufkante und spaltenweise die aufgezeichneten Statistiken, Methodennamen und Zeilennummern. Das genaue Format der CSV-Dateien kann, bei Bedarf, aus den Dateien auf dem USB-Stick im Anhang oder dem Quell-Code entnommen werden. In den Dateien des ausführbaren Beispiels und im Ordner mit den R-Skripten finden sich entsprechende Dateien.

Hierbei wird ein Graph aufgebaut, in welchem `PipeConnector`-Objekte die Kanten und `CallGraphNode`-Objekte die Knoten des Graphens bilden. Innerhalb dieser Klassen werden dann, für die spätere Darstellung die eingelesenen Metriken gespeichert. Die `PipeConnector`-GameObjekte sind gleichzeitig auch die visuell sichtbaren Objekte innerhalb der Szene, während die `CallGraphNode`-Objekte lediglich mit den bereits existierenden Methoden-Blöcken der `SEECity` verknüpft werden. Die programmierte Logik der einzelnen Oberflächen-Elemente verteilt sich über einzelne Skripte, welche an die entsprechenden GameObjects gebunden sind, so wird beispielsweise durch das Ändern einer Schaltfläche in der Oberfläche, oder das Anklicken einer 3D-Kante im entsprechenden Objekt ein Event ausgelöst, welches dann an die `PerformanceEdges`-Hauptlogik zurückgegeben wird. Beim Klick auf eine Kante, wird dann beispielsweise die Logik zum Darstellen des Quell-Codes aufgerufen.

3.5.7.2 Visualisierung des Quell-Codes

Der dargestellte Quell-Code wird über ein selbst implementiertes Syntax-Highlighting visuell leichter lesbar gemacht. In Abbildung 3.7 war dies bereits zu sehen. Das Syntax-Highlighting basiert auf regulären Ausdrücken, welche Schlüsselworte und primitive Datentypen erkennen. Zeichenketten und Kommentare werden ebenfalls erkannt und es wird verhindert, dass Schlüsselworte innerhalb von Kommentaren und Zeichenketten hervorgehoben werden. Ein Syntax-Highlighting, basierend auf abstrakten Syntax-Bäumen, wäre perspektivisch besser, war für dem Umfang dieser Arbeit, mangels geeigneter Fremdbibliotheken, jedoch zu aufwändig.

3.5.7.3 Darstellung der 3D-Kanten

Anforderung an die Darstellung der Aufrufkanten war, dass die Richtung der Kanten erkennbar ist und dass mindestens eine visuelle Eigenschaft der Kanten dynamisch veränderbar ist. Wichtig war, dass die veränderbare Eigenschaft die Sichtbarkeit unwichtiger Kanten stark reduziert. Würden alle Kanten gleich dick dargestellt werden und lediglich die Farbe abhängig von den Statistiken verändert werden, wäre die Ansicht deutlich unübersichtlicher, als wenn unwichtige Kanten allein durch ihre visuelle Eigenschaft automatisch graduell verschwinden.

Unity bietet keine nativen Mechanismen zum Darstellen von flexibel veränderbaren Bögen oder Splines an. Eine betrachtete Möglichkeit war die Nutzung der `LineRenderer` in Unity zum Darstellen von Linien. Viele kurze Linien könnten verwendet werden, um einen Bogen darzustellen. Der `LineRenderer` generiert jedoch keine `Meshes`, also Objekte mit welchen übliche Interaktionen durch das Eventsystem möglich sind. Für die Darstellung von Kanten in SEE gab es zum Planungszeitpunkt bereits eine Implementierung für die Darstellung von Kanten, welche auf dem `LineRenderer` basiert. Sie basiert auf der *TinySpline*-Bibliothek von Marcel Steinbeck [58], welche die einzelnen Punkte auf einer Spline errechnet, welche dann durch den `LineRenderer` dargestellt werden. Nachteil dieser Darstellung war, dass in dieser Lösung keine Mausklicks auf die Kanten möglich waren [2]. Zudem soll die Aufrufrichtung leicht über eine Animation erkennbar sein. Die vorhandene Implementierung stellte die Aufrufrichtung lediglich über einen Farbverlauf dar. Eine weitere vorhandene Kantendarstellung, welche auf sich bewegenden Kugeln basiert, hatte ebenfalls den Nachteil, dass diese Kugeln nur schwer anklickbar wären. Um beide vorhandenen Implementierungen anklickbar zu machen, müssten zum Beispiel unsichtbare Kollisionsboxen über die Linien gelegt werden.

3.5.7.4 Tatsächliche Implementierung der Kanten

Es wurde sich für eine auf Zylindern basierende Neuimplementierung entschieden, da diese ohne großen Entwicklungsaufwand umsetzbar sind und drei wichtige Punkte erfüllen. Die Zylinder, welche solide Objekte mit Kollisionsboxen sind, lassen sich leichter in das Event-System von Unity integrieren, wodurch Mausklicks auf diese ermöglicht werden. Sie sind leicht animierbar und lassen sich einfach in ihrer Breite verändern. Die Kanten werden aus einer festlegbaren Anzahl an Zylindern zusammengesetzt, welche, ähnlich wie bei einer Kette, von Startpunkt bis Zielpunkt miteinander verbunden sind. Abbildung 3.9 illustriert dies, indem einige dieser Zylinder halb-transparent dargestellt werden. Der Bogen wird erzeugt indem die Zylinder über eine halbe Sinuskurve platziert werden. Die Zylinder werden je nach Steigung an der platzierten Stelle entsprechend angewinkelt, so dass die Illusion eines verbundenen Rohres

[2] Im SEE Projekt wird jedoch von anderen Teilnehmern an einer Erweiterung gearbeitet, welche dies zukünftig ermöglichen soll.

entsteht. Die Positionierung über diese einfache mathematische Funktion ermöglicht ein sehr schnelles Anpassen der Kantenbögen und der Verbindungspunkte während der Laufzeit. Dies ermöglicht die aktive Veränderung der Position und Skalierung der Kanten. Die Höhe des Bogens ist an die Breite der Kante gekoppelt. Je breiter eine Kante ist, desto höher geht auch die Kante. Dies verhindert für viele Fälle, dass mehrere Kanten sich überlappen. Eine komplette Überschneidungsfreiheit kann so jedoch nicht garantiert werden. Für die Animation der Kanten wurde ein Shader verwendet, welcher basierend auf dem aktuellem Zeitpunkt, alternierend zwei unterschiedliche Farben der Zylinder darstellt, sodass diese die Zylinder entlang laufen. Die Farben sind während der Laufzeit anpassbar und werden für das Hervorheben der Kanten genutzt, wenn sie ausgewählt sind oder mit dem Mauszeiger berührt werden. Ein Nachteil dieser Darstellung ist, dass Kanten mit gleichem Start- und Endpunkt keinen schönen Bogen aufspannen, sondern nur einen Turm bilden. Beispielsweise in Abbildung 3.3 auf Seite 35 ist dieses Artefakt unten links in der Abbildung zu sehen.

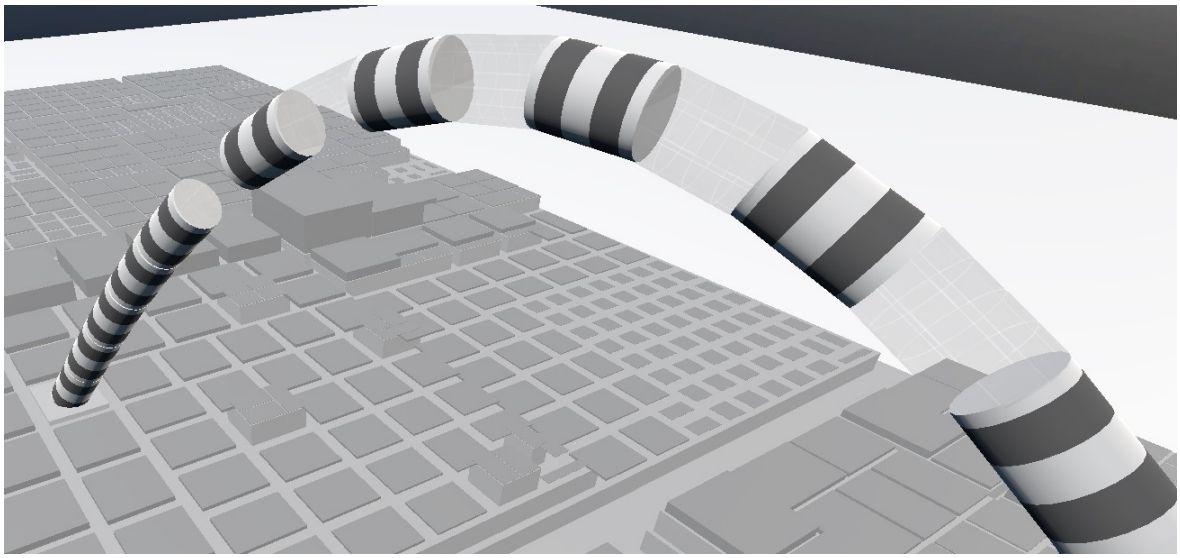


Abbildung 3.9: Aufbau einer Kante mit halbtransparenten Segmenten

Quell-Code Der Quellcode der Visualisierung in SEE befindet sich auf dem USB-Stick im Anhang. Eine genauere Beschreibung wo er zu finden ist befindet sich in Anhang A.1 auf Seite 149. Insbesondere ist dort vermerkt, welcher Quell-Code Eigenleistung war, da dieser Quell-Code in das Gesamtprojekt integriert ist.

3.6 Profiler

Es wurde ein Java-Profiler implementiert, welcher zwei unterschiedliche Profiling-Modi unterstützt: Sampling und Instrumentierung. Wie in Abschnitt 2.2.2 beschrieben, funktioniert ein Sampling-Profiler, indem in kleinen Abständen der aktuelle Zustand der Anwendung abgefragt wird und so Hot-Spots ermittelt werden. Eine Messung der Methodenlaufzeiten findet nicht statt. Die Instrumentierung fügt in laufenden Anwendungen zusätzlichen Quellcode ein, welcher die exakte Laufzeit und Aufrufanzahl von einzelnen Methoden misst.Beide Profiler funktionieren für Java-Anwendungen, indem sie sich mit einer bereits laufenden „Java Virtual Machine“ (JVM) verbinden. Eine Instrumentierung vor Start der Anwendung ist nicht notwendig.

Der Profiler ist eine separate Anwendung von SEE und nicht direkt integriert, da SEE in C# implementiert ist. Da Java-Anwendungen untersucht und instrumentiert werden sollen, eignet sich eine Implementierung in Java besser. Der Profiler lässt sich über eine grafische Oberfläche benutzen.Der Quellcode beider Profiler befindet sich auf dem USB-Stick im Anhang. Eine genauere Beschreibung wo er zu finden ist befindet auch hierzu sich in Anhang A.1 auf Seite 149.

3.6.1 Erhobene Metriken

Im Folgenden bezeichnet M einen Methodenaufruf (*Caller* und *Callee)* oder eine einzige Methode, ohne den Aufrufkontext (nur Callee). Für beide Varianten werden jeweils alle Statistiken berechnet, da sie für die Darstellung der Kanten und der farblichen Hervorhebung der Methoden-Blöcke beide notwendig sind. M bezieht sich nicht auf einen einzige Durchführung, sondern fasst die Menge aller Aufrufe einer Methode zusammen. Über die Menge der Aufrufe und ihre einzelnen, gemessenen Laufzeiten können dann Statistiken zusammengefasst werden. Der Sampling-Profiler kann die Laufzeit einzelner Methoden nicht bestimmen. Er ermittelt lediglich Anzahl der Beobachtungen, die $samples(M)$. Die Anzahl der Samples gibt approximativ an, in welchen Methoden sich die Anwendung am häufigsten befunden hat. Relativ zu der am häufigsten beobachteten Methode kann hiermit die $heat(M)$ berechnet werden. Sie entspricht der „Hitze“ in der Darstellung. Sie gibt an, wie stark die Metrik in der Visualisierung dargestellt wird. Die Methoden mit der höchsten $heat(M)$ bieten potentiell das größte Optimierungspotential, da sich die Anwendung in ihnen am häufigsten aufgehalten hat.

Zusammenhang zwischen gesampleter Heat und Instrumentierung Sei $\hat{M}$ die Menge aller beobachteten Methoden beziehungsweise Aufrufe M, dann ist die, mit dem Sampling-Profiler erhobene Heat, wie folgt definiert:

$$\text{heat}_s(M) = \frac{\text{samples}(M)}{\max\{\text{samples}(M') | M' \in \hat{M}\}}$$

Die mit dem Sampler erhobene *Heat* wird innerhalb der Anwendung als *Sampled Heat* bezeichnet. Der Instrumentation-Profiler erhebt statt den Samples exakte Laufzeiten, ermittelt Statistiken über sie, und zählt die Anzahl der Aufrufe. Sind Aufrufanzahl calls(M) und die mittlere Laufzeitdauer mean(M) einer Methode durch das instrumentierte Profiling bekannt, kann die *Heat* mit diesen Informationen exakt berechnet werden:

$$\text{total}(M) = \text{calls}(M) \cdot \text{mean}(M)$$

$$\text{heat}(M) = \frac{\text{total}(M)}{\max\{\text{total}(M') | M' \in \hat{M}\}}$$

Durch die unterschiedlichen Messverfahren sind heat(M) und $\text{heat}_s(M)$ jedoch nicht identisch. Insbesondere ist die, mit dem Sampling-Profiler, berechnete *Heat* nur eine Approximation. Bei einem hinreichend langem Beobachtungszeitraum kann jedoch davon ausgegangen werden, dass die Metriken in der Visualisierung ähnliche Darstellungen erzeugen. Auf diesen Zusammenhang wird auf Seite 67 in Abschnitt 4.1.1 noch einmal kurz, mit einem Visuellen Beispiel, eingegangen.

Nachteil der Heat ist, dass an ihr nicht erkennbar ist, ob eine Methode einen hohen Wert hat, weil sie häufig aufgerufen wurde oder weil sie eine hohe Laufzeit hat. Insbesondere können Methoden mit einer hohen Laufzeit durch den Sampling-Profiler übersehen werden, wenn die Methode nur selten aufgerufen wird.

Der Instrumentation-Profiler kann die Laufzeit einzelner Methoden ermitteln und so langsame Methoden besser erkennen. Durch die Berechnung der Quantile kann er Methoden identifizieren, welche nur in seltenen Fällen langsam sind. Im Instrumentation-Profiler werden die Statistiken sowohl für die *total time*, als auch für die *self time* ermittelt. Die Berechnung der Metriken folgt der, in den Anforderungen gewählten, Definition. Da es zusätzlich einen Modus gibt, welcher die Quantile approximiert, wird die Berechnung dieser Metriken in Abschnitt 3.6.3 näher beschrieben.

3.6.2 Sampling-Profiler

Der Sampling-Profiler überprüft in festlegbaren Zeitintervallen den aktuellen *Stack-Trace* der laufenden Threads des untersuchten Prozesses ab und speichert diese zwischen. Hierfür wird die `HotSpotVirtualMachine`-Komponente der JVM HotSpots Schnittstelle verwendet, welche von einer laufenden JVM bereitgestellt wird. Abbildung 3.10 illustriert den Kommunikationsweg zwischen der JVM des Profilers und der JVM der getesteten Anwendung (System under Test). Die `HotSpotVirtualMachine` liefert auf Anfrage des Samplers die notwendigen *Stack-Traces*, welche dann in einem separaten Thread verarbeitet werden, um die untersuchte Anwendung möglichst wenig zu beeinflussen.

Abbildung 3.11 zeigt die Benutzeroberfläche des Sampling-Profilers. Zu sehen ist hier eine Tabelle aller beobachteten Methodenaufrufe mit *Caller* und *Callee* während eines laufendem Sampling-Prozesses. Rechts ist die Anzahl der gesehenen Samples gelistet.

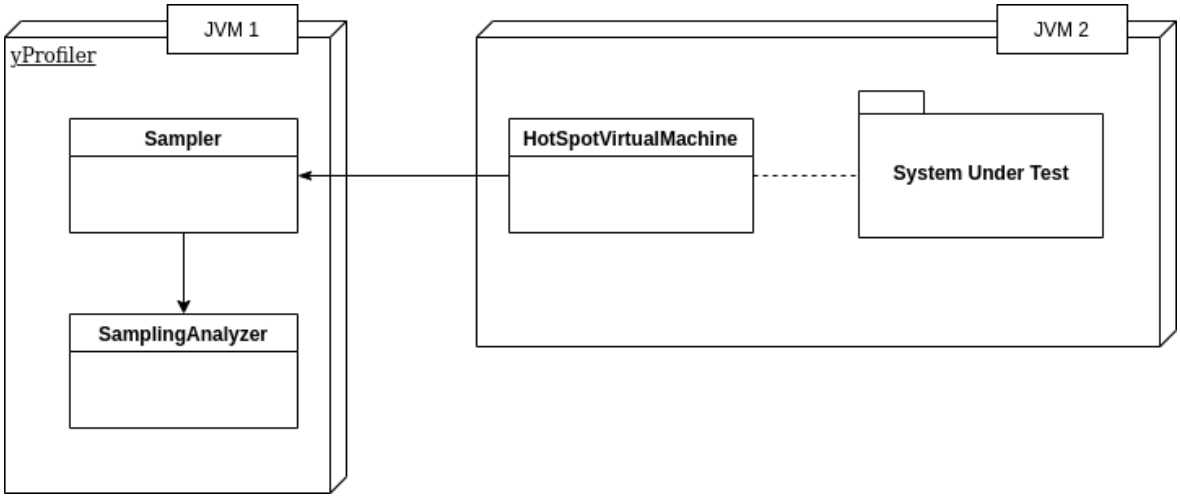


Abbildung 3.10: Aufbau des Sampling-Profilers. Die Pfeile geben den Informationsfluss der abgefragten *Stack-Traces* an.

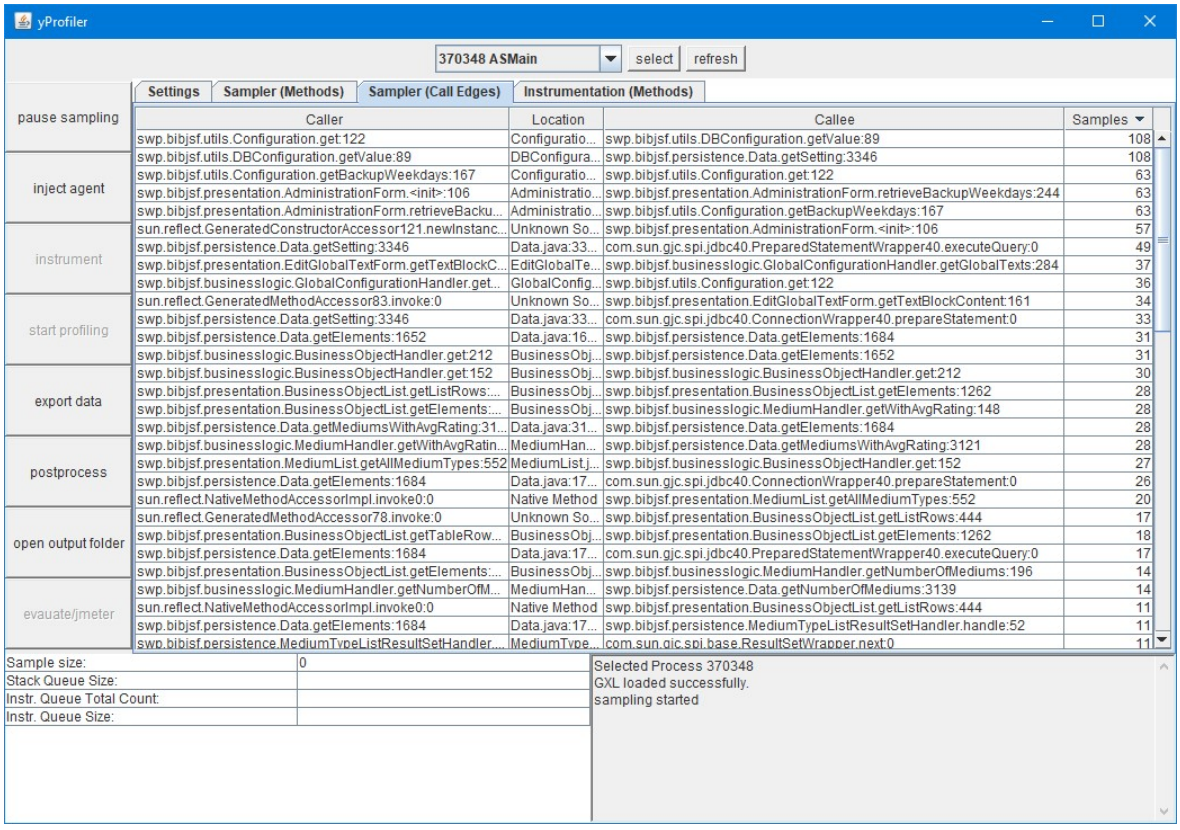


Abbildung 3.11: Oberfläche des „yProfiler“ mit angezeigten Aufrufkanten, welche durch den Sampling-Profiler erhoben wurden.

3.6.2.1 Benutzung des Sampling-Profilers

Der Profiler wird verwendet, indem die Java-Anwendung, welche untersucht werden soll gestartet wird. Nach Start der Anwendung kann der Prozess in dem Auswahlmenü am oberen Rand der „yProfiler"-Oberfläche in Abbildung 3.11 ausgewählt werden. Um den Sampling-Profiler zu starten wird nur ein Mausklick auf *start sampling* am linken Fensterrand benötigt. Dieser Button wird danach zu *pause sampling* umbenannt. Während dem Sampling-Vorgang tauchen die Ergebnisse bereits in der Tabelle auf. Zum Exportieren der Daten am Ende, wird auf *export data* geklickt. „Open output folder" öffnet den Ordner, in dem die exportierten Daten liegen. Die restlichen Buttons dienen der Bedienung des Instrumentation-Profilers und werden im entsprechenden Kapitel beschrieben. Die zwei Panele in der unteren Hälfte der Oberfläche zeigen Debug-Informationen und Log-Informationen des Profilers an. Die grafische Oberfläche des Profilers ist nicht Schwerpunkt dieser Arbeit und wurde nur zur vereinfachten Benutzung des Profilers und für eine schnelle Darstellung der Daten während der Profiler-Laufzeit eingebaut. Die Daten können zwar bereits in der Tabellenform analysiert werden. Die grafische Oberfläche des Profilers wurde in Rahmen dieser Arbeit jedoch nicht durch andere Personen auf Benutzbarkeit überprüft.

3.6.2.2 Sampling-Rate

Der Abstand, mit der die Stack-Samples abgefragt werden ist frei wählbar. Die Standardeinstellung liegt bei 50 Millisekunden. Wie Mytkowicz et al. [72] feststellten, ist regelmäßig Sampling jedoch nicht hinreichend zufällig und kann das Laufzeitverhalten der untersuchten Anwendung unerwünscht beeinflussen, dadurch dass die Anwendung in regelmäßigen Abständen pausiert wird. Der eingestellte Zeitabstand ist deshalb nicht fix, sondern wird für jede Iteration zufällig zwischen 0 und der angegebenen Abstand ausgewählt. Da der Einfluss auf die Anwendung auch beim Sampling recht stark ausfallen kann, wie in der Evaluation in Abschnitt 4.2.1.4 sichtbar wird, sollte er je nach untersuchter Anwendung entsprechend gewählt werden.

3.6.2.3 Analyse der Stack-Traces

Bei der Verarbeitung der Stack-Traces werden die „Stack-Traces" zeilenweise eingelesen und die einzelnen Methodenaufrufe gesammelt. Ein Methodenaufruf entspricht jeweils zwei Zeilen eines Stack-Traces, wie er in Textausschnitt 3.1 exemplarisch zu sehen ist. Eine tiefere Zeile entspricht dem *Caller*, der aufrufenden Methode und die vorherige Zeile entspricht dem *Callee*, der aufgerufenen Methode. So ist die Methode `getTableRow` auf Zeile 18 der Caller der Methode `getCreator` auf Zeile 17. Es werden alle Aufrufe zwischen jeweils zwei Methoden ausgelesen und für diese gezählt, wie häufig diese in allen gesammelten Stack-Traces vorkommen. Nur Aufrufe, in denen mindestens der Caller oder der Callee im beobachteten Bereich liegen, werden gezählt. Die Gesamtsumme der Samples einer Methode mit mehreren eingehenden Aufrufkanten wird für die Visualisierung erst in SEE berechnet.

Der beobachtete Bereich ist durch reguläre Ausdrücke in der Einstellungen des Profilers definierbar. In der Regel wird der Paketname angegeben. Für die Anwendung *libAwesome*, zu welcher der abgebildete *Stack-Trace* gehört, eignet sich folgender Ausdruck: `swp\.bibjsf.*`.

```
"http−listener−1(5)" #33 daemon prio=5 os_prio=0 tid=0x0000027bad806800 nid=0x31a4 runnable [0x000000b84f6fd000]
   java.lang.Thread.State: RUNNABLE
        at java.lang.Object.hashCode(Native Method)
        at java.util.WeakHashMap.hash(WeakHashMap.java:298)
        at java.util.WeakHashMap.put(WeakHashMap.java:449)
        at org.apache.derby.client.am.ClientConnection.prepareStatementX(Unknown Source)
        at org.apache.derby.client.am.ClientConnection.prepareStatement(Unknown Source)
        − locked <0x00000000fe81a820> (a org.apache.derby.client.net.NetConnection)
        at com.sun.gjc.spi.base.ConnectionHolder.prepareStatement(ConnectionHolder.java:586)
        at com.sun.gjc.spi.jdbc40.ConnectionWrapper40.prepareCachedStatement(ConnectionWrapper40.java:255)
        at com.sun.gjc.spi.jdbc40.ConnectionWrapper40.prepareCachedStatement(ConnectionWrapper40.java:52)
        at com.sun.gjc.spi.ManagedConnectionImpl.prepareCachedStatement(ManagedConnectionImpl.java:992)
        at com.sun.gjc.spi.jdbc40.ConnectionWrapper40.prepareStatement(ConnectionWrapper40.java:173)
        at swp.bibjsf.persistence.Data.getElement(Data.java:1606)
        at swp.bibjsf.businesslogic.BusinessObjectHandler.get(BusinessObjectHandler.java:86)
        at swp.bibjsf.models.Medium.getMediumType(Medium.java:945)
        at swp.bibjsf.models.Medium.getCreator(Medium.java:1058)
        at swp.bibjsf.presentation.MediumList.getTableRow(MediumList.java:233)
        at swp.bibjsf.presentation.MediumList.getTableRow(MediumList.java:59)
        at swp.bibjsf.presentation.BusinessObjectList.getTableRows(BusinessObjectList.java:366)
        at sun.reflect.GeneratedMethodAccessor143.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:498)
        at javax.el.BeanELResolver.getValue(BeanELResolver.java:362)
        at com.sun.faces.el.DemuxCompositeELResolver._getValue(DemuxCompositeELResolver.java:180)
        at com.sun.faces.el.DemuxCompositeELResolver.getValue(DemuxCompositeELResolver.java:208)
        at com.sun.el.parser.AstValue.getValue(AstValue.java:139)
        at com.sun.el.parser.AstValue.getValue(AstValue.java:203)
```

Text 3.1: Beispiel eines *Stack-Traces* aus *libAwesome*

3.6.2.4 Überladene Methodennamen

In Java können Methoden innerhalb einer Klasse mit gleichem Namen mehrfach definiert werden, solange sie sich durch ihre Parameter eindeutig identifizieren lassen. Diese Methoden werden „überladene Methoden“ genannt. Die Methodenparameter sind in den *Stack-Traces* jedoch nicht enthalten. Die Zeilennummern im *Stack-Trace* geben die Position der Methodenaufrufe innerhalb des *Callers* an, jedoch nicht die erste Zeilennummer der entsprechenden Methode. Die Zeilennummer kann deshalb nicht ohne Weiteres verwendet werden, um überladene Methoden auseinander zu halten.

Um aktive Zeilen in der Visualisierung darzustellen wird zusätzlich ein Zähler für jede auftauchende Zeile geführt, sodass in der Visualisierung die entsprechenden Code-Zeile, wie in Abbildung 3.7 auf Seite 40 hervorgehoben werden können.

Es gibt im Profiler deshalb zwei unterschiedliche Vorgehen um überladene Methoden handhaben. Im Standardfall werden Aufrufe überladener Methoden nicht unterschieden und in der Auswertung als identisch behandelt. Werden vor Start des Profilers zusätzliche Informationen über den Quellcode importiert, kann anhand der Zeilennummer aus dem *Stack-Trace* die Methode eindeutig identifiziert werden, um so überladene Methoden zu trennen.

Die zusätzlichen Quell-Code Informationen werden über das GXL Format eingelesen, welches auch für die Visualisierung der Software-Stadt benötigt wird. Diese GXL-Dateien werden durch ein vorhandenes statische Analysetools von *Axivion* [99], welches innerhalb der Arbeitsgruppe verfügbar ist, erstellt. In diesen Informationen sind, neben dem vollständigen Aufrufgraphen der Anwendung, auch die Startzeilen aller Methoden enthalten. Gegeben einer beliebigen Zeile und eines Dateinamens, wird so die Methode gesucht, welche diese Zeile enthält. Auch die eindeutigen Methodensignaturen werden zusätzlich zwischengespeichert, damit sie in der Ausgabe des Profilers mit ausgegeben werden können. Sie werden dann in Visualisierung genutzt, um sie den visuellen Methoden-Objekten zuzuordnen. Für das Einlesen des GXL-Formates wurde ein GXL-Parser mit den, für diesen Zweck notwendigen Funktionen implementiert. Dieser Parser ist unter Umständen für kommende Arbeiten erweiterbar.

3.6.2.5 Threads

Es wird in diesem Profiler nicht nach Threads separiert. Dies wäre technisch möglich, und nachträglich erweiterbar, wurde jedoch für diese Arbeit nicht als notwendig erachtet, da hauptsächlich Web-Anwendungen getestet wurden, deren Threads häufig die selben Funktionen erfüllen. Für andere Anwendungstypen ist diese Separierung jedoch durchaus sinnvoll. Ein Java-Thread kann sich zudem in den Zuständen `NEW`, `RUNNABLE`, `WAITING`, `BLOCKED`, `TIMED_WAITING` und `TERMINATED` befinden. Eine Aufteilung der gemessenen Zeiten auf die Thread-Zustände wäre ebenfalls denkbar, wurde jedoch nicht implementiert, um die Komplexität der Darstellung nicht zu erhöhen. Insbesondere kann der Instrumentation-Profiler, welcher in Abschnitt 3.6.3 beschrieben wird, den Thread-Status nicht in seine Ergebnisse integrieren. Eine Auftrennung dieser Informationen im Sampling-Profiler würde also zu inkonsistenten Messwerten führen.

3.6.2.6 Ausgabe

Der Sampling-Profiler gibt eine CSV-Datei mit einer Zeile für jede aufgetretene Aufrufkante aus. Hier enthalten sind Methodennamen, Dateiname, Zeilennummer von *Caller* und *Callee* und die Anzahl der gezählten Samples pro Aufruf, welche in der Visualisierung als *Sampled Heat* dargestellt werden. Zusätzlich zu der Sample-Anzahl werden noch die zeilenweisen Zähler mit ausgegeben, welche für die Hervorhebung der Zeilen in den Quell-Text-Fenstern verwendet werden. Der Ausgabeordner ist, wenn nicht in den Einstellungen anders eingestellt, `C:/temp/yprofiler/`.

3.6.3 Instrumentation-Profiler

Der zweite Profiling-Modus verwendet Techniken, um die untersuchte Anwendung zu instrumentieren. Hierbei werden untersuchten Methoden während der Laufzeit so verändert, dass sie zusätzlichen Code zum Messen der Ausführungszeiten ausführen.

Hierfür wird zunächst ein „Agent“ in die untersuchte JVM injiziert. Abbildung 3.12 zeigt die Kommunikation zwischen den beiden JVMs und der Teilkomponenten. Nachdem der `YProfilingAgent` in die untersuchte JVM injiziert und dort gestartet wurde, wird eine Netwerkverbindung zwischen `YProfilingAgent` und `IPCServcer` („Inter Process Communication Server“) hergestellt. Über diesen Kanal werden Steuerbefehle ausgetauscht und die gesammelten Daten zurückgeschickt. Die JVM ermöglicht es den Bytecode von Java-Methoden, während der Laufzeit einer Anwendung zu verändern. Über den `YProfilingAgent` wird diese Instrumentierung ausgelöst. Mit der Bibliothek *Javassist* [100] wird der Bytecode verändert und neu geladen. Bytecode, welcher erst nach dem Zeitpunkt der Injektion des Agenten geladen wird, wird automatisch erkannt und instrumentiert.

Durch die Verwendung von *Javassist* ist die Instrumentierung recht einfach, da eine Arbeit, direkt mit dem Bytecode, nicht nötig ist. Zum Hinzufügen von Code in einer Methode werden lediglich die Methoden `insertBefore` und `insertAfter` aus *Javassist* aufgerufen, welche Java-Code als Parameter erwarten. Der Quell-Code 3.2 zeigt exemplarisch für eine Methode, welcher Code hinzugefügt wurde. Es wird jeder Methodeninhalt in einem neuen `try-finally`-Block umschlossen, welcher mit einem Aufruf von `reportMethodStart` beginnt und `reportMethodEnd` endet. Die Verwendung von `finally` ist wichtig, da hierdurch alle möglichen Austrittspunkte der unveränderten Methode, inklusive Exceptions, abgedeckt werden.

Die instrumentierten Methoden rufen zu Beginn und zum Ende der Ausführung der Metho-

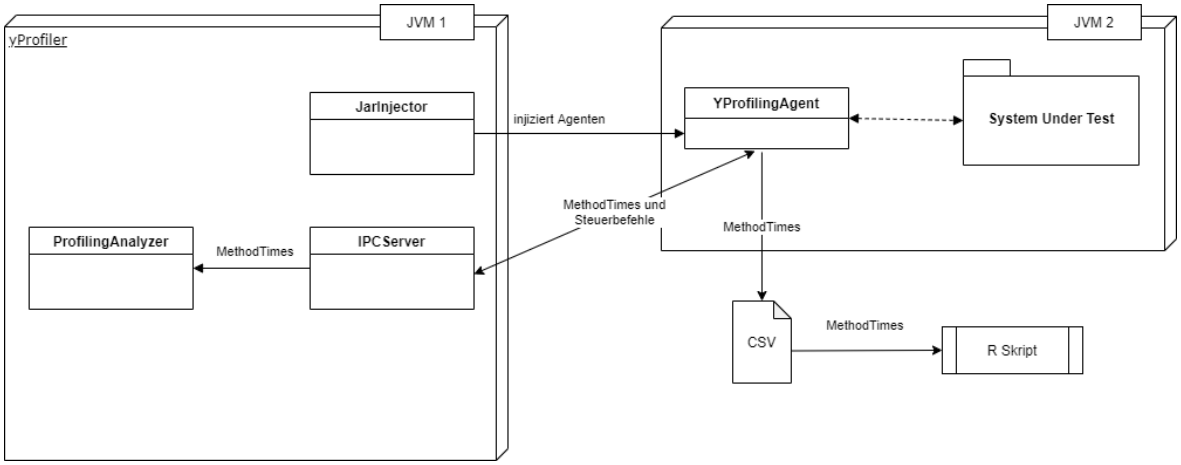


Abbildung 3.12: Aufbau des Profilers mit Instrumentierung

```
1  public int exampleMethod(int c) {
2      YProfilingAgent.reportMethodStart(...)
3      try{
4          int a = 4;
5          int b = a + c;
6          return b + 4;
7      }finally{
8          YProfilingAgent.reportMethodEnd(...)
9      }
10 }
```

Text 3.2: Beispiel einer Methode mit injiziertem Profiling-Code (Java-Pseudocode). Die markierten Zeilen wurden durch den Agenten hinzugefügt.

de den `YProfilingAgent` auf. In ihm wird dann die Laufzeit gemessen. Quell-Text 3.3 zeigt in Java-ähnlichem Pseudo-Code beide aufgerufenen Methoden. Zusätzlich zum Zwischenspeichern der Methodenlaufzeit wird in jedem Thread ein Stack verwaltet, mit welchem die Aufrufhierachie zwischengespeichert wird. Hierdurch kann, zusätzlich zum Namen der aktuellen Methode, auch der Name des *Callers* ermittelt und ausgegeben werden.

Zusätzlich zu der Gesamtlaufzeit wird die *self time* berechnet, indem nach Ausführungsende einer Methode ein Zähler für die *outer time* im vorherigem Stack-Eintrag, dem *Callee*, erhöht wird. Diese *outer time* wird für die Berechnung der *self time* von der *total time* subtrahiert. Diese Subtraktion ist im Pseudo-Code in Quell-Text 3.3 zu sehen. Nicht instrumentierte Methoden können nicht eingerechnet werden und werden deshalb als *self time* gewertet. Dies entspricht dem Verhalten anderer Profiler, wie in Abschnitt 2.2.2.3 beschrieben wurde. Ein `Thread.sleep(1000)` würde demnach als die *self time* der Methode um eine Sekunde erhöhen, obwohl die Wartezeit eigentlich innerhalb der Methode `sleep` stattfindet.

Für die Weitererarbeitung dieser Daten gibt es mehrere Vorgehen, denn es können je nach Auslastung der Anwendung sehr große Datenmengen anfallen. Die Profiling-Mechanismen sollen eine möglichst geringe Menge an zusätzlicher Laufzeit benötigen um das Laufzeitverhalten der Anwendung möglichst wenig zu beeinflussen. Aus diesem Grund werden die Daten nicht innerhalb der JVM der untersuchten Anwendung weiterverarbeitet, sondern in unkomprimierter Form ausgegeben. Abbildung 3.12 zeigt zwei, von dem Agenten ausgehende, Kanten für die beiden möglichen Ausgaberichtungen.

```
1  class TracedData{
2      public String name;
3      public long startTime;
4      public long outerTime = 0;
5  }
6
7  // Für jeden Thread existiert ein eigenes Stack−Objekt.
8  Stack<TracedData> stack = new Stack<TracedData>();
9
10 void reportMethodStart(String name, long startTime){
11     TracedData t = new TracedData();
12     t.name = name;
13     t.startTime = startTime;
14     stack.add(t)
15 }
16
17 void reportMethodEnd(long endtime){
18     TracedData t = stack.pop();
19
20     long runtime = endtime − t.startTime;
21     long selftime = runtime − t.outerTime;
22     String caller = t.name;
23
24     // füge Callee outerTime hinzu
25     stack.peek().outerTime += runtime;
26
27     // Ausgabe von gemessenen Daten.
28     output(caller, t.name, runtime, selftime);
29 }
```

Text 3.3: Vereinfachter Java-Code der Methoden reportMethodStart und reportMethodEnd im Profiling-Agent.

Die Daten können direkt in eine CSV-Datei geschrieben werden oder über den Netzwerk-Kommunikationskanal zurück an die Profiler-JVM übertragen werden. Das Schreiben in die CSV-Datei ist mit dem Java Standard Logger umgesetzt, da diese Implementierung bereits für viele Schreibzugriffe gut geeignet ist. Das Senden der Daten an die Profiler-JVM wird via Standard-Java Netzwerk-Sockets gelöst.

In beiden Fällen findet keine Komprimierung der Daten statt, um keinen zusätzlichen Rechenaufwand innerhalb des Profiling-Agents innerhalb der JVM zu erzeugen. Dies führt jedoch zu großen Datenmengen, insbesondere wenn die Ausgabe in CSV-Dateien verwendet wird.

Die CSV-Dateien werden im Nachhinein von einem R-Skript ausgewertet. Dieses R-Skript ist bei Bedarf leicht erweiterbar um die Auswertungen zu verfeinern. Wird das Netzwerk-Logging verwendet, werden die Methodenlaufzeiten vom `ProfilingAnalyzer` in einem separaten Thread gesammelt und dort weiterverarbeitet, jedoch nicht in eine Datei geschrieben. Hier werden lediglich die ausgewerteten Statistiken in Dateien geschrieben.

3.6.3.1 Benutzung des Instrumentation-Profilers

Abbildung 3.11 auf Seite 47 stellt auf der linken Seite der Benutzeroberfläche Knöpfe zum Starten des Sampling-Profilers und des Instrumentation-Profilers bereit. Um den Instrumentation-Profiler zu starten muss zunächst mit *inject agent* der `YProfilingAgent` in die zu untersu-

chende JVM injiziert werden. Sobald der Agent gestartet wurde und sich mit der JVM des Profilers über eine Netzwerkverbindung verbunden hat wird der *instrument*-Knopf auswählbar. Sobald dieser angeklickt wird, werden alle Methoden, entsprechend der Einstellungen im *Settings*-Tab instrumentiert. Sobald die Instrumentierung abgeschlossen ist, kann das Profiling über *start profiling* gestartet werden. Über den selben Knopf kann es danach wieder pausiert werden. Der Knopf *postprocess* startet die Verarbeitung der Rohdaten mit einem optionalen integrierten R-Skript, welches in Abschnitt 3.6.3.5 näher beschrieben wird.

3.6.3.2 Top-n Instrumentierung

Um die Menge an Methoden zu reduzieren, welche instrumentiert werden, kann in den Einstellungen der reguläre Ausdruck zum Filtern der Methoden angepasst werden, sodass er nur einzelne Teilpakete oder bestimmte Methodennamen auswählt. Hierdurch kann der zusätzliche Rechenaufwand stark reduziert werden.

Mit der **top-n**-Methode, welche über die Einstellungen aktivierbar ist, können die n häufigsten Methoden aus den Messungen des Sampling-Profilers ausgewählt werden, sodass ausschließlich die Methoden instrumentiert werden, welche während einem vorangegangenem Sampling-Profiling am auffälligsten waren. Mit der Einstellung **profile Call-Tails** werden zusätzlich zu den häufigsten Methoden auch alle Methoden instrumentiert, welche die ausgewählten Methoden direkt oder indirekt aufgerufen haben. Hierdurch bleibt der Aufrufgraph für die Visualisierung erhalten. Diese Kombination aus Sampling-Profiler und Instrumentierung ermöglicht eine starke Reduktion der Instrumentierung mit der Intention, den Overhead durch den Profiler zu reduzieren.

3.6.3.3 Einstellungen

Abbildung 3.13 zeigt die verfügbaren Einstellungen des Profilers. Die Standard-Einstellungen können im Quell-Code des Profilers geändert werden. Hier können für beide Profiler in einer vereinten Oberfläche Einstellungen vorgenommen werden. Es können reguläre Ausdrücke angegeben werden, welche als Filter für die zu untersuchenden Methoden verwendet werden. Es kann eingestellt werden, welcher Modus für das Logging der Performance-Daten verwendet werden soll, welche Dateinamen verwendet werden und die GXL-Datei kann geladen werden. Die Einstellungen sind in den ihnen entsprechenden Abschnitten näher erklärt. Insbesondere können hier auch die Parameter für die Approximationen verändert werden, welche in Abschnitt 3.6.4 beschrieben werden. In der letzten Zeile wird automatisch berechnet, welchen Wertebereich die Approximation abdeckt.

3.6.3.4 Ausgabe

Die vom Profiler ausgegebenen Statistiken beinhalten, pro Methode und pro Methodenaufruf, die in den Anforderungen festgelegten Metriken: Aufrufanzahl, arithmetisches Mittel, ausgewählte Quantile, Minimum und Maximum. Diese werden jeweils in CSV-Dateien ausgegeben. Es werden nur die Quantile für 50% (Median), 90%, 95% und 99% ausgegeben. Eine Ergänzung der Ausgabe um weitere Quantile ist jedoch durch minimale Anpassung des Quellcodes möglich. Die durch die R-Skripte errechneten Statistiken werden im gleichen Ordner ausgeben. Der Ausgabeordner ist, wenn nicht in den Einstellungen anders eingestellt, `C:/temp/yprofiler/`.

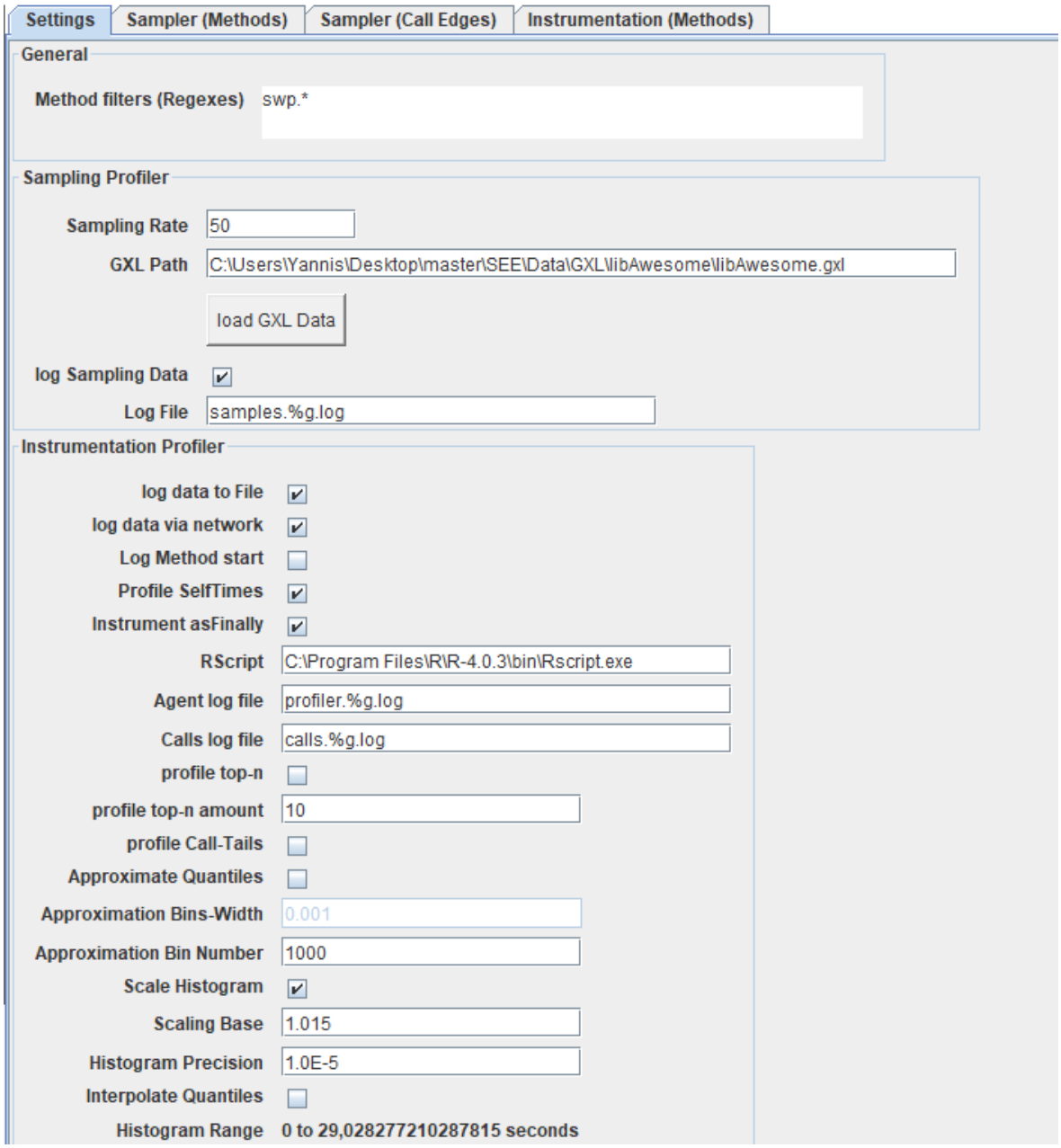


Abbildung 3.13: Einstellungen im Profiler

3.6.3.5 Berechnung der Statistiken

Der Profiler kann die Daten, welche über das Netzwerk an ihn übertragen wurden oder in der Log-Datei verfügbar sind auf unterschiedliche Arten weiterverarbeiten. Alle drei Modi haben als Ausgabe CSV-Dateien, welche die in Abschnitt 3.1 und 3.6.1 festgelegten Metriken enthalten. Dies sind ausgewählte Quantile, Median, Mittelwert, Minimum, Maximum und die Aufrufanzahl der Methoden als *total time* und *self time*. Die Statistiken werden in allen drei Methoden sowohl für Aufrufkanten, aber auch für Methoden ohne ihren *Caller* berechnet. Die kontextlosen Methodenlaufzeiten werden in der Visualisierung für die Einfärbung einzelner Methoden-Blöcke verwendet.

Modus 1: Exakte Berechnung mit einem R-Skipt Die exakte Auswertung der, in die CSV gespeicherten Rohdaten ist durch die Nutzung eines integrierten R-Skriptes möglich. Das eingebundene R-Skript berechnet für alle Methoden und Aufrufkanten die Statistiken und speichert die Ergebnisse in CSV-Dateien. Nachteil ist die zusätzliche Abhängigkeit von R und der Paketsammlung *tidyverse*, welche zusätzlich installiert sein müssen. Für kürzere Tests ist diese Variante gut geeignet. Insbesondere ist so eine Nachbearbeitung mit einem angepassten R-Skript möglich, da im Nachhinein alle Rohdaten weiterhin verfügbar sind. Bei einem größeren Beobachtungszeitraum werden die Log-Dateien jedoch sehr groß, sodass die nachträgliche Berechnung der Statistiken mit dem R-Skript zu zeitintensiv wird. Das verwendete R-Skript ist im Quell-Code des yProfilers enthalten. Dieser ist auf dem USB-Stick Anhang A.1 zu finden.

Modus 2: Berechnung im yProfiler Alternativ können die Rohdaten auch im Arbeitsspeicher der Profiler-JVM gesammelt werden und dort nach der Messung ausgewertet werden. So werden keine großen Mengen an Rohdaten auf die Festplatte geschrieben und es werden dennoch exakte Ergebnisse berechnet. Ein möglicher Nachteil gegenüber der Berechnung mit dem R-Skript ist jedoch, dass die Daten über einen Netzwerk-Socket vom Agenten in den Profiler-Prozess übertragen werden. Dieser Overhead könnte bei großen Datenvolumen problematischer sein, als das Logging in eine Datei im Dateisystem. Insbesondere falls der Profiler über einen längeren Zeitraum eingesetzt werden soll, könnte großer Arbeitsspeicherverbrauch anfallen. In den Profiling-Aktivitäten, welche für die Evaluation dieser Arbeit durchgeführt wurden, kam es jedoch nicht zu Speicherproblemen.

Modus 3: Approximation im yProfiler Während des Profilings fallen große Datenmengen an. Schon in einem kurzen Test können Millionen an Messpunkten anfallen. Das exakte Berechnen des Medians und anderer Quantile wird mit wachsender Anzahl an Messpunkten sehr rechenintensiv, da alle Messpunkte sortiert werden müssen.

Für längere Tests können die Quantile deshalb approximiert werden, damit nicht alle Rohdaten zwischengespeichert und sortiert werden müssen. So kann potentiell die Berechnungszeit der Quantile und der Speicherverbrauch stark reduziert werden.

Hierfür wurden zwei Optionen in Betracht gezogen. Der „Remedian“ von Rousseeuw und Bessett [101] berechnet hierarchisch auf kleinen Teilmengen den Median und schiebt die errechneten Zwischenergebnisse in der Hierarchie weiter. So ist nur wenig Speicher nötig. Der Nachteil dieser Methode ist, dass Sie für jedes Quantil separat durchgeführt werden muss und, dass bereits während der Laufzeit Quantile, der Teilmengen errechnet werden müssen. Der Remedian wurde deshalb nicht implementiert.

Die implementierte Variante ist eine Einordnung aller Messwerte in ein Histogramm. Durch

dieses Histogramm muss, für die Berechnung der Quantile, lediglich durchgezählt werden, bis das gesuchte Quantil erreicht wird. Abschnitt 3.6.4 beschreibt dies näher. Nachteil hierbei ist, dass die Ergebnisse diskretisiert werden. Der Ergebnisbereich ist also auf feste Zeitschritte begrenzt. Zudem ist durch die Breite des Histogrammes auch der Gesamtbereich begrenzt. Letzteres ist in diesem Kontext weniger problematisch, da eine obere Grenze in der Regel gut festgelegt werden kann. Auch die Diskretisierung des Ergebnisbereiches ist vernachlässigbar, da in der Regel die exakten Werte für eine Analyse nicht notwendig sind, sondern nur das grobe relative Verhältnis der Methode zu anderen Methoden. Der Abschnitt 3.6.4 beschreibt ausführlich zwei unterschiedliche, implementierte Histogramm-Arten.

3.6.4 Quantilberechnung mit Histogrammen

Um nicht alle gemessenen Werte zwischenspeichern zu müssen wird ein Histogramm angelegt, in welches gemessene Methodenlaufzeiten einsortiert werden. Die Quantile können dann aus diesem Histogramm abgelesen werden, indem von links nach rechts durch alle Bin-Zähler gezählt wird, bis der Bin gefunden wurde, in welchen das gesuchte Element einsortiert wurde. Abbildung 3.14 zeigt dies am Beispiel des Medians. Die vertikale Linie zeigt den echten Median aller Messwerte. Der Algorithmus kennt die Gesamtanzahl der in das Histogramm einsortierten Elemente und zählt nun von links nach rechts alle Bins durch, bis die Hälfte erreicht ist. Die linke Hälfte ist blau und die Rechte rot eingefärbt. Hier kennt der Algorithmus nun lediglich den Wertebereich des aktuellen Bins, jedoch nicht den exakten Median. Der Median wird approximiert indem das arithmetische Mittel der Bin-Grenzen errechnet wird. Für andere Quantile funktioniert das Vorgehen analog, da der Median ebenfalls nur ein Quantil ist. Bei anderen Quantilen oder Perzentilen wird nur nicht bei der Hälfte, sondern bei der jeweils gesuchten Grenze gestoppt.

Das gesuchte Quantil kann exakt zwischen zwei, nicht notwendigerweise nebeneinander liegenden, Bins liegen. In diesem Spezialfall wird das arithmetische Mittel zwischen beiden Bins verwendet. Dieser Spezialfall wird in unterschiedlichen Definitionen von Quantilen unterschiedlich errechnet [97], ist in diesem Kontext jedoch nur von geringer Bedeutung. Die gleiche Interpolation zwischen zwei Werten wird auch in den beiden exakten Quantilberechnungsmodi des Profilers durchgeführt.

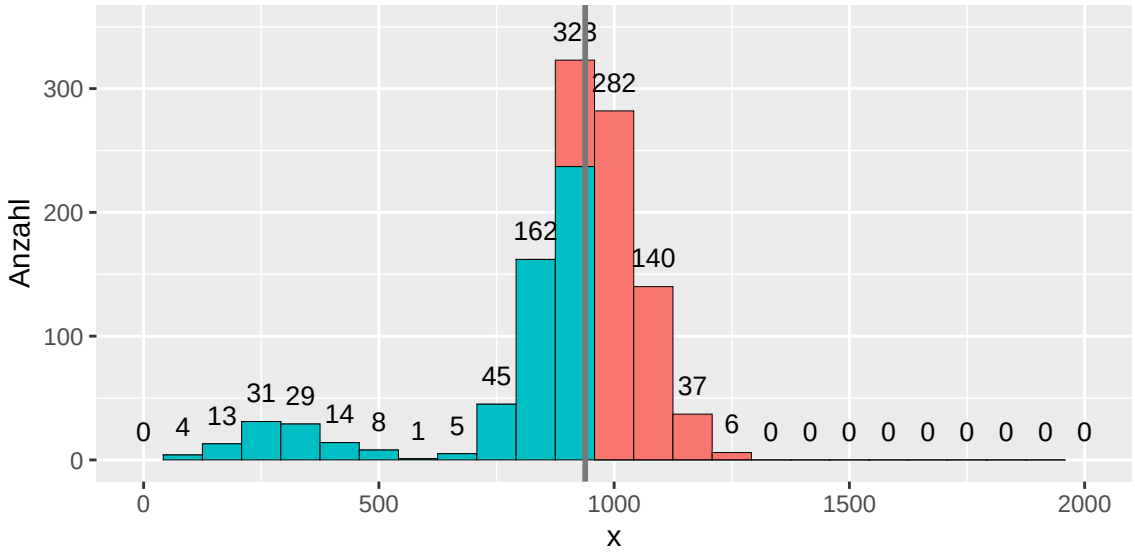


Abbildung 3.14: Berechnung des Medians mit einem Histograms

Für die Approximation der Quantile wird ein Histogram und ein Vorgehen zum schnellen Einsortieren von eingehenden Messwerte in dieses Histogramm definiert.

Ein Histogramm H für einen festen Wertebereich teilt diesen Wertebereich in disjunkte Bins (Teilbereiche) ein. Abbildung 3.14 zeigt diese Teilbereiche durch eingezeichnete Balken. Ein Histogramm besitzt eine endliche Menge an Bins. Die Bin-Grenzen b_i mit $i \in \mathbb{N}_0$ geben jeweils immer den niedrigsten Wert des Bins i an. Eine Zahl x würde dem Bin i zugeteilt werden, für welchen $b_i \leq x < b_{i+1}$ gilt. Die äußeren Ränder des Histograms sind offen. Zahlen die außerhalb der Bins des Histograms liegen werden in die äußersten Bins einsortiert. Im folgenden wird angenommen, dass alle Messwerte im Wertebereich des Histogramms liegen.

Jede Zahl x kann einem Bin i zugeordnet werden. Um Werte in der Implementierung schnell einem Bin zuordnen zu können wird eine Funktion $f : \mathbb{R} \mapsto \mathbb{N}_0$ definiert, welche diese Zuordnung von Zeitwerten auf Bins umsetzt. Eine weitere Funktion $F : \mathbb{N}_0 \mapsto \mathbb{R}$ dient als Inverse zu f und bildet von den Indizes der Bins zurück auf die Zeitwerte ab. Da f auf Histogram-Bins abbildet und dabei Runden muss, ist die inverse Funktion nur eine Approximation des Originalwertes.

Die Abweichung zwischen echtem und approximiertem Wert ist definiert als:

$$e(x) = F(f(x)) - x$$

Der relative Fehler, im Verhältnis zum echten Wert ist $\frac{e(x)}{x}$.

Um diese Abweichung möglichst gering zu halten, soll F auf den Erwartungswert der Werte eines des Bins abbilden. Sei für die eingehenden Zahlen $x \in [b_i, b_{i+1})$ die Zufallsvariable X_i definiert, so muss also gelten: $F[i] = E[X_i]$.

Unter der Annahme, dass die Zahlen innerhalb eines Bins gleichverteilt sind, kann als Erwartungswert der Mittelwert angenommen werden.

In der Implementierung sind die x-Werte lediglich ganze Zahlen und als Nanosekunden angegeben. Für die Theorie können sie jedoch ebenfalls als reelle Zahlen angenommen werden. Mit Hilfe dieser Abbildung werden die gesammelten Messewerte in der Anwendung in das Histogramm eingeordnet und für die jeweiligen Bins ein Zähler hochgezählt.

3.6.4.1 Konstante Bin-Breite

Bei einem linearen Histogramm haben die Bins alle die gleichen Abstände. Die Breite der Bins ist demnach konstant, wie in Abbildung 3.15 dargestellt. Die Bin–Breite w kann frei gewählt werden. Zusammen mit der Anzahl der Bins H_l definiert sich der Wertebereich des Histogramms von 0 bis $w \cdot H_l$. Für die Bin-Breite w sind die Bin-Grenzen b_i wie folgt definiert. Abbildung 3.15 illustriert dies für $w = 10$.

$$\begin{aligned} b_0 &= 0 \\ b_i &= b_{i-1} + w \\ &= i \cdot w \end{aligned}$$

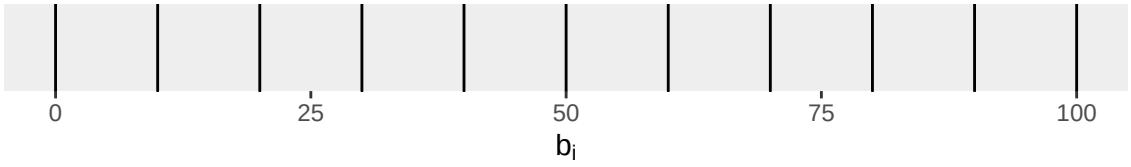


Abbildung 3.15: Darstellung der Bins mit einer Bin-Breite von 10

Die Funktionen f und F sind wie folgt definiert:

$$f(x) = \left\lfloor \frac{x}{s} \right\rfloor$$
$$F(i) = i \cdot w + \frac{w}{2}$$

Es wird also lediglich durch eine ganzzahlige Division der entsprechende Bin für einen Wert x bestimmt und mit F wieder auf den Erwartungswert zurückgerechnet.

Fehlerbereich Die stärkste Differenz zwischen echtem Wert x und der Approximation $F(f(x))$ tritt auf, wenn der richtige Wert am Rand eines Bins liegt. Der Fehler $e(x)$ entspricht also maximal der Differenz aus der Mitte des Bins bis zu einem der Ränder, also die Hälfte $\frac{w}{2}$

Wie in Abbildung 3.16 erkennbar, schrumpft der relative Fehler bei steigendem tatsächlichen Wert. Folgende Rechnung zeigt, dass er mit steigendem tatsächlichem Wert x gegen 0 konvergiert. Das i ist der zum x gehörende bin.

$$\lim_{x\to\infty} \frac{F(i) - x}{x} = \lim_{x\to\infty} \frac{\frac{w}{2}}{x}$$
$$= 0$$

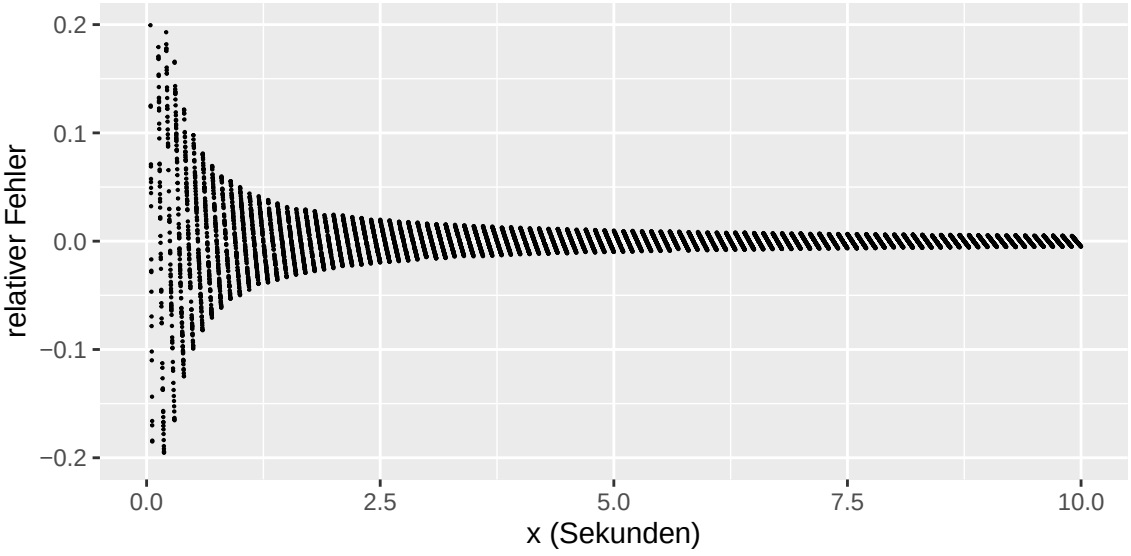


Abbildung 3.16: Relative Abweichung im Verhältnis zum echten Wert bei konstanter Bin-Breite

3.6.4.2 Wachsende Bin-Breite

Mit konstanter Bin-Breite wird der Approximationsfehler im Verhältnis zu dem Wertebereich des Bins, mit steigendem i immer geringer. Mit wachsenden Messwerten wird die Approximation also sehr präzise. Es ist hinreichend, dass der durch die Approximation entstehende Fehler, relativ zum echten Wert, konstant ist, da bei größeren Zahlen keine höhere Genauigkeit erforderlich ist.

Eine Vergrößerung der Bin-Breite in Abhängigkeit des Zahlenbereiches, der von dem Bin umschlossen wird, führt zu einem Fehler, dessen Verhältnis zum b_i konstant bleibt. Hierdurch wird zudem die Anzahl der benötigten Bins stark reduziert. Die Bins werden mit einen wählbaren Faktor $\beta > 1$ zu ihrem Vorgänger vergrößert. Dieser Faktor definiert zudem den Fehlerkorridor des relativen Fehlers. Die Bin-Grenzen b_i und die Breite w_i der Bins und die werden hierfür wie folgt definiert. Abbildung 3.17 illustriert den wachsenden Abstand zwischen den Grenzen der Bins.

Bin-Grenzen:

$$\begin{aligned} b_0 &= 1 \\ b_i &= b_{i-1} \cdot \beta \quad (\text{für } i > 0) \\ &= \beta^i \end{aligned}$$

Bin-Breite:

$$\begin{aligned} w_i &= b_{i+1} - b_i \\ &= \beta^{i+1} - \beta^i \\ &= \beta^i \cdot \beta - \beta^i \cdot 1 \\ &= \beta^i \cdot (\beta - 1) \end{aligned}$$

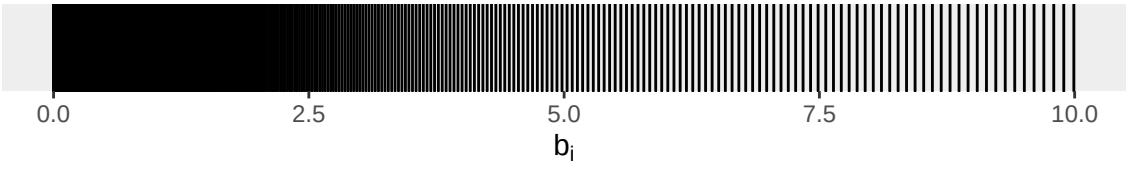


Abbildung 3.17: Wachsende Bin-Breite

Da die Bin-Grenzen b_i exponentiell mit der Basis β platziert werden, können eingehende Werte über den Logarithmus mit Basis β den entsprechenden Bins zugeordnet werden. Die Inverse hierzu ist die Potenz mit der Basis β. Der recht Teil der Addition entspricht dem Erwartungswert innerhalb des Bins.

$$\begin{aligned} f(x) &= \lfloor log_\beta(x) \rfloor \\ F(i) &= \beta^i + \frac{\beta^{i+1} - \beta^i}{2} \end{aligned}$$

Fehlerbereich Die stärkste Differenz zwischen echtem Wert x und der Approximation $F(f(x))$ tritt auch hier auf, wenn der echte Wert an einem der Ränder eines Bins liegt, da F auch hier auf den Erwartungswert des Bins abbildet.

Da β ein wählbarer Parameter ist, lässt sich über diese Konstante leicht ein fester Fehlerkorridor festlegen, welcher in Abbildung 3.18 zu sehen ist. Die Abbildung zeigt für zufällige x im Bereich 0 bis 10 Sekunden die entsprechenden relativen Fehler $\frac{e(x)}{x}$. Die x-Werte werden als ganzzahlige Nanosekunden in die Bins eingeordnet, jedoch in der Grafik als Sekunden

angegeben. Das β für die Bin-Breite in dieser Abbildung war $\beta = 1.01$. Es ist zu sehen, dass der relative Fehler, sowohl in positive, als auch negative Richtung nicht größer oder kleiner als $\pm\frac{1-\beta}{2}$ ist.

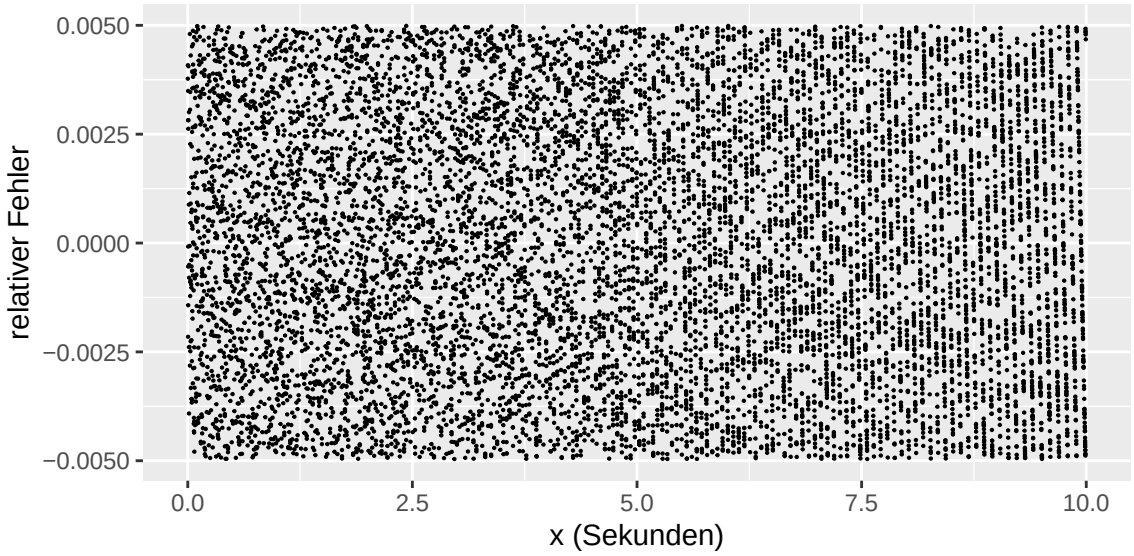


Abbildung 3.18: Abweichung im Verhältnis zum echten Wert bei wachsender Bin-Breite

3.6.4.3 Approximation der Quantile mit Echtdaten

Mit echten Profiling-Daten wurden die Approximationen in Abbildungen mit der Implementierung im *yProfiler* getestet. Hierfür wurden die approximierten Mediane mit den echten, mit dem R-Skript errechneten, Medianen verglichen. In Abbildung 3.19 und 3.20 werden die tatsächlichen Werte verglichen. Die grünen Punkte in den Abbildungen sind mit R berechnete Mediane der jeweiligen Methoden. Die approximierten Werte, werden durch einen kleineren schwarzen Punkt angegeben. Wie erwartet zeigen die Approximationen kleine Abweichungen, vor Allem bei der Approximation mit wachsender Bin-Breite. Die Präzision der Approximationen kann jedoch über ihre Parameter skaliert werden. Deshalb können die beiden Abbildungen untereinander nicht direkt verglichen werden.

Abbildung 3.21 und 3.22 zeigen die relativen Abweichungen vom echtem Median. Sie entsprechen den Abbildungen 3.16 und 3.18, errechnen jedoch den relativen Fehler des Medians und nicht den von einzelnen, in das Histogramm einsortierten, Messwerten. Die relativen Fehler mit einem echten Median von 0 sind auf Grund der Division durch 0 undefiniert und werden in der Grafik nur als 0 angezeigt.

In Abbildung 3.21 ist, wie in Abbildung 3.16, eine Konvergenz des relativen Fehlers zu sehen. Die Ansammlungen gleich hoher Mediane, sind vermutlich Aufrufketten, welche so schnell abgearbeitet wurden, dass die Methodenlaufzeiten der aufgerufenen Methoden jeweils als gleich lang gemessen wurden. Die großen Fehler mit Faktor -4 entstehen dadurch, dass in mit der konstanten Bin-Breite vor Allem bei sehr geringen Zahlenwerten die Abweichung potentiell sehr hoch sein kann.

Abbildung 3.22 zeigt diese Konvergenz nicht mehr, stattdessen ist jedoch zu sehen, dass der relative Fehler den durch $\beta = 0.055$ definierten Bereich von -0.0275 bis 0.0275 nicht über- oder unterschreitet. Auch mit der Methode der wachsenden Bin-Breite gibt es Ansammlungen, insbesondere im Bereich der sehr niedrigen Mediane. Auch diese lassen sich auf triviale Methoden der Aufrufketten zurückführen, welche nur sehr geringe Laufzeit benötigen und deshalb in die niedrigsten Bins fallen. Da das Phänomen der gleichen Fehlerrate jedoch nur bei sehr kleinen Zahlen auftritt, und die interessanteren, höheren Werte, wie in Abbildung 3.19 und 3.20 sichtbar ist, nicht betroffen sind, ist dieses Phänomen für eine Performance-Analyse nur von geringer Relevanz.

3.6.4.4 Fazit

Durch das Nutzen von wachsenden Bins kann die Anzahl der benötigten Bins reduziert werden um den Speicherverbrauch gering zu halten. Hierdurch lässt sich im Vergleich zu den Bins mit konstanter Breite, bei gleicher Anzahl an Bins, zudem ein deutlich größerer Zahlenbereich abdecken. Durch Tuning der Parameter ist vor allem die Fehlerrate komfortabel einstellbar. Die optimalen Einstellungen hängen jedoch von der untersuchten Anwendung ab und können in der Oberfläche des Profilers vorgenommen werden.

Zwischen den exakten Berechnungen im *yProfiler* und den exakten Werten, welche mit R berechnet wurden, wurden im direkten Vergleich nur sehr geringe Differenzen beobachtet. Diese lassen sich vermutlich auf Rundungsfehler zurückführen.

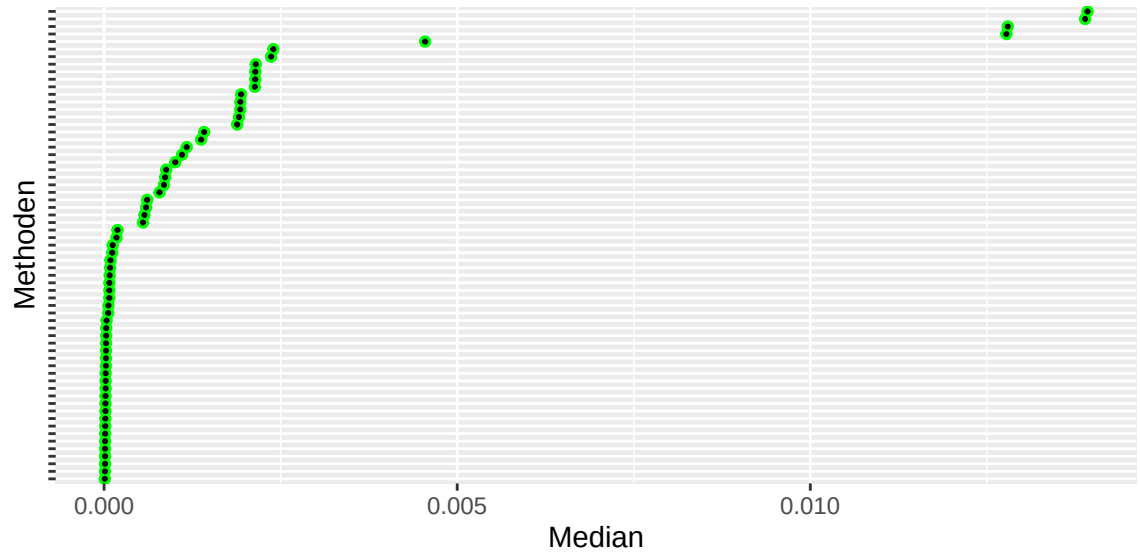


Abbildung 3.19: Gegenüberstellung der echten Mediane und der Approximation bei fester Bin-Breite von 1000 Nanosekunden. Die Methoden sind auf der Y-Achse aufsteigend sortiert nach echtem Median. (Nur Methoden mit echtem Median > 0.00001)

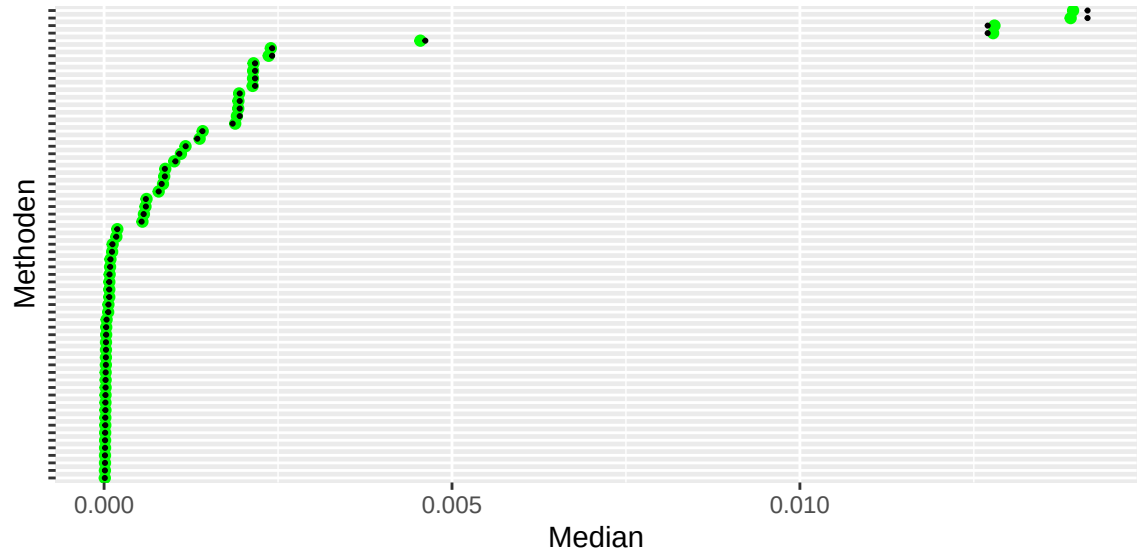


Abbildung 3.20: Gegenüberstellung der echten Mediane und der Approximation mit wachsenden Bins mit $\beta = 1.055$. Die Methoden sind auf der Y-Achse aufsteigend sortiert nach echtem Median. (Nur Methoden mit echtem Median > 0.00001)

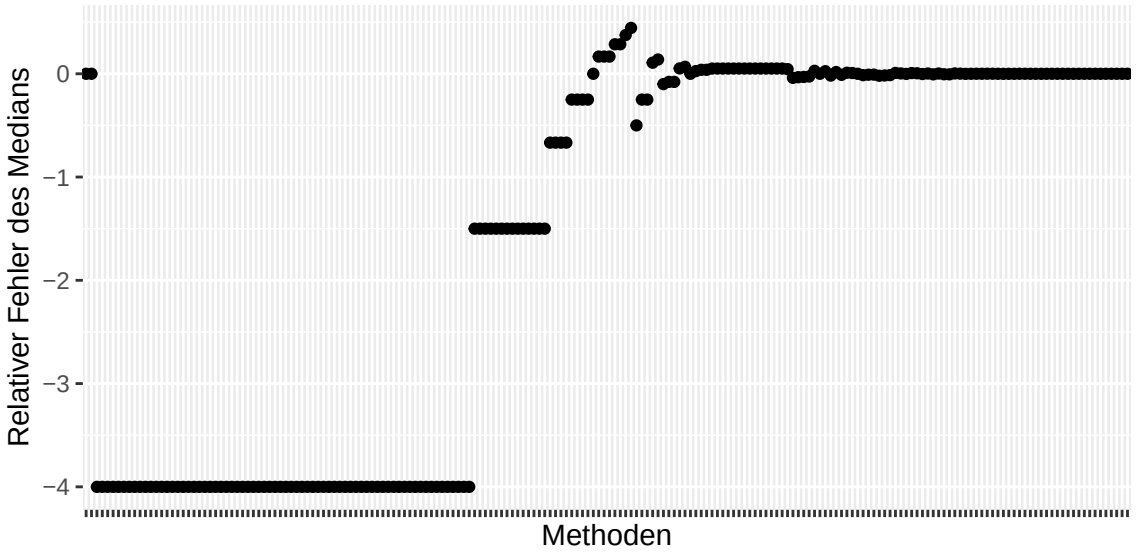


Abbildung 3.21: Die relativen Fehler des Medians bei fester Bin-Breite von 1000 Nanosekunden. Die Methoden sind auf der X-Achse aufsteigend sortiert nach echtem Median.

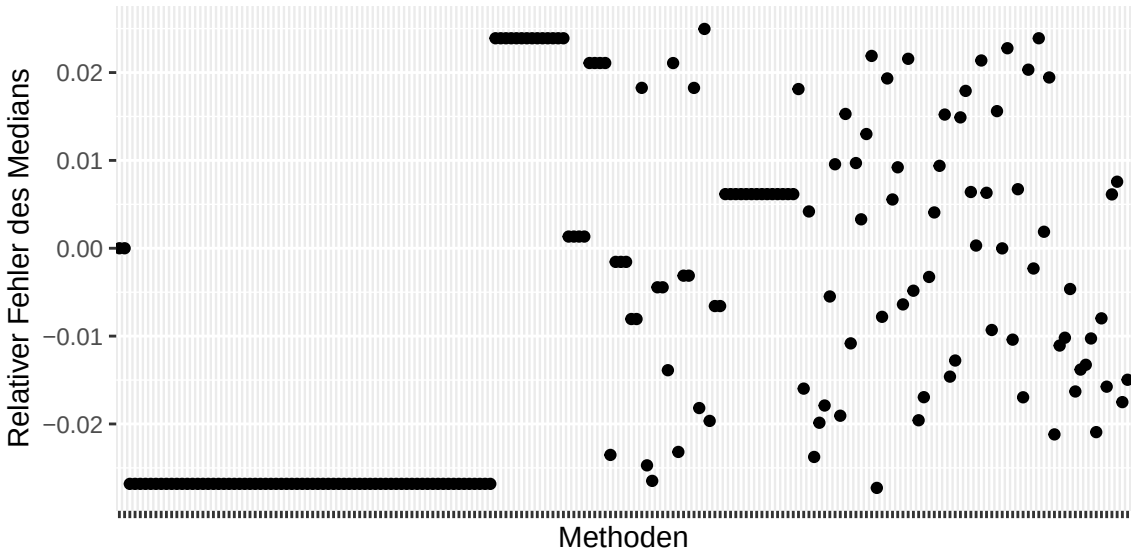


Abbildung 3.22: Die relativen Fehler des Medians mit wachsenden Bin-Breite mit $\beta = 1.055$. Der relative Fehler befindet demnach sich im Bereich -0.0275 bis 0.0275. Die Methoden sind auf der X-Achse aufsteigend sortiert nach echtem Median.

KAPITEL 4

Evaluation

Dieses Kapitel widmet sich der Evaluation der entwickelten Anwendungen. Im ersten Abschnitt 4.1 werden zunächst einige exemplarische Performance-Analysen durchgeführt, welche beispielhaft mögliche Anwendungszenarien demonstrieren. In Abschnitt 4.2 wird dann untersucht, wie stark der Einfluss des Profilers auf die getestete Anwendung ist. Hierfür wurden Lasttests durchgeführt, welche in verschiedenen Szenarien und Anwendungen die Auswirkung messen.

Die wichtigste Frage, die sich zum Ende dieser Arbeit stellt ist, ob die entworfene Darstellung praxistauglich ist und, ob sie von Entwickern akzeptiert wird. Diese Frage wird in der Benutzerstudie in Abschnitt 4.3 ausführlich untersucht.

4.1 Anwendung von Profiler und Visualisierung

Die Benutzung des Profilers erfordert mehrere Schritte. Für die Darstellung der Anwendung innerhalb von *SEE* muss zunächst eine GXL-Datei für die geprüfte Anwendung vorhanden sein. Diese kann erzeugt werden, indem während des Buildvorgangs der Anwendung ein spezieller Compiler von *Axivion* [99] eingehängt wird. Die GXL-Datei kann optional, jedoch nicht notwendigerweise, auch im Profiler angegeben werden, um überladene Methoden erkennen zu können. Nach Start, der zu untersuchenden Anwendung, wird der Profiler und einer oder beide Profiling-Modi gestartet.

Die Anwendung muss während der Profilerlaufzeit in irgendeiner Weise benutzt werden. Dies kann durch Starten eines automatisierten Testfalls oder durch die manuelle Bedienung in der Oberfläche getan werden. Wird der Sampling-Profiler verwendet, muss die Aktion häufiger wiederholt werden, da der Sampling-Profiler sonst nur wenige bis keine der ausgeführten Methoden erfassen kann. Im Kontext von Lasttests werden über viele Iterationen einzelne Testfälle mehrfach und insbesondere parallel durchgeführt. Hier eignet sich der Sampling-Profiler. Wird eine lokale Single-User-Anwendung getestet ist das Nutzen des Instrumentation-Profilers möglicherweise passender, da der Anwendungsfall in der Anwendung nur einmalig durchlaufen wird. Wird nur die Startseite einer Web-Anwendung untersucht, reicht es aus die Seite im Web-Browser manuell mehrfach zu aktualisieren. Nach Erheben der Performance-Daten mit dem Profiler liegen die Metriken in CSV-Dateien vor und können in SEE für die Darstellung angegeben werden. Hierfür wird im Unity-Editor der Pfad angegeben. Die Anwendung wird über den Editor gestartet.

Die folgenden Benutzungsbeispiele haben nicht den Anspruch die untersuchten Anwendungen zu optimieren. Sinn dieser exemplarischen Darstellung ist die Illustration einer möglichen Benutzungsweise und eine Betrachtung der Visualisierung im Anwenderkontext.

4.1.1 Untersuchung von libAwesome

Der Quellcode, der untersuchten Anwendung, wurde nicht von mir implementiert. In diesem Abschnitt wird exemplarisch dargestellt, wie erfolgreich zeitintensivere Stellen gefunden werden können. Die Anwendung *libAwesome* wurde mit dem Lasttest-Tool *JMeter* belastet. Der Lasttest-Fall ruft wiederholt die Bücher-Übersicht aus Abbildung 2.18 der Anwendung auf, sodass Methoden der Anwendung ausgeführt werden. Im Ruhezustand können keine Profilingdaten erhoben werden, da die Anwendung keine Hintergrundaktivitäten hat.

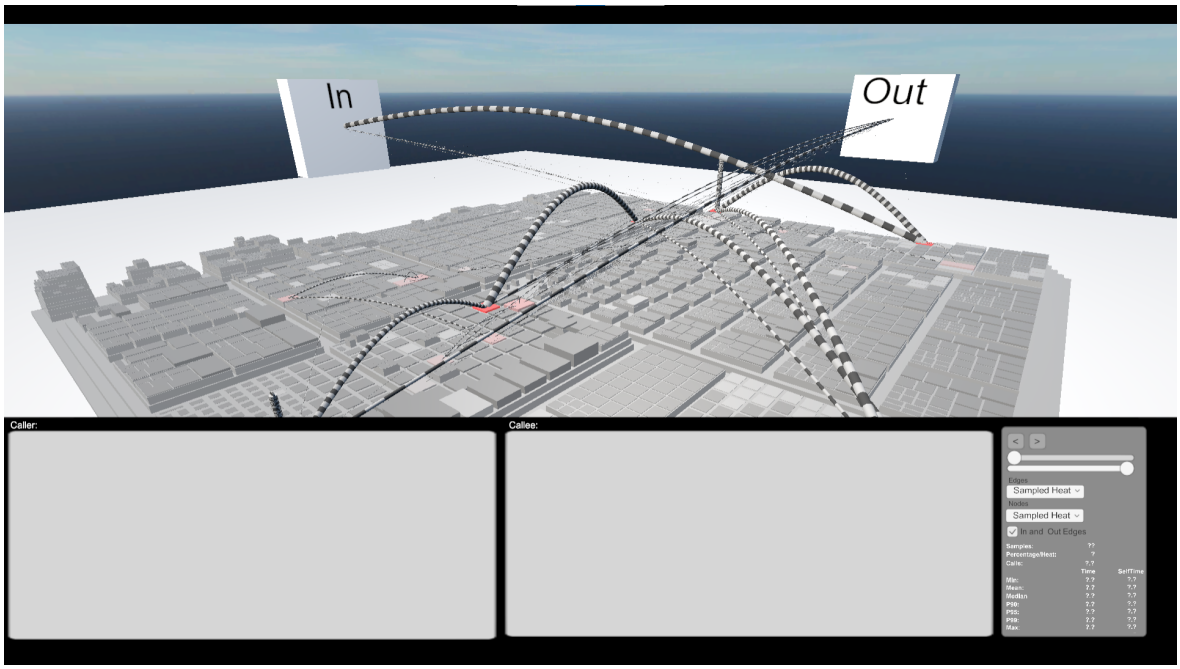


Abbildung 4.1: Visualisierung der Ergebnisse Sampling-Profilers.

Abbildung 4.1 zeigt die erste Ansicht, welche beim Start des Profilers zu sehen ist. Dargestellt sind sowohl für die Kanten, als auch die Methoden-Blöcke die *Sampled Heat*, wie im Einstellungsmenü sichtbar ist. Die Sampled Heat ist die Heat, bzw. die Anzahl der Samples, welche mit dem Sampling-Profiler erhoben wurde. Ein paar Kanten hier stechen heraus, da sie breiter sind als die anderen. Werden die breiten Kanten nacheinander ausgewählt und der Quellcode inspiziert, kann auf diesem Wege schnell herausgefunden werden, dass der Grund für die hohe Anzahl an Samples darin liegt, dass eine SQL-Abfrage ausgeführt wird. In Abbildung 4.2 ist die ausgewählte Kante, gemeinsam mit dem Quellcode des *Callees* zu sehen. Im Quellcode markiert ist tiefrot eine Zeile, welche einen `handle`-Methode aufruft. Innerhalb dieses `handle` werden die einzelnen Datenreihen der SQL-Abfrage verarbeitet. In blassem Rot sind zudem die Zeilen markiert, welche die SQL-Abfrage initial aufbauen. Die blassere Hervorhebung bedeutet, dass die Methoden beim Sampling seltener aufgetaucht sind und sich deshalb zeitlich seltener in Ausführung befanden.

Der Nutzen des Instrumentation-Profilers wird deutlich, wenn die Ansicht umgestellt wird, sodass der Mittelwert der gemessenen Aufruflaufzeiten dargestellt wird. Abbildung 4.3 zeigt dies. Die auffälligsten Kanten unterscheiden sich stark von denen, welche in den Sampling-Ergebnissen aus Abbildung 4.1 zu sehen sind. Grund hierfür ist, dass einige *langsame*[1] Methoden im Verhältnis deutlich seltener aufgerufen wurden. Der Sampling-Profiler kann dies

[1] Wirklich spürbar langsam sind die Methoden nicht wirklich, lediglich im Verhältnis zu den anderen Methoden sind sie langsamer.

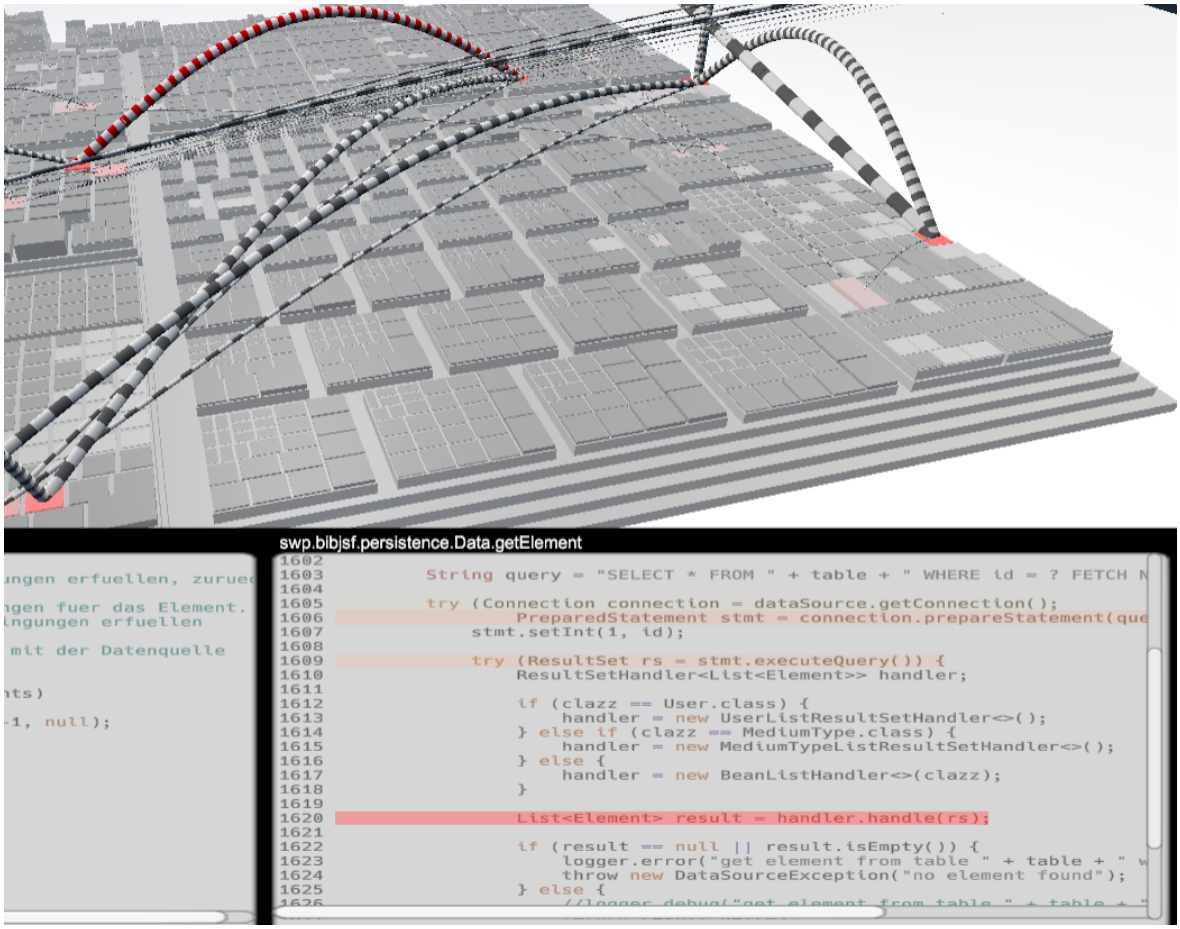


Abbildung 4.2: Ausgewählter Aufruf mit markierten zeitintensiven Zeilen

jedoch nicht erkennen. Abbildung 4.4 auf Seite 69 zeigt die Aufrufhäufigkeit im Vergleich.

Die Darstellung des Medians und den Perzentilen ähneln sich, mit diesen Profiling-Ergebnissen, optisch sehr stark, da es offenbar keine größeren Ausreißer gab, welche vom Mittelwert stark abwichen. Sie bieten in diesem Fall für die Analyse also den gleichen Erkenntnisgewinn wie der Mittelwert. Der Erkenntnisgewinn gegenüber dem Sampling-Profiler ist hier jedoch durchaus wertvoll, denn nur durch die mit Instrumentierung erhobenen Daten wird die Methode in Abbildung 4.3 auf Seite 68 mit der höchsten Laufzeit richtig sichtbar.

Ähnlichkeit zwischen den Heat-Darstellungen Da in Abschnitt 3 erwähnt wurde, dass heat und heat_s in der Darstellung ähnlich aussehen, kann die heat aus den Ergebnissen des Instrumentation-Profilers errechnet werden, indem die Aufrufanzahl mit der mittleren Laufzeit multipliziert wird. Abbildung 4.5 zeigt, dass die Darstellung dieses Wertes tatsächlich sehr ähnlich zu den Ergebnissen des Sampling-Profilers in Abbildung 4.1 auf Seite 66 aussieht.

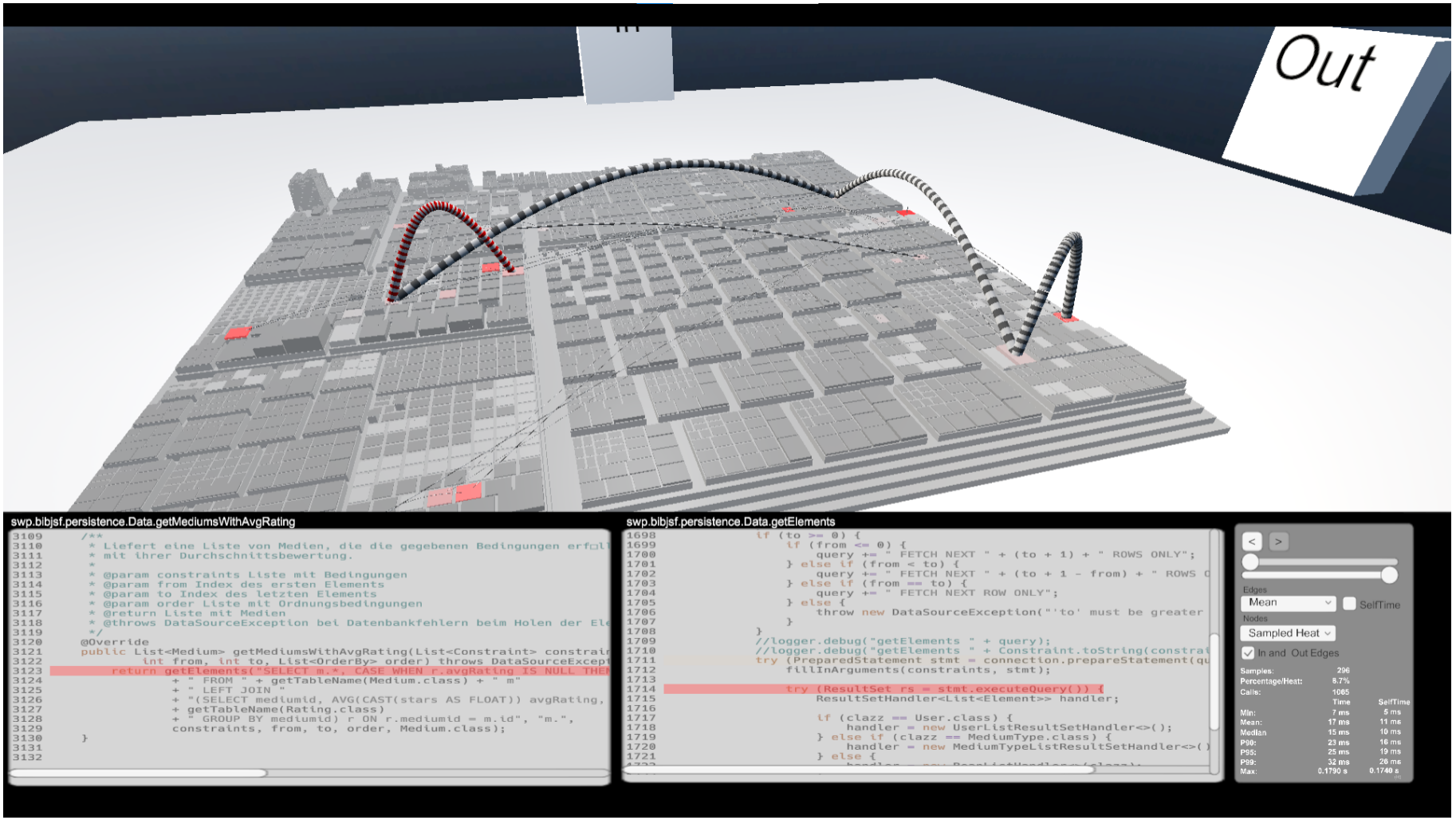


Abbildung 4.3: Durch den Instrumentation-Profiler aufgedeckter Methodenaufruf.

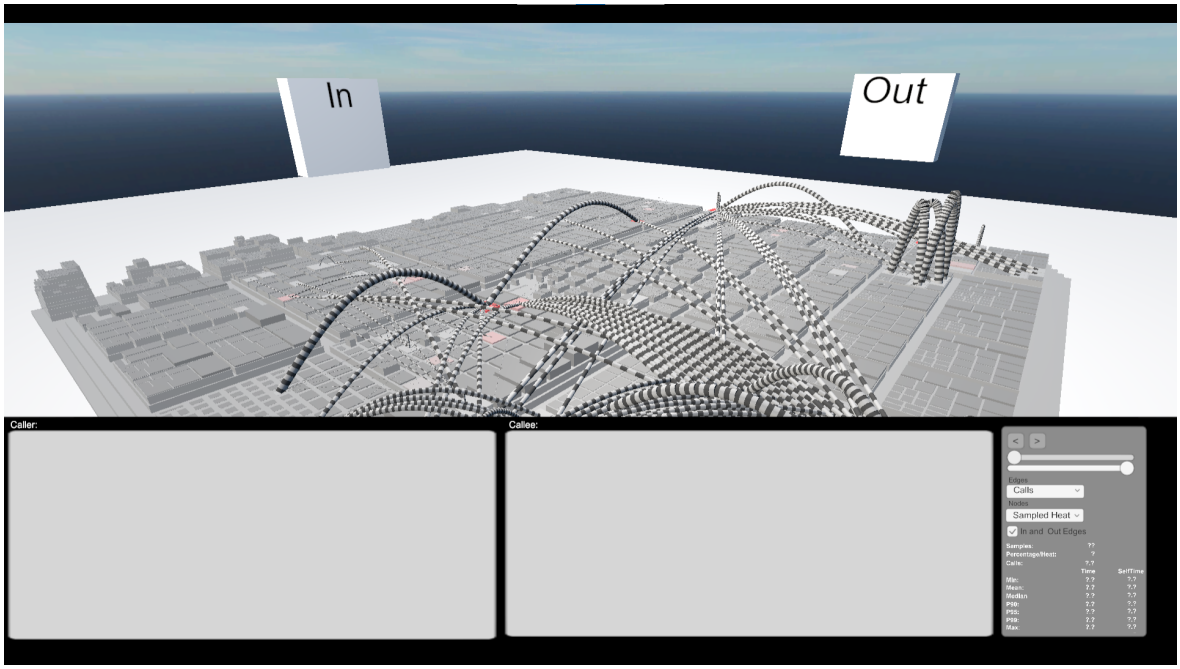


Abbildung 4.4: Visualisierung der Aufrufanzahl des Sampling-Profilers.

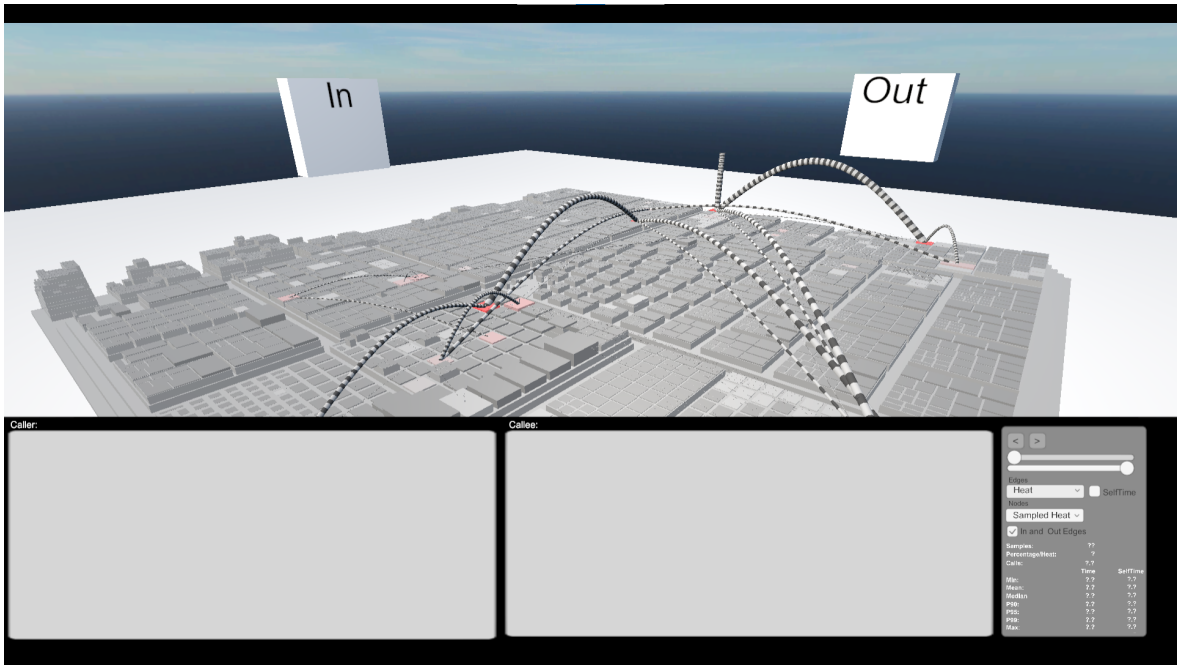


Abbildung 4.5: Berechnete Heat aus den Daten des Instrumentation-Profilers. Sie ähnelt stark der Darstellung der Ergebnisse des Sampling-Profilers aus Abbildung 4.1

Self Times Die Darstellung der *self times* ist, wie in Abschnitt 3.5.1 beschrieben, ebenfalls möglich. Hierdurch bleibt jedoch in der Regel nur die letzte Kante der Aufrufkette übrig. Die Abbildung 4.6 im Anhang auf Seite 70 zeigt dies exemplarisch. Sie ist das Gegenstück zu Abbildung 4.3. In diesem Fall entsteht durch das Darstellen der *self time* kein Erkenntnisgewinn. Sind jedoch in der Darstellung deutlich mehr Kanten dargestellt, hätte das Betrachten der *self time* zu einer höheren Übersicht führen können.

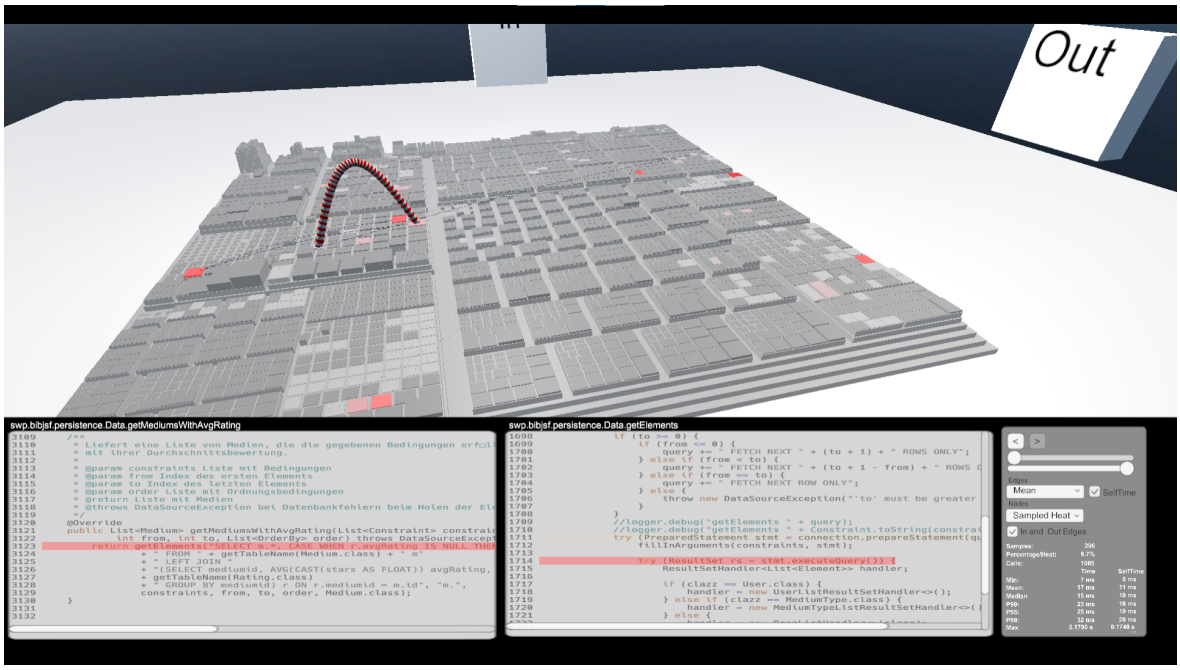


Abbildung 4.6: Darstellung der *self times* mit gleicher Perspektive wie in Abbildung 4.3. Es ist nur noch die letzte Kante des *Stacks* sichtbar.

4.1.2 Ungleich verteilte Methodenlaufzeit

Auch, wenn es in der vorherigen Untersuchung nicht so war, kann es in speziellen Fällen sein, dass Laufzeiten von Methoden stark schwanken oder in nur sehr seltenen Fällen langsam sind. Diese Fälle können von einem Sampling-Profiler oder mit dem Mittelwert der Laufzeit nur schwer erfasst werden. Es wurde ein Dummy-Fehler eingebaut, welcher dieses Verhalten simuliert, indem eine Methode in 5 Prozent der Aufrufe deutlich langsamer ist als sonst. Der angepasste Quell-Code ist in Text 4.1 zu sehen. In Abbildung 4.7 sind Darstellungen von 4 Metriken dargestellt. Zwar ist die Aufrufkante in der „Sampled Heat“, Mean und im 95. Perzentil sichtbar, jedoch nicht so deutlich erkennbar wie im 99. Perzentil. Insbesondere fällt der Unterschied auf, wenn in der Anwendung zwischen den Metriken gewechselt wird.

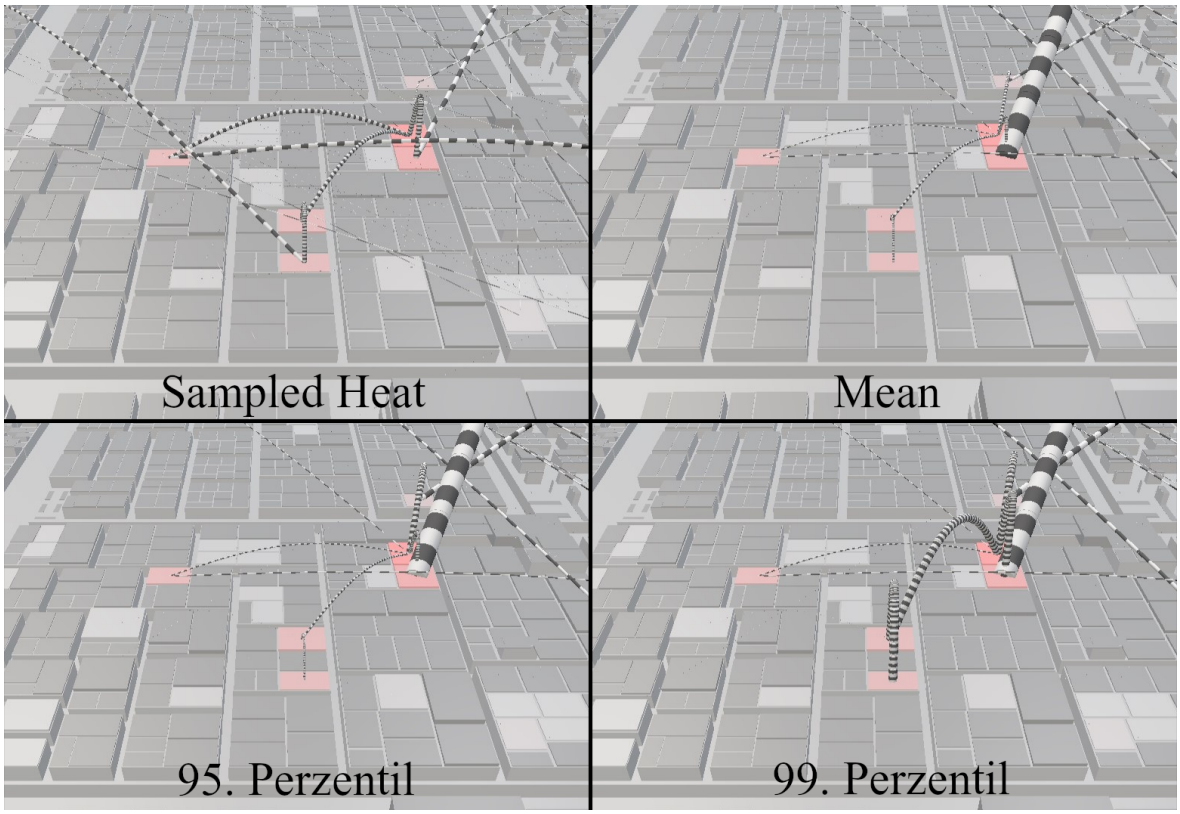


Abbildung 4.7: Vergleich unterschiedlicher Metriken zeigt, dass die Erhebung von Perzentilen einen Mehrwert bieten kann.

```
76 public String getValue(final String key) {
77
78     if(random.nextDouble() > 0.95){
79         sleep(20);
80     }else{
81         sleep(1);
82     }
83
84     return PropertyFile.getValue(config_filename, key);
85 }
86
87 private void sleep(int time){
88     try {
89         Thread.sleep(time);
90     } catch (InterruptedException e) {
91         e.printStackTrace();
92     }
93 }
```

Text 4.1: Veränderter Quellcode von FileConfiguration.java, für eine 5% Wahrscheinlichkeit auf eine erhöhte Methodenlaufzeit. Es wurden Zeilen 78-82 und 87-93 ergänzt.

4.1.3 Aufgaben der Benutzerstudie

Für die Benutzerstudie, welche ausführlich in Abschnitt 4.3 beschrieben wird, wurden drei unterschiedliche Fehler innerhalb der Anwendung *libAwesome* platziert. Die Fehler haben alle gemeinsam, dass sie im Quell-Code recht einfach als obsolet erkennbar sind, sie jedoch die Methodenlaufzeit verlangsamen. Im Abschnitt 4.3.2 werden die 3D Darstellungen und mögliche Lösungswege zu diesen drei Aufgaben beschrieben.

4.1.4 Gitblit

Die Java-Anwendung *Gitblit* [94] konnte mit dem implementierten Profiler erfolgreich untersucht werden. Gitblit ist eine Open-Source Anwendung für die Verwaltung von Git-Repositories. Die Besonderheit ist, dass sie ausschließlich auf Java basiert und deshalb gut für einen Test des Profilers geeignet ist. Sie funktioniert als Web-Anwendung und integriert hierfür in ihrer Standalone-Installation den Web-Server „Jetty“[102].

Sowohl der Sampling-Profiler, als auch der Instrumentierungs-Profiler konnten angewendet werden. Im Zeitbereich, in dem die Anwendung profiliert wurde, wurde lediglich die Startseite händisch sehr häufig aufgerufen. Die Anwendung wurde zuvor nicht benutzt. Sie befand sich also in der Standardkonfiguration und es lag für diese Untersuchung noch keine Datenbasis in Form von eingerichten Git-Repositories vor. Bereits in der tabellarischen Darstellung im Profiler war sichtbar, dass ein Hintergrundprozess der Anwendung auch aktiv ist, während keine Seitenaufrufe ausgeführt werden. Die hierzu gehörenden Methodenaufrufe überdecken in ihrer Häufigkeit die Methodenaufrufe, welche durch den Aufruf der Startseite ausgeführt werden.

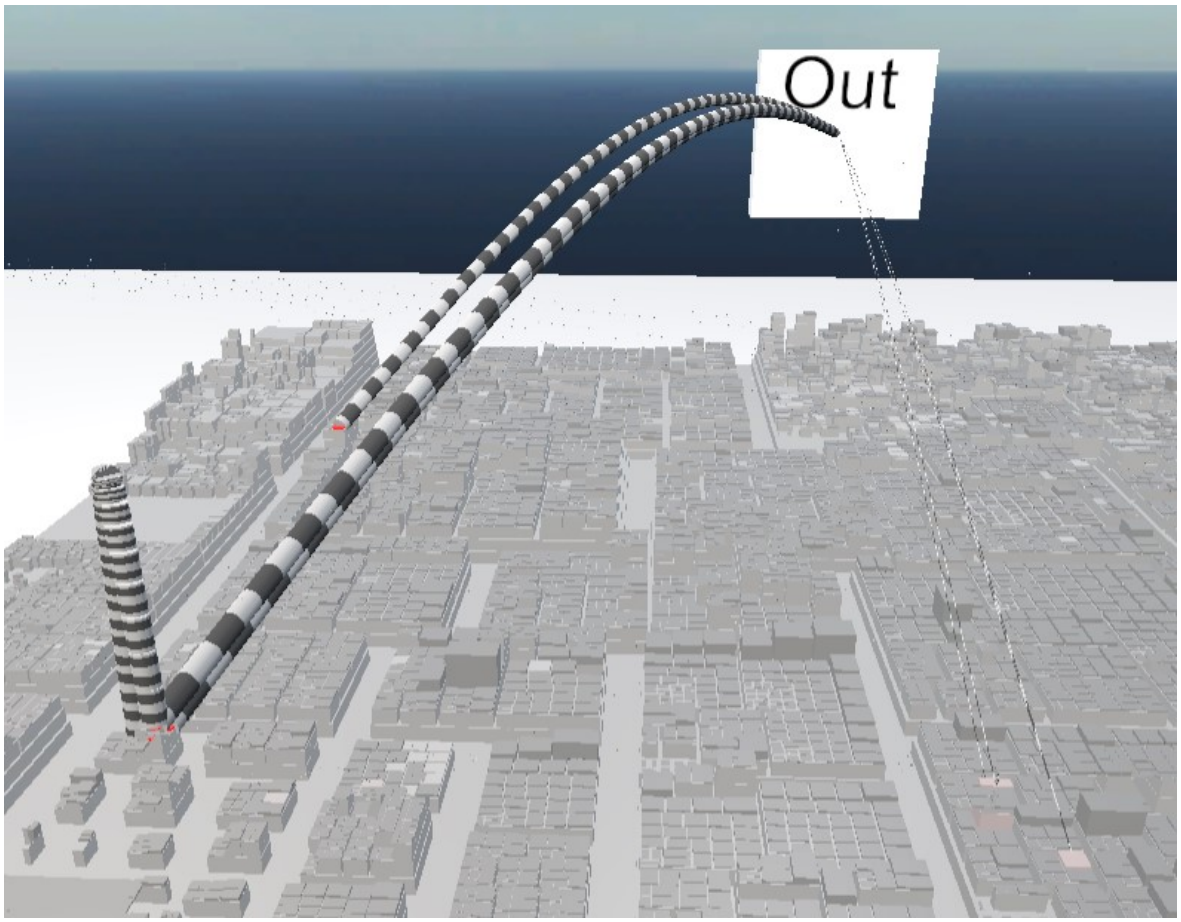


Abbildung 4.8: Darstellung der gesampleten *Heat*

Abbildung 4.8 zeigt innerhalb der Software-City von Gitblit einige breite eingehende Kanten und einige deutlich kleinere Kanten. Die dickeren Kanten sind die, welche durch den Hintergrundprozess regelmäßig aufgerufen werden. Die entsprechen Kanten sind Methodenaufrufe von `java.net.ServerSocket.accept` und `org.eclipse.jetty.server.Server.join`, den Hauptteil der Zeit verwendete die Anwendung im Messzeitraum also mit Warten auf neue, ein-

gehende Client-Verbindungen. Durch die Verwendung der Filterfunktion können diese Kanten gefiltert werden, sodass die kleinen Kanten hochskaliert und somit sichtbarer werden. Diese gefilterte Ansicht ist in Abbildung 4.9 zu sehen. Zuvor nur schwer zu sehen waren, die mit ① markierten Kanten. Sie rufen die Methode `doFilter` auf. Als projektfremde Person ist dieser Methode jedoch nur schwer eine Logik zuordenbar, obwohl der Quellcode ebenfalls einsehbar war. Im, mit ② markierten Bereich, befindet sich eine Aufrufkette `getSetting` → `read` → `exists`. Scheinbar wird hier beim Abrufen von Einstellungen geprüft, ob bestimmte Dateien vorhanden sind. Möglicherweise gibt es hier Optimierungspotential dadurch, dass Dateien nicht bei jedem Aufruf von `getSetting` geprüft werden müssen. Dies ist jedoch nur eine initiale Vermutung und muss in einer genaueren Analyse des Quellcodes geprüften werden. Die durch das Profiling gemessenen Laufzeitwerte der Methoden deuten insbesondere auch nicht darauf hin, dass hier Optimierungen durchgeführt werden müssen, da der Mittelwert und das Maximum dieser Methode unter einer Millisekunde liegen, wie dem Informationspanel zu entnehmen war.

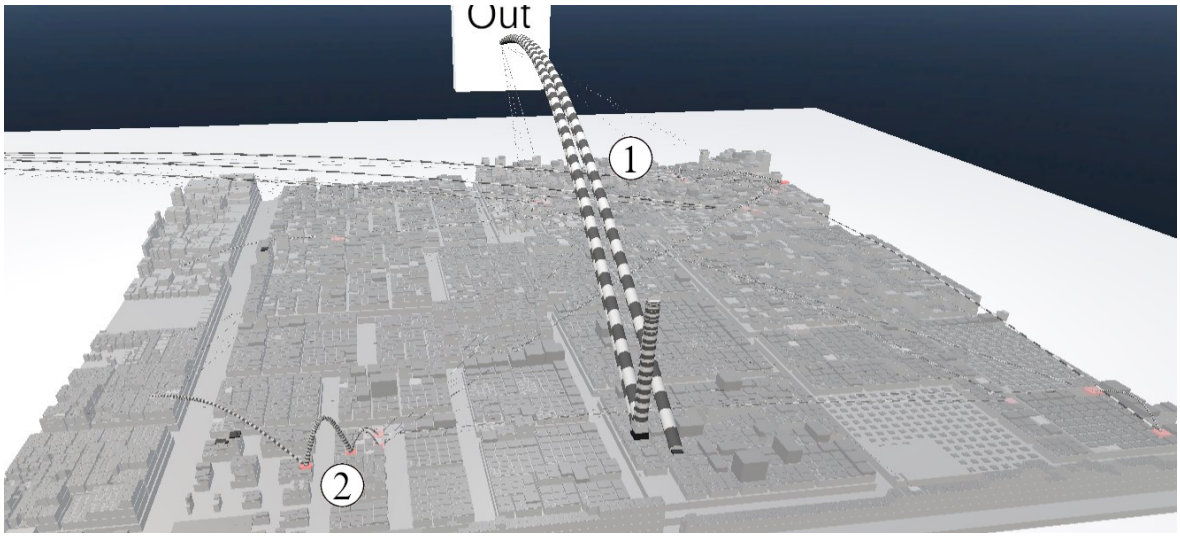


Abbildung 4.9: Darstellung der gesampleten *Heat* mit herausgefilterten großen Kanten.

In einem weiteren Schritt wird, an Stelle der „Sampling Heat“, der, mit dem Instrumentation-Profiler erhobene, Mittelwert der Methodenlaufzeit für die Darstellung der Kanten verwendet. Abbildung 4.10 zeigt, wie sich die Ansicht stark ändert und neuen Kanten sichtbar werden. Die nun am stärksten sichtbare Methode ist weiterhin `doFilter`. Im Mittelwert ist diese Methode demnach die Langsamste. Bei 24 Milisekunden besteht vermutlich jedoch kein akuter Optimierungsbedarf. Durch die Darstellung der Perzentile verändert sich die Ansicht nur wenig, es scheint also keine Methoden zu geben, welche eine verhältnismäßig große Diskrepanz zwischen Mittelwert und den Extremwerten in hohen Perzentilen oder Maximum haben.

Untersuchung mit vorhandenem Repository Für einen zweiten Profiling-Lauf wurde in Gitblit ein großes Git-Repository[2] eingebunden, um Gitblit so dazu zu bringen intern weitere Funktionen aufzurufen, da beim Aufruf der Seite mehr Informationen geladen werden müssen. Während des Profiling-Zeitraums wurde die Commit-Historie des Projekts in der Gitblit-Oberfläche wiederholt manuell aufgerufen. Hierdurch verändert sich die Darstellung der Profiling-Informationen in SEE zwar nicht stark, der in Abbildung 4.11 sichtbare Abschnitt enthält nun jedoch, zuvor nicht nicht vorhandene, Methodenaufrufe. Diese sind

[2]Das Git-Repository des gesamten SEE-Projektes.

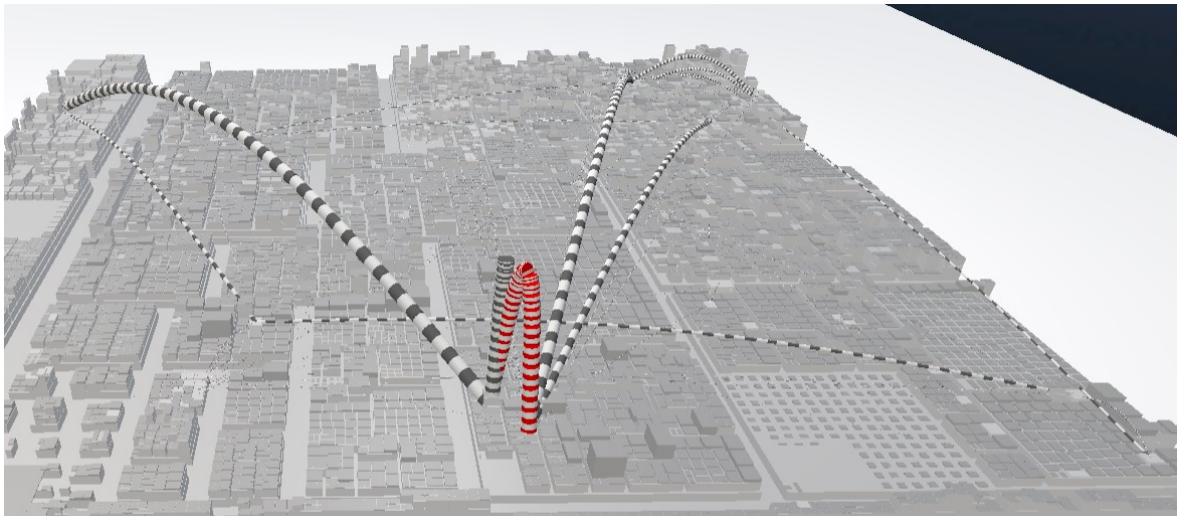


Abbildung 4.10: Darstellung der gemessenen Mittelwerte in Gitblit

insbesondere mit dargestelltem Mittelwert erkennbar. Sie sind jedoch auch mit der „Sampling Heat“ sichtbar. Mit der Navigation über die Code-Fenster sind zudem schnell die Endpunkte der Aufrufketten in diesem Bereich findbar. Ein Punkt, in dem viel Zeit verbracht wird, ist eine `for`-Schleife über alle Commits der Git-Historie. Die zweite auffällige Stelle ist der Zugriff auf eine Datenbank. Als leicht auffällig ist zudem ein Sortiervorgang durch `Collections.sort()` identifizerbar. Die in der ersten Untersuchung bereits auffälligsten Aufrufkanten des Webservers und `doFilter` bleiben weiterhin am stärksten ausgeprägt.

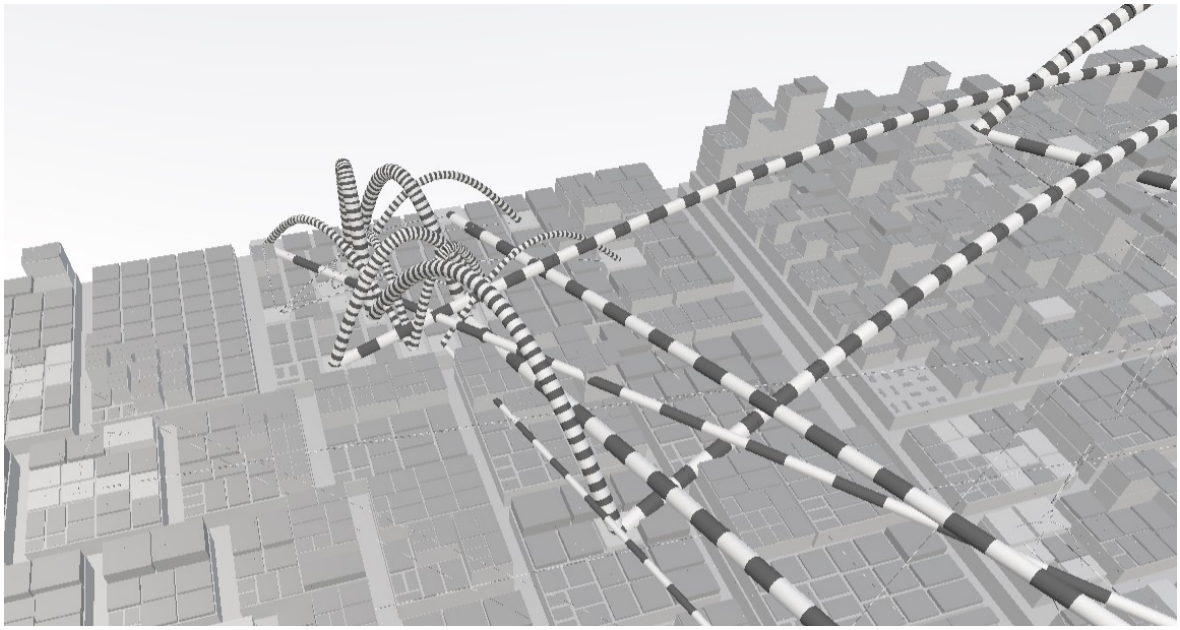


Abbildung 4.11: Darstellung der Aufrufe in *Gitblit*, welche die Tabelle aller *Git-Commits* erzeugt.

4.2 Evaluation von Sampling und Instrumentierung

Mit Hilfe von Lasttests sollen die die Seitenantwortzeiten der getesteten Web-Anwendung während der Profiling-Zeiträume der beiden Profiler-Typen untersucht werden. Es ist eine Veränderung der Antwortzeiten zu erwarten, da die untersuchte Anwendung, durch den Sampling-Profiler, in regelmäßigen Zeiträumen immer wieder sehr kurz pausiert wird und, durch die Instrumentierung, im zweiten Profiling-Modus zusätzlichen Code ausführen muss.

4.2.1 libAwesome

Es wurden mehrere Lasttests gegen die Anwendung libAwesome durchgeführt. Hierbei wurden unterschiedliche Benutzerzahlen und Aufrufraten verwendet. Während der Testlaufzeit wurden der Sampling-Profiler und der Instrumentation-Profiler verwendet um die Auswirkung auf die Antwortzeiten zu bestimmen. Während der `baseline`-Phase wurden ausschließlich HTTP-Requests des Lasttest mit „JMeter“ ausgeführt. Eine `warmup`-Phase soll unerwünschte Effekte beim Vergleich der Messungen vermeiden. Die erste Minute des Tests könnte Funktionen der Anwendung enthalten, welche zu Beginn nur einmalig durchgeführt werden, wie erstmaliges Laden von Daten in den Speicher und Just-in-Time-Optimierungen durch die JVM [103]. Die Phase, in welcher der Sampling-Profiler verwendet wird, heißt `sampling`. Zu dem Instrumentation-Profiler gehören drei Phasen. Die Phase `inject` verbindet sich lediglich mit der Ziel-JVM, in `instrument` werden die zu untersuchenden Java-Klassen instrumentiert und der Profiler-Code wird eingefügt. Hierbei sind in den Tests deutlich erhöhte Antwortzeiten sichtbar. Nach dem Instrumentieren ist der Profiler nicht automatisch aktiv. Erst mit Beginn der Phase `profile` werden die Methodenlaufzeiten gemessen und abgespeichert. Im ersten Test wird in eine Log-file geschrieben und in zweiten die Netzwerk-Funktionalität des Profilers verwendet, wodurch keine Dateizugriffe stattfinden, sondern die gesammelten Informationen über einen Netzwerk-Socket übertragen und im *yProfiler* die Statistiken berechnet werden.

In Test 1 bis 3 wurden jeweils die gleichen Lastprofile verwendet. Mit 5 parallelen Lasttreiber-Threads des Test-Tools wurden jeweils mit je 100 Milisekunden Abstand HTTP-Aufrufe gemacht. In Test 4 und 5 wurden 50 parallele Threads genutzt. Es wurde für eine leichter lesbare Analyse nur der Aufruf der Seite `medien_liste.xml` gemacht, um die Analyse visuell einfacher zu gestalten und Seiteneffekte zu vermeiden. In anderen Lasttests werden in der Regel noch Ressourcen, Bilder oder ähnliches mit geladen. Für die Tests in der Benutzerevalutation in Abschnitt 4.3 wurden mehr unterschiedliche Aufrufe durchgeführt.

4.2.1.1 Test 1 (Profiling in Log-Datei)

Für diesen Tests wurde der Profiling Modus verwendet, bei welchem die alle Rohdaten in eine Datei geschrieben werden, welche nach dem Test von einem R-Skript nachträglich analysiert werden. Der erste Testlauf ist in Abbildung 4.12 abgebildet. Es wurde nach der `baseline`-Phase lediglich der Instrumentation-Profiler eingeschaltet und anschließend wieder gestoppt. Der Sampling-Profiler wurde noch nicht verwendet. Es ist visuell ein deutlicher Anstieg der Antwortzeiten in den Phasen `instrument` und `profile` zu erkennen. Tabelle 4.1 zeigt den Anstieg der Mittelwerte der jeweiligen Bereiche. Das Abweichen der Aufrufrate in `inject` ist der kurzen Phasenzeit geschuldet. Die niedrigere Aufrufrate in `profile` begründet sich darin, dass die Seitenaufrufe länger dauern und somit ein geringerer Durchsatz durch JMeter möglich ist. Der Faktor der Verlangsamung von `baseline` zu `profile` beträgt 1.86. Das bedeutet, dass durch den Einsatz der Instrumentierung die Seitenantwortzeit nahezu verdoppelt hat.

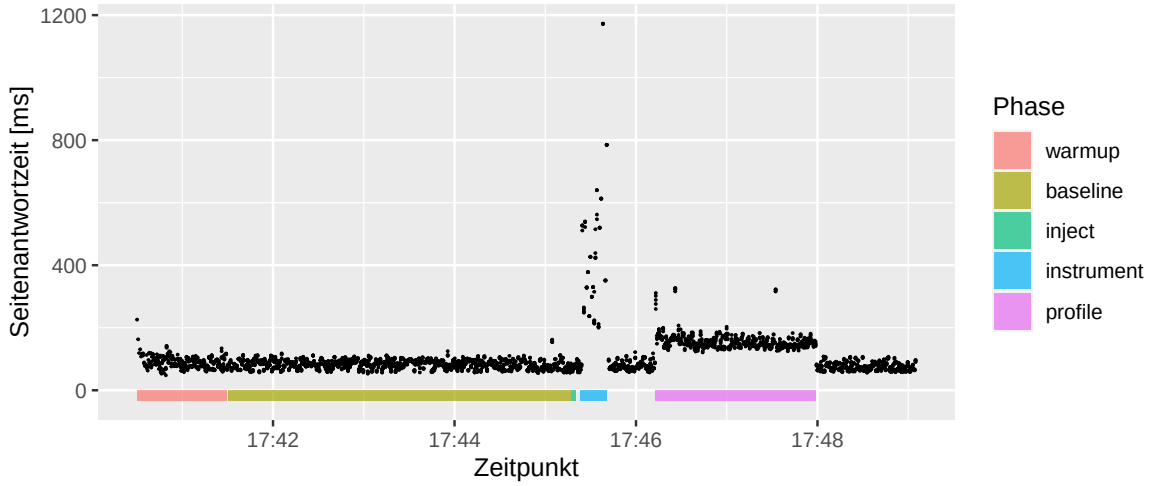


Abbildung 4.12: Testverlauf von Test 1

Phase	Seitenaufrufe	Dauer (s)	Aufrufe/s	Mean (ms)
warmup	436	59.48	7.33	87.96
baseline	1920	226.75	8.47	83.23
inject	30	2.89	10.36	72.07
instrument	100	17.31	5.78	419.80
profile	805	106.29	7.57	155.41

Tabelle 4.1: Statistiken der untersuchten Phasen in Test 1

4.2.1.2 Test 2 (Profiling via Netzwerk)

Dieser Tests gleicht Test 1 vom Aufbau her, mit der Ausnahme, dass die Profiling-Logs nicht in eine Datei geschrieben werden, sondern über einen Netzwerk-Socket an den Hauptprozess des Profilers weitergegeben werden. Dort werden sie dann zwischengespeichert. Die Phasenlängen variieren etwas, da sie händisch ausgelöst wurden. Die Mittelwerte von `baseline` in Text 1 und Test 2 unterscheiden sich um circa $3ms$. Da die Tests auf einem herkömmlichen Windows 10 Rechner ausgeführt wurden, können Beeinflussungen durch andere Applikationen nicht ausgeschlossen werden. Die kleinen Unterschiede sollten deshalb in dieser Analyse generell als Rauschen betrachtet werden. Der Faktor, um den die Seitenantwortzeit in Phase `profile` ansteigt, beträgt 2.07. Der Faktor ist also etwas größer als in Test 1. Der Unterschied ist jedoch nicht so groß, dass zwischen diesen beiden Tests eine Messmethode als die bedeutend besser bestimmt werden kann. Beide Modi hatten in etwa eine Verdoppelung der Antwortzeiten zur Folge.

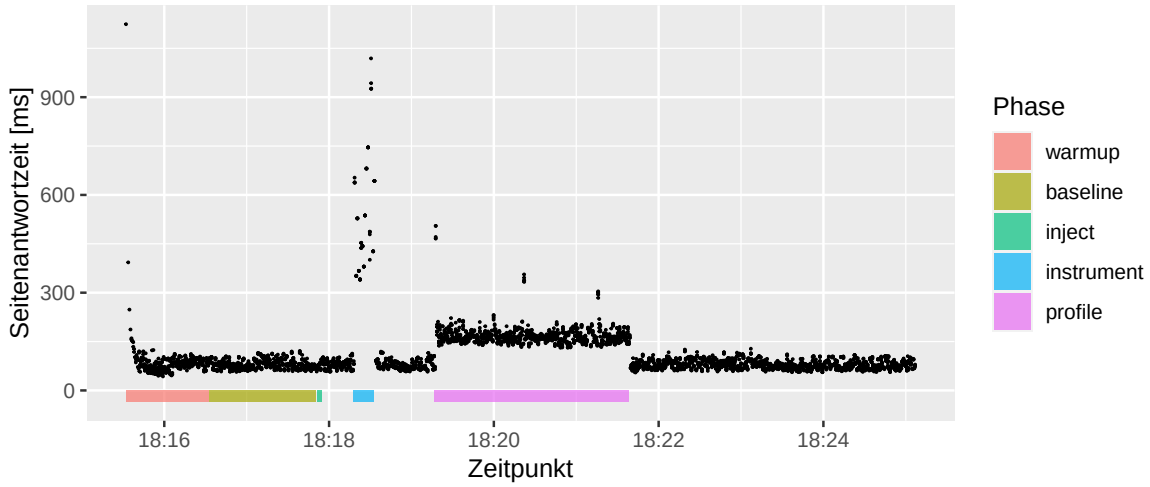


Abbildung 4.13: Testverlauf von Test 2

Phase	Seitenaufrufe	Dauer (s)	Aufrufe/s	Mean (ms)
warmup	442	59.83	7.39	84.58
baseline	665	77.90	8.54	80.67
inject	35	3.48	10.05	74.97
instrument	80	14.65	5.46	465.60
profile	1050	141.28	7.43	167.07

Tabelle 4.2: Statistiken der untersuchten Phasen in Test 2

4.2.1.3 Test 3 (Reduzierte Instrumentierung)

In Test 3 wurden in der Profiling-Phase nicht alle Methoden der Anwendung instrumentiert, um einen geringeren Einfluss auf die Antwortzeiten zu erzielen. Hierfür wird die **top-n**-Einstellung verwendet, welche nur die häufigsten, vom Sampling-Profiler gesehenen, Methoden für die Instrumentierung auswählt. Zusätzlich wurde die Einstellung **profile Call-Tails** verwendet, wodurch alle Methoden, welche direkt oder indirekt die top-n Methoden aufrufen, ebenfalls instrumentiert werden. Insgesamt wurden 84 Methoden instrumentiert. Im Vergleich wurden in Test 1 und Test 2 für über 200 Klassen jeweils alle Methoden instrumentiert.

Für diese Reduzierung der Instrumentierten Methoden, wurde vor dem instrumentierten Profiling der Sampling-Profiler aktiviert, wodurch es eine zusätzliche Phase gibt. Zudem wurde die `baseline` Phase nach dem `sampling` fortgesetzt. Abbildung 4.14 sieht bereits deutlich flacher aus, als Test 1 und 2. Dies zeigt sich auch in Tabelle 4.3. Die Mittelwerte von `sampling` sind nur minimal größer, als die der `baseline`. Auch die `profile` Phase ist, mit einem Wachstumsfaktor zur `baseline` nur 1.12 ebenfalls nur minimal größer. Ein Benutzer der Anwendung würde diese Erhöhung deutlich weniger wahrnehmen, als die Verdoppelung in Test 1 und 2.

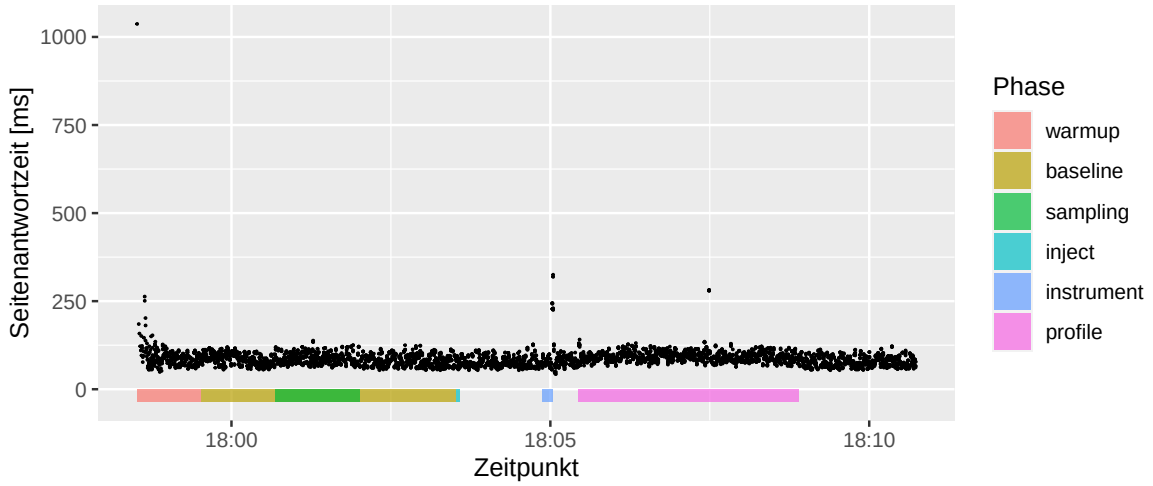


Abbildung 4.14: Testverlauf von Test 3

Phase	Seitenaufrufe	Dauer (s)	Aufrufe/s	Mean (ms)
warmup	437	59.72	7.32	90.75
baseline	1345	239.02	5.63	82.70
sampling	675	79.88	8.45	87.62
inject	35	3.51	9.97	74.37
instrument	90	10.09	8.92	95.83
profile	1730	207.65	8.33	92.98

Tabelle 4.3: Statistiken der untersuchten Phasen Test 3

4.2.1.4 Test 4 (Erhöhte Last)

In diesem Test wurde die generierte Last verzehnfacht. Es wurden zwei unterschiedliche Sampling-Phasen durchgeführt `sampling` entspricht mit einem $20ms$ Sampling-Intervall den gleichen Einstellungen wie in den vorherigen Tests. `sampling2` hingegen verwendet nur einen Sampling-Intervall von $200ms$. Nach den Sampling-Phasen wurde noch ein instrumentiertes Profiling durchgeführt, welches bei der hohen Aufrufrate des Tests jedoch die Anwendung überlastet hat. Die Seitenantwortzeiten über $2000ms$ werden in Abbildung 4.15 nicht angezeigt, in Tabelle 4.4 sind jedoch die sehr hohen Seitantwortzeiten zu sehen. Ein Profiling mit kompletter Instrumentierung ist also bei höheren Lasten mit dem entwickelten Profiler nicht möglich. Test 5 zeigt, dass dies mit reduzierter Instrumentierung dennoch möglich sein kann. Der, in `sampling2` vergrößerte Sampling-Intervall, zeigt jedoch deutlich niedrigere Antwortzeiten im Mittelwert, als in der Phase mit einem schmaleren Sampling-Intervall. Tatsächlich ist der gemessene Mittelwert sogar minimal niedriger als in den `baseline` Abschnitten. Dies ist jedoch eher durch Messrauschen zu begründen, kann jedoch auch durch externe Faktoren

wie Laufzeitoptimierungen durch die JVM oder Hintergrundprozesses des Betriebssystems begründet sein. Erkennbar ist hierdurch allerdings deutlich, dass eine höhere Sampling-Rate zu einem geringeren, negativen Impakt auf die Antwortzeiten der Anwendung führt. Generell sind in Abbildung 4.15 viele regelmäßig auftretende Ausreißer sichtbar, welche bei den vorherigen Tests nicht sichtbar waren. Diese werden sehr wahrscheinlich durch die hohe Auslastung des Systems durch die höhere Anzahl von parallelen Test-Threads verursacht.

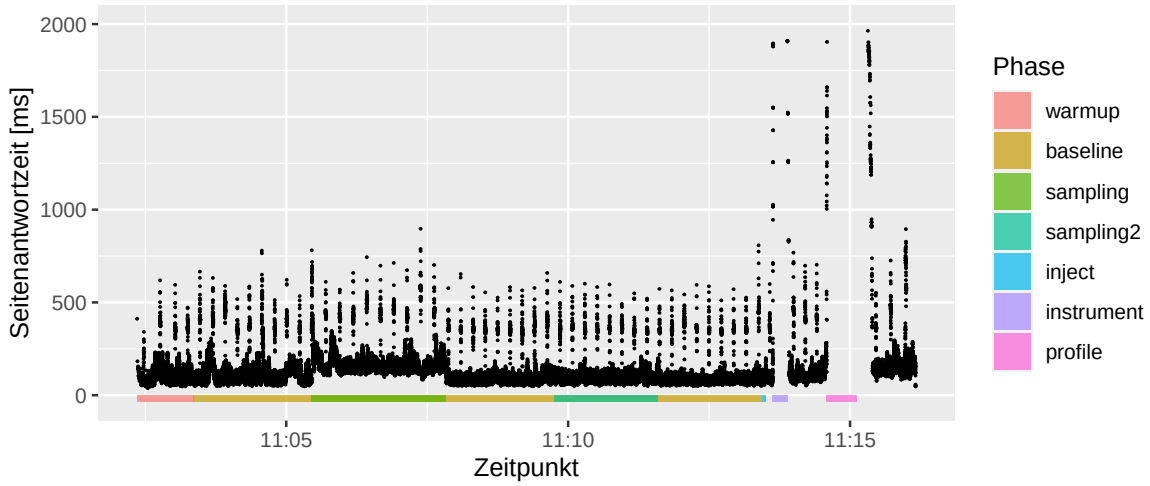


Abbildung 4.15: Testverlauf von Test 4

Phase	Seitenaufrufe	Dauer (s)	Aufrufe/s	Mean (ms)
warmup	4063	59.97	67.75	113.56
baseline	28581	604.66	47.27	105.79
sampling	10440	142.95	73.03	175.65
sampling2	8956	109.95	81.46	104.73
inject	336	3.97	84.53	89.69
instrument	333	16.82	19.79	2208.60
profile	191	31.92	5.98	9678.89

Tabelle 4.4: Statistiken der untersuchten Phasen in Test 4

4.2.1.5 Test 5 (Erhöhte Last mit top-n)

Um die **top-n**-Einstellung, welche in Test 3 bereits verwendet wurde, bei höherer Last zu überprüfen wurde für die selbe Lastgröße wie in Test 4 erneut die reduzierte Instrumentierung eingesetzt. Durch die „top-n“ und „profile Call-Tails“ Funktion wurden nur 86 Methoden instrumentiert. Dies hat, wie in Abbildung 4.16 und Tabelle 4.5 zu sehen ist, deutliche Auswirkungen auf die Seitenantwortzeiten, da die Anwendung nun, im Vergleich zu Test 4, während dem Profiling zwar langsam ist, jedoch benutzbar bleibt. Der Faktor von `baseline` auf `profile` war hier 1.85.

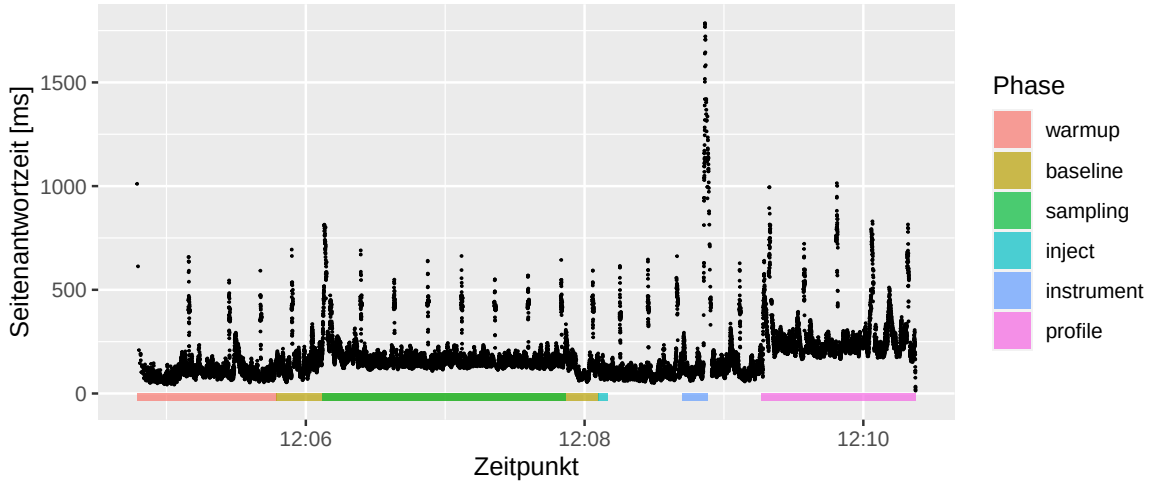


Abbildung 4.16: Testverlauf von Test 5

Phase	Seitenaufrufe	Dauer (s)	Aufrufe/s	Mean (ms)
warmup	4048	59.95	67.53	116.09
baseline	2574	138.62	18.57	143.32
sampling	7633	104.99	72.70	178.80
inject	321	3.91	82.08	102.58
instrument	804	10.90	73.73	205.22
profile	4297	66.50	64.62	264.60

Tabelle 4.5: Statistiken der untersuchten Phasen in Test 5

4.2.1.6 Fazit

In Tests mit höherer Last steigt die Seitenantwortzeit in den Profiling-Tests deutlich an. Ab einem zu hohem Datendurchsatz kommt sowohl die Übertragung der Profiling-Daten via Netzwerk, als auch das Schreiben in die Log-Datei nicht hinterher und die Seitenantwortzeiten steigen auf nicht tragbare Extremwerte von über 10 Sekunden an. Die Wachstumsfaktoren, die in den vorherigen Abschnitten bestimmt wurden, sind lediglich als exemplarisch zu verstehen, da der Performance-Einfluss stark von den System, der erzeugten Last und der untersuchten Anwendung abhängig ist. Soll die Instrumentierung in größeren Last-Situationen eingesetzt werden, ist es ratsam nur ausgewählte Methoden zu instrumentieren oder die `top-n`-Einstellung zu verwenden. In vielen Fällen wird die Nutzung des Sampling-Profiler, welcher einen deutlich niedrigeren *Overhead* hat, ausreichend sein.

Um zu überprüfen, ob das Schreiben der Log-Dateien in das Dateisystem der Flaschenhals ist, wurde in einem weiteren Test als Speicherort eine RAM-Disk genutzt. Eine RAM-Disk ist ein als Dateisystem eingehängter Teil des Arbeitsspeichers. Hierdurch werden deutlich höhere Zugriffzeiten ermöglicht. Hierbei konnten jedoch keine Verbesserungen der Profiling-Performance festgestellt werden. Der Flaschenhals ist eher, dass in vielen, normalerweise trivialen Methoden, wie „gettern“ und „settern“, zusätzlicher Overhead durch die Zeitmessung und die Verwaltung des Stacks hinzukommt.

4.2.2 Gitblit

Wie in Abschnitt 4.1.4 zu sehen war, konnten für Gitblit sowohl der Sampling-Profiler, als auch der Instrumentation-Profiler erfolgreich eingesetzt werden. Eine detailliertere Analyse der Seitenantwortzeit und eine damit einhergehende Analyse des Profiler-Overheads wurde in Gitblit jedoch nicht durchgeführt.

4.2.3 Minecraft

Beide Profiler-Modi wurden an dem Viedeospiel „Minecraft“, welches in Java implementiert ist, getestet. Der Sampling-Profiler zeichnet erfolgreich große Mengen an Informationen auf. Hierbei ist im Spiel keine spürbare Veränderung bemerkbar. Leider sind keine lesbaren Symbolnamen enthalten und alle Klassen und Methodennamen bestehen nur aus zufälligen Zeichenketten. Beispiele sind: `fba.br`, `aqu.br` und `dpq.e`. Zusätzlich tauchen noch einige Einträge der Game-Engine auf. Die Instrumentierung funktionierte leider nicht. Der Grund hierfür konnte jedoch nicht identifiziert werden. Möglicherweise wurde die Anwendung bewusst gegen Instrumentieren isoliert. Auch VisualVM ist nur in der Lage seinen Sampling-Profiler zu verwenden.

4.3 Benutzerstudie

Die Qualität der Visualisierung kann durch die reine Betrachtung der Funktionen nur schwer beurteilt werden. In einer Benutzerstudie soll evaluiert werden, wie echte Personen mit der Darstellung umgehen, ob sie innerhalb der Anwendung typische Aufgaben lösen können und wie die Anwendung im Vergleich zu einer klassischen, vergleichbaren Darstellung abschneidet. Die entwickelte Darstellung wird deshalb mit dem Profiler VisualVM verglichen.

4.3.1 Forschungsfragen

Die entwickelte Darstellung der Performance-Informationen in SEE wird auf Benutzbarkeit und Zweckmäßigkeit überprüft. Die Hauptforschungsfragen hierzu sind:

F1: Ist die Performance-Analyse in SEE gebrauchstauglich?

F2: Ist die Performance-Analyse in SEE mindestens so effektiv wie eine klassische Darstellung?

F3: Ist die Performance-Analyse in SEE mindestens so effizient wie eine klassische Darstellung?

F4: Welches Verbesserungspotential gibt es?

Effektiv bedeutet hier, ob in der Benutzerstudie gestellte Aufgaben erfolgreich gelöst werden können oder nicht. Die Effizienz meint die Lösungsgeschwindigkeit. Dauert die Bearbeitung der Aufgaben in der implementierten Umgebung deutlich länger, bietet die entwickelte Anwendung keinen Vorteil gegenüber klassischen Anwendungen und benötigt weitere Optimierungen. Zur Beantwortung von F2 wurden konkrete Aufgaben gestellt, welche die Testpersonen lösen sollten. Diese Aufgaben wurden ebenfalls in einer alternativen Anwendung bearbeitet, welche vergleichbare Funktionen bereitstellt. Hierbei wurden jeweils Bearbeitungszeiten gemessen, um F3 beantworten zu können. Für die Beantwortung von F1 wurden hierzu im Fragebogen entsprechende Fragen gestellt. Zusätzlich wurde der „System Usability Score“ (SUS) [104] berechnet, welcher Aufschluss über die Benutzbarkeit des Systems gibt. Freitextantworten und Gespräche mit den Testpersonen gaben Rückmeldung bezüglich F4. Hier wurden Ideen gesammelt, um die Anwendung zukünftig besser zu gestalten.

Da die Teilnahme für eine Testperson nicht länger als 60 bis 90 Minuten dauern und die Komplexität des Experimentes niedrig bleiben sollte, wurde die Visualisierung für die Studie eingeschränkt. Es wurde lediglich die Darstellung der Sampling-Profiler Ergebnisse verwendet, damit die Testpersonen nicht erst den Unterschied zwischen den einzelnen Metriken und der *self time* verstehen müssen. Zudem wurden keine Zahlenwerte im Informationspanel angezeigt. Die Testszenarien wurden so gewählt, dass die Filterfunktion nicht benötigt wird. Die Schieberegler für diesen Filter wurden deshalb ebenfalls deaktiviert. Die Testpersonen führen nur die Visualisierung aus, da das Profiling bereits vor Studienbeginn durchgeführt wurde, sodass alle Testpersonen die gleichen Profiling-Ergebnisse sehen.

4.3.2 Vorbereitung

In der Studie wurden zwei Anwendungen miteinander verglichen: Die 3D-Ansicht für SEE und eine vergleichbare textuelle 2D Darstellung innerhalb der Netbeans [93] Entwicklungsumgebung, welche den Profiler „VisualVM“ [60] integriert. Im Folgenden ist mit „SEE“ primär

der, innerhalb dieser Arbeit, entwickelte Teil der Anwendung gemeint und nur sekundär die gesamte Anwendung SEE. „Netbeans“ bezieht sich in den folgenden Abschnitten auf die Kombination aus der Entwicklungsumgebung und den integrierten Profiler, obwohl Netbeans und VisualVM auch separat voneinander verwendbar wären.

Beide Anwendungen stellen Profiling-Ergebnisse hierarchisch dar und können diese direkt mit dem Quellcode verknüpfen. In SEE lässt sich der Quellcode durch Mausklicks auf die 3D-Objekte abrufen. In Netbeans hingegen über den Kontextmenüeintrag „Open Source“. In Abschnitt 4.3.3 wird deutlicher, wie Netbeans und SEE funktionieren. Zwei, für die Testpersonen erstellte Videos erläutern die Aufgabenstellung näher. Weiteres zu diesen Videos in Abschnitt 4.3.5.2.

Alle Testpersonen haben beide Anwendungen ausprobiert und bewertet. Hierfür gab es jeweils einen Fragebogen, welcher unter anderem auch den „System Usability Score“-Fragebogen [104] beinhaltet. Jede Testperson löste pro Anwendung jeweils eine Aufgabe, in welcher sie einen *Performance-Bug* finden musste. Diese Performance-Bugs wurden im Voraus händisch in die Anwendung *libAwesome* eingebaut. Hiermit sind Stellen in der Software gemeint, welche durch ergänzten Code verlangsamt werden. Dieser neue Code hat keine Auswirklungen auf die sonstige Funktionalität der Anwendung. Es gibt drei Szenarien E, B und C. Szenario E ist eine Einführungsaufgabe, bei welcher Fragen gestellt werden konnten. Szenario B und C sind die eigenständig zu lösenden Aufgaben[3]. Für diese Szenarien wurde die Anwendung bereits im Voraus profiliert, sodass die Testpersonen nur die Visualisierung evaluieren, jedoch nicht den Profiling-Prozess an sich.

Im Folgenden werden die drei Performance-Bugs beschrieben.

4.3.2.1 Performance-Bug E

Der Performance-Bug in Szenario E (siehe Abbildung 4.2) ist eine absichtliche Verlangsamung durch einen Aufruf von `Thread.sleep(20)` auf Zeile 199. Da diese Methode häufig aufgerufen wird, summieren sich die 20 gewarteten Millisekunden über die vielen Aufrufe stark auf. Der umschließende `try-catch`-Block wurde ebenfalls hinzugefügt. Der gesuchte Fehler befindet sich also in der Methode `getNumberOfMediums()`. Insgesamt hinzugefügt wurden Zeilen 198 bis 202.

4.3.2.2 Performance-Bug B

Der Performance-Bug in Szenario B (siehe Abbildung 4.3) gibt eine große Menge an „hello world“-Textausgaben auf der Standardausgabe aus. Egal in was für einer Anwendung, diese Ausgabe ist in der Getter-Methode `getTitle()` fehlplatziert. Ausgaben auf der Standardausgabe können in großen Mengen bemerkbare Auswirkungen auf die Laufzeit haben, wie sich in dargestellten Profiling-Ergebnissen zeigt. Insgesamt hinzugefügt wurden Zeilen 867 bis 870.

4.3.2.3 Performance-Bug C

Szenario C ist etwas komplexer. In Abbildung 4.4 ist die Methode `getCurrentUser()` zu sehen. In ihr wurde ein großer try-catch-Block platziert, welcher einen HTTP-Aufruf gegen eine URL sendet und die Größe der Antwort auf der Standardausgabe ausgibt. Der gesamte Block ist für die Methode unerheblich. Es ist jedoch vorstellbar, dass für eine Methode mit

[3]Der Grund für die Namen B und C anstelle von A und B ist, dass eine frühe Überlegung für ein Etwaiges Szenario A frühzeitig verworfen wurde.

```
187 /**
188  * Liefert die Anzahl der (nicht geloeschten) Medien, die die
189  * gegebenen Constraints erfuellen.
190  *
191  * @param constraints zu erfuellenden Constraints der zu zaehlenden Medien
192  * @return Anzahl der (nicht geloeschten) Medien, die die
193  * gegebenen Constraints erfuellen
194  * @throws DataSourceException bei Problemen mit der Persistenz-Komponente.
195  */
196 public int getNumberOfMediums(List<Constraint> constraints) throws DataSourceException {
197
198     try {
199         Thread.sleep(20);
200     } catch (InterruptedException e) {
201         e.printStackTrace();
202     }
203
204     return persistence.getNumberOfMediums(addUnDeleted(constraints));
205 }
```

Text 4.2: Performance-Bug in Szenario E. Zeilen 198 bis 202 wurden hinzugefügt.

```
862 /**
863  * Liefert den Titel des Mediums
864  *
865  * @return Der Titel des Mediums
866  */
867 public final String getTitle() {
868     for (int i=0; i<1000; i++){
869         System.out.println("hello world "+ i);
870     }
871     return title;
872 }
```

Text 4.3: Performance-Bug in Szenario B. Zeilen 868 bis 870 wurden hinzugefügt.

dem Namen `getCurrentUser` externe Aufrufe gemacht werden. Es kann demnach sein, dass dieser Performance-Bug nicht direkt als offensichtlich unnötig identifiziert wird. Der langsame Teil ist Zeile 147, auf welcher der Inhalt geladen wird. Die Standardausgabe ist hier nicht das wesentliche Problem. Benannt werden muss von den Testpersonen jedoch nur die Methode `getCurrentUser` und keine spezielle Zeile. Insgesamt hinzugefügt wurden Zeilen 143 bis 157. Die aufgerufene Domain befand sich zum Zeitpunkt dieser Arbeit in meinem Besitz. Die Rückgabe der Seite war eine einzeilige Antwort mit dem Inhalt `Hello World`.

```
130 /**
131  * Gibt den aktuell angemeldeten Nutzer zurück.
132  *
133  * @return Aktuell angemeldeter Nutzer oder null, wenn diese Bean zu keinem
134  * Nutzer gehört.
135  */
136 protected User getCurrentUser() {
137     ExternalContext context = getExternalContext();
138 
139     if (context == null) {
140         return null;
141     }
142 
143     try {
144         //
145         URL url = new URL("http://syrver.de/");;
146 
147         BufferedReader read = new BufferedReader(new InputStreamReader(url.openStream()));
148         StringBuilder page = new StringBuilder();
149         String line;
150         while ((line = read.readLine()) != null){
151             page.append(line);
152         }
153         System.out.println("Loaded testdomain. Response size: " + page.length());
154         read.close();
155     } catch (Exception e) {
156         e.printStackTrace();
157     }
158 
159 
160     return (User) context.getSessionMap().get("currentUser");
161 }
```

Text 4.4: Performance-Bug in Szenario C. Zeilen 143 bis 157 wurden hinzugefügt.

4.3.3 Darstellungen in SEE und Netbeans

Für alle Szenarien wurden vor der Studiendurchführung die Performancedaten erhoben. Die Abbildungen 4.17, 4.18, 4.19 zeigen für alle drei Szenarien wie die aufgezeichneten Performancedaten in SEE dargestellt aussehen. Die Kamera wurde für die Abbildungen so umpositioniert, dass möglichst alle Kanten gut sichtbar sind. Abbildung 4.20, 4.21 und 4.22 ab Seite 92 zeigen analog die Darstellung in Netbeans. In Netbeans wurde der hierarchische Aufruf-Baum für die Abbildungen teilweise aufgeklappt, sodass die gesuchte Methode in dem Bildschirmfoto sichtbar ist. Zusätzlich ist die Hot-Spot-Ansicht bereits geöffnet, welche die Testpersonen selber öffnen mussten.

Eine Gemeinsamkeit, die alle drei Aufgaben in SEE haben ist, dass die größte ausgehende Kante jeweils auch den Performance-Fehler als Caller (linkes Quellcode-Fenster) enthält. Es wäre interessant gewesen, ebenfalls eine Aufgabe mit einem Fehler zu verwenden, welcher mittiger im Aufrufgraphen positioniert ist. Die Testpersonen haben jedoch in vielen Fällen nicht den „trivialen Lösungsweg“ gewählt oder haben trotzdem den Fehler nicht sofort erkannt. Einen vergleichbaren „trivialen Lösungsweg“ gibt es auch in Netbeans. Dieser wird in den folgenden Beschreibungen der möglichen Lösungswege erläutert. Weiteres beobachtetes Verhalten der Testpersonen wird in Abschnitt 4.3.6.3 und 4.3.6.4 der Auswertung beschrieben. Im Folgenden werden mögliche Lösungswege beschrieben, um den Ablauf der Aufgaben zu verdeutlichen.

4.3.3.1 Lösungsweg Einführungsaufgabe E (SEE)

Ein möglicher Lösungsweg beginnt mit dem Auswählen der Kante ①, da diese am auffälligsten ist. Sie spaltet sich an der aufgerufenen Methode in Kante ② und ③ auf. Nur eine dieser beiden ausgehenden Kanten führt zum gesuchten Performance-Bug. Navigiert die Person hier also zunächst Kante ② entlang, muss sie eventuell später zurück navigieren. Von Kante ③ ausgehend kann die Person dann weiter zu Kante ④ navigieren. In allen Schritten schaut die Person sich zumindest kurz den angezeigten Quell-Code an, welcher durch das Auswählen der Kanten eingeblendet wird. Sobald Kante ④, wie in Abbildung 4.17, ausgewählt ist, kann die Person potentiell den Fehler erkennen, da er im rechten Fenster sichtbar ist. Der angezeigte Fehler ist hier das im Abschnitt 4.3.2.1 beschriebene, nicht notwendige `Thread.sleep`. Dieses ist in Abbildung 4.17 im rechten Quell-Code Fenster sichtbar und als Hot-Spot markiert.

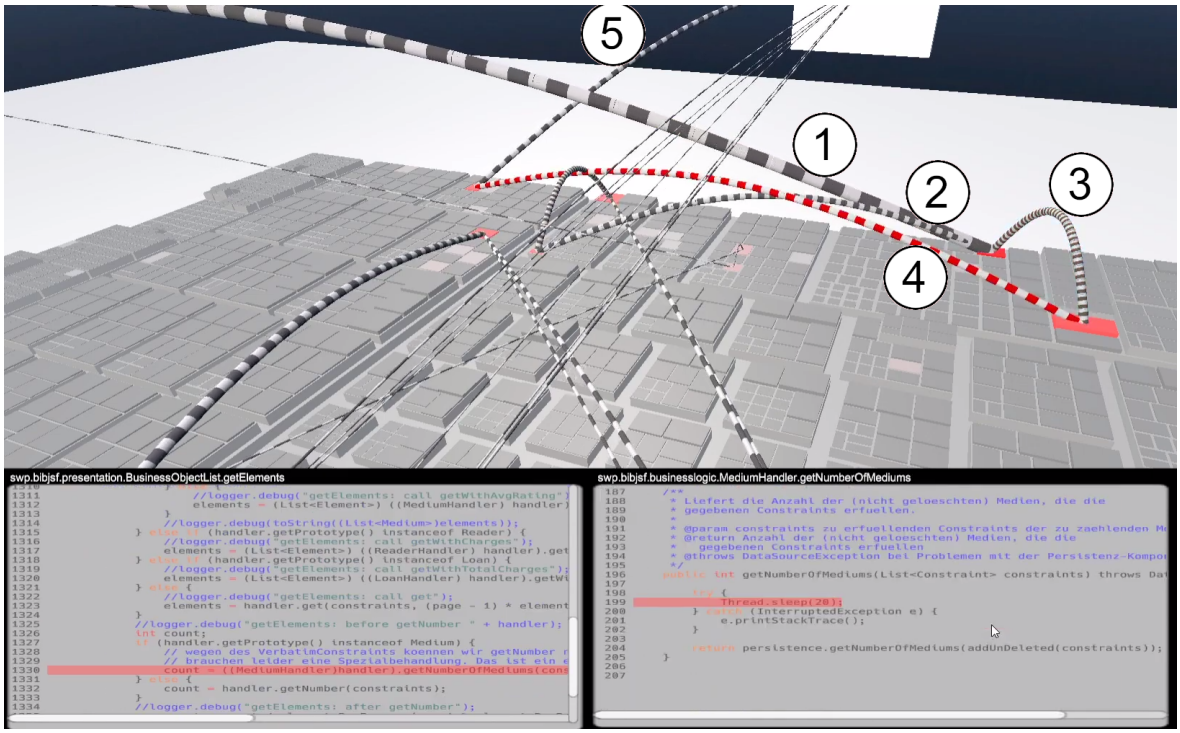


Abbildung 4.17: Gesuchter Methodenaufruf in der Einführungsaufgabe (SEE)

4.3.3.2 Lösungsweg Aufgabe B (SEE)

In der Einführungsaufgabe war die dickste Kante nicht direkt die gesuchte Lösung. Wählt die Testperson in Aufgabe B direkt Kante ⑥ aus, kann sie im linken Quell-Code-Fenster bereits den Fehler erkennen. Versucht sie jedoch, wie in der Einführungsaufgabe bei den eingehenden Kanten zu beginnen, kann sie sich über mehrere Pfade durch den Aufrufgraphen navigieren. Sobald sie bei Kante ④, ⑤ oder, ⑥ angekommen ist, kann sie den Fehler bereits sehen. Der Fehler ist die rot, als Hot-Spot, markierte Zeile und die umschließende `for`-Schleife, die hervorgehobenen Zeilen geben hier also einen recht deutlichen Hinweis.

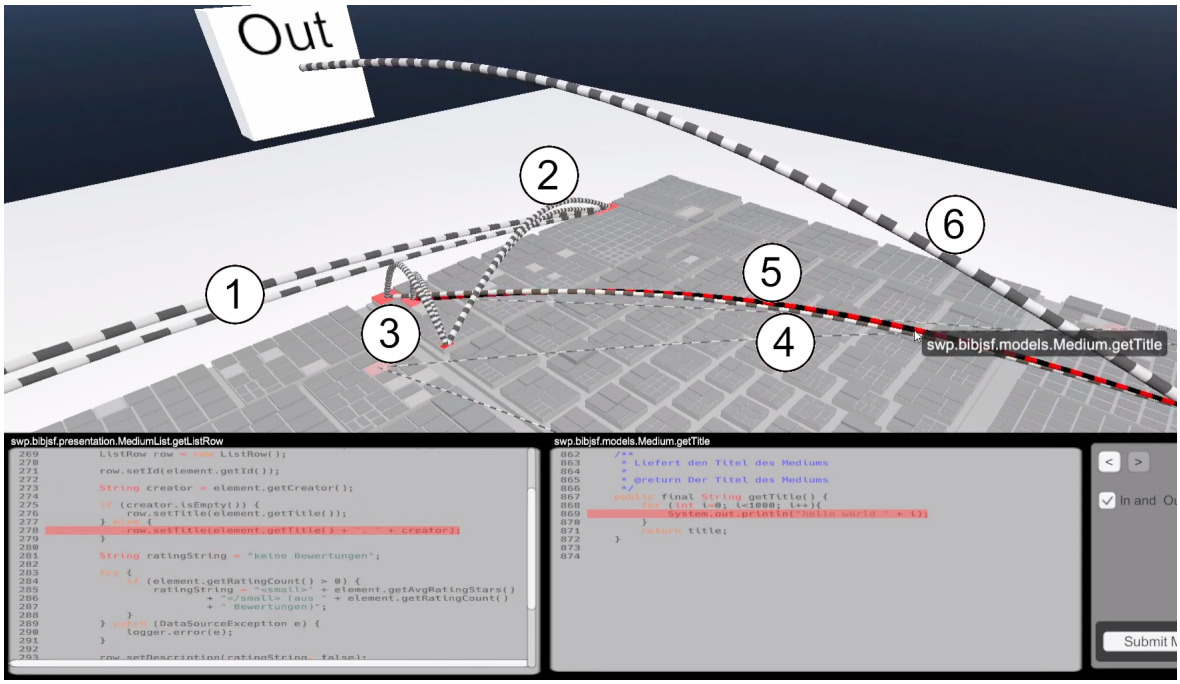


Abbildung 4.18: Gesuchter Methodenaufruf in Aufgabe B (SEE)

4.3.3.3 Lösungsweg Aufgabe C (SEE)

Ähnlich wie in Aufgabe B kann auch hier direkt zu der ausgehenden Kante ③ navigiert werden. Der Fehler ist jedoch etwas versteckter als in Aufgabe B. Da die drei dicken Kanten bei ② relativ viele eingehende Kanten bei ① haben, welche direkt oder indirekt die Methode `getCurrentUser()` aufrufen, gibt es insgesamt wenige dicke eingehende Kanten. Der Fokus könnte also eher auf die drei größten Kanten gelenkt werden. Um den Fehler zu erkennen, muss verstanden werden, dass die als Hot-Spot markierte Zeile einen HTTP-Aufruf macht, welcher im Kontext der Methode nicht notwendig ist. Hinweise darauf, dass dieser Abschnitt verdächtig ist, geben die im Quell-Code stehende URL und die Ausgaben auf dem Standard-Ausgabe mit einem Text, welcher das Wort `testdomain` enthält. Da jedoch nur ein kleiner Teil des Fehlerkonstruktes als Hot-Spot markiert wird, könnte diese Aufgabe eher zu Verwirrungen führen, als Aufgabe B. Dies zeigt sich auch durch eine höherere Fehlerrate in der Auswertung der Benutzerstudie in Abschnitt 4.3.6.2.

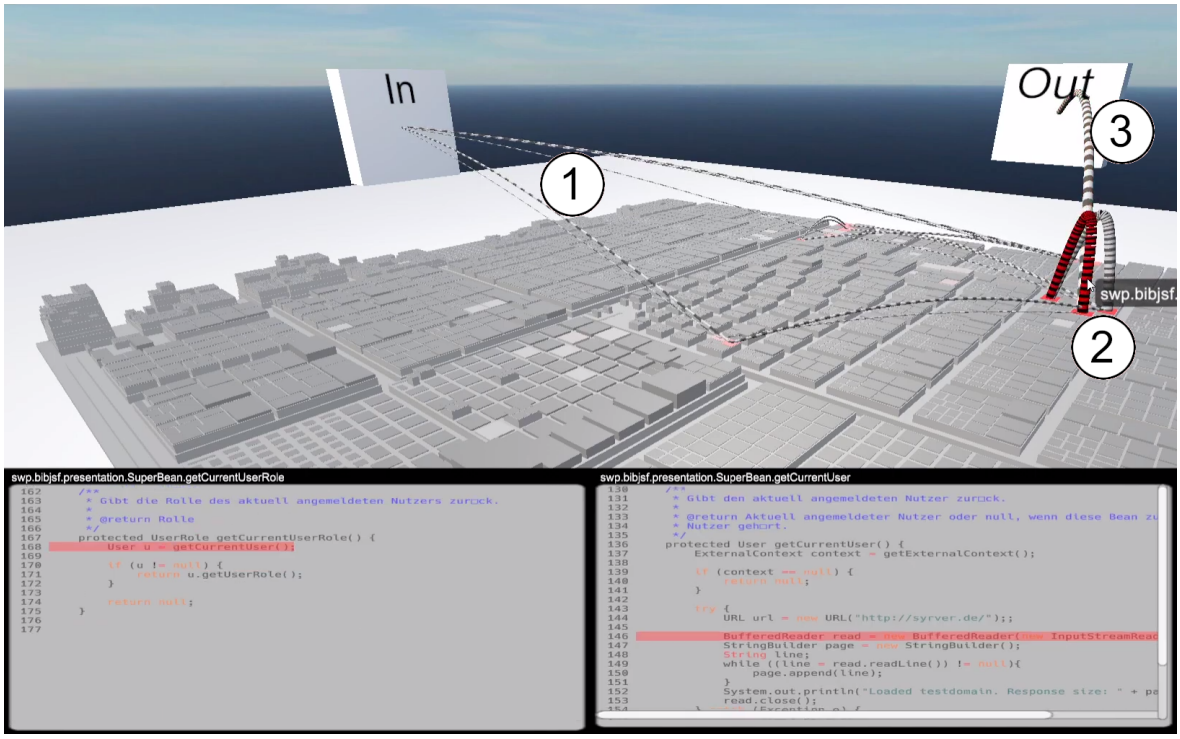


Abbildung 4.19: Gesuchter Methodenaufruf in Aufgabe C (SEE)

4.3.3.4 Lösungsweg Einführungsaufgabe E (Netbeans)

In Netbeans und in VisualVM, stehen dem Anwender 3 Ansichten zur Verfügung. Die *Forward-Calls*, die *Hot-Spots* und die *Backward-Calls*. Die Testpersonen sollten für die Bearbeitung der Aufgaben nur die *Forward-Calls* und die *Hot-Spots*-Ansicht verwenden. Die *Forward-Calls* sind standardmäßig geöffnet. Die *Hot-Spots*-Ansicht muss zusätzlich geöffnet werden. In Abbildung 4.20 ist im oberen Abschnitt des Arbeitsbereiches die *Forward-Calls*-Ansicht sichtbar und im unteren Abschnitt die *Hot-Spots*-Ansicht. In den *Forwards-Calls* ist der Aufrufbaum, zeilenweise aufklappbar, dargestellt. Hier tauchen einzelne Methoden mehrfach auf, wenn sie von unterschiedlichen Aufrufpfaden aufgerufen werden. In der *Hot-Spots*-Ansicht tauchen Methoden nur einmalig auf und es wird die Gesamtstatistik angezeigt. Diese beiden Ansichten entsprechen respektive den Aufrufkanten in der Visualisierung in SEE und den, farblich hervorgehobenen, Methoden-Blöcken.

Die Testperson bekommt gezeigt, wie sie den Baum nach dem Paketnamen `swp` filtert und wie sie die Ansichten wechselt. Während der Einführungsaufgabe wird sie ebenfalls darauf hingewiesen, dass die *Hot-Spots*-Tabelle zunächst nach *Total Time* sortiert und die Liste nach `swp` gefiltert werden sollte.

Die Testperson öffnet in der *Forward-Calls* Ansicht einzelne Teilbäume und betrachtet die, rechts von den Methodennamen angegebenen, approximierten Gesamtlaufzeiten. Ist der Laufzeitanteil hoch, so navigiert sie weiter in einen Teilbaum hinein oder schaut sich für diese Methode den Quell-Code an. Findet sie im Quell-Code eine verdächtige Stelle, benennt sie diese und gibt so die Aufgabe ab oder sucht in weiteren Methoden nach anderen Auffälligkeiten. Ein aufgeklappter Teilbaum in den *Forward-Calls* ist im oberen Teil von Abbildung 4.20 zu sehen. Findet die Person den Fehler nicht in dieser Ansicht, kann sie die Hot-Spot Ansicht öffnen. In der *Hot-Spots* Ansicht ist der Fehler recht weit oben in der Liste auffindbar. Das Fehlerverständnis innerhalb des Quell-Codes birgt für die drei Aufgaben die selben Schwierigkeiten wie bereits für die Aufgaben in SEE beschrieben wurde.

4.3.3.5 Lösungsweg Aufgabe B und C (Netbeans)

Die Lösungswege in Aufgabe B und C unterscheiden sich nicht wesentlich zu dem der Einführungsaufgabe. Wie in SEE auch, gibt es in Aufgabe C und B einen „trivialen Lösungsweg“. In der *Hot-Spots*-Ansicht sind jeweils die oberste Methode der *Hot-Spots*-Ansicht die gesuchte Methode. Dies ist zu sehen in Abbildung 4.21 und 4.22. Auf die Einführungsaufgabe trifft dies nicht zu, hier ist die gesuchte Methode erst als fünfter Eintrag gelistet (Siehe Abbildung, 4.20). Genau wie in SEE wurde dieser „triviale Lösungsweg“ jedoch von den Testpersonen kaum, oder erst im späteren Bearbeitungsverlauf genutzt.

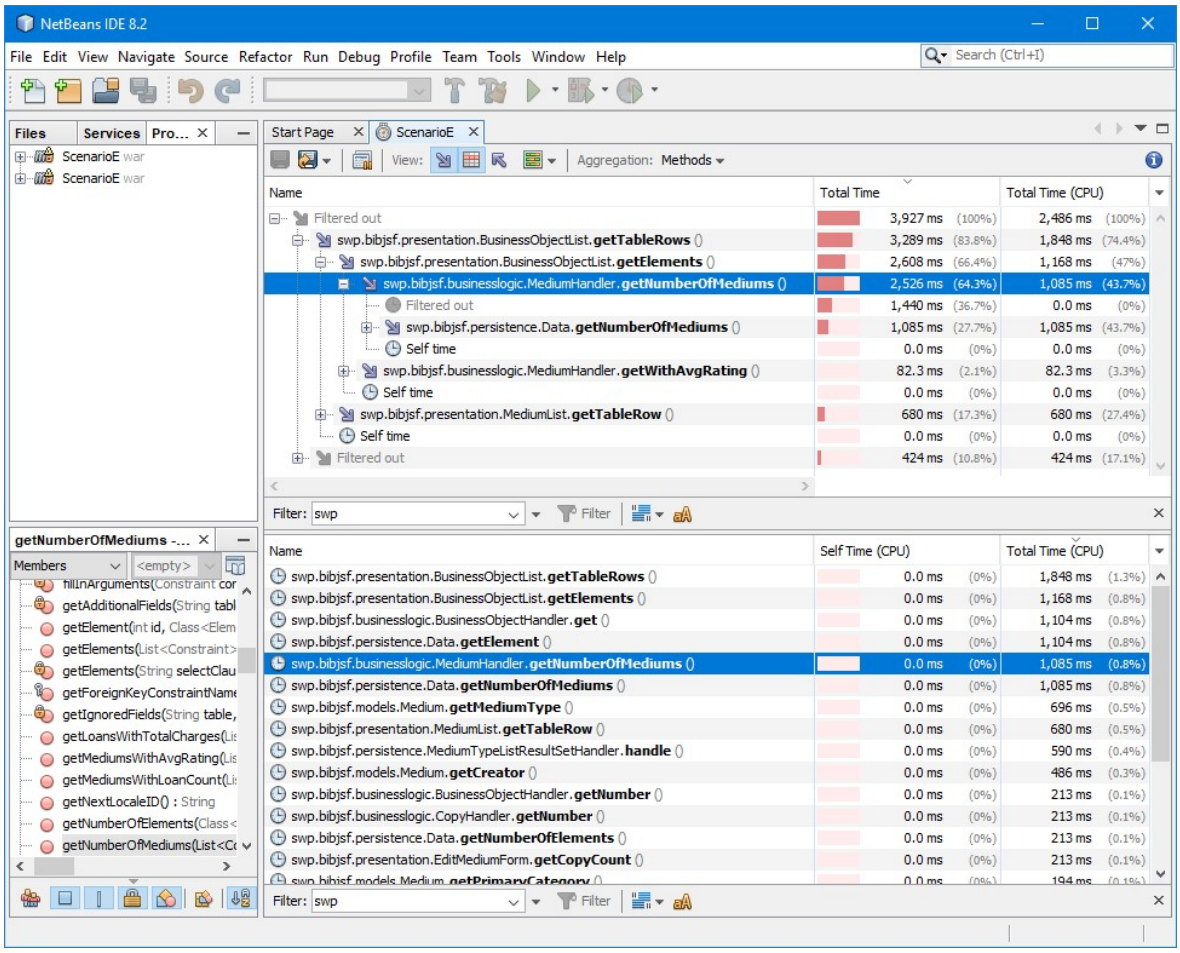


Abbildung 4.20: Gesuchter Methodenaufruf in der Einführungsaufgabe in der *Forward-Call*-Ansicht und der *Hot-Spots*-Ansicht von Netbeans

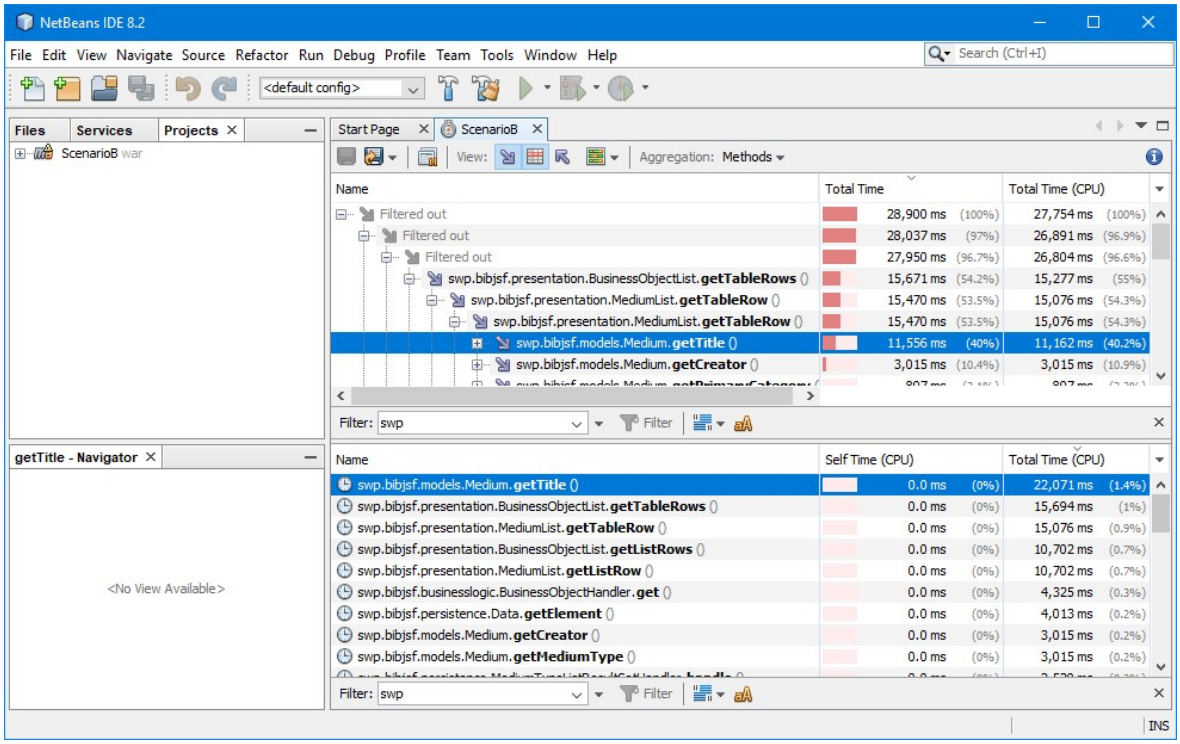


Abbildung 4.21: Gesuchter Methodenaufruf in Aufgabe B in der *Forward-Call*-Ansicht und der *Hot-Spots*-Ansicht von Netbeans

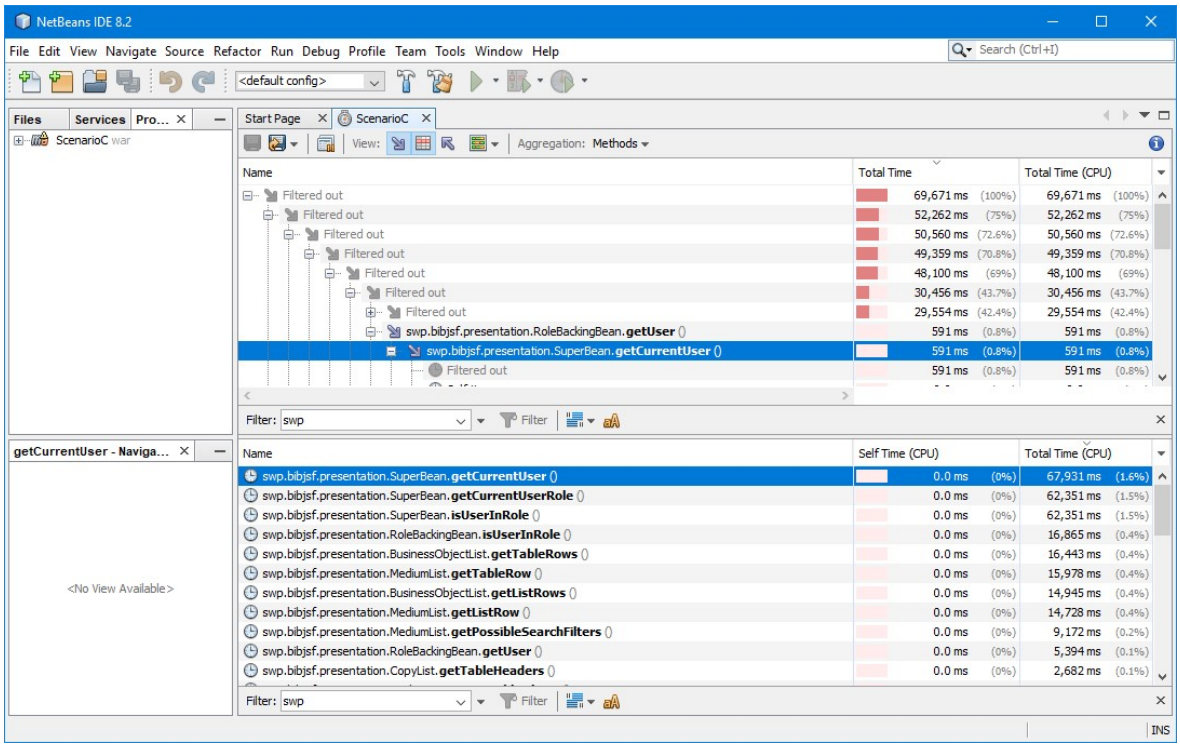


Abbildung 4.22: Gesuchter Methodenaufruf in Aufgabe C in der *Forward-Call*-Ansicht und der *Hot-Spots*-Ansicht von Netbeans

4.3.4 Ablauf

Für die Studie wurden Teilnehmer aus mehreren Umfeldern eingeladen. Alle Teilnehmer sollten bereits grundlegende Erfahrungen in der objektorientierten Softwareentwicklung und dem Softwaretest haben.

Eingeladen wurden Arbeitskollegen und Arbeitskolleginnen meines aktuellen Arbeitgebers *Dataport AöR* und Bekannte, welche großteils ehemalige Studierende der Universtiät Bremen sind. Hinzu kommen Studierende aus dem Umkreis des SEE-Projektes von Rainer Koschke. Die letzte Gruppe hat bereits engere Erfahrungen mit dem SEE-Projekt und stellt deshalb eine Besonderheit dar. Auch Rainer Koschke, der Erstgutachter dieser Arbeit, hat an der Studie teilgenommen und zählt zu dieser Gruppe.

Die Aufgabenstellungen und die Reihenfolge der Aufgaben sollten möglichst gleich verteilt sein und werden deshalb rotierend gestellt. Tabelle 4.6 zeigt eine mögliche Reihenfolge. Auch die Reihenfolge der Software kann für die Auswertung eine wichtige Variable sein, weshalb nicht nur die Aufgaben B und C, sondern zusätzlich auch die Anwendungen alterniert wurden. So ergaben sich vier unterschiedliche Aufgabenkonstellationen, welche möglichst gleichmäßig mit Testpersonen belegt werden sollen.

Testperson	**Gruppe**	**Aufgabe 1**	**Aufgabe 2**
1	1	C in SEE	B in Netbeans
2	2	C in Netbeans	B in SEE
3	3	B in SEE	C in Netbeans
4	4	B in Netbeans	C in SEE
5	1	C in SEE	B in Netbeans
6	2	C in Netbeans	B in SEE
7	3	B in SEE	C in Netbeans
8	4	B in Netbeans	C in SEE
...	...	...	...

Tabelle 4.6: Geplante Aufteilung der Testpersonen auf vier Kombinationen der Anwendungen und Aufgaben

4.3.5 Experimentablauf

Die Aufgaben und eine zugehörige Präsentation wurden auf dem Computer der Studienleitung durchgeführt. Die Testpersonen verbinden sich mit der Bildschirm-Fernsteuerungsssoftware TeamViewer [105] oder AnyDesk [106] auf diesen Rechner. Die Studienleitung kann also beobachten, was die Testperson in den Anwendungen macht. Eine Vor-Ort-Studie ist, bedingt durch COVID-19 nicht möglich gewesen. Während der Studie wird über einen beliebigen Telefon oder Voice-Over-IP Dienst telefoniert, hierfür wurden das betriebsinterne Kommunikationstool *Skype for Business* [107], *Discord* [108] und Zoom [109] verwendet. Eine Videoübertragung der Personen, zusätzlich zu der Bildschirmübertragung, fand nicht statt. Der begleitende Fragebogen wurde, von der Studienleitung nicht direkt einsehbar, auf dem Computer der Testperson ausgefüllt.

Jedes Experiment hatte folgende festgelegte Struktur:

1. Anruf mit Testperson
2. Öffnen des Fragebogens
   (a) Zustimmung der Teilnahmeerklärung
3. Fragebogen Abschnitt 1 (Allgemein)
4. Verbindung mit Teamviewer/AnyDesk
5. Allgemeine Einführung (Folien)
6. Aufgabe 1
   (a) Einführunsvideo
   (b) Einführungsaufgabe
   (c) Aufgabe
   (d) Fragebogen Abschnitt 2a (Anwendungsbezogen)
7. Aufgabe 2
   (a) Einführunsvideo
   (b) Einführungsaufgabe
   (c) Aufgabe
   (d) Fragebogen Abschnitt 2b (Anwendungsbezogen)
8. Fragebogen Abschnitt 3 (Vergleich)

Nachdem die Verbindung zwischen Studienleitung und Testperson aufgebaut ist und die Teilnahmeerklärung bestätigt wurde, wurde zunächst der allgemeine Teil des Fragebogens ausgefüllt. Die Teilnahmeerklärung kann dem Fragebogen in Anhang A.4 entnommen werden. Zusätzlich lag der Testperson ein Dokument mit der Teilnahmeerklärung und Kommentaren zur Datenschutz-Grundverordnung vor. Auch dieses Dokument ist dem Anhang in A.7 zu entnehmen. Anschließend gab es eine kurze, frei gesprochene Einführung, welche mit der Unterstützung von Präsentationsfolien den Ablauf der Studie erklärt. Es wird erwähnt, dass es zwei Aufgaben geben wird, welche jeweils mit einer Einführung in die Anwendung beginnen. Bestandteil dieser Einführungen war jeweils ein circa fünf Minuten langes Instruktionsvideo, in welchem die Benutzung der Anwendung erklärt wird. Die Instruktionsvideos sind in Abschnitt 4.3.5.2 verfügbar. Nachdem die Testperson das Video angesehen hat, hat sie nun die Möglichkeit, ohne Zeitdruck, sich das Einführungsszenario E in der jeweiligen Anwendung anzuschauen. Hier können, die im Video gelernten Funktionen, ausprobiert werden und Fragen an die Studienleitung gestellt werden. Die Studienleitung sorgt dafür, dass in den Einführungsszenarien die wichtigen Funktionen alle einmal ausprobiert werden. In Szenario E ist bereits ein Performance-Bug platziert. Dieser wurde entweder von der Testperson eigenständig gefunden, oder die Studienleitung unterstützte bei der Suche, indem grobe Hinweise zum Vorgehen gegeben wurden, wie beispielsweise:

> „Schau dich genauer bei den dickeren Kanten/Rohren um.“
>
> „Der platzierte Fehler ist möglicherweise einfacher zu Erkennen als du vermutest. In den Aufgaben ist es nicht unbedingt notwendig den Code fachlich bis ins tiefste Detail zu verstehen.“

Sobald die Testperson sich sicher genug fühlt, die richtige Aufgabe zu starten, wird das Szenario gewechselt. Die Testperson löst ohne inhaltliche Unterstützung der Studienleitung die Aufgabe und meldet zurück, sobald sie den Performance-Bug gefunden hat. In der Aufgabe in SEE passiert diese Rückmeldung über einen Abgabe-Knopf. In der Netbeans-Aufgabe musste verbal deutlich gemacht werden, wenn abgegeben wird und die Lösungszeit wurde manuell durch die Studienleitung gestoppt. Die Lösungszeit in der SEE-Aufgabe wird automatisch gespeichert. Zusätzlich wurde automatisch die Zeit gestoppt, welche benötigt wurde bis der Quell-Code des Fehlers zum ersten mal geöffnet wurde. Über diese zusätzliche Zeitmessung wussten die Testpersonen nicht Bescheid. Sie wussten jedoch, dass im Allgemeinen die Bearbeitungszeit ermittelt und untersucht wird.

Die Testperson hatte jederzeit die Möglichkeit abzubrechen, wenn sie glaubt den Fehler nicht finden zu können. Für die Aufgaben waren ungefähr zehn Minuten vorgesehen. Fand die Testperson den Fehler in dieser Zeit nicht, oder wirkte zunehmend verzweifelt, wurde erneut angeboten, dass die Aufgabe abgebrochen werden kann. Falls die Testperson jedoch noch Fortschritte bei der Suche machte, wurde nicht zwangsweise nach diesen zehn Minuten abgebrochen.

In der zweiten Aufgabe wird erneut Szenario E für die Einführung verwendet. Da für Aufgabe zwei pro Testperson die jeweils andere Software verwendet wurde, wurde auch die Einführungsaufgabe in ihrer zweiten Durchführung mit der jeweils anderen Software durchgeführt. Abgesehen von der Einführungsaufgabe hat keine Testperson hat eine Aufgabe oder Software zweifach verwendet.

Nach den Aufgaben wurden jeweils Abschnitte des Fragebogens ausgefüllt. Im Anschluss gab es noch die Möglichkeit sich in einem freien Gespräch über die Anwendung zu unterhalten. Hier gemachte Kommentare werden von der Studienleitung als Notizen aufgenommen, jedoch nicht wörtlich in dieser Arbeit zitiert. Eine Audioaufnahme und Transkription findet nicht statt.

4.3.5.1 Fragebogen

Der Fragebogen befindet sich im Anhang im Abschnitt A.4 sowohl im Form von Bildschirmfotos der Web-Oberfläche in der die Testpersonen ihn ausgefüllt haben, als auch in tabellarischer Textform. Der Fragebogen beinhaltet vier Abschnitte. Der erste Abschnitt beinhaltet allgemeine Fragen zu den persönlichen Erfahrungen der Testperson mit Softwareentwicklung, -test und -profiling. Abschnitt 2 und 3 beziehen sich auf die gelösten Aufgaben in SEE und Netbeans. Hier werden Fragen zu der Aufgabe und Fragen der verwendeten Anwendung gestellt. Es wurde speziell nach der Nützlichkeit einzelner Funktionen gefragt.

Pro Anwendung wurden zudem jeweils zehn Fragen des „System Usability Score“ (SUS) [104] gestellt, mit welchem einfach die Benutzbarkeit einer Anwendung oder eines Systems geprüft werden kann. Die Fragen des SUS sind im Original auf Englisch. Da es, meinen Recherchen nach, keine standardisierte deutsche Version gibt, wurde die deutsche Übersetzung gewählt, welche ich am häufigsten finden konnte [110, 57]. Unter anderem steht diese Formulierung auch auf Wikipedia [111]. Die dort angegebene Quelle ist jedoch lediglich ein Webblog [112] von 2011. Alternative Übersetzungen unterscheiden sich jedoch nur geringfügig im Wortlaut.

Alle Fragen des Fragebogens befinden sich im Anhang in Abschnitt A.4.

Während des Verlaufs der Studie wurde ein kleiner Teil an Fragen ergänzt und bei wenigen Fragen die Antwortmöglichkeiten erweitert. Testpersonen mit der unveränderten Version des Fragebogens hatten immer die Möglichkeit keine Antworten zu geben oder freie Antworten im direkten Gespräch zu benennen. Zusätzlich gab es Freitextfelder. Ist im Gespräch deutlich geworden, dass die Antwortmöglichkeiten die Antwort der Testperson einschränken, so wurde dies im Nachhinein betrachtet und entsprechend für die Auswertung angepasst.

4.3.5.2 Präsentationsfolien und Instruktionsvideos

Die Testpersonen wurden zu Beginn der Studie mit einer kurzen Folien-Präsentation instruiert. Es wurden grobe Erklärungen zur Art der in den Aufgaben dargestellten Metriken erläutert und das Formular für den Fragebogen vorgestellt. Die Folien sind im Anhang A.6 einsehbar.

Die für die Einführung gezeigten Videos beinhalten visuelle und sprachliche Erklärungen. Die Videos wurden den Testpersonen über YouTube [113] bereitgestellt. Hier sind die Videos auch nach Abgabe dieser Arbeit noch einige Zeit verfügbar. Zusätzlich befinden sich die Transkriptionen der Videos im Anhang in Text A.1 und Text A.2 und in digitaler Form auf dem beiliegenden USB-Stick. Die Videos selbst sind ebenfalls auf diesem USB-Stick verfügbar. Siehe Hierzu Anhang A.1.

- Netbeans Instruktionsvideo auf: `https://youtu.be/SQhtTMLBmG0`
- SEE Instruktionsvideo auf: `https://youtu.be/hcJAJRPg8Y0`

4.3.5.3 Telemetrie

Um Bearbeitungszeiten und das Verhalten der Testpersonen im Nachhinein analysieren zu können wurden „Telemetrie“-Funktionen in die Anwendung eingebaut, welche einige Benutzeraktionen in ein Protokoll schreiben. Zu diesen Informationen gehören Mausklicks auf Aufrufkanten oder Methoden-Blöcke, sämtliche Kamera-Bewegungen und die Abgabe der ausgewählten Methode über den Abgabe-Knopf. Aus diesen Informationen, zusammen mit den Zeitpunkten konnten die Bearbeitungszeiten und ein Betrachtungsprotokoll der einzelnen Methoden erhoben werden. Es wurden keine personenbezogenen Daten erhoben. Für die Bearbeitungszeiten in Netbeans wurden die Zeiten manuell gemessen.

4.3.5.4 Pilotierung

Vor dem Start der Studie wurde der geplante Ablauf in einem Pilotdurchlauf mit einer Testperson getestet. Ziel dieses Testlaufes war es herauszufinden, ob die Aufgabenstellungen grundsätzlich lösbar sind und ob die technische Umsetzung funktioniert. Es war ursprünglich geplant, dass die Testpersonen die Wahl bekommen zwischen zwei Teilnahmeoptionen: Remote-Desktop und einer lokalen Installation auf dem privaten Rechner der Testperson . Die Testperson des Piloten hat zunächst die lokalen Installationsdateien verwendet. Da es hier jedoch zu technischen Problemen kam, wurde die zweite Option im weiterem Studienverlauf verworfen.

Die Pilot-Testperson hat alle Aufgabenkombinationen bearbeitet, wohingegen die weiteren Testpersonen nur vier Aufgaben lösten, von denen jeweils zwei Einführungsaufgaben für die jeweilige Software sind.

Die Testperson hatte in den Einführungsaufgaben zunächst Schwierigkeiten die platzierten Programmfehler zu finden. Dies lag vermutlich an einer noch zu unstrukturierten Erklärung der Benutzungsweise. Die Einführungszenarien dienen jedoch genau dazu, dass die Testpersonen sich in Ruhe mit der Software vertraut machen können und von der Studienleitung Unterstützung bekommen können. Aus diesem Grund wurde sich für die weiteren Testpersonen für ein Instruktionsvideo entschieden.

Die Pilot-Testperson hatte Probleme bei Aufgabe B in Netbeans die Funktion `getTitle()` als gesuchte Methode zu identifizieren, da die Funktion `getCreator()` in einigen Teilbäumen stärker ausgeprägt war.

Es ist negativ aufgefallen, dass in Netbeans die fünf aufgezeichneten Threads getrennt voneinander dargestellt wurden. Da in SEE die Threads auch zusammengefasst werden, sollte dies in Netbeans ebenfalls so sein. Deshalb wurde bei den weiteren Testpersonen vor Beginn der Aufgabe die Ansicht in Netbeans entsprechend durch die Studienleitung so eingestellt, dass die Threads auch in Netbeans zusammengefasst werden.

Ähnlich ist dies auch mit der Filter-Funktion. Die Pilot-Testperson wurde instruiert, zu Beginn eigenständig diese Funktion zu benutzen, um nach `swp` zu filtern. Dies wurde in allen weiteren Durchläufen von der Studienleitung vorbereitet.

Vorgenommene Änderungen nach der Pilotierung

- Zugang zur Studie via Team-Viewer oder AnyDesk. Keine lokale Installation.
- Änderungen am Fragebogen.
- Verwendung von Instruktionsvideos.
- Anpassung der Steuerung der Software, sodass sie besser mit Remote-Desk-Software benutzbar ist.
- Verdeutlichung des Bugs in Szenario B.
- Beschränkung der Ansicht in Netbeans auf Forward-Calls und Hot-Spots.
- Voreinstellung von Filter und Thread-Gruppierungen.

In Szenario B waren in der Darstellung in Netbeans einige Teilbäume der hierarchischen Darstellung des Aufrufbaumes inkonsistent zueinander. Grund hierfür war ein zu kurzer Profiling-Zeitraum, wodurch in einigen der Teilbäumen zu wenige Samples vorlagen. Hierfür wurde der Profiler über einen längeren Zeitraum erneut ausgeführt, um für die Studie konsistentere Ergebnisse zu erhalten. Im Fragebogen wurden Freitext-Antwortfelder für Fragen ergänzt. Zudem wurde in einer Frage der Wortlaut klarer formuliert.

4.3.5.5 Änderungen während der Durchführung

Es wurden während und zwischen den einzelnen Experimenten weitere Unklarheiten oder Fehler in der Anwendung gefunden. Diese Fehler führten in den bis zur Entdeckung durchgeführten Experimente zu keinen bedeutenden Verständnisproblemen bei den Testpersonen. Es wurde deshalb entschieden, dass es sinnvoll sei, die Fehler zu beheben, damit zukünftige Testpersonen nicht auf Verständnisprobleme stoßen.

Ab der vierten Testperson wurde der Auswahldialog, welcher angezeigt wird, wenn es mehrere mögliche Zielkanten für eine im Quell-Code ausgewählte Zeile gibt visuell aufgebessert. Es

wurde ab der fünften Testperson eine Frage zu den Vorerfahrungen mit Netbeans und nach dem Studiengang hinzugefügt. Diese Fragen wurde jedoch in der Auswertung nicht weiter beachtet. Die größte Auswirkung auf die Bearbeitung der Aufgaben könnte jedoch ein Fehler gehabt haben, welcher dafür gesorgt hat, dass bei fünf Testpersonen die Methoden-Blöcke nicht rot eingefärbt waren. Die Beobachtung der Testpersonen zeigte jedoch, dass sie sich visuell eher an den Kanten, statt den Blöcken orientiert haben. Betroffen waren hiervon die Testpersonen TP7, TP8, TP15, TP16 und TP18. Eine detailliertere Auflistung aller Änderungen und betroffenen Testpersonen ist im Anhang in Tabelle A.1 zu finden. Es konnte jedoch bei keiner Änderung oder bei keinem Fehler eine Auswirkung auf das Verhalten der Person festgestellt werden.

4.3.6 Auswertung

Die folgenden Abschnitte beschäftigen sich mit der Auswertung der durchgeführten Studie. Zunächst wird ein Überblick über die Testpersonen und ihre Erfahrungswerte gegeben. Danach werden die Kennzahlen der bearbeiteten Aufgaben betrachtet.

4.3.6.1 Testpersonen

Es haben, zusätzlich zu der Pilot-Testperson, 16 Testpersonen teilgenommen. Die Pilot-Testperson wird in allen folgenden Auswertungen nicht weiter berücksichtigt. Diese Testpersonen lassen sich im wesentlichen in zwei Teilgruppen einordnen: Am SEE-Projekt beteiligte Studierende und wissenschaftliche Mitarbeiter und projektfremde Informatiker. Letztere bestehen zum Großteil aus direkten Arbeitskollegen und Studenten.

Tabelle 4.7 zeigt die Aufteilung auf diese Gruppen mitsamt den zugewiesenen Aufgaben und Lösungszeiten. Durch das alternieren der Aufgaben gibt es zudem die vier verschiedenen Aufgabenkombinationen. Es wurde dafür gesorgt, dass die beiden Personengruppen sich möglichst gleichmäßig auf die Aufgabenkombinationen aufteilen, weshalb nicht alle Nummern für die Testpersonen verwendet wurden.

11 Personen gaben an, in der Softwareentwicklung zu arbeiten. Vier dieser Personen gaben zusätzlich an, im Softwaretest tätig zu sein. Zwei weitere Personen gaben an, ausschließlich Softwaretester zu sein. Vier Personen gaben an, sich aktuell noch im Studium zu befinden (drei im Bachelor, eine im Master), wohingegen elf Personen bereits ihr Studium und drei Personen eine Ausbildung als Fachinformatiker abgeschlossen haben. Die Angaben zum Tätigkeitsverhältnis varrierten zu stark, sodass hier keine Gruppierung nach akademischem Abschluss sinnvoll erschien.

Die Testpersonen haben eine Selbsteinschätzung über ihre Erfahrungen im Bereich Software-Entwicklung, Software-Test und Performance-Analyse und Profiling gegeben. In Tabelle 4.7 sind die Angaben pro Person angegeben. Sie wurden auf einer Skala von 1 (keine Erfahrung) bis 5 (viel Erfahrung) angegeben. Fast alle Personen haben ihre eigene Erfahrung in der Softwareentwicklung mindestens mit 3 bewertet. Abbildung 4.23 zeigt zudem die Aufteilung ihrer selbst eingeschätzten Erfahrungswerte in ihr ist deutlich sichtbar, dass deutlich niedrigere Selbsteinschätzungen für Software-Test und Performance-Analyse und Profiling abgegeben wurden.

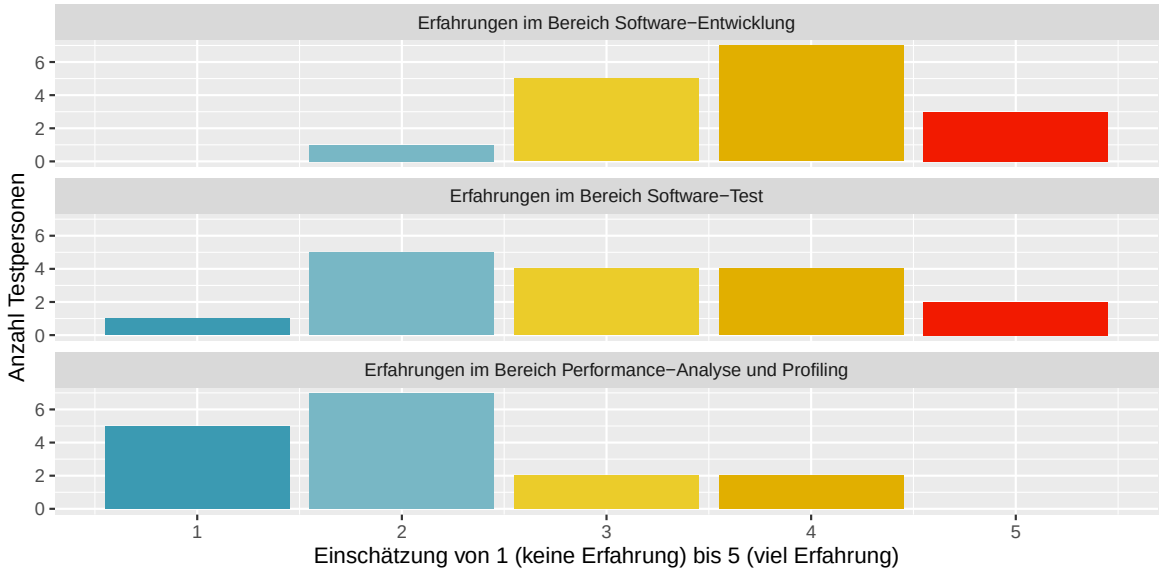


Abbildung 4.23: Selbsteinschätzungen der Erfahrungswerte der Testpersonen

Es werden im Weiteren keine Unterscheidungen zwischen den, in dem Fragebogen getätigten, Angaben bezüglich akademischem Abschluss und Angestelltenverhältnis gemacht. Betrachtet werden lediglich die Selbsteinschätzungen und der Unterschied zwischen SEE-Projektteilnehmern und sonstigen Informatikern. Insbesondere werden die weiteren erhobenen Informationen zu den Personen nicht in direkter Verbindung mit den Nummern der Testpersonen publiziert, da sie, durch die speziellen Antworten, Aufschluss auf die Person geben könnten.

Testperson 25 weicht etwas ab, da sie aktuell Doktorand im Maschinenbau ist[4]. Sie wurde hinzugezogen, da sie in ihrem täglichen Aufgabenfeld ebenfalls Software entwickelt und wird deshalb, im Sinne der Studie, zu den Informatikern gezählt.

[4]Für die Erwähnung dieser Information wurde das Einverständnis durch diese Person eingeholt.

Person	Gruppe	Erfahrung			Aufgabe		Erstaufruf (s)	Lösungszeit (s)	
TP2	IT	3	4	4	SEE	C	15	238	✗
					Netbeans	B	68	73	
TP3	IT	4	2	1	Netbeans	C	53	233	
					SEE	B	38	46	
TP4	IT	4	3	3	SEE	B	153	191	
					Netbeans	C	464	684	
TP5	IT	3	5	2	Netbeans	B	69	81	
					SEE	C	20	265	
TP6	IT	4	2	2	SEE	C	5	341	
					Netbeans	B	12	18	
TP7	IT	3	3	2	Netbeans	C	63	827	✗
					SEE	B	7	107	
TP8	IT	4	2	1	SEE	B	7	35	
					Netbeans	C	95	235	
TP9	IT	5	2	2	Netbeans	B	90	91	
					SEE	C	6	580	
TP11	IT	3	2	1	Netbeans	C	210	635	✗
					SEE	B	2	143	
TP13	IT	5	5	4	Netbeans	B	144	164	
					SEE	C	9	470	✗
TP14	SEE	3	3	2	SEE	C	10	496	
					Netbeans	B	45	85	
TP15	SEE	5	4	2	Netbeans	C	111	208	
					SEE	B	39	87	
TP16	SEE	4	4	2	SEE	B	77	129	
					Netbeans	C	25	472	
TP17	SEE	4	4	3	Netbeans	B	83	87	
					SEE	C	5	341	
TP18	SEE	4	3	1	SEE	C	23	381	✗
					Netbeans	B	20	28	
TP25	IT	2	1	1	Netbeans	B	64	82	
					SEE	C	20	183	

Tabelle 4.7: Übersicht über die Testpersonen. Mit ✗ gekennzeichnete Aufgaben wurden fehlerhaft bearbeitet. Die 3 Zahlen in der Spalte „Erfahrung“ geben der Selbsteinschätzung der Erfahrung im Bereich Softwareentwicklung, Softwaretest und Performance-Analyse (selbe Reihenfolge). Zu jeder Person gehören zwei Zeilen. Die obere Aufgabe ist die zuerst gelöste Aufgabe. Die Spalte „Erstaufruf“ gibt an, nach wie vielen Sekunden die Testperson den Quellcode mit dem enthaltenen Fehler zum ersten mal geöffnet hat. Die Lösungszeit ist die Dauer bis zum Aufgabenabschluss.

4.3.6.2 Effektivität

Tabelle 4.7 zeigte bereits die fehlerhaft bearbeiteten Aufgaben pro Person. Tabelle 4.8 fasst diese zusammen und stellt sie, nach Aufgabenkombination gruppiert, mit den erfolgreichen Bearbeitungen gegenüber. Aufgabe B wurde von allen Testperson erfolgreich gelöst. Aufgabe C jedoch konnte in manchen Fällen gar nicht oder nur teilweise gelöst werden. Aufgabe C wurde in SEE zwar häufiger nicht gelöst, jedoch wurde die Aufgabe in SEE auch insgesamt häufiger gelöst. Als „nicht gelöst“, zählt eine Aufgabe, wenn die Testperson aufgegeben hat, eine falsche Lösung abgegeben hat oder durch die Studienleitung abgebrochen wurde, da nach mindestens zehn Minuten Bearbeitungszeit die Testperson den Fehler mehrfach gesehen hat, jedoch nie als Fehler erkannt hat.

Der Grund dafür, dass Aufgabe C häufiger nicht gelöst wurde, lag meiner eigenen Einschätzung nach, welche auf der Rückmeldung der Testpersonen basiert, eher am Erkennen des Fehlers im Quell-Code und weniger an der Darstellung. Sowohl in Netbeans, als auch in der Darstellung in SEE haben alle Testpersonen die gesuchte Methode nach recht geringer Zeit geöffnet.

Da jedoch die meisten Testpersonen alle Aufgaben lösen konnten, oder zumindest den gesuchten Abschnitt über längere Zeiträume untersucht haben, kann die 3D-Darstellung als valide Alternative zur 2D-Darstellung betrachtet werden, auch wenn keine der Darstellungen als bedeutend ineffektiver erachtet werden kann.

Aufgabe	Software	Fehlerhaft	Erfolgreich	Gesamt	Erfolgrate
B	Netbeans	0	9	9	100.00%
B	SEE	0	7	7	100.00%
C	Netbeans	2	5	7	71.43%
C	SEE	3	6	9	66.67%

Tabelle 4.8: Erfolgreiche und fehlgeschlagene Aufgaben

Im Folgenden werden die von den Testpersonen gemachten Fehler näher beschrieben. Abschnitt 4.3.6.3 setzt sich mit den Verhaltensmustern bei erfolgreichen Lösungswegen auseinander.

Erster Fehler in SEE Die erste Testperson (TP2) hat nach einigem Suchen und Betrachten der gesuchten Funktion in Aufgabe C den Fehler nicht erkannt. Sie hat zum Beenden der Aufgabe willkürlich eine der Kanten ausgewählt, welche zufällig den gesuchten Fehler enthielt. Die Aufgabe wurde als fehlerhaft gewertet. Die Testperson hat nach kurzer Erläuterung des Fehlers im Nachhinein das Problem nachvollziehen können.

Zweiter Fehler in SEE Diese Testperson (TP13) hat sich die gesuchte Methode in Aufgabe C mehrfach angeschaut, hat jedoch nach einiger Zeit deutlich gemacht, dass die gesuchte Stelle nicht der Fehler seien könne. Trotz einem Hinweis darauf, dass die Testperson schon an den richtigen Stellen sucht, wurde jedoch das Experiment abgebrochen, da deutlich wurde, dass die Testperson die gesuchte Methode für sich ausgeschlossen hat. Hinweise dieser Art wurden nur gemacht, wenn Testpersonen kurz vor einem Abbruch waren. Erfolgreich gelöste Aufgaben wurden nicht mit Hinweisen kommentiert. Auch diese Testperson hat, zum Abschluss, zufällig noch die richtige Kante ausgewählt. Die Aufgabe wurde dennoch als fehlerhaft bewertet.

Dritter Fehler in SEE Die Testperson 18 identifizierte den `BufferedReader` aus Aufgabe C als zeitintensivste Stelle, empfand sie dennoch als innerhalb der Methode logisch platziert, da Benutzerprofil-Infos geladen werden könnten. Sie hat sich deshalb letztendlich für eine falsche Methode entschieden.

Fehler in Netbeans Testperson 7 hat sich im Aufrufbaum etwas verirrt. Sie hat mehrfach die gesuchte Methode in Aufgabe C `getCurrentUser` gesehen und ein paar mal geöffnet. Nach elf Minuten gab die Person an, sich unsicher zu sein und die Lösung nicht zu finden. Die Testperson wollte jedoch noch weiter suchen. Sie bestätigte beim erneuten Betrachten der gesuchten Methode `getCurrentUser`, dass sie den Fehler hier nicht vermutet. Die Aufgabe wurde daraufhin abgebrochen. Die Testperson hat mit der *Hot-Spots*-Ansicht begonnen und sich danach vermehrt in die *Forward-Calls*-Ansicht begeben. Der zweite Fehler in Netbeans wurde von Testperson 11 gemacht. Sie stellte nach einigen Minuten Bearbeitungszeit verbal fest, dass in `getCurrentUser` etwas nachgeladen wird, betonte jedoch, dass dies nicht der Fehler seinen könne. Aufgrund dieser Aussage wurde kurz darauf die Aufgabe durch die Studienleitung beendet, da die Testperson die gesuchte Stelle sicher ausgeschlossen hat.

Angegebene Lösungsansätze Die Testpersonen hatten in einem Freitext-Feld die Möglichkeit anzugeben, welchen Lösungsansatz sie zum Beheben des Fehlers hätten. Diese Antworten wurden verwendet, um zu überprüfen, ob die abgegebenen Lösungen auch tatsächlich korrekt sind, oder ob die Testpersonen sie nur zufällig korrekt abgegeben haben. Für Aufgabe B war zumeist eindeutig, dass die Personen den Fehler verstanden haben. In Aufgabe C schlugen einige der Testpersonen Lösungsansätze vor, welche Caching-Mechanismen zum Zwischenspeichern des Web-Aufrufes vorschlagen. Sie haben teilweise nicht erkannt, dass dieser Aufruf keinerlei sinnvolle Funktion erfüllt und komplett entfernt werden kann. Diese Antworten wurden trotzdem als korrekt gelöst gewertet, da es nur um die Identifizierung des Performance-Bugs ging, jedoch nicht darum ihn *korrekt* zu beheben.

4.3.6.3 Verhaltensmuster in SEE

In Abschnitt 4.3.3 wurden bereits mögliche Lösungswege anhand von Abbildungen beschrieben. In Folgendem werden die tatsächlich beobachteten Benutzerverhalten betrachtet.

Abbildung 4.24 stellt die Anklickhäufigkeiten der einzelnen Kanten durch die Testpersonen in Form einer Heatmap dar. Hierfür wurde die Gesamtsumme der Klicks aller Testpersonen auf die Kanten berechnet. Blau entspricht einer geringen Anzahl an Klicks und Rot einer sehr hohen Anzahl. Die Dicke der Kanten entspricht der Dicke, wie sie für die Testpersonen zu sehen war. Die Kanten waren für die Testperson farblos, wenn sie nicht markiert waren.

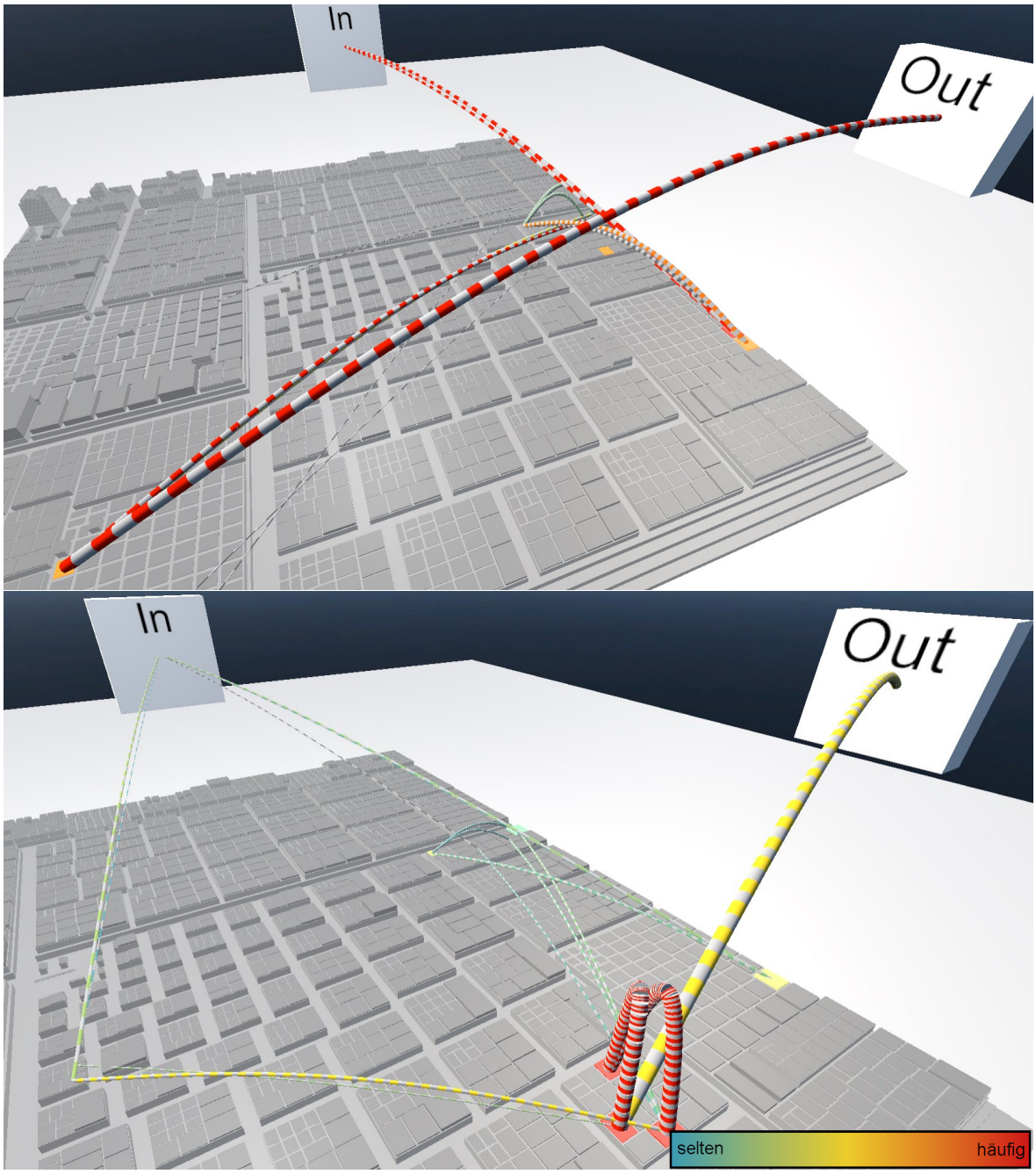


Abbildung 4.24: Kanten abhängig von der Auswahlhäufigkeit aller Benutzer entsprechend eingefärbt. (Aufgabe B oben; Aufgabe C unten)

Es wird so visualisiert, dass die Testpersonen, wie erwartet, häufiger dickere Kanten ausgewählt haben. Diese Kanten sind insbesondere auch die, welche für die Fehlersuche relevant sind. Insbesondere in Aufgabe C kam es jedoch bei einigen Testpersonen dazu, dass sie zwar zunächst im korrekten Abschnitt gesucht haben, da sie jedoch den Fehler nicht erkannt haben noch in anderen Bereichen gesucht haben. Dies hat zur Folge, dass in Aufgabe C auch die dünnere Kanten häufiger angeklickt wurden.

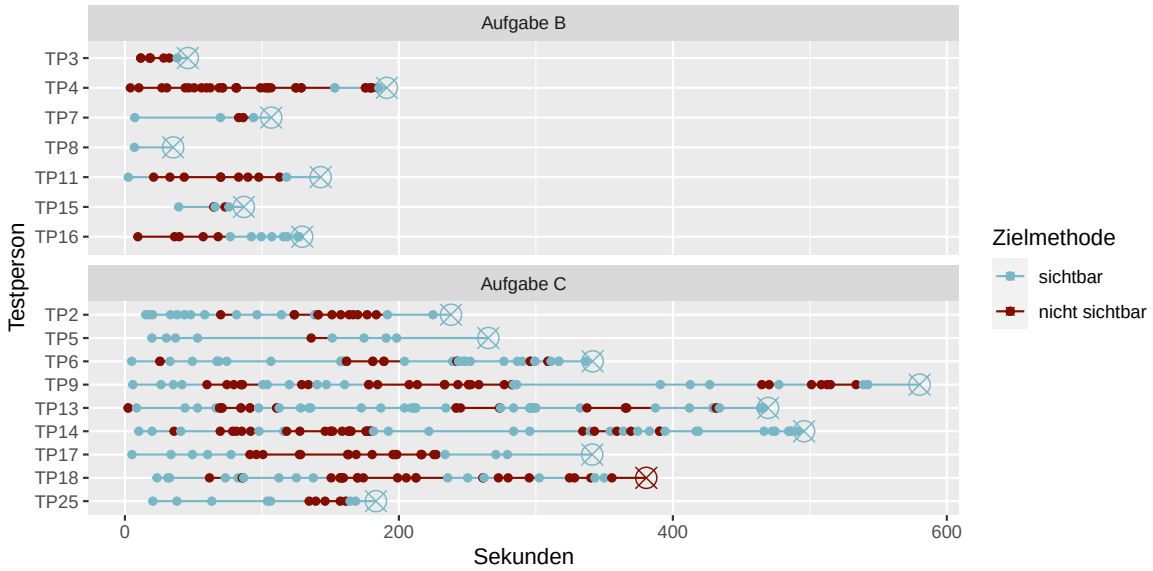


Abbildung 4.25: Auswahlverlauf der Kanten und Methoden über den zeitlichen Bearbeitungsverlauf.

Abbildung 4.25 zeigt für alle Testpersonen den Auswahlverlauf der einzelnen Methoden über Zeit. Die Farbe gibt an, ob die ausgewählte Kante als „korrekt“ gewertet werden würde. Korrekt ausgewählte Kanten sind diejenigen, bei denen entweder im *Caller* oder *Callee* Quellcode-Fenster die gesuchte Methode angezeigt wird. Das bedeutet, dass die Testperson, die Methode mit dem Performance-Bug in der Quell-Code Ansicht solange geöffnet und möglicherweise angeschaut hat, bis in der Zeitleiste ein rot eingefärbter Punkt auftaucht, dadurch, dass sie eine andere Kante/Methode ausgewählt hat. Die umkreisten Kreuze markieren den Abgabezeitpunkt. Testperson 2 und 13 haben zwar die richtige Methode abgegeben, haben jedoch geraten oder waren sich sehr unsicher. In allen Auswertungen wurde die Aufgabe für diese Personen deshalb als falsch gewertet.

Es fällt auf, dass in Aufgabe C, viele der Testpersonen zu Beginn direkt die richtigen Kanten geöffnet haben. Im weiteren Zeitverlauf haben sich viele Testpersonen dann vermehrt falsche Methoden angeschaut. Zum Ende hin wurde jedoch meistens die richtige Methode ausgewählt.

In Aufgabe B fällt auf, dass keine der Testpersonen lange überlegt hat, ab dem Moment, wo sie zum ersten Mal die gesuchte Methode geöffnet hat. Testperson 4 hat zu Beginn recht lange falsche Methoden ausgewählt. Auch Testperson 16 verhielt sich ähnlich. Das liegt daran, dass diese beiden Testpersonen bei den von außerhalb eingehenden Kanten mit ihrer Suche begonnen haben und von dort aus tiefer in den Aufrufgraphen vorgedrungen sind. Die anderen Testpersonen in Aufgabe B haben sich eher daran orientiert zunächst die dicksten Kanten auszuwählen oder bei den ausgehenden Kanten mit ihrer Suche zu beginnen.

Auch für Netbeans wurden in Abschnitt 4.3.3 mögliche Lösungswege anhand von Abbildungen gezeigt. In Folgendem werden die tatsächlich beobachteten Benutzerverhalten für Netbeans betrachtet.

Die Personen wurden nicht explizit in verschiedene Vehaltenskategorien einsortiert, dennoch konnten folgende Muster beobachtet werden:

- Beginn bei einer der größten Kanten
- Beginn bei den eingehenden Kanten von Außen
- Beginn bei einer ausgehenden Kante nach Außen
- Entlanglaufen des Aufrufbaumes
- Willkürliches Anklicken von größeren Kanten und Analyse des Quell-Codes

Aufgabe E In der Einführungsaufgabe war es für viele Testpersonen nicht eindeutig, dass in dieser Aufgabe bereits ein Fehler platziert ist. Sie haben sich stattdessen lediglich alle Funktionen angeschaut. Die Testpersonen wurden dann darauf hingewiesen, den Fehler noch zu suchen. In dieser Aufgabe kam es zu keinen besonderen Schwierigkeiten. Nur wenige der Personen benötigten, zusätzlich zu dem Einführungsvideo, noch einen Hinweis, wie sie den Fehler finden können.

Aufgabe B Aufgabe B stellte sich als deutlich einfacher heraus. Die Testpersonen haben alle, nachdem sie die Methode erstmals geöffnet haben nicht mehr lange überlegt und die Lösung abgegeben. Teilweise haben sie noch kurz andere potentielle Methoden angeschaut, haben sich jedoch dann schnell entschieden.

Aufgabe C Die Testpersonen haben in Aufgabe C vergleichbar schnell zu der richtigen Methode navigiert. Fast alle Testpersonen haben mit ihrem ersten Mausklick eine Kante ausgewählt, welche den Fehler entweder im Callee oder im Caller enthält. Im Gegensatz zu Aufgabe B konnten einige der Testpersonen jedoch nicht sofort erkennen, dass sich in der Methode unnötiger Quell-Code befindet. Viele Testpersonen beschrieben nach dem Experiment, dass sie sich die richtige Stelle zwar lange angeschaut haben, den externen Serveraufruf jedoch nicht als zwingend unnötig erkannt haben. Da die Methode `getCurrentUser` heißt, wurde teilweise vermutet, dass hier eine User-Session geladen wird. Möglicherweise entstand eine Art Tunnelblick, wodurch die Testpersonen sich die rot markierte Zeile stärker angeschaut haben, als die umliegenden Zeilen. Die Zeile, welche als Hot-Spot markiert ist, ist selber lediglich das Einlesen des HTTP-Aufrufes mit einem `BufferedReader`. Verdächtigere Codezeilen sind jedoch nur im näherem Umfeld zu finden.

4.3.6.4 Verhaltensmuster in Netbeans

Der Hauptfokus der Studie liegt auf dem Benutzerverhalten in SEE. Auch, weil in Netbeans keine automatischen Protokolierungsskripte integriert wurden, liegen zu Netbeans weniger auswertbare Informationen vor. Die Aufgaben wurden in Netbeans größtenteils richtig bearbeitet. In Abschnitt 4.3.3 wurde die fehlerhaft bearbeitete Aufgabe bereits beschrieben.

Analog zu den Verhaltensmustern in der SEE Anwendung können auch in Netbeans ähnliche Muster erkannt werden.

- Vollständiges Aufklappen des Aufrufbaums an den rechenintensivsten Stellen und anschließendem Untersuchen des Quell-Codes
- Vollständiges Aufklappen des Aufrufbaumes mit Unterbrechungen für Quell-Code-Analysen.
- Beginn mit der Forward-Call-Ansicht
- Beginn mit der Hot-Spot-Ansicht

Aufgabe E Diese Aufgabe wurde hauptsächlich für die Orientierung in der Anwendung genutzt. Insbesondere in Netbeans wurden durch den Studienleiter auf alle verfügbaren Funktionen hingewiesen. Es traten keine größeren Probleme bei der Suche nach dem Fehler des Einführungszenarios auf.

Aufgabe B Die meisten Testpersonen haben mit der *Forward-Calls*-Ansicht begonnen. In den meisten Fällen konnten die Fehler so auch gefunden werden oder zumindest die gesuchte Methode erstmalig geöffnet werden. Nach erstmaligen Öffnen wurde in Aufgabe C der Regel noch weiter in anderen Methoden gesucht, während in Aufgabe B meistens direkt abgegeben wurde. Ein paar der Testpersonen haben sehr früh oder direkt zu Beginn die Hot-Spots Ansicht aufgerufen und diese verwendet, um danach über die Funktion „in Forward-Calls finden“ die ausgewählte Methode in den Forward-Calls aufzurufen. Auch diese Vorgehensweise zeigte Erfolg.

Aufgabe C In Aufgabe C kam es zu zwei Problemen, da die Funktion `getCurrentUser` von mehreren unterschiedlichen Methoden aus aufgerufen wird, taucht sie in der *Forward-Calls*-Ansicht mehrfach auf und ist deshalb an manchen Stellen weniger auffällig. In der *Hot-Spots*-Ansicht und in der in der Studie nicht verwendeten *Backward-Calls*-Ansicht ist sie jedoch besser findbar. Sehr ähnlich wie in SEE war die eigentliche Identifizierung des Fehlers in Aufgabe C. Die Personen haben sich die gesuchte Methode häufig angeschaut, ohne zu erkennen, dass der dort implementierte HTTP-Aufruf keinen Zweck erfüllt. Auch hier wurde eine Art Tunnelblick auf den betrachteten Quell-Code beschrieben.

Vergleich zu SEE Bezüglich des Code-Verständnisses sind die Testpersonen auf die gleichen Probleme gestoßen. Die Ansicht in SEE hatte den Vorteil, dass die Personen sehr direkt Hot-Spots sehen konnten und die Relationen zueinander besser einschätzen konnten. Netbeans hat den Vorteil, dass die klassische Ansicht gewohnter und somit leichter zu navigieren ist. Die Notwendigkeit einer Kamerasteuerung in SEE ist eher ein Nachteil, da sie die Komplexität der Anwendung erhöht.

4.3.6.5 Bearbeitungsgeschwindigkeit

Um die Effizienz der beiden Anwendungen zu vergleichen, wird die Bearbeitungsgeschwindigkeit der Aufgaben miteinander verglichen. Die Aufgaben wurden von den Testpersonen teilweise mit stark unterschiedlicher Geschwindigkeit bearbeitet. Abbildung 4.26 zeigt recht deutlich, dass über alle Testpersonen hinweg Aufgabe B schneller gelöst wurde, als Aufgabe C. Dieser Unterschied ist in der Grafik für Netbeans und in SEE gleichermaßen sichtbar. Die Aufgaben wurden im Median in Netbeans etwas schneller gelöst als in SEE. Visuelle Unterschiede gibt es hier jedoch nicht so stark, wie im Unterschied zwischen den Aufgaben selbst.

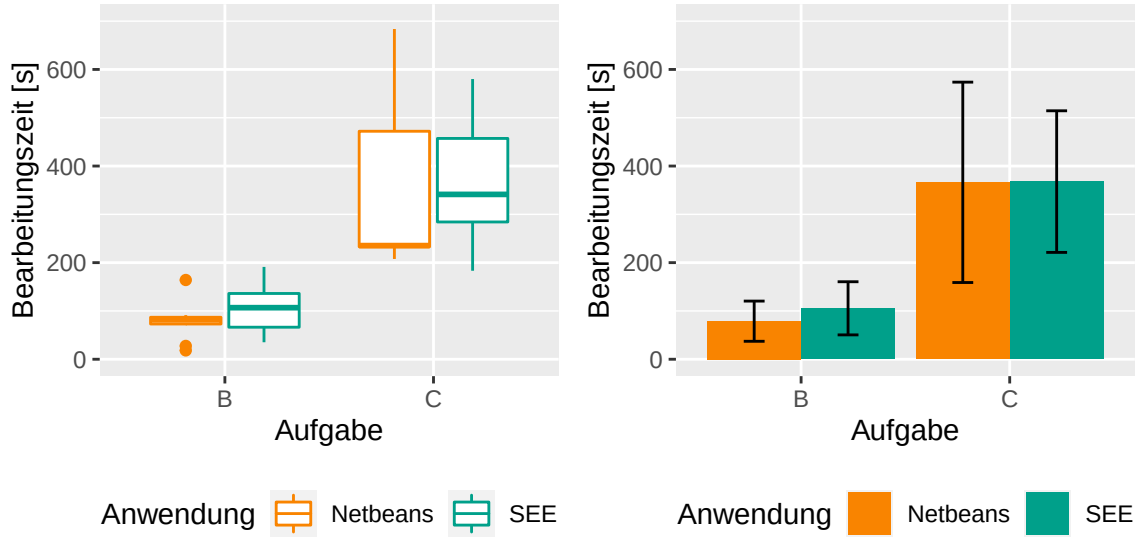


Abbildung 4.26: Lösungsgeschwindigkeiten erfolgreich bearbeiteter Aufgaben. (links Boxplot; rechts Mittelwert und Standardabweichung)

Die Zeit, welche vergangen ist, bis die Testpersonen die gesuchte Methode zum ersten Mal geöffnet haben ist in Abbildung 4.27 zu sehen. Sie ist in SEE in beiden Aufgaben niedriger als in Netbeans. Dies ist das umgekehrte Verhältnis zu dem Trend, der sich in der Lösungsgeschwindigkeit zeigt. Der Unterschied lässt sich damit begründen, dass in Netbeans zunächst der Aufrufbaum aufgeklappt werden muss und in SEE direkt die auffälligsten Kanten zu sehen sind. Die grafische 3D-Darstellung bietet hier also einen Vorteil für die initiale Orientierung, da die Testpersonen schneller zu den gesuchten Kanten gelenkt werden.

Eine langsamere Lösungszeit in SEE könnte durch mehrere Variablen begründet werden: Das Arbeiten in einer dreidimensionalen Umgebung ist ungewohnter, als das Arbeiten mit textuellen Darstellungen; Für die Navigation in SEE sind deutlich mehr Aktionen nötig. In Netbeans sind lediglich Mausklicks notwendig. In SEE hingegen muss die Kamera bewegt werden, um Details näher anschauen zu können. Hierfür könnte zusätzlicher Aufwand für die Orientierung nötig sein. Dies könnte insbesondere auf Personen zutreffen, welche in digitaler 3D-Orientierung weniger geübt sind. Diese Variablen konnten im Rahmen dieser Arbeit jedoch nicht genauer untersucht werden. Des Weiteren haben sich die Testpersonen teilweise jedoch auch bewusst Zeit gelassen, um sich die Anwendung anzuschauen und die Aufgabe gewissenhaft zu lösen. Dies gilt jedoch ebenso für auch für Netbeans.

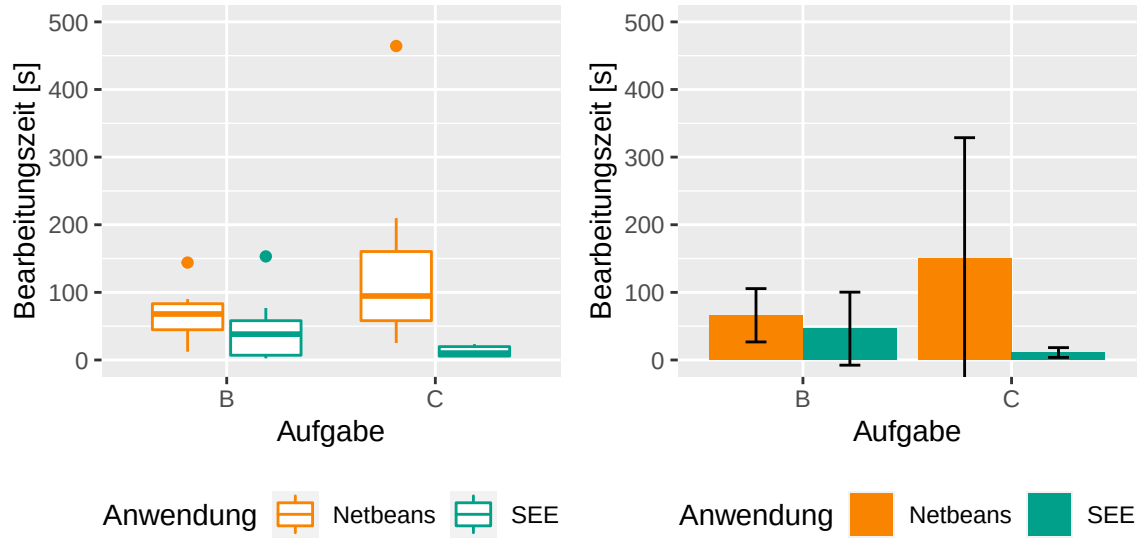


Abbildung 4.27: Dauer bis die gesuchte Methode zum ersten mal geöffnet wurde. (links Boxplot; rechts Mittelwert und Standardabweichung)

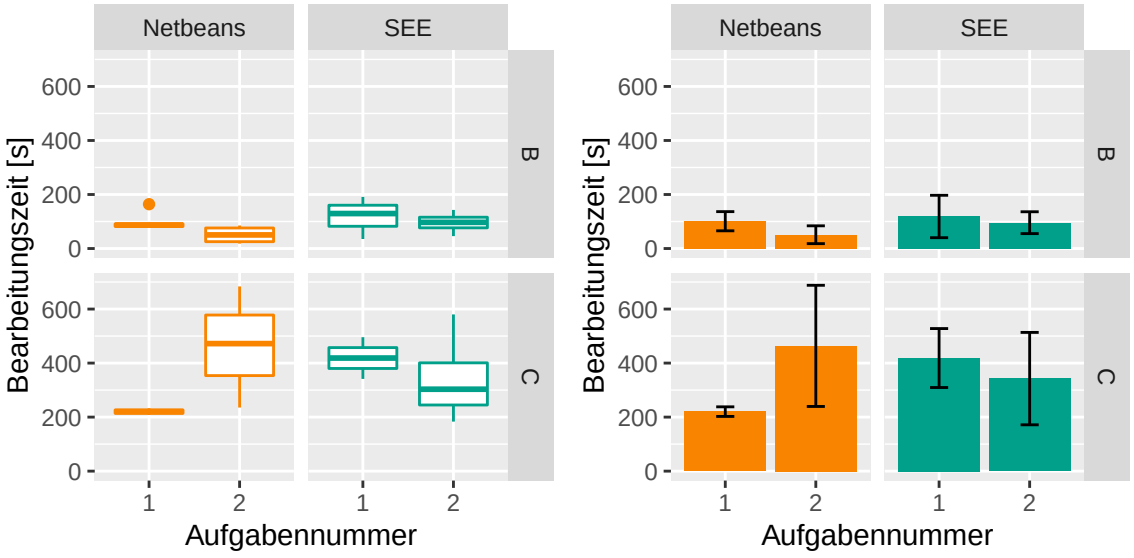


Abbildung 4.28: Lösungsgeschwindigkeiten erfolgreich bearbeiteter Aufgaben mit Auftrennung nach Aufgabennummer (links Boxplot; rechts Mittelwert und Standardabweichung)

Aufgabenreihenfolge Neben der gestellten Aufgabe gibt es weitere Variablen mit möglichem Einfluss auf die Lösungsgeschwindigkeit der Aufgaben. Abbildung 4.28 trennt die Lösungszeiten abhängig davon auf, ob die bearbeitete Aufgabe die erste oder zweite bearbeitete Aufgabe war, denn es ist möglich, dass die Testpersonen durch das Bearbeiten der ersten Aufgabe in der zweiten bereits besser mit der Aufgabenstellung vertraut waren, obwohl in der zweiten Aufgabe die jeweils andere Anwendung verwendet wird.

Drei der Vier Aufgabe-Software-Kombinationen wurden im Median und Mittelwert in der zweiten Aufgabe etwas schneller bearbeitet als in der ersten. Die Ausnahme bildet die Aufgabe C in Netbeans. Die Anzahl der Samples pro Kombination sind jedoch sehr gering. Mit statistischen Tests wird in Abschnitt 4.3.6.6 für keinen der vier Fälle ein signifikanter Unterschied zwischen erster und zweiter Aufgabe festgestellt.

SEE-Projekt In Abschnitt 4.3.6.9 werden die Teilnehmer des SEE-Projektes näher betrachtet. Bezüglich der Lösungsgeschwindigkeiten konnten keine Vorteile für die Darstellung in SEE durch die Projektteilnehmer beobachtet werden. Insgesamt schienen die Projektteilnehmer jedoch besser abzuschneiden, da Aufgabe C von ihnen seltener fehlerhaft bearbeitet wurde.

Erfahrungen der Benutzer Abbildung 4.29 zeigt die Lösungszeiten in Abhängigkeit der, durch die Testpersonen angegebenen Erfahrungswerte, im Bereich der Softwareentwicklung. Die Testpersonen konnten zwischen den fünf Werten 1 bis 5 wählen, wobei 1 wenig Erfahrung und 5 sehr viel Erfahrung entsprach. Mit einer linearen Regression wurde eine Trendlinie zwischen den Datenpunkten gezogen. Abbildungen 4.30 und 4.31 zeigen entsprechende Grafiken für die angegebene Erfahrung im Softwaretest und dem Software-Profiling. Für diese Grafiken wurden die fehlerhaft bearbeiteten Aufgaben mit einbezogen. Diese sind durch Form und Farbe in den Grafiken gekennzeichnet. In Abbildung 4.29 ist für Aufgabe C in Netbeans zu sehen, dass die zwei nicht erfolgreichen Testpersonen ebenfalls auch die in ihrer Gruppe niedrigste Erfahrung in der Softwareentwicklung angegeben haben. In SEE wiederum verteilen die fehlerhaften Bearbeitungen sich stärker über die Erfahrungsstufen. In allen drei Grafiken ist kein eindeutiger Trend zu erkennen, der darauf schließen lässt, dass die Aufgaben mit steigender, selbst eingeschätzten, Erfahrung schneller bearbeitet werden können. Für manche Aufgaben-Software-Kombinationen ist der Trend eher aufwärts und für andere wiederum abwärts. Für eine tiefergehende Korellationsanalyse sollten in der Zukunft mehr Daten erhoben werden.

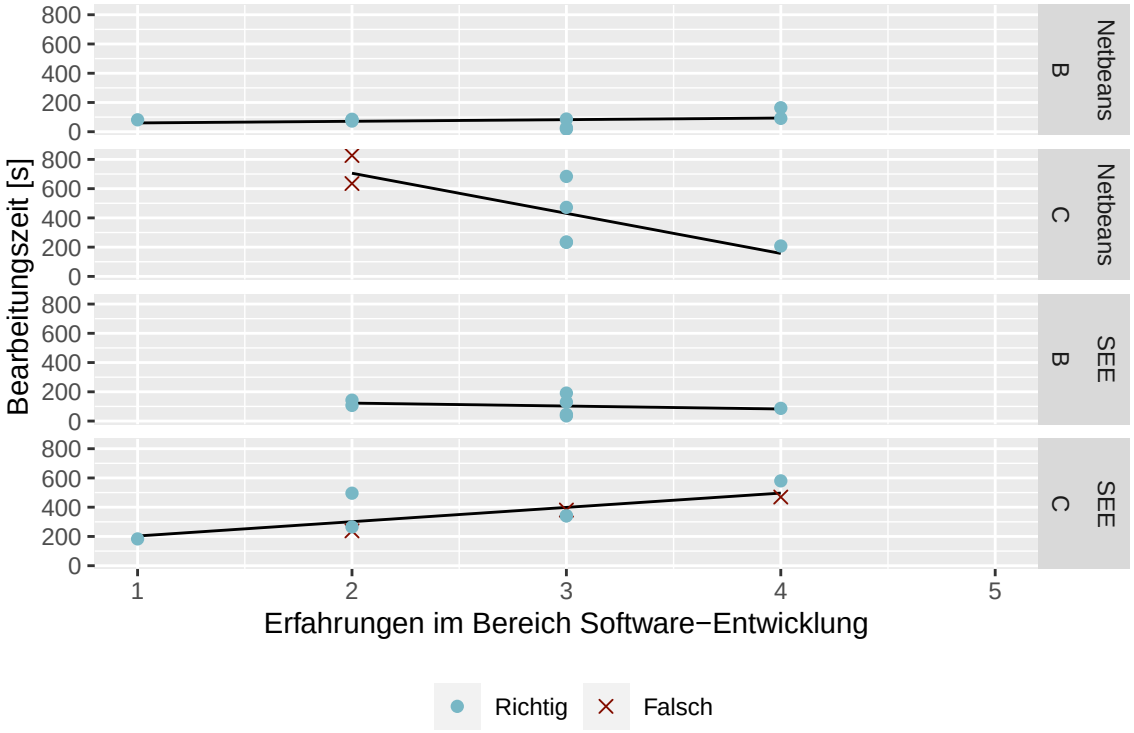


Abbildung 4.29: Lösungszeiten in Abhängigkeit der durch die Testperson angegebenen Erfahrung in der Softwareentwicklung mit linearem Modell.

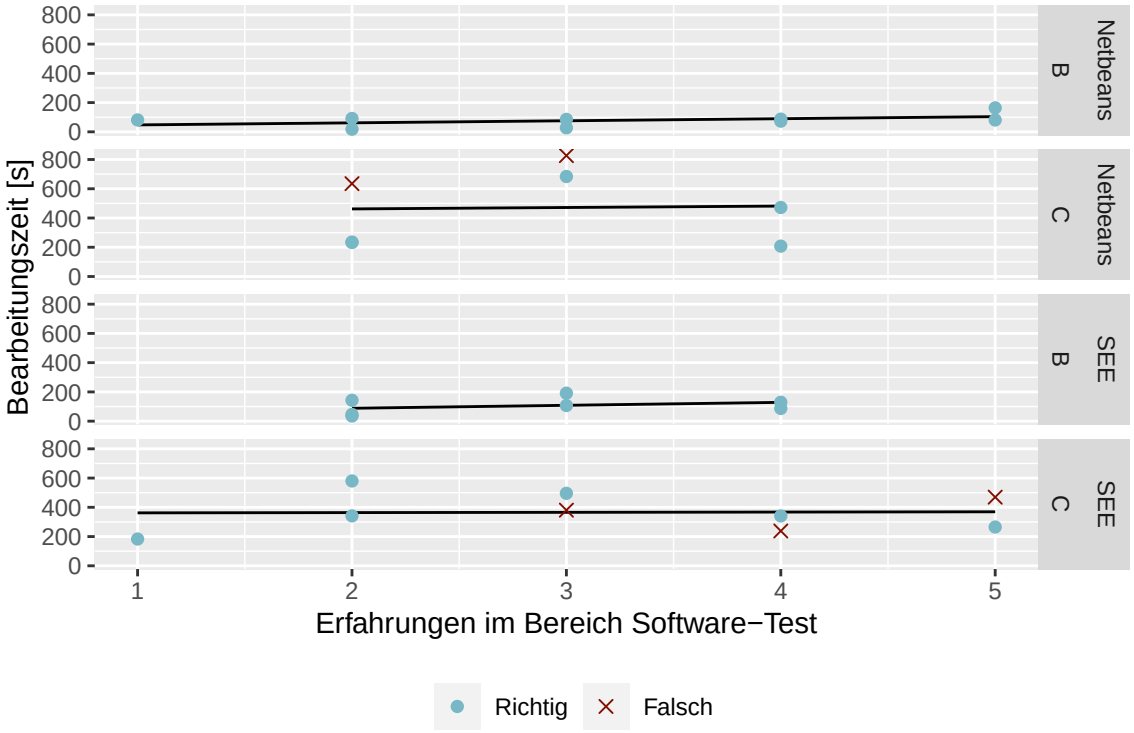


Abbildung 4.30: Lösungszeiten in Abhängigkeit der durch die Testperson angegebenen Erfahrung im Softwaretest mit linearem Modell.

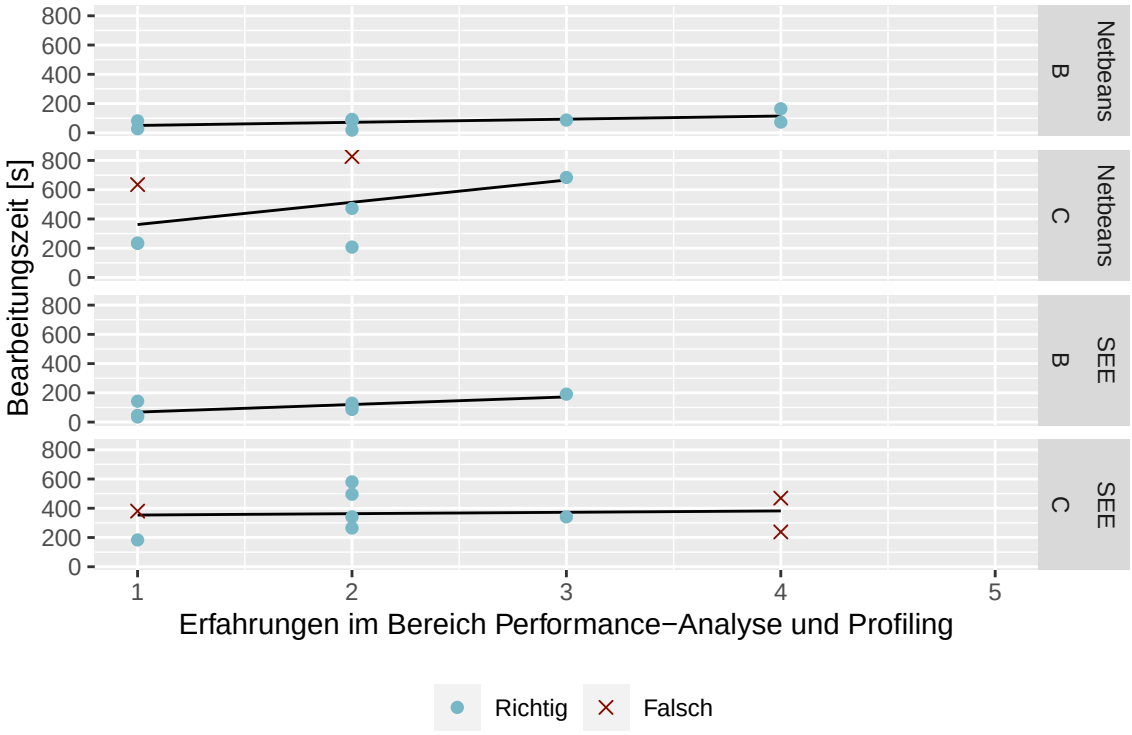


Abbildung 4.31: Lösungszeiten in Abhängigkeit der durch die Testperson angegebenen Erfahrung mit Software-Profiling mit linearem Modell.

4.3.6.6 Signifikanztests

In der visuellen Betrachtung der Messergebnisse bildet sich folgende Tendenz ab: Die Aufgaben C und B benötigen unterschiedlich lange Lösungszeiten und die Aufgaben werden in Netbeans durchschnittlich schneller gelöst als in SEE. Ersteres ist in Abbildung 4.25 und 4.26 visuell sehr deutlich zu erkennen. Hierfür soll folgende Signifikanzanalyse zeigen, ob diese Beobachtungen statistisch signifikant sind.

Für die Signifikanztests war ursprünglich die Verwendung von T-Tests [114] geplant, auf Grund der geringen Datenmenge in einigen der durchgeführten Tests kann die Normalverteilung der zugrundeliegenden Daten jedoch nur unzuverlässig ermittelt werden. Insbesondere durch die große Menge bekannter und unbekannter Variablen in einer Benutzerstudie lässt sich dies nur schwer prüfen. Statt der Prüfung der Normalverteilung, beispielsweise mit Hilfe eines Shapiro-Tests [115] und der Nutzung von T-Tests, werden deshalb Wilcoxon-Mann-Whitney-Tests [116] (kurz: U-Tests) verwendet. In den durchgeführten Tests werden jeweils zwei Gruppen an Messwerten verglichen. Die W-Statistik des U-Tests gibt an, wie stark sich diese Gruppen unterscheiden. Der p-Wert gibt die Wahrscheinlichkeit an, dass dieser Unterschied nicht durch den Zufall bestimmt ist. Es für alle hier Tests ein Signifikanzniveau von 0.05 gefordert. Die Nullhypothesen können also nur dann abgelehnt werden, wenn der p-Wert unter 0.05 liegt.

Generell sind die Ergebnisse der Signifikanztests jedoch mit Vorsicht zu verstehen, da es in dieser Art von Experiment viele unbekannte Variablen gibt.

Aufgabenreihenfolge In Abschnitt 4.3.6.5 wurden in den meisten Aufgabenkombinationen nur geringe Unterschiede zwischen einer als erstes oder als zweites bearbeiteten Aufgabe beobachtet. Um die gesammelten Ergebnisse in weiteren Betrachtungen nicht nach der Aufgabenreihenfolge voneinander trennen zu müssen, wird zunächst geprüft, ob die Reihenfolge der Aufgaben einen signifikanten Unterschied auf die Lösungszeit ausmacht. Falls die Null-Hypothese verworfen wird, kann die Aufgabenreihenfolge in den darauffolgenden Tests ignoriert werden.

Au den Aufgaben B und C und den Anwendungen N (Netbeans) und S (SEE) ergeben sich folgende vier Aufgaben-Software-Kombinationen:

$$K = \{(S, B), (S, C), (N, B), (N, C)\}$$

Für jedes $k \in K$ lassen sich nun folgende Hypothesen überprüfen.

$H_k^1.1$: Die Aufgabenkombination k wird unterschiedlich schnell gelöst, je ne nachdem, ob sie zuerst oder als zweites gelöst wird.

$H_k^1.0$: Die Aufgabenkombination k wird gleich schnell gelöst, unabhängig davon, ob sie zuerst oder als zweites gelößt wird.

Aufgabenkombination	Werte bei erster Aufgabe	Werte bei zweiter Aufgabe	W-Statistik	p-Wert
$k = (S, B)$	191.17	45.92	7.0000	0.8571
	35.22	106.81		
	129.42	142.89		
		86.86		
$k = (S, C)$	341.43	265.29	6.0000	0.5333
	495.8	580.13		
		341.13		
		183.25		
$k = (N, B)$	81.18	73	18.0000	0.0635
	91.4	18.34		
	164.07	85.15		
	86.88	27.61		
	81.63			
$k = (N, C)$	232.79	683.68	0.0000	0.2000
	207.66	235.3		
		472.04		

Tabelle 4.9: U-Tests für die Aufgabenreihenfolge

Die jeweils mit dem U-Test berechneten p-Werte liegen alle über dem Signifikanzniveau von 0.05, weshalb die Nullhypothesen nicht abgelehnt werden können. Es müssten weitere Experimente gemacht werden, um sie sicher verwerfen zu können. Für die folgenden Analysen, wird nun, aufgrund der visuellen Darstellungen in Abbildung 4.28 und da die Nullhypothesen nicht abgelehnt werden konnten, die Nullhypothesen $H_k.0$ angenommen. Für die folgenden Untersuchungen hat dies den Vorteil, dass mehr Datenpunkte zur Verfügung stehen, da die Daten nicht zusätzlich nach Reihenfolge der Aufgabenstellung voneinander getrennt werden müssen.

Unterschied zwischen Aufgabe B und C Der deutlich sichtbare Unterschied zwischen Aufgaben B und C in Abbildung 4.26 soll mit den folgenden Hypothesen untersucht werden.

Seien die folgenden Hypothesen für $s \in \{N, S\}$ definiert:

$H_s^2.1$: Die Aufgaben B und C werden unterschiedlich schnell gelöst.

$H_s^2.0$: Die Aufgaben B und C werden gleich schnell gelöst.

Software	Zeiten in Aufgabe C	Zeiten in Aufgabe B	W-Statistik	p-Wert
$s = S$	191.17	341.43	41.0000	0.0023
	35.22	495.8		
	129.42	265.29		
	45.92	580.13		
	106.81	341.13		
	142.89	183.25		
	86.86			
$s = N$	81.18	232.79	45.0000	0.0010
	91.4	207.66		
	164.07	683.68		
	86.88	235.3		
	81.63	472.04		
	73			
	18.34			
	85.15			
	27.61			

Tabelle 4.10: U-Tests für den Unterschied zwischen den Aufgaben

Der sich hier ergebende p-Wert für SEE ist 0.0023. Für Netbeans liegt er bei 0.001. Da beide unter dem Signifikanzniveau 0.05 liegen, können die Nullhypothesen $H_s^2.0$ abgelehnt werden und es ist anzunehmen, dass die Aufgaben im Mittel unterschiedlich schnell gelöst werden. Die Alternativhypothesen $H_S^2.1$ und $H_N^2.1$ werden angenommen. Die Folge hieraus ist, dass weitere Tests nach gestellter Aufgabe getrennt werden müssen.

Unterschied zwischen den Anwendungen Da in Abbildung 4.26 auf Seite 108 der Mittelwert der Lösungsgeschwindigkeit in Netbeans beider Aufgaben niedriger ist, als in SEE, wird geprüft, ob dieser Unterschied signifikant ist:

Definiere für beide Aufgaben $t \in \{B, C\}$ die folgenden Hypothesen:

$H_t^3.1$: Aufgabe t wird in den Anwendungen unterschiedlich schnell gelöst.

$H_t^3.0$: Aufgabe t wird in den Anwendungen gleich schnell gelöst.

Aufgabe	Zeiten in SEE	Zeiten in Netbeans	W-Statistik	p-Wert
$t = B$	191.17	81.18	43.0000	0.2523
	35.22	91.4		
	129.42	164.07		
	45.92	86.88		
	106.81	81.63		
	142.89	73		
	86.86	18.34		
		85.15		
		27.61		
$t = C$	341.43	232.79	17.0000	0.7922
	495.8	207.66		
	265.29	683.68		
	580.13	235.3		
	341.13	472.04		
	183.25			

Tabelle 4.11: U-Tests für den Unterschied zwischen den verwendeten Anwendungen

Mit dem U-Test ergibt sich für beide Aufgaben $t \in \{B, C\}$ ein hoher p-Wert. Da beide Werte über 0.05 liegen, können die Nullhypothesen $H_B^3 0$ und $H_C^3 0$ nicht abgelehnt werden.

Der Unterschied der Lösungsgeschwindigkeiten zwischen den Anwendungen kann demnach nicht als signifikant unterschiedlich nachgewiesen werden.

Praxisrelevanz Insbesondere kann kein für die Praxis bedeutsamer Unterschied bei der Bearbeitungszeit beobachtet werden. Die gemessenen Unterschiede betragen lediglich wenige Sekunden, dieser Unterschied ist in der Praxis vernachlässigbar. Insbesondere im Verhältnis zu dem sonstigen Aufwand, welcher für die Profiling-Aktivitäten notwendig ist, sind die geringen Unterschiede nur von geringer Bedeutung. Wichtiger ist, dass die Benutzer die Anwendung akzeptieren und in der Lage sind, typische Aufgaben mit ihr zu lösen.

In Abschnitt 4.3.6.7 und 4.3.6.8 wird hierauf weiter eingegangen.

4.3.6.7 Bevorzugte Anwendung

Im Kontrast zu der Effizienz und der Effektivität von SEE im Vergleich zu der klassischen Darstellung steht der subjektive Faktor „Spaß“ und die persönlich gewählte Präferenz, welche der Anwendungen den Personen besser gefallen hat. Es wurde bewusst die sehr subjektive Frage danach, wie viel Spaß die Testpersonen bei der Bearbeitung der Aufgaben hatten, gestellt.

Ziel dieser Frage ist es, die allgemeine Stimmung der Testperson aufzufangen und möglichst kompakt in einer Zahl zusammenzufassen. Die Anwortmöglichkeiten lagen auf einer Skala von 1 („kaum Spaß“) bis 5 („viel Spaß“). Mit zehn gezielteren Fragen wird in Abschnitt 4.3.6.8 versucht, die subjektive Wahrnehmung präziser zu erfassen.

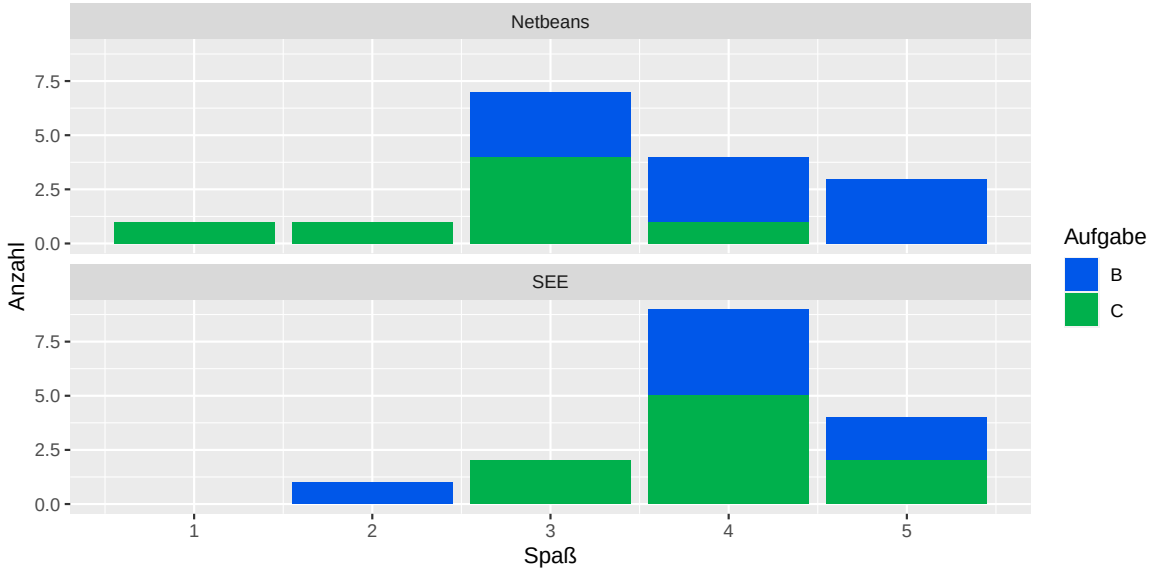


Abbildung 4.32: Spaß während der Aufgaben

Abbildung 4.32 zeigt, dass der 3D-Anwendung ein leicht größerer *Spaßfaktor* zugesichert wird. Im Mittelwert schneidet hier SEE auf der Skala von 1 bis 5 mit 4.0 etwas besser ab, als Netbeans mit 3.44. Dies geht einher mit der persönlichen Präferenz der Testpersonen, wie Abbildung 4.33 zeigt. Hier wurde am häufigsten die 3D Ansicht als die präferierte Darstellung gewählt. Die Frage danach, in welcher Anwendung die Testpersonen ihrem persönlichen Gefühl nach schneller zu einer Lösung kamen spiegelt diese Präferenz jedoch nur in geringerem Maße wieder, wie in Abbildung 4.34 zu sehen ist.

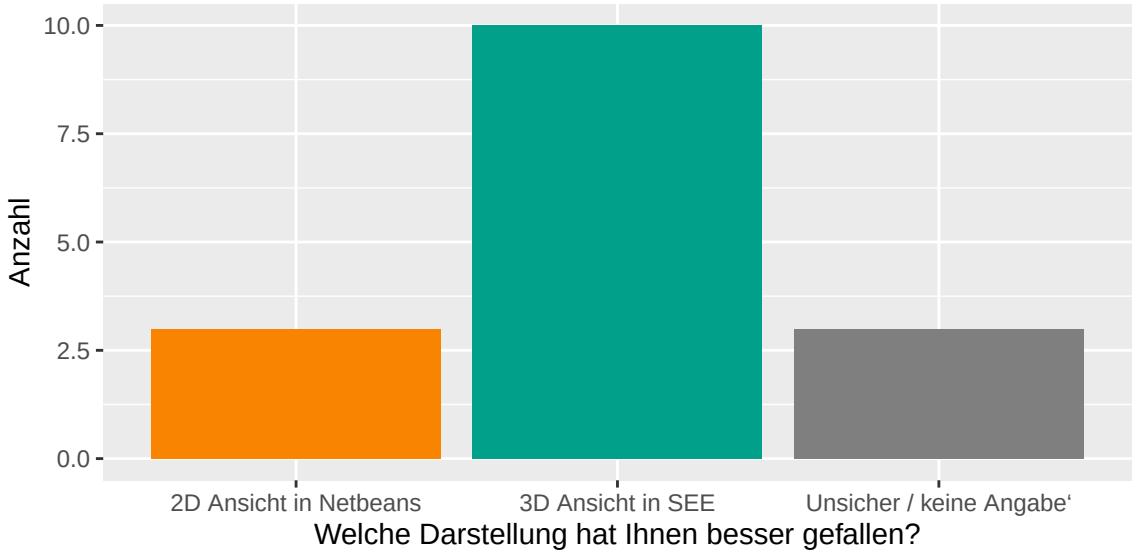


Abbildung 4.33: Präferierte Anwendung

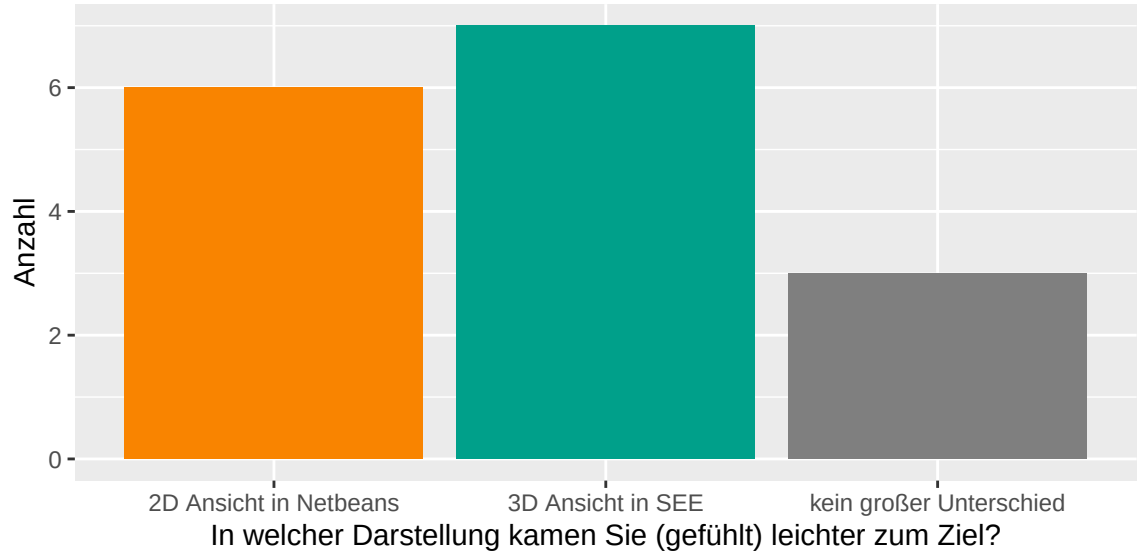


Abbildung 4.34: Wahrgenommene Bearbeitungsgeschwindigkeit

Es ist jedoch nicht auszuschließen, dass die Testpersonen bei dieser konkret gestellten Frage eher dazu tendieren, das Studienergebnis mit wohlwollender Intention zu beeinflussen.

In Abbildung 4.32 ist zudem erkennbar, dass Personen, wenn sie Aufgabe C in Netbeans bearbeitet haben, eher schlechtere Bewertungen abgegeben haben. Folgender U-Test kann diesen Einfluss für Netbeans bestätigen, da die Nullhypothese durch den p-Wert von 0.022 abgelehnt werden kann. Für SEE kann dieser Unterschied in der Bewertung weder visuell, noch durch einen Signifikanztest gezeigt werden.

Für $s \in \{S, N\}$:

$H_s^4.1$: Die bearbeitete Aufgabe hat Einfluss auf den wahrgenommenen *Spaß* beim Verwenden der Anwendung s.

$H_s^4.0$: Die bearbeitete Aufgabe hat keinen Einfluss auf den *Spaß* beim Verwenden der Anwendung s.

Software	Spaß in Aufgabe B	Spaß in Aufgabe C	W-Statistik	p-Wert
$s = S$	4, 4, 5, 2, 5, 4, 4	3, 3, 4, 5, 4, 4, 4, 5, 4	34.0000	0.8137
$s = N$	4, 3, 4, 5, 5, 4, 3, 5, 3	3, 2, 1, 3, 3, 4, 3	52.5000	0.0220

Tabelle 4.12: U-Tests für den Unterschied im Spaßfaktor zwischen Aufgabe B und C

4.3.6.8 Benutzbarkeit

Die Anzahl der erfolgreich gelösten Aufgaben lässt bereits darauf schließen, dass das System im generellen benutzbar ist und seinen Zweck erfüllt. Hierzu wird mit den beantworteten Fragen der „System Usability Score“ (SUS) errechnet. Der SUS wurde 1996 von Brooke [104] eingeführt und ist ein häufig für Benutzbarkeitsstudien verwendeter und anerkannter Satz an Fragen, welche insgesamt eine Gesamtwertung im Bereich von 0 und 100 ergeben. Nach Bangor [117] kann ein SUS ab circa 50 als „OK“ und ab circa 71 als „Good“ interpretiert werden. Ab 90 wäre die Anwendung „Best Imaginable“. Die Mediane sind 68.8 für Netbeans und 78.8 für SEE.

Der SUS hier dient dazu, die Benutzbarkeit der beiden Anwendungen zu bewerten und zu vergleichen. Abbildung 4.35 zeigt Boxplots für die SUS-Bewertungen der beiden Anwendungen, getrennt nach Anwendungen. Die rechte Grafik unterscheidet zusätzlich nach der gestellten Aufgabe.

Im Vergleich zu den Ergebnissen der Effizienz und Effektivität schneidet hier SEE etwas besser ab. Dies kann jedoch durchaus darauf zurückzuführen sein, dass die Testpersonen den Fragebogen gutwillig ausgefüllt haben. Deutlich sichtbar ist jedoch auch, dass der SUS in beiden Anwendungen niedriger ist, wenn in der Anwendung Aufgabe C bearbeitet wurde, welche von den Testpersonen als schwieriger aufgefasst wurde.

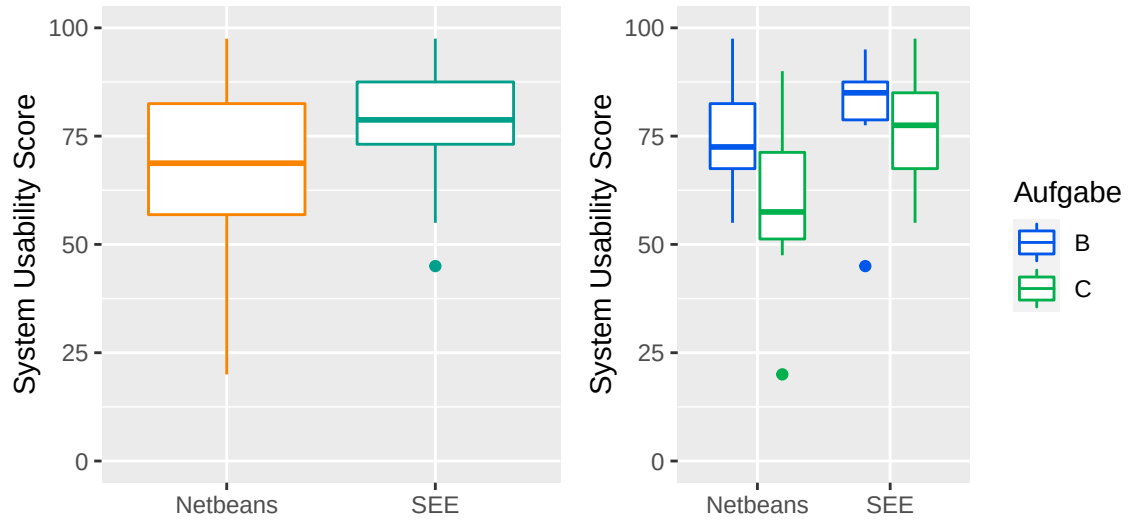


Abbildung 4.35: Boxplots der „System Usability Scores“ von SEE und Netbeans. Rechts getrennt nach Aufgabe

Ein U-Test für die Hypothese, dass die bearbeitete Aufgabe einen Einfluss auf den SUS der Anwendung hat, wie der rechte Teil in Abbildung 4.35 vermuten lässt, zeigt für ein Signifikanzniveau von 0.05 keine signifikanten Ergebnisse. Auf Grundlage der visuellen Darstellung ist jedoch recht deutlich zu erkennen, dass der SUS für eine Anwendung eher niedriger war, wenn die Anwendung für das Lösen von Aufgabe C verwendet wurde.

Für $s \in \{S, N\}$:

$H5_s^{SUS}.1$: Die bearbeitete Aufgabe hat Einfluss auf den SUS der Anwendung s.

$H5_s^{SUS}.0$: Die bearbeitete Aufgabe hat keinen Einfluss auf den SUS der Anwendung s

Software	SUS bei Aufgabe B	SUS bei Aufgabe C	W	P-Wert
$s = S$	87.5	77.5	22.5000	0.3651
	77.5	87.5		
	85	60		
	45	97.5		
	80	85		
	87.5	67.5		
	95	75		
		55		
		77.5		
$s = N$	82.5	57.5	17.5000	0.1515
	55	47.5		
	72.5	20		
	70	90		
	97.5	60		
	75	82.5		
	62.5	55		
	67.5			
	82.5			

Tabelle 4.13: U-Tests für den Unterschied im SUS abhängig der bearbeiteten Aufgabe

Den Testpersonen wurde nach der Bearbeitung der Aufgaben einer Anwendung folgende Frage gestellt: „Glauben sie, dass diese Darstellung in einem Softwareprojekt erfolgreich eingesetzt werden kann?“. Für beide Anwendungen antworteten die Personen hier überwiegend mit „eher ja“ und „ja“.Nur wenige Personen antworteten hier mit nein. Überwiegend wurden die Anwendungen also eher positiv aufgenommen. Beiden Anwendungen wurden also durchaus ein positiver Nutzen zugesprochen. Tabelle 4.14 schlüsselt die Antworten Nach verwendeter Software auf.

Antwort	Netbeans	SEE
Ja	5	5
eher ja	8	7
neutral	2	3
eher nicht	1	1
Nein	0	0

Tabelle 4.14: „Glauben Sie, dass diese Darstellung in einem Softwareprojekt erfolgreich eingesetzt werden kann?“

4.3.6.9 SEE-Projektteilnehmer

Der größte Unterschied zwischen den Testpersonen ist, dass ein Teil der Personen am Projekt SEE beteiligt ist. Es besteht die Möglichkeit, dass die Projektteilnehmer durch ihre Erfahrungen mit SEE einen Vorteil in der Bearbeitung der Aufgaben mit der Anwendung SEE haben oder die Anwendung ihnen deshalb besser gefällt. Da Personen aus dem SEE-Projekt bereits vertrauter mit der 3D-Anwendung SEE sind, ist es vorstellbar, dass diese Personengruppe eher dazu tendiert diese Anwendung zu mögen.

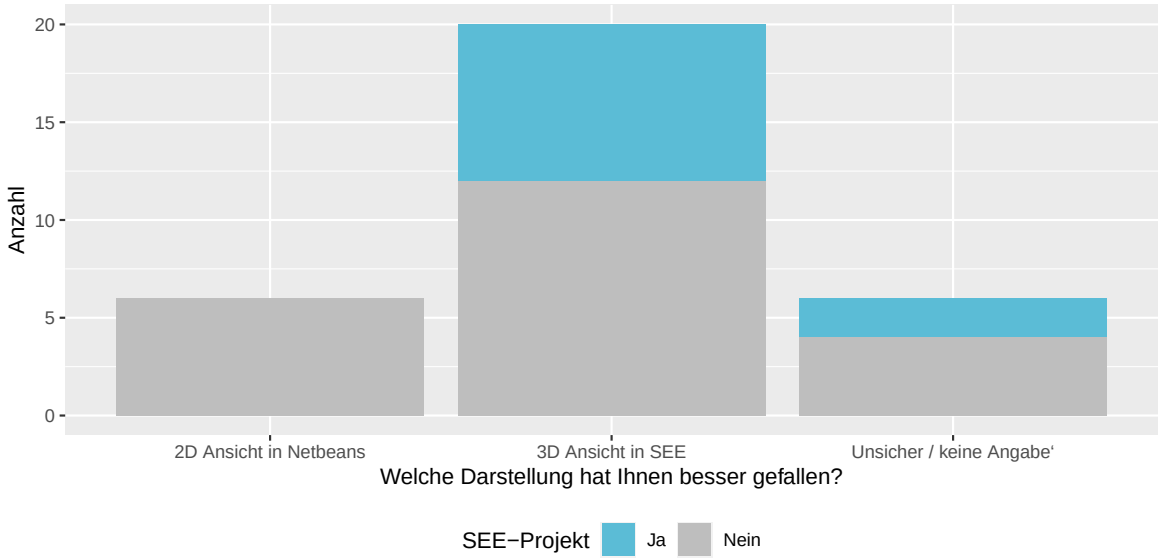


Abbildung 4.36: Präferierte Anwendung, gruppiert nach SEE-Projekt-Teilnehmern

Abbildung 4.36 zeigt erneut die Umfrageergebnisse aus Grafik 4.33, jedoch mit unterschiedlichen Einfärbungen, abhängig davon, ob die Personen mit dem SEE-Projekt vertraut sind oder nicht. Keine der Personen aus dem SEE-Projekt hat Netbeans als präferierte Darstellung gewählt hat. Allerdings wurde die 3D-Darstellung in SEE auch ohne diese, potentiell SEE präferierende Personengruppe, insgesamt häufiger bevorzugt.

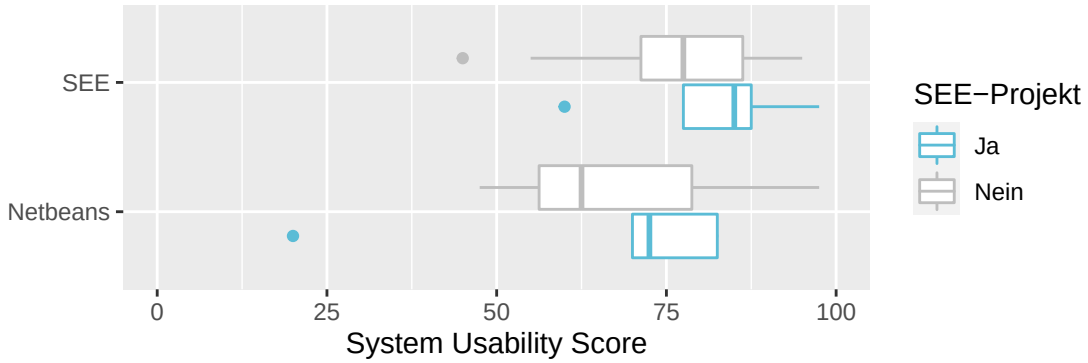


Abbildung 4.37: Boxplots der „System Usability Scores“ von SEE und Netbeans. Rechts getrennt nach SEE-Projektteilnehmern

Auch der Vergleich der SUS-Bewertungen von SEE-Projektteilnehmern in Abbildung 4.37 lässt keinen besonders starken Bias vermuten. SEE-Projektteilnehmer haben sowohl Netbeans,

als auch SEE insgesamt eher höher bewertet. Ein Bias zum Vorteil von SEE ist in der visuellen Betrachtung ist nicht zu erkennen.

Abbildung 4.38 zeigt die Lösungszeiten für die beiden Personengruppen in den vier unterschiedlichen Aufgabenkombinationen. Die Grafiken könnten tendentiell auf bessere Ergebnisse durch die SEE-Projektteilnehmer schließen lassen. Insbesondere wurden vier, der nicht erfolgreich abgeschlossenen Aufgaben, durch Nicht-SEE-Teilnehmer bearbeitet und nur eine durch einen SEE-Projektteilnehmer. Eine schnellere Bearbeitung der Aufgaben durch Projektteilnehmer in der SEE-Anwendung gegenüber der Netbeans anwendung ist der Abbildung jedoch nicht deutlich ableitbar.

Eine Durchführung von Signifikanztests wäre, auf Grund der geringen Datenmenge, bezüglich der SEE-Projektteilnehmer, nur von geringer Aussagekraft. Ein auffälliges Bevorzugen oder ein besseres Abschneiden in der Darstellung in SEE konnte bei den Teilnehmern des SEE-Projektes nicht erkannt werden. Über beide Anwendungen und Aufgaben hinweg wurde jedoch durchaus eine etwas höhere Effizienz und Effektivität der SEE-Projektteilnehmer festgestellt. Als Grund hierfür sollten, ohne weitergehende Untersuchungen, die individuellen Fähigkeiten der Personen angenommen werden.

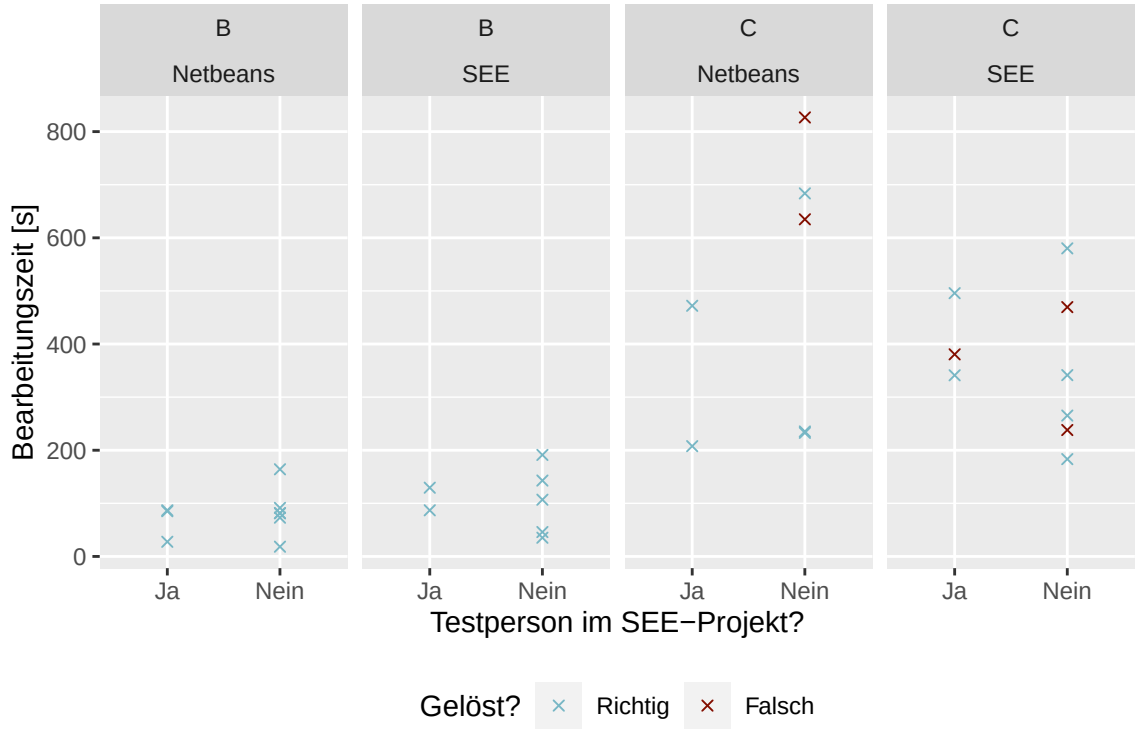


Abbildung 4.38: Lösungszeiten getrennt nach SEE-Projektteilnehmern und Projektfremden.

4.3.6.10 Bewegungsprotokolle

Es wurden, in den mit SEE bearbeiteten Aufgaben, die fokussierten Positionen und der Kamerapositionen automatisch protokolliert. Diese Bewegungsprotokolle wurden angelegt, um eventuelle Muster zu erkennen. Diese Analyse wurde aus zeitlichen Gründen jedoch nicht weiterverfolgt. Abbildung 4.39 zeigt die gesammelten Positionen aller Testpersonen. Auch hier zeigt sich erneut, dass Aufgabe B schneller gelöst wurde als C, da in Aufgabe C deutlich mehr Bewegungen aufgezeichnet wurden. Die Bewegungsprotokolle geben ohne weitere Analyse nur wenig Erkenntnisse. Zu sehen ist jedoch, dass einige Testpersonen deutlich weniger Bewegungen gemacht haben, als andere. Insbesondere Testpersonen 5, 9 und 14 haben in Aufgabe C sehr viel mit der Kamera gearbeitet. Testpersonen 4 und 13 wiederum haben die Kamera laut Protokoll gar nicht bewegt, da nur geradlinige Bewegungen zu sehen sind, welche durch die Bewegungen mit den Pfeiltasten entstehen. Zirkulare Bewegungen entstehen durch die Bewegung mit der Kamera, welche um den fokussierten Punkt rotiert. Die Arbeitsgruppe, an welcher diese Masterarbeit geschrieben wurde hat in der Vergangenheit bereits Analysen, teilweise durch Clustering von mehreren Pfaden, durchgeführt, um Bewegungsprotokolle genauer zu untersuchen [118, 47, 119]. Hierauf könnte für zukünftige Arbeiten weiter aufgebaut werden.

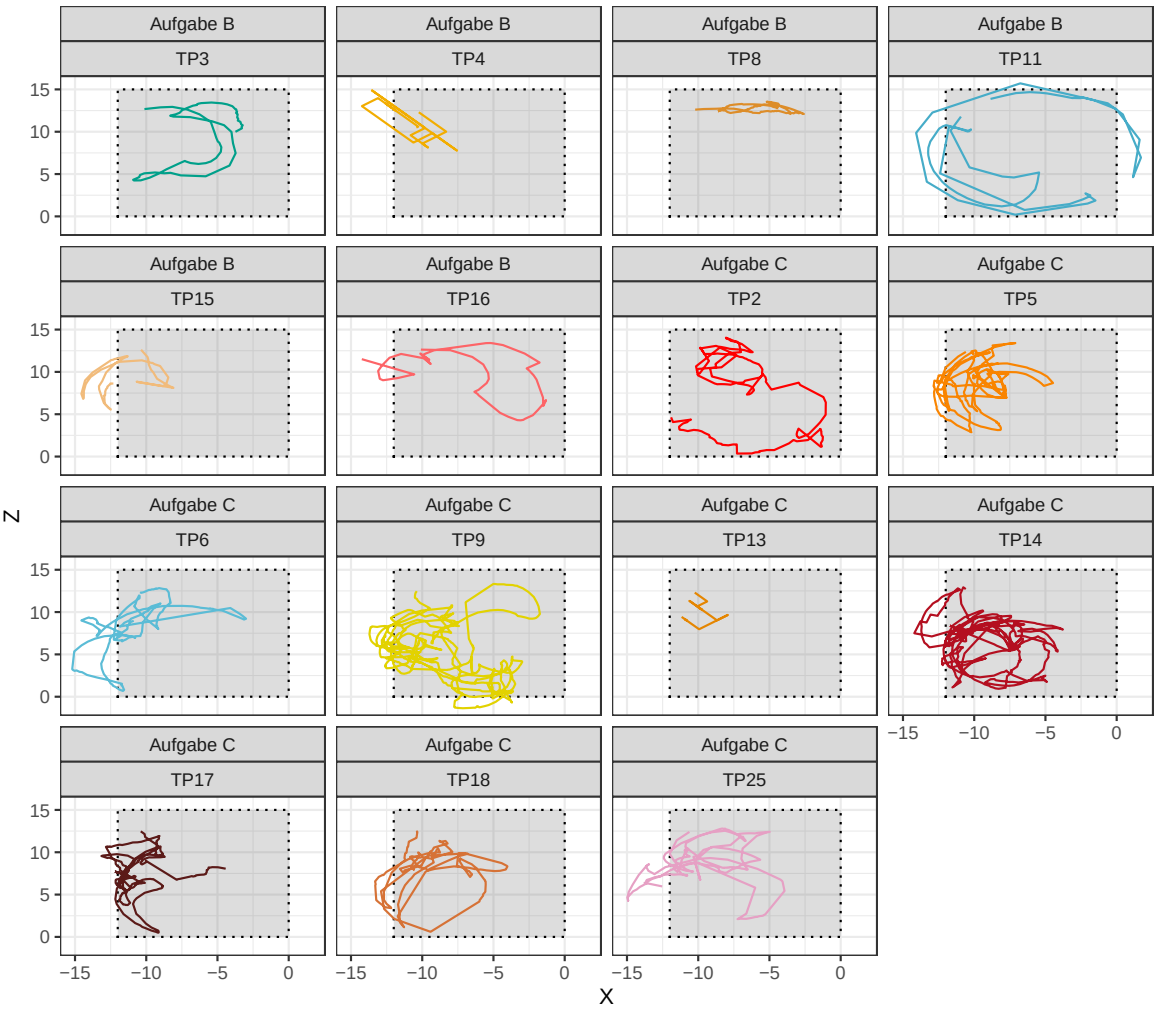


Abbildung 4.39: Bewegungsprotokolle der Kamerapositionen. Grau eingezeichnet ist die Position der Softwarestadt.

4.3.6.11 Spezielle Funktionen

Die Testpersonen wurden im Fragebogen bezüglich spezieller Funktionen gefragt, wie hilfreich sie diese fanden. Abbildung 4.40 zeigt die Aufteilung der Bewertungen zu den einzelnen Funktionalitäten nach Nützlichkeit. Die 3D-Aufrufkanten werden tendenziell als eher hilfreich und sehr hilfreich empfunden. Auch die Einblendung des Programmcodes wird als sehr hilfreich bewertet. Dies ist nicht weiter verwunderlich, da beide Funktionen essentiell sind, um die Aufgaben zu lösen. Weniger vorhersehbar waren die Antworten bezüglich der weiteren Funktionalitäten: Das gleichzeitige Anzeigen von Caller und Callee und die hervorgehobenen Zeilen wurden häufig als sehr hilfreich empfunden.

Weniger hilfreich, gar nicht hilfreich und gar nicht benutzt wurde die 3D-Darstellung der Projekthierachie (3D Anordnung und Gruppierung der Häuser). Vereinzeltes Feedback deutete hier darauf hin, dass diese Funktion den Testpersonen vermutlich erst nützlich werden könne, wenn sie mit dem untersuchten Softwareprojekt vertrauter seien und sich innerhalb der Hierarchie besser auskennen. Indirekt ist diese Darstellung jedoch wichtig, da durch die räumliche Verteilung der einzelnen Programmkomponenten erst eine räumliche Verbindung durch die 3D-Kanten möglich wird, sodass die Kanten übersichtlich verteilt werden.

Ebenso wurden auch die Methodennamen-Tooltips häufig gar nicht verwendet, welche eingeblendet werden, wenn der Mauszeiger über eine Methode oder Kante gehalten wird. Auch hier kam teilweise die Rückmeldung, dass die Methodennamen erst dann hilfreich würden, wenn die Person das Projekt besser kennen würde.

Weiterhin wurden die Vor- und Zurückbuttons von vielen Personen nicht genutzt. Die Personen, welche diese Funktion verwendet haben, fanden sie jedoch teilweise hilfreich.

Besonders positiv bewertet wurde jedoch, dass Caller und Callee immer gleichzeitig angezeigt wurden, sodass gleichzeitig die Position des Methodenaufrufes und der Inhalt der Methode zu sehen waren.

Die Bewertungen der Funktionalitäten in Netbeans ähnelt sich in zwei Punkten mit der in SEE: Die Anzeigbarkeit des Programmcodes wurden als wichtig angesehen und die Vorwärtsaufrufe, welche mit den 3D-Kanten vergleichbar sind, wurden ebenfalls ähnlich bewertet. Dies ist zu sehen in Abbildung 4.41. Die hierarchische Darstellung des Call-Graphens in Netbeans entspricht ebenfalls in etwa der Funktionalität der 3D-Aufrufkanten zwischen den Methoden in SEE. Beide wurden jeweils recht ähnlich bewertet. Die Bewertungen der *Hot Spots*-Ansicht hingegen streut deutlich mehr. Möglicherweise, weil die Vorwärtsaufrufe als erstes sichtbar waren und länger erklärt wurden.

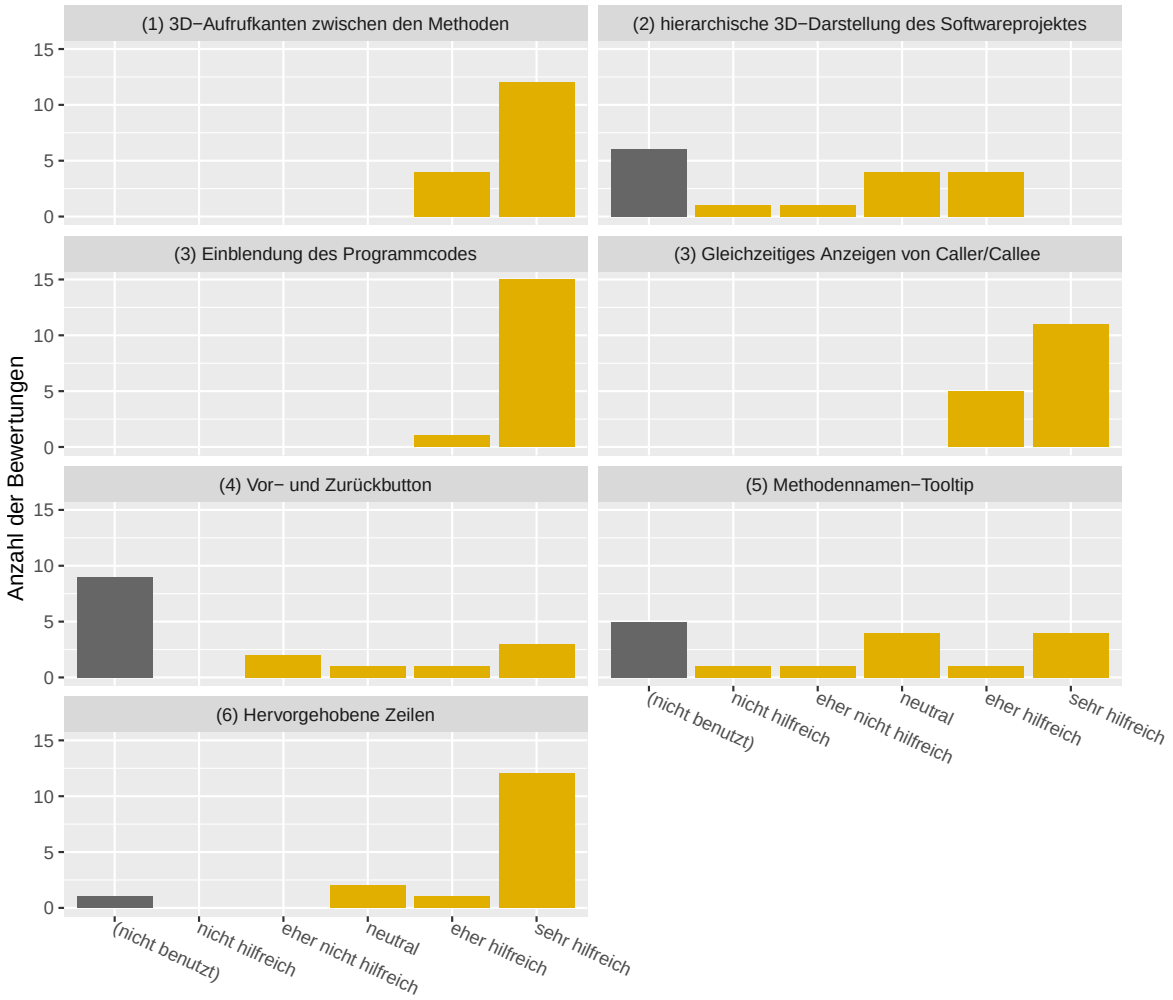


Abbildung 4.40: Nützlichkeit der einzelnen Funktionen in SEE

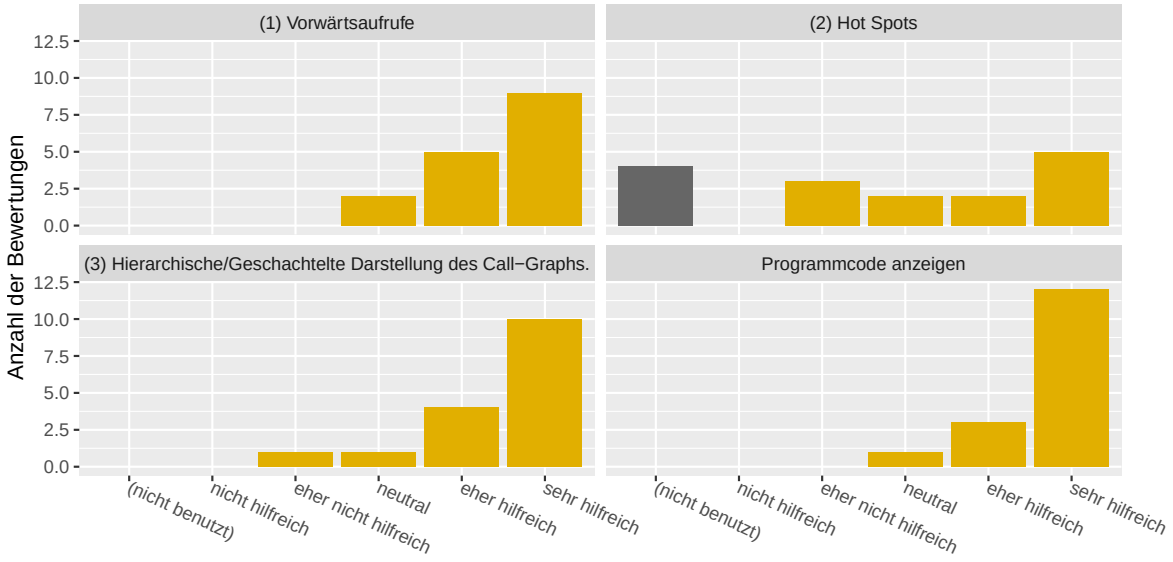


Abbildung 4.41: Nützlichkeit der einzelnen Funktionen in Netbeans

4.3.6.12 Sonstige Rückmeldungen und Feature-Wünsche

Die Testpersonen hatten die Möglichkeit im Fragebogen Freitext-Antworten zu geben oder Anmerkungen mündlich während der Bearbeitung der Aufgaben zu machen. Um direkte Zitate einzelner Testpersonen zu vermeiden, wurden während der Durchführung der Experimente Notizen zu dem Verhalten innerhalb der Anwendung und zu spontanem, mündlichen Feedback gemacht. Im Folgenden wird das häufigste Feedback wiedergegeben.

Ungleichgewicht der Fehler Die Testpersonen haben jeweils Performance-Bug B und C gelöst. Ein häufiger Kommentar war, dass die Fehler an sich unterschiedlich schwer zu identifizieren waren und, dass dies möglicherweise Einfluss auf die Bewertung hatte. Dass ein Unterschied in der Schwierigkeit der Aufgaben vorhanden ist, deuten auch die Bearbeitungszeiten in Abschnitt 4.3.6.5 an. Um einer Beeinflussung der Aufgaben auf die Anwendung in der Auswertung entgegenzuwirken, wurden die Aufgaben-Software Kombinationen möglichst gleichmäßig verteilt.

Kombination der Darstellungen Einige Testpersonen vermuten, dass das zusätzliche Verwenden einer zusätzlichen Anwendung für die 3D-Darstellung in der Praxis vielen Benutzern zu aufwändig sein könnte. Weiterhin schlagen einige Teilnehmer vor, dass eine Kombination beider Anwendungen vorstellbar wäre, da sie häufig Funktionen beider Darstellungen hilfreich fanden. Ich persönlich stelle ich mir hier vor, dass die 3D-Ansicht von SEE als ein integriertes Plug-In innerhalb von Entwicklungsumgebungen in einem modularen Fenster, wie üblicherweise Debugger, Projektbaum und Konsolenausgabe, sichtbar sein könnte. Diese Ansicht könnte weiterhin eng mit den anderen Funktionen der IDE verknüpft sein. Alternativ wäre auch das vollständige Nachbauen einer voll integrierten Entwicklungsumgebung innerhalb von SEE denkbar.

Filter- und Suchfunktionen Die gestellten Aufgaben waren recht übersichtlich. Einige Testpersonen befürchten jedoch, dass eine größere Komplexität der Anwendung oder der Testszenarien zu einer unübersichtlicheren 3D-Ansicht führt. Ein konkreter Vorschlag war die Möglichkeit Kanten als „kein Performance-Problem“ zu markieren, um diese Kante und ihren Aufrufkontext zu filtern. Dies wäre insbesondere dann wichtig, wenn Methoden vorhanden sind, welche aufgaben erfüllen, welche legitimer Weise eine große Menge an CPU-Zeit benötigen (SQL-Abfragen, Dateizugriffe, Benutzerinteraktionen). Zudem wäre eine Funktion praktisch, mit welcher nach Komponentennamen gefiltert werden kann. Die Suche nach einem Methoden- oder Paketnamen könnte dann alle Treffer visuell hervorheben oder die anderen Objekte ausblenden.

Vor- und Zurück-Button Ein Teil der Testpersonen hat explizit erwähnt, dass die Vor- und Zurück-Button verwirrend sind. Den Testpersonen wurde die korrekte Bedeutung dieser Knöpfe auf Nachfrage erklärt. Intuitiv, dachten Testpersonen jedoch fälschlicherweise, dass es sich um ein Vor- und Zurückspringen durch den Stack handelt. Es handelt sich jedoch lediglich um das Navigieren innerhalb der Historie der ausgewählten Objekte.

Vor- und Rücksprünge durch den Stack Die im vorherigen Absatz erwartete Funktionalität ist über die markierten Zeilen im Quellcode des rechten Quellcode-Fensters bereits teilweise möglich. Ein Mausklick auf markierte Zeilen ermöglicht es, tiefer in den Aufrufbaum zu navigieren. Ein Zurückspringen in die andere Richtung ist jedoch noch nicht möglich.

Zusätzliche Anzeige der Zahlen In der Studie wurden für die 3D-Ansicht absichtlich keine Zahlen eingeblendet, damit die Testpersonen die 3D-Visualisierung nutzen. Einige Testpersonen haben sich jedoch die Integration von Zahlen gewünscht. Diese Integration wäre entweder bei den Mouseover-Pop-Ups möglich oder in einem zusätzlichen Panel. Im vollen Funktionsumfang der Visualisierung sind diese Zahlen zumindest im Informationspanel einsehbar. Für einige Nutzer könnte diese Anzeige jedoch auch zu klein und unauffällig sein.

Tastaturlose Steuerung Es wurde sich gewünscht, die Anwendung komplett mit der Maus steuern zu können. Das räumliche Bewegen mit den Pfeiltasten könnte ersetzt werden, durch einen *Drag & Drop*-basierten Bewegungsmodus. Durch Greifen und Ziehen der Software-City könnte die Kamera durch den Raum gezogen werden. Eine vergleichbare Bewegungssteuerung ist bekannt aus Kartenanwendungen wie „Google Maps“ [120]. Diese Funktionalität wurde zwischenzeitlich durch andere Kontributoren im SEE-Gesamtprojekt implementiert.

Markierbare Kanten Eine Testperson wünschte sich, dass sie Kanten, welche sie bereits besucht hat und für „nicht relevant“ befunden hat farblich markieren könne. Durch ein Markierungssystem von Elementen könnten, auch über mehrere Sitzungen hinweg, Elemente auseinandergehalten oder gefiltert werden.

Zu kleine Code-Fenster Nur wenige Personen haben sich an den kleinen Quell-Code-Fenstern gestört. In der Zukunft sollten die Quell-Code-Fenster jedoch vergrößerbar sein. Ebenso wäre eine Anpassbarkeit der Textgröße wünschenswert.

4.3.6.13 Threats to Validity

Auswahl des Referenzobjektes Die Auswahl von Netbeans in Kombination mit VisualVM ist nur einer von vielen möglichen Vergleichskandidaten. Möglicherweise schneiden andere Visualisierungen deutlich besser ab, als VisualVM. Die Testpersonen könnten, alleinig durch die Tatsache, dass sie eine andere Entwicklungsumgebung wie zum Beispiel *Visual Studio* [7] oder *IntelliJ IDEA* [91] bevorzugen, Netbeans grundsätzlich negativ gegenüber eingestellt sein.

Untersuchte Anwendung Für die Evaluation wurde lediglich die Web-Anwendung *libAwesome* verwendet. Andere Anwendungen und Anwendungstypen könnten potentiell bedeutende Unterschiede in ihrer Architektur, Größe oder Benutzungsprofilen haben. Es ist davon auszugehen, dass sowohl Netbeans, als auch SEE in mit anderen getesteten Anwendungen sehr unterschiedliche Ergebnisse erzielt hätten. Dies zeigte sich bereits darin, dass die Auswahl der gestellten Aufgaben scheinbar den errechneten SUS beeinflusste, auch wenn die Signifikanztests dies nicht belegen konnten.

Programmiersprache Für die Studie wurden Softwareentwickler unabhängig ihrer bevorzugten Programmiersprache eingeladen. Es wurde davon ausgegangen, dass sie unabhängig davon, wie vertraut sie mit Java sind, die Aufgaben lösen können, da sie selber keinen Code schreiben mussten. Eine Auswirkung der individuellen Erfahrung mit Java, insbesondere auf die Lösungsgeschwindigkeit, ist jedoch nicht auszuschließen. Weiterhin wurde nur eine objektorientierte Anwendung getestet. *Funktionale* Programmierparadigmen funktionieren teilweise fundamental anders, als klassische objektorientierte Programmierung mit Java und könnten in dieser Darstellung schlechter abschneiden.

Auswahl der Teilnehmer Es wurden hauptsächlich Studierende der Universität Bremen und Angestellte meines Arbeitgebers *Dataport AöR* [82] für die Studie eingeladen. Die Teilnehmer sind vermutlich nicht repräsentativ für alle Softwareentwickler. Insbesondere könnten geübte Videospieler oder Videospielentwickler aufgrund der 3D-Darstellung mehr Erfolg mit dieser haben. Ältere Personengruppen, welche schon lange mit klassischen Tools arbeiten, könnten wiederum effektiver mit den klassischen Tools umgehen. Es gab unter den Testpersonen zwar repräsentative Personen aus beiden Gruppen, eine tiefergehende Untersuchung war durch die geringe Teilnehmerzahl jedoch nicht möglich.

Auswahl der Aufgabenstellungen Es ist möglich, dass die Aufgabenstellungen ungünstig gewählt wurden. Die Aufgaben könnten beispielsweise generell zu einfach sein. Sie könnten ebenso unbeabsichtigt so gewählt worden sein, dass ungesehene Faktoren sie in einer Anwendung leichter bearbeitbar machen. Ebenso ist es möglich, dass eine bestimmte, in dieser Studie nicht getestete, Art von Fehlern in der Ansicht in SEE nicht erkennbar sind, welche jedoch in klassischen Ansichten problemlos erkennbar sind.

Gutmütigkeit der Testpersonen Da es sich um eine studentische Arbeit handelt, ist es möglich, dass die Testpersonen dadurch in ihren Antworten beeinflusst sind und sich bemühen Antworten zu geben, welche das Studienergebnis gut aussehen lassen. Es wurde während der Studie bei allen Testpersonen mehrfach betont, dass jedes Ergebnis brauchbar ist und eine bewusste Einflussnahme nicht hilfreich ist.

Subjektivität Da ich, als auswertende Person, alleiniger Autor der Arbeit bin, können subjektive Ideen und Einschätzungen, insbesondere bei der Analyse des Verhaltens der Testpersonen Einfluss auf die Ergebnisse genommen haben.

Unterschiedliche Profiling-Ergebnisse Da für beide Darstellungen unterschiedliche Profiler verwendet wurden und die Profiler jeweils zeitlich versetzt aufgezeichnet eingesetzt wurden, ist es möglich, dass die dargestellten Ergebnisse die versteckten Fehler unterschiedlich stark gemessen wurden.

4.3.6.14 Fazit zur Benutzerstudie

Die folgenden, in Abschnitt 4.3.1 definierten, Forschungsfragen werden nun noch einmal betrachtet:

F1: Ist die Performance-Analyse in SEE gebrauchstauglich?

F2: Ist die Performance-Analyse in SEE mindestens so effektiv eine klassische Darstellung?

F3: Ist die Performance-Analyse in SEE mindestens so effizient eine klassische Darstellung?

F4: Welches Verbesserungspotential gibt es?

Frage F2 wurde nicht auf statistische Signifikanz geprüft, die Aufgaben konnten jedoch in beiden Anwendungen von der Mehrheit der Personen gelöst werden. In SEE wurden jedoch häufiger Fehler gemacht als in Netbeans. Für Frage F3 gab es eher Anzeichen dafür, dass die klassische Darstellung effizienter ist. Hierfür konnte jedoch keine statistische Signifikanz gezeigt werden. Die gemessenen Unterschiede sind jedoch für die Praxis nicht zwingend relevant, da in den meisten Fällen der Großteil der Zeit mit dem Verständnis des Quell-Codes und dem Entwurf von Testszenarien aufgebracht wird. F2 und F3 lassen sich also nicht sicher beantworten. Die gemessenen Ergebnisse deuten stark an, dass die Anwendungen sich nur geringfügig unterscheiden und beide Darstellungen einsetzbar sind.

Der „System Usability Score“ und die präferierte Anwendung der meisten Testpersonen zeigte eine tendenziell größere Begeisterung der Testpersonen an der 3D-Darstellung in SEE. Auf dieser Basis und der Konkurrenzfähigkeit zur klassischen Anwendung, welche beim Untersuchen von F2 und F3 deutlich geworden ist, wird Frage F1 mit „ja“ beantwortet. Frage F4 wurde in Abschnitt 4.3.6.12 durch zahlreiche Rückmeldungen und Feature-Wünsche der Testpersonen reflektiert.

Insbesondere wird durch diese Studie deutlich, dass eine 3D-Darstellung der Performance-Informationen eine valide Alternative für Entwickler bieten kann.

KAPITEL 5

Abschluss

5.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde ein Java-Profiler für die Erhebung von Performancedaten und eine interaktive 3D-Darstellung für die Visualisierung dieser entwickelt.

Durch das exemplarische Untersuchen von zwei Web-Anwendungen wurde gezeigt, wie die Darstellung und die entwickelten Profiler verwendet werden können und welchen Mehrwert sie bieten. Hierbei wurde zusätzlich geprüft, welche Auswirkung der Profiler auf die getestete Anwendung hat. Der Sampling-Profiler kann, durch die einstellbare Sampling-Rate ohne großen Overhead betrieben werden. Der Instrumentation-Profiler hat, insbesondere bei hoher Aktivität der Anwendung, eine große Auswirkung auf die Antwortzeit der Anwendung. Es lassen sich jedoch auch gezielt nur einzelne Methoden instrumentieren, wodurch der Overhead verringert wird.

Der entwickelte Profiler bietet, gegenüber anderen Profilern, den Vorteil, dass zusätzlich zu klassischen, von Profilern erhobenen Metriken, auch Perzentile der Methodenlaufzeiten berechnet werden. Diese sind im Bereich Last- und Performancetest häufig untersuchte Metriken, da sie Ausreißer gezielter mit in die Auswertung einbeziehen und filtern können, als das arithmetische Mittel. Der *yProfiler* bietet zudem die Grundlage, um zukünftig eine engere Integration des Profilers in die SEE-Umgebung zu ermöglichen. Die optionale Verwendung der Quantilapproximation schafft zudem die Basis für einen Dauerbetrieb des Instrumentation-Profilers in Langzeittests oder in einer Dauerüberwachung. Für die Approximation der Quantile wurde ein Histogramm-Typ mit wachsender Bin-Breite entworfen, wodurch ein konstanter, relativer Approximationsfehler ermöglicht wird und der Speicherverbrauch des Histogramms effektiv reduziert werden kann. Die Visualisierung der, mit dem Profiler gemessenen, Statistiken lässt sich in einer 3D-Umgebung in SEE interaktiv untersuchen. Die 3D-Visualisierung unterstützt diesen Prozess insbesondere dadurch, dass der Aufrufkontext einzelner Methoden deutlich erkennbar ist.

Nach der Entwicklung dieser beiden Anwendungen wurde ein Teil der Visualisierung in einer Benutzerstudie evaluiert. In dieser Studie wurde untersucht, wie Benutzer mit der Visualisierung, im Vergleich zu einer klassischen Darstellung, umgehen und welche sie bevorzugen. Die Testpersonen kamen größtenteils gut mit der entwickelten 3D-Anwendung zurecht, weshalb die Darstellungsform grundsätzlich als valide Alternative zur klassischen Darstellung betrachtet werden kann. Bezüglich der Bearbeitungszeit konnten keine signifikanten Unterschiede zwischen den beiden Anwendungen oder anderen Variablen beobachtet werden. Auffällig war jedoch, dass der Quell-Code der gesuchten Methoden in SEE häufig bereits in den ersten paar Sekunden geöffnet wurde. Der einzige signifikante Unterschied, in der Bearbeitungszeit, wurde zwischen den Aufgaben B und C festgestellt. Die gemessenen Mittelwerte der Bearbeitungszeiten weisen zudem keine, für die Praxis relevanten, Unterschiede zwischen der neuen

und der klassischen Darstellung auf. In beiden Darstellungen konnten die Aufgaben nach wenigen Minuten gelöst werden. Das spricht dafür, dass die entwickelte Darstellung eine valide Alternative für die klassische Darstellung ist. Die entwickelte Visualisierung für SEE hat im Median einen System Usability Score von 78.8 erhalten und schnitt somit etwas besser ab, als die klassische Darstellung mit einem Median von 68.8. Es konnte jedoch auch gezeigt werden, dass bei der klassischen Darstellung die bearbeiteten Aufgaben einen signifikanten Einfluss auf die Bewertung hatten.

Obwohl eine der zwei gestellten Aufgaben schwieriger erschien, da sie in 5 von insgesamt 16 Fällen fehlerhaft bearbeitet wurde, hatten die Testpersonen zumindest Ansätze und konnten lediglich den Fehler im Quell-Code nicht erkennen. Die Aufgabe wurde sowohl in der klassischen, als auch in der neuen Visualisierung fehlerhaft bearbeitet. Hier bietet die entworfene Ansicht also keinen Nachteil gegenüber der Ansicht klassischer Profiler. Durch die Rückmeldung der Testpersonen konnten zudem weitere hilfreiche Funktionen ermittelt werden, durch welche die Visualisierung zukünftig erweitert werden kann.

5.2 Ausblick

Die Rückmeldungen der Testpersonen zu der Darstellung könnten näher betrachtet und implementiert werden. Einige der gewünschten Funktionen bieten großes Potential die Visualisierung weiter zu verbessern.

Bereits während der Entwicklung dieser Masterarbeit wurde von anderen Entwicklern im SEE-Projekt an einer verbesserten Darstellung der 3D-Kanten gearbeitet. Diese Implementierungen könnten vereint werden, sodass zukünftig eine einheitliche Kanten-Repräsentation verwendet wird. Auch an der Darstellung von Quell-Code wurde in der Zwischenzeit in anderen Abschlussarbeiten gearbeitet. Auch diese Darstellung sollte vereint oder übernommen werden, sodass nicht zwei unterschiedliche Quell-Text-Fenster existieren. Die neue Darstellung ermöglicht zudem das Editieren des Quell-Codes.

Nach einer Verbesserung der Darstellung könnten erneut Benutzerstudien durchgeführt werden. Insbesondere wäre eine größerer Teilnehmerkreis interessant, um weitere unterschiedliche Aufgabenstellungen und eine repräsentativere Personengruppe zu untersuchen. In der Studie wurden relativ einfache Aufgaben untersucht. Vor Allem die Untersuchung von komplexen Aufgaben mit Experten, an Stelle von Personen, welche nur einen geringen Erfahrungsschatz im Bereich Profiling haben, wäre deshalb interessant. Idealerweise würden in solch einem Experiment auch Anwendungen untersucht werden, mit denen die Personen bereits vertraut sind, da in der Studie dieser Arbeit die meisten Testpersonen die untersuchte Anwendung *libAwesome* im Voraus nicht kannten.

In der Benutzerstudie wurden ausschließlich die Profiling-Ergebnisse des Sampling-Profiler dargestellt. Eine Evaluation einer Anwendung, in welcher die Testpersonen auch auf die Darstellung von Perzentilen und der *self time* nutzen müssen, um die Aufgaben zu lösen, würde weiteren Aufschluss darüber geben, ob diese Metriken tatsächlich helfen, Performanceprobleme zu erkennen. Eine Fallstudie über den Einsatz der Profiling-Tools und der Visualisierung mit produktiv eingesetzten Anwendungen, für welche Optimierung der Performanz noch ausstehen, wäre ebenfalls wünschenswert. Auch hierbei sollten vor Allem Entwickler mit einbezogen werden, welche sich gut mit den untersuchten Anwendungen auskennen. Insbesondere der Mehrwert, der messbaren Perzentile, konnte in dieser Arbeit nicht für echte Praxisfälle gezeigt werden und sollte deshalb zukünftig näher untersucht werden.

In der Benutzerstudie wurden die aufgezeichneten Bewegungsprotokolle der Testpersonen nur

grob untersucht. Eine detailliertere Analyse dieser aufgezeichneten Daten, ähnlich wie sie in der betreuenden Arbeitsgruppe bereits zuvor [118, 47, 119] durchgeführt wurde, könnte weitere Erkenntnisse über unterschiedliche Benutzerverhalten bringen.

Die geplante Implementierung sah vor, dass die Profiling-Ergebnisse in einer *Live*-Ansicht dargestellt werden können. Dies konnte im Rahmen dieser Masterarbeit zeitlich nicht mehr umgesetzt werden. Hierfür ist eine engere Kopplung von SEE und dem *yProfiler* notwendig. Hierfür könnten in dieser Ansicht die erhobenen Statistiken nicht über den gesamten Messzeitraum errechnet werden, sondern über kürzere Zeitfenster, wie beispielsweise die letzten 30 Sekunden oder 5 Minuten. Kürzliche Änderungen im Verhalten der getesteten Anwendung würden mit einem langem Messzeitraum erst verzögert auffallen. Als Ergänzung zu der Live-Ansicht könnte ein Zeitstrahl integriert werden, mit welchem es möglich ist, zwischen einzelnen Zeitsegmenten hin- und herzuwechseln, um die Entwicklung der Anwendung über die Zeit untersuchen zu können.

Um in einem Regressionstest Profiling-Ergebnisse miteinander vergleichen zu können, wäre die Darstellung der Differenz von zwei Messungen in der Darstellung hilfreich. So könnten Methoden erkannt werden, welche nach einem Software-Update langsamer geworden sind. Hierfür sollte unbedingt der *Performance Evaluation Blueprint* von Bergel et al. [79] mit beachtet werden, welcher in Abschnitt 2.2.2.4 kurz vorgestellt wurde.

Während der Entwicklung der Visualisierung und der Evaluation der Profiler ist es nicht dazu gekommen, dass in der Visualisierung so viele Kanten dargestellt werden, dass die Darstellung unübersichtlich wird, eine solche Situation ist jedoch durchaus vorstellbar. Größere Anwendungen oder Anwendungen mit mehr parallel verwendeten Funktionen könnten zu einer größeren Zahl an sichtbaren Aufrufkanten führen. Hierfür sollte die Darstellung um ein automatisches Bündeln von Kanten erweitert werden, ähnlich, wie in der zweidimensionalen Darstellung in *ExplorViz* in Abbildung 2.12 auf Seite 16.

Um die Profiler enger mit den Last- und Performancetests zu verbinden wäre es denkbar, dass in der Visualisierung auch die Seitenantwortzeiten auf HTTP-Ebene dargestellt werden, welche von den entsprechenden Testtools erhoben werden. Es wäre denkbar, dass diese die eingehenden und ausgehenden Kanten der In- und Out-Blöcke ersetzen.

SEE unterstützt *Virtual Reality*-Brillen und anderen alternative Eingabegeräte. Diese wurden in dieser Arbeit nicht betrachtet. Um die in dieser Arbeit entwickelten Konzepte vollständig in die SEE-Umgebung zu integrieren, müssen sie kompatibel gemacht werden. Hierzu gehört vor Allem, dass die zweidimensionale Oberfläche für Quell-Code und Einstellungen auch in *VR* nutzbar gemacht wird. Auch die Kompatibilität mit den kollaborativen Funktionen von SEE, durch welche das gleichzeitige Verwenden der Darstellung durch mehrere Personen und Endgeräte möglich wäre, sollte zukünftig integriert werden, um ein gemeinsames Analysieren der Anwendungsperformance zu ermöglichen.

Danksagung: Mit Abschluss dieser Arbeit möchte ich mich an dieser Stelle noch persönlich bei meinen Gutachtern und der Universität Bremen für die Betreuung und viele lehrreiche Jahre bedanken. Besonderer Dank gilt ebenfalls allen Personen, welche an meiner Studie teilgenommen haben.

ABBILDUNGSVERZEICHNIS

LITERATURVERZEICHNIS

[1] Z. M. Jiang and A. E. Hassan, "A survey on load testing of large-scale software systems" *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1091–1118, 2015.

[2] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis" *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, 2009.

[3] G. Kroah Hartman *et al.*, "Linux kernel development" in *Linux Symposium*, pp. 239–244, 2007.

[4] R. Koschke, "Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey" *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 15, no. 2, pp. 87–109, 2003.

[5] F. Beck, O. Moseler, S. Diehl, and G. D. Rey, "In situ understanding of performance bottlenecks through visually augmented code" in *2013 21st International Conference on Program Comprehension (ICPC)*, pp. 63–72, IEEE, 2013.

[6] P. Khaloo, M. Maghoumi, E. Taranta, D. Bettner, and J. Laviola, "Code park: A new 3d code visualization tool" in *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 43–53, IEEE, 2017.

[7] Microsoft, "Visual Studio" `https://visualstudio.microsoft.com/`, 2009. (letzter Zugriff m 09.08.2021).

[8] P. Caserta and O. Zendra, "Visualization of the static aspects of software: A survey" *IEEE transactions on visualization and computer graphics*, vol. 17, no. 7, pp. 913–933, 2010.

[9] S. G. Eick, J. L. Steffen, E. E. Sumner, *et al.*, "Seesoft-a tool for visualizing line oriented software statistics" *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 957–968, 1992.

[10] J. I. Maletic, A. Marcus, and L. Feng, "Source Viewer 3D (sv3D)-a framework for software visualization" in *25th International Conference on Software Engineering, 2003. Proceedings.*, pp. 812–813, IEEE, 2003.

[11] R. Kern, "Github Repository: line_profiler." `https://github.com/pyutils/line_profiler`. (letzter Zugriff am 20.08.2021).

[12] T. A. Corbi, "Program understanding: Challenge for the 1990s" *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989.

[13] R. Fjelstad and W. Hamlen, "Applications program maintenance study: Report to our respondents" *Proceedings Guide*, vol. 48, pp. 13–27, 1979.

[14] G.-C. Roman and K. C. Cox, "Program visualization: The art of mapping programs to pictures" in *Proceedings of the 14th International Conference on Software Engineering*, pp. 412–420, 1992.

[15] T. D. Hendrix, J. H. Cross, S. Maghsoodloo, and M. L. McKinney, "Do visualizations improve program comprehensibility? Experiments with control structure diagrams for Java" in *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pp. 382–386, 2000.

[16] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization" *IEEE transactions on evolutionary computation*, vol. 1, no. 1, pp. 67–82, 1997.

[17] D. J. Gilmore and T. R. G. Green, "Comprehension and recall of miniature programs" *International Journal of Man-Machine Studies*, vol. 21, no. 1, pp. 31–48, 1984.

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

[19] S. Carpendale and Y. Ghanam, "A survey paper on software architecture visualization" tech. rep., University of Calgary, 2008.

[20] H. A. Basit, M. Hammad, and R. Koschke, "A survey on goal-oriented visualization of clone data" in *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, pp. 46–55, IEEE, 2015.

[21] M. Hammad, H. A. Basit, S. Jarzabek, and R. Koschke, "A systematic mapping study of clone visualization" *Computer Science Review*, vol. 37, p. 100266, 2020.

[22] Y. Zhang, H. A. Basit, S. Jarzabek, D. Anh, and M. Low, "Query-based filtering and graphical view generation for clone analysis" in *2008 IEEE International Conference on Software Maintenance*, pp. 376–385, IEEE, 2008.

[23] J. Harder and N. Göde, "Efficiently handling clone data: RCF and cyclone" in *Proceedings of the 5th International Workshop on Software Clones*, pp. 81–82, 2011.

[24] Y. Ueda, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Gemini: Code clone analysis tool" in *Proc. International Symposium on Empirical Software Engineering (ISESE 2002)*, 2002.

[25] J. Rumbaugh, I. Jacobson, and G. Booch, "The unified modeling language" *Reference manual*, 1999.

[26] K. Ogami, R. G. Kula, H. Hata, T. Ishio, and K. Matsumoto, "Using High-Rising Cities to Visualize Performance in Real-Time" in *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 33–42, 2017.

[27] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, "Graphviz—open source graph drawing tools" in *International Symposium on Graph Drawing*, pp. 483–484, Springer, 2001.

[28] J. Fekete, D. Wang, N. Dang, A. Aris, and C. Plaisant, "Interactive poster: Overlaying graph links on treemaps" in *Proceedings of the IEEE Symposium on Information Visualization Conference Compendium (InfoVis 03)*, pp. 82–83, 2003.

[29] B. Cornelissen, A. Zaidman, A. Van Deursen, and B. Van Rompaey, "Trace visualization for program comprehension: A controlled experiment" in *2009 IEEE 17th International Conference on Program Comprehension*, pp. 100–109, IEEE, 2009.

[30] D. Holten, "Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data" *IEEE Transactions on visualization and computer graphics*, vol. 12, no. 5, pp. 741–748, 2006.

[31] B. Johnson and B. Shneiderman, "Tree-maps: A space-filling approach to the visualization of hierarchical information structures" *Readings in Information Visualization: Using Vision to Think*, pp. 152–159, 1999.

[32] J. Rilling and S. P. Mudur, "On the use of metaballs to visually map source code structures and analysis results onto 3d space" in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pp. 299–308, IEEE, 2002.

[33] S. P. Reiss, "An engine for the 3D visualization of program information" *Journal of Visual Languages & Computing*, vol. 6, no. 3, pp. 299–323, 1995.

[34] A. Savidis, P. Papadakos, and G. Zargianakis, "Immersive 3d Visualizations for Software-Design Prototyping and Inspection" in *International Symposium on Visual Computing*, pp. 879–890, Springer, 2008.

[35] R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: A controlled experiment" in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 551–560, 2011.

[36] C. Knight and M. Munro, "Virtual but visible software" in *2000 IEEE Conference on Information Visualization. An International Conference on Computer Visualization and Graphics*, pp. 198–205, IEEE, 2000.

[37] C. R. Dos Santos, P. Gros, P. Abel, D. Loisel, N. Trichaud, and J.-P. Paris, "Mapping information onto 3D virtual worlds" in *2000 IEEE Conference on Information Visualization. An International Conference on Computer Visualization and Graphics*, pp. 379–386, IEEE, 2000.

[38] A. Dieberger and A. U. Frank, "A city metaphor to support navigation in complex information spaces" *Journal of Visual Languages & Computing*, vol. 9, no. 6, pp. 597–622, 1998.

[39] C. Jeffery, "The City Metaphor in Software Visualization" 2019.

[40] S. Lisberger, "Tron" 1982. (Film).

[41] T. Panas, R. Berrigan, and J. Grundy, "A 3D metaphor for software production visualization" *Proceedings of the International Conference on Information Visualisation*, vol. 2003-January, no. May 2014, pp. 314–319, 2003.

[42] F. Pfahler, R. Minelli, C. Nagy, and M. Lanza, "Visualizing Evolving Software Cities" in *2020 Working Conference on Software Visualization (VISSOFT)*, pp. 22–26, IEEE, 2020.

[43] G. Balogh, A. Szabolics, and A. Beszédes, "CodeMetropolis: Eclipse over the city of source code" in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 271–276, IEEE, 2015.

[44] F. Fittkau, S. Finke, W. Hasselbring, and J. Waller, "Comparing Trace Visualizations for Program Comprehension through Controlled Experiments" *IEEE International Conference on Program Comprehension*, vol. 2015-Augus, pp. 266–276, 2015.

[45] F. Fittkau, A. Krause, and W. Hasselbring, "Software landscape and application visualization for system comprehension with ExplorViz" *Information and Software Technology*, vol. 87, pp. 259–277, 2017.

[46] F. Fittkau, A. Krause, and W. Hasselbring, "Exploring software cities in virtual reality" in *2015 ieee 3rd working conference on software visualization (vissoft)*, pp. 130–134, IEEE, 2015.

[47] M.-O. Rüdel, J. Ganser, and R. Koschke, "A controlled experiment on spatial orientation in vr-based software cities" in *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 21–31, IEEE, 2018.

[48] S. Romano, N. Capece, U. Erra, G. Scanniello, and M. Lanza, "On the use of virtual reality in software visualization: The case of the city metaphor" *Information and Software Technology*, vol. 114, pp. 92–106, 2019.

[49] L. Merino, J. Fuchs, M. Blumenschein, C. Anslow, M. Ghafari, O. Nierstrasz, M. Behrisch, and D. A. Keim, "On the Impact of the Medium in the Effectiveness of 3D Software Visualizations" in *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 11–21, IEEE, 2017.

[50] M. Balzer, A. Noack, O. Deussen, and C. Lewerentz, "Software landscapes: Visualizing the structure of large software systems" in *IEEE TCVG*, 2004.

[51] M. Balzer and O. Deussen, "Hierarchy based 3D visualization of large software structures" in *IEEE Visualization 2004*, pp. 4p–4p, IEEE, 2004.

[52] R. Koschke, "SEE" `https://see.uni-bremen.de/`. (letzter Zugriff am 09.08.2021).

[53] R. Koschke, "Auge in Auge mit Ihrer Softwarearchitektur" `https://www.youtube.com/watch?v=eC7Zu4ch-E4`, Nov. 2020. (letzter Zugriff am 09.08.2021).

[54] R. Koschke, "Animation of the Evolution of Linux Net Subsystem with Tree Map Layout" `https://www.youtube.com/watch?v=w7gDIHzEW0Q`, Mar. 2009. (letzter Zugriff am 09.08.2021).

[55] R. C. Holt, A. Winter, and A. Schurr, "GXL: Toward a standard exchange format" in *Proceedings Seventh Working Conference on Reverse Engineering*, pp. 162–171, IEEE, 2000.

[56] R. Ferenc, S. E. Sim, R. C. Holt, R. Koschke, and T. Gyimóthy, "Towards a standard schema for C/C++" in *Proceedings Eighth Working Conference on Reverse Engineering*, pp. 49–58, IEEE, 2001.

[57] L. Kipka, "Bachelorarbeit: Software-Debugging mithilfe von Code Cities" 2020.

[58] M. Steinbeck and R. Koschke, "Tinyspline: A small, yet powerful library for interpolating, transforming, and querying nurbs, b-splines, and bézier curves" in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 572–576, IEEE, 2021.

[59] Unity Technologies, "Unity" `https://unity.com/de`. (letzter Zugriff am 09.08.2021).

[60] T. H. Jiri Sedlacek, "VisualVM" `https://visualvm.github.io/`. (letzter Zugriff am 09.08.2021).

[61] C.-P. Bezemer, J. Pouwelse, and B. Gregg, "Understanding software performance regressions using differential flame graphs" in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 535–539, IEEE, 2015.

[62] B. Gregg, "Flame graphs" `http://www.brendangregg.com/flamegraphs.html`, 2012.

[63] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. Van Wijk, and A. Van Deursen, "Understanding execution traces using massive sequence and circular bundle views" in *15th IEEE International Conference on Program Comprehension (ICPC'07)*, pp. 49–58, IEEE, 2007.

[64] R. Wettel, *Software systems as cities*. PhD thesis, Università della Svizzera italiana, 2010.

[65] "ExplorViz" `https://www.explorviz.net/features.php`. (letzter Zugriff am 09.08.2021).

[66] F. Fittkau, J. Waller, P. C. Brauer, and W. Hasselbring, "Scalable and live trace processing with Kieker utilizing cloud computing" CEUR Workshop Proceedings, 2013.

[67] F. Fittkau and W. Hasselbring, "Elastic application-level monitoring for large software landscapes in the cloud" in *European Conference on Service-Oriented and Cloud Computing*, pp. 80–94, Springer, 2015.

[68] M. Harward, W. Irwin, and N. Churcher, "In situ software visualisation" in *2010 21st Australian Software Engineering Conference*, pp. 171–180, IEEE, 2010.

[69] T. M. Ahmed, C.-P. Bezemer, T.-H. Chen, A. E. Hassan, and W. Shang, "Studying the Effectiveness of Application Performance Management (APM) Tools for Detecting Performance Regressions for Web Applications: An Experience Report" in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pp. 1–12, 2016.

[70] New Relic, Inc, "New Relic" `https://www.newrelic.com`. (letzter Zugriff am 09.08.2021).

[71] Dynatrace LLC, "Dynatrace" `http://www.dynatrace.com/`. (letzter Zugriff am 09.08.2021).

[72] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Evaluating the accuracy of Java profilers" *ACM Sigplan Notices*, vol. 45, no. 6, pp. 187–197, 2010.

[73] J. H. Andrews, "Testing using log file analysis: tools, methods, and issues" in *Proceedings 13th IEEE International Conference on Automated Software Engineering (Cat. No. 98EX239)*, pp. 157–166, IEEE, 1998.

[74] J. Thiel, "An overview of software performance analysis tools and techniques: From gprof to dtrace" *Washington University in St. Louis, Tech. Rep*, pp. 1–19, 2006.

[75] Oracle, "java.lang.instrument (Java Platform SE 8)" `https://docs.oracle.com/javase/8/docs/api/index.html?java/lang/instrument/package-summary.html`. (letzter Zugriff am 09.08.2021).

[76] Oracle, "The jstack Utility" `https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr016.html`. (letzter Zugriff am 09.08.2021).

[77] A. Ragozin, "Safepoints in HotSpot JVM" http://blog.ragozin.info/2012/10/safepoints-in-hotspot-jvm.html, 2012. (letzter Zugriff am 09.08.2021).

[78] "YourKIT" https://www.yourkit.com/java/profiler/. (letzter Zugriff am 09.08.2021).

[79] J. P. Sandoval Alcocer, A. Bergel, S. Ducasse, and M. Denker, "Performance evolution blueprint: Understanding the impact of software evolution on performance" in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 1–9, 2013.

[80] "JProfiler" https://www.ej-technologies.com/products/jprofiler/overview.html. (letzter Zugriff am 09.08.2021).

[81] E. Tufte, "The visual display of quantitative information" *Cheshire: Graphic Press.–2001.–213 p*, 2001.

[82] Dataport AöR, "Dataport" https://www.dataport.de/. (letzter Zugriff am 09.08.2021).

[83] Apache Software Foundation, "Apache JMeter" https://jmeter.apache.org/. (letzter Zugriff am 09.08.2021).

[84] Microsoft, "Schnellstart: Erstellen eines Auslastungstestprojekts" https://docs.microsoft.com/de-de/visualstudio/test/quickstart-create-a-load-test-project. (letzter Zugriff am 09.08.2021).

[85] K6, "K6" https://k6.io/, 2020. (letzter Zugriff am 09.08.2021).

[86] D. Grossman, M. C. McCabe, C. Staton, B. Bailey, O. Frieder, and D. C. Roberts, "Performance testing a large finance application" *IEEE Software*, vol. 13, no. 5, pp. 50–54, 1996.

[87] "K6 - Metrics" https://k6.io/docs/using-k6/metrics/. (letzter Zugriff am 09.08.2021).

[88] F. Assmann, S. Bollen, C. Schmidt, O. Schwartz, D. Schwarz, and F. Thielke, "libAwesome (Software-Projekt 2 Abschlussaufgabe)" 2014.

[89] R Core Team, *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2020.

[90] H. Wickham, M. Averick, J. Bryan, W. Chang, L. D. McGowan, R. François, G. Grolemund, A. Hayes, L. Henry, J. Hester, M. Kuhn, T. L. Pedersen, E. Miller, S. M. Bache, K. Müller, J. Ooms, D. Robinson, D. P. Seidel, V. Spinu, K. Takahashi, D. Vaughan, C. Wilke, K. Woo, and H. Yutani, "Welcome to the tidyverse" *Journal of Open Source Software*, vol. 4, no. 43, p. 1686, 2019.

[91] "IntelliJ IDEA: The Capable & Ergonomic Java IDE by JetBrains" https://www.jetbrains.com/idea/. (letzter Zugriff am 10.08.2021).

[92] "RStudio" https://www.rstudio.com/. (letzter Zugriff am 14.08.2021).

[93] The Apache Software Foundation, "Apache NetBeans" https://netbeans.apache.org/. (letzter Zugriff am 13.08.2021).

[94] "Gitblit" http://gitblit.github.io/gitblit/. (letzter Zugriff am 10.08.2021).

[95] Mojang, "Minecraft" https://www.minecraft.net/. (letzter Zugriff am 13.08.2021).

[96] R. J. Hyndman and Y. Fan, "Sample quantiles in statistical packages" *The American Statistician*, vol. 50, no. 4, pp. 361–365, 1996.

[97] RDocumentation, "quantile: Sample Quantiles" `https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/quantile`. (letzter Zugriff am 09.08.2021).

[98] F. Fittkau, *Live trace visualization for system and program comprehension in large software landscapes*. BoD–Books on Demand, 2015.

[99] Axivion GmbH, "Technologieführer für statische Codeanalyse + Architekturprüfung | Axivion" `https://www.axivion.com/`. (letzter Zugriff am 10.08.2021).

[100] "Javassist by jboss-javassist" `https://www.javassist.org/`. (letzter Zugriff am 10.08.2021).

[101] P. J. Rousseeuw and G. W. Bassett Jr, "The remedian: A robust averaging method for large data sets" *Journal of the American Statistical Association*, vol. 85, no. 409, pp. 97–104, 1990.

[102] Eclipse Foundation, "Eclipse Jetty | The Eclipse Foundation" `https://www.eclipse.org/jetty/`. (letzter Zugriff am 10.08.2021).

[103] R. Westrelin, "How the JIT compiler boosts Java performance in OpenJDK" `https://developers.redhat.com/articles/2021/06/23/how-jit-compiler-boosts-java-performance-openjdk`. (letzter Zugriff am 10.08.2021).

[104] J. Brooke *et al.*, "SUS: A quick and dirty usability scale" *Usability evaluation in industry*, vol. 189, no. 194, pp. 4–7, 1996.

[105] "TeamViewer: The Remote Desktop Software" `https://www.teamviewer.com/en-us/`. (letzter Zugriff am 10.08.2021).

[106] "The Fast Remote Desktop Application – AnyDesk https://anydesk.com/en" `https://anydesk.com/en`. (letzter Zugriff am 10.08.2021).

[107] "Skype for Business – mit der Sicherheit und Kontrolle von Microsoft" `https://www.skype.com/de/business/`. (letzter Zugriff am 10.08.2021).

[108] "Discord | Ein Ort zum Treffen und zum Unterhalten" `https://discord.com/`. (letzter Zugriff am 10.08.2021).

[109] "Videokonferenzen, Cloud-Telefon, Webinare, Chat | Zoom" `https://zoom.us/`. (letzter Zugriff am 10.08.2021).

[110] E. Ludewig, *Usability und UX für Dummies*. Wiley, 2020.

[111] "System Usability Scale - Wikipedia" `https://de.wikipedia.org/wiki/System_Usability_Scale`. (letzter Zugriff am 07.08.2021).

[112] M. Rauer, "Measuring usability with the System Usability Scale (SUS)" `https://blog.seibert-media.net/blog/2011/04/11/usablility-analysen-system-usability-scale-sus/`. (letzter Zugriff am 07.08.2021).

[113] "YouTube" `http://youtube.com/`. (letzter Zugriff am 10.08.2021).

[114] Student, "The probable error of a mean" *Biometrika*, pp. 1–25, 1908.

[115] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)" *Biometrika*, vol. 52, pp. 591–611, Dec. 1965.

[116] F. Wilcoxon, "Probability tables for individual comparisons by ranking methods" *Biometrics*, vol. 3, no. 3, pp. 119–122, 1947.

[117] A. Bangor, P. Kortum, and J. Miller, "Determining what individual SUS scores mean: Adding an adjective rating scale" *Journal of usability studies*, vol. 4, no. 3, pp. 114–123, 2009.

[118] R. Koschke and M. Steinbeck, "Clustering paths with dynamic time warping" in *IEEE Working Conference on Software Visualization*, p. 12 pages, IEEE Computer Society Press, 2020.

[119] M. Steinbeck, R. Koschke, and M. O. Rudel, "Comparing the evostreets visualization technique in two-and three-dimensional environments a controlled experiment" *IEEE International Conference on Program Comprehension*, vol. 2019-May, pp. 231–242, 2019.

[120] "Google Maps" `maps.google.com`. (letzter Zugriff am 13.08.2021).

Anhang

A.1 USB-Stick

Der Arbeit liegt ein USB-Stick mit mehreren Inhalten und dem Quell-Code bei. Unter anderem ist ein digitales Exemplar dieser Arbeit enthalten. Zudem liegen ausführbare Programme und der Quell-Code der entwickelten Anwendungen vor. Auf dem USB-Stick liegen weitere `README.txt` Dateien, welche zusätzliche technische Informationen, insbesondere bezüglich des Quell-Codes bieten. Alle Anwendungen wurden nur unter Windows 10 getestet. Sollten beim Start oder beim Kompilieren Probleme auftreten unterstütze ich hier gerne. Die Zip-Dateien sollten idealerweise vorher vom USB-Stick auf eine schnellere Festplatte kopiert und dort entpackt werden.

Startbare Beispiele/Anwendungen:

- `startbare_beispiele/libAweseomeVisualisierung.zip`
  (Startbare Demo für die Performance-Edges mit Performance-Daten aus *libAwesome* ohne veränderten Quell-Code. Startbar über `SEECity.exe`)
- `startbare_beispiele/studie.zip`
  (Dieser Ordner enthält alle, in der Studien verwendeten, Aufgaben in startbarer Version. Hierfür liegt eine portable JDK, Netbeans und der Quellcode für libAwesome mit in den Ordnern. Die SEE-Aufgaben lassen sich in den Ordnern über `SEECity.exe` starten. Die Netbeans-Aufgaben werden über die `start_netbeans.bat` gestartet.)
- `startbare_beispiele/yprofiler.zip`
  (Sowohl der Quell-Code, als auch diese yprofiler.zip enthalt fertig gebaute, ausführbare .jar Dateien. Zum Ausführen muss die `run_profiler_with_output.bat` ausgeführt werden. In dieser Datei ist der Pfad zu einer, ebenfalls beiliegenden JDK 8 angegeben. Die, für die Verwendung des Profilers, notwendigen Dateien tools.jar und attach.dll liegen den ZIP-Dateien ebenfalls bei.)

Von mir entwickelter Quell-Code:

- `quellcode/SEE.zip/SEE/Assets/SEE/Game/PerformanceEdges/`
  (Enthält den Quellcode für die Darstellung der in dieser Arbeit implementierten Funktionen. Der Quellcode oberhalb dieses Ordners ist das Gesamtprojekts SEE, welches von weiteren Autoren entwickelt wurde. Der ausgecheckte Branch `YannisAbgabe` enthält den letzten Stand (für Unity-Version `2019.4.20f1`. Der Branch `YannisPerformanceEdges` enthält einen Versuch auf Unity Version `2020.3.14f1` zu upgraden, ist jedoch noch in Arbeit.)

- `quellcode/yprofiler.zip`
  (Der yProfiler wurde in Gänze von mir entwickelt. Wie dem startbaren Beispiels des yProfilers liegen auch diesem Ordner alle notwendigen Dateien zum Start der Anwendung bei. Dies beinhaltet ein JDK 8, die tools.jar und die attach.dll. Diese zusätzlichen Dateien wurden nicht von mir entwickelt.)

- `quellcode/RScripts.zip`
  (Enthält alle für die Auswertung verwendeten R-Skripte)

Nicht selbst entwickelter Quell-Code:

- `quellcode/SEE.zip`
  (Der Quellcode über dem oben genannten Ordners `PerformanceEdges` ist das Gesamtprojekt SEE, welches von weiteren Autoren entwickelt wurde. **Achtung:** Zu Entwicklungszwecken ist in den zu dieser Arbeit gehörenden Branches das libAwesome Projekt mit im Projekt getracked. Dies sollte, vor einem *Merge* in den master, entfernt werden.)

- `quellcode/libAwesome.zip`
  (Das, häufig in dieser Arbeit getestete Softwareprojekt, welches jedoch nicht von mir entwickelt wurde. Der, für die Evaluation, eingefügte Code ist über die Branches `bug/ScenarioB`, `bug/ScenarioC` und `bug/ScenarioE` und `bug/parameter` findbar. Diese Änderungen sind jedoch auch innerhalb der Ausarbeitung dokumentiert.)

Videos:

- `videos/Demovideos/`
  (Demovideos für die implementierten Funktionen. Eine Beschreibung der Videos in Form von Untertiteln im `.srt`-Format liegt bei.)

- `videos/Studie Instruktionsvideos/`
  (Vertonte Instruktionsvideos für die Testpersonen der Studie.)

Die Dateien auf dem USB-Stick haben folgende SHA1-Checksummen. Mit diesen Summen kann geprüft werden, ob die Dateien auf dem USB-Stick, nach Abgabe der Arbeit, verändert oder beschädigt wurden.

```
$ find . -type f -exec sha1sum {} \;
5944b8dcece42ae16038d6c036134c30a44b1cfd ./startbare_beispiele/libAweseomeVisualisierung.zip
533b28752841f2d2d556226555dbdfc22c373a05 ./startbare_beispiele/studie.zip
905eda280ad865e8ec8c2cae59608004d14e6ad2 ./startbare_beispiele/yprofiler.zip
504fc6ba0645f4fc1866cdffacd7d2971722e3d3 ./videos/Studie Instruktionsvideos/SEE.mp4
ba288ca6c60164490c553ec46fe2b967f7f74dd3 ./videos/Studie Instruktionsvideos/Netbeans.mp4
cc58374faa147b75fd4256609879bd757610bf89 ./videos/youtube_links.txt
24afbab47286538d362cec435c2825f0201fabed ./videos/Demovideos/Visualisierung in SEE.mp4
1c1b5e4f1d4b82b8d362061116ac2f711dbf3356 ./videos/Demovideos/yProfiler.mp4
a0aae30f1eb2a0b7af27b9f7db613c234ad1b81c ./videos/Demovideos/Demovideo yProfiler.srt
4c39f2935fb9985c836dba9f079145a462bbb1cc ./videos/Demovideos/Demovideo Visualisierung in SEE.srt
f3a3a0bdc2aba50e185b6bf5c1b5f1b3bdaa3a3b ./README.txt
810b84d607028c78384ec8644b6856c840d50f5e ./quellcode/SEE.zip
75a37211aa0202c70d019aeca2c57fff51d0f2a4 ./quellcode/RScripts.zip
77d0d445c22ad63ac38e1ba22f8a8236b27d8d44 ./quellcode/yprofiler.zip
f6332ed754e30c0cd62d8be959fd7356886fba8d ./quellcode/libAwesome.zip
1c715c17e75ec69e41648a2f323a5c92e31e3409 ./quellcode/SEE_README.txt
```

A.2 Videos

Im Rahmen dieser Arbeit wurden Videos erstellt und Youtube als Plattform für den Austausch verwendet. Die Videos sind auch nach Abgabe der Arbeit für unbestimmte Zeit verfügbar. Die Videos liegen auf dem USB vor, falls sie zukünftig nicht mehr auf YouTube verfügbar seien sollten. Text A.1 und A.2 sind Transkriptionen der beiden Instruktionsvideos.

- Demovideo Profiler `https://youtu.be/Bq5Ef31JUWo`
- Demovideo Visualisierung `https://youtu.be/Qgb8azxCW_k`
- Instruktionsvideo für Netbeans `https://youtu.be/SQhtTMLBmG0`
- Instruktionsvideo für SEE `https://youtu.be/hcJAJRPg8Y0`

Bevor die richtige Aufgabe beginnt, schauen wir uns gemeinsam eine Einführungsaufgabe an.
Bei dieser Aufgabe schauen wir uns zunächst die Steuerung der Anwendung und danach den Inhalt an.
Die Aufgabe, sowohl in der Einführungsaufgabe, als auch in der richtigen Aufgabe, wird es sein, einen Performance–Bug zu finden.
Mit Performance–Bug ist hier eine Stelle im Programm gemeint, die sich absichtlich langsamer gemacht habe, indem ich hier Code hinzugefügt habe, der das Programm an der Stelle etwas langsamer macht.
In allen Aufgaben ist dieser Performance–Bug relativ einfach zu erkennen und dürfte auch Auffallen, obwohl die Anwendung ansonsten völlig unbekannt ist. In der Einführungsaufgabe hier, werden wir nun ungefähr sehen, wie kompliziert diese Aufgaben sind. Die Aufgabe endet damit, dass der Bug gefunden und benannt wird. Es ist nicht notwendig – beziehungsweise es ist nicht gewünscht – den Code zu verändern oder den Bug zu beheben.
Im Nachhinein kann in der Umfrage [sic] noch angegeben werden, wie der Bug behoben werden könnte. Aber das ist optional. Gesteuert wird diese Anwendung zum einen mit den Pfeiltasten. Hiermit können wir uns so im Raum herumbewegen.
Und mit Hilfe der Maus und der rechten Maustaste – damit können wir die Kamera drehen.
Wir können und dann auch gleichzeitig Bewegen und die Kamera drehen. Und mit dem Mausrad können wir in die Szene reinzoomen und wieder herauszoomen.
Wir sehen hier eine SEE–City. Das ist eine 3D–Visualisierung von der Bürchereianwendung libAwesome.
Die einzelnen Häuser, die wir hier sehen, sind einzelne Methoden aus der Anwendung libAwesome.
Wenn wir mit der Maus darüber gehen, wird uns der Methodenname samt Paket und Klassenname angezeigt. Also der komplette Pfad der Methode. Mit der Maus können die Methoden ebenfalls angeklickt werden um den Quellcode entsprechend anzuzeigen.
Dann sehen wir hier im rechten Quellcode–Fenster die entsprechende Methode angezeigt.
Der Rest der Datei, in der sich diese Methde befindet wird allerdings nicht angezeigt.
Die Rohre, beziehungsweise Kanten, die wir hier sehen; Die sich zwischen den Häusern aufspannen; Das sind einzelne Aufrufkanten, des Aufrufgraphen. Sie sind eine Kombination aus mehreren unterschiedlichen Stack–Traces, die im vorhinein aufgezeichnet wurden. Die Richtung, in die die Rohre animiert sind, geben jeweils die Aufrufrichtung an. Das heißt von dieser Methode hier wird die Methode dort hinten aufgerufen.
Die Dicke der Rohre zeigt an, wie häufig die Methode in den aufgezeichneten Stack–Traces innerhalb des Messzeitraumes vorkam. Sie geben also an, an welchen Stellen sich die Software wie häufig aufgehalten hat.
Die dicken Kanten sind demnach die Hot–Spots, in denen die Anwendung sich am häufigsten befand und welche wir für die Analyse am ehesten betrachten sollten.
Nicht nur die einzelnen Methoden können angeklickt werden, sondern auch die einzelnen Aufrufkanten.
Wenn wir mir der Maus darüber gehen, sehen wir, dass sie schon leicht gehighlighted [sic] werden.
Klicken wir eine von den Methoden jetzt an, können wir auf der rechten Seite die Aufgerufene Methode sehen und auf der linken Seite zusätzlich noch die Stelle, von welcher die Methode aufgerufen worden ist.
In den beiden angezeigten Methoden sehen wir jetzt verschiedene hervorgehobene Zeilen. Auf der linken Seite einmal hier die aufrufende Methode. Die entspricht der rot markierten Kante in der 3D–Ansicht. Wir sehen auch "getElement" – den Namen.
Auf der rechten Seite sehen wir dann auch noch mal hervorgehobene Zeilen. Von diesen Zeilen ausgehend sind dann auch wieder einzelne Aufrufkanten. Diese werden, wenn man mit der Maus über eine der Zeilen geht, auch oben in der 3D–Ansicht hervorgehoben – und die können dann in der 3D–Ansicht ausgewählt werden.
Auch hier ist die Stärke der Hervorhebung entsprechend der dicke der einzelnen Kanten.

Das generelle Vorgehen um den Performance–Bug zu finden, ist es nun, die dickeren Kanten anzuschauen und jeweils in den Quellcode zu schauen, ob man den Performance–Bug erkennen kann.
Sobald der Performance–Bug erkannt wurde, soll die Aufgabe abgegeben werden, indem unten rechts – hier – auf den Button "submit submit [sic] Method" geklickt wird.
Hierbei ist es egal, ob der Bug sich jetzt auf der linken oder auf der rechten Seite befindet. Das wird beides gleich gewertet.
Ich habe die hier einmal Abgegeben. Und jetzt steht hier unten kurz "Submitted". Und nach ein paar Sekunden hätte man dann theoretisch die Möglichkeit noch einmal neu Abzugeben, für den Fall, dass der [sic] nicht die richtige Methode erkannt wurde oder einfach noch mal weiter geschaut werden kann – weiter geschaut werden möchte.
Zuletzt noch einmal ein paar kleine Funktionen, die nicht ganz so wichtig sind, aber trotzdem vorhanden sind. Man kann die ein– und ausgehenden Kanten – also die von außerhalb des Projektes eingehenden und die von innerhalb des Projektes nach außen ausgehenden Kanten. Hier den großen In und Out Block hier an den Seiten. Die kann man deaktivieren um sich ein bisschen mehr Übersicht zu verschaffen. Das ist allerdings nicht unbedingt notwendig. Vor allem, weil man durch die großen eingehenden und ausgehenden Kanten die großen Hot–Spots eher noch erkennt. Dann sind hier noch Vor– und Zurück–Button. Mit denen kann man über die Kanten, die man ausgewählt hat, wieder vor und zurück und vorwärts springen. Das kann bei der Navigation hilfreich werden.

Vor Start der richtigen Anwendung, ist nur aber noch genügend Zeit um die Anwendung in Ruhe auszuprobieren und dabei weitere Fragen zu stellen.

Text A.1: Transkription des Instruktionsvideos für SEE

Bevor die richtige Aufgabe beginnt, schauen wir uns gemeinsam eine Einführungsaufgabe an.
Bei dieser Aufgabe schauen wir uns zunächst die Steuerung der Anwendung und danach den Inhalt an.
Die Aufgabe, sowohl in der Einführungsaufgabe, als auch in der richtigen Aufgabe, wird es sein, einen Performance–Bug zu finden.
Mit Performance–Bug ist hier eine Stelle im Programm gemeint, die sich absichtlich langsamer gemacht habe, indem ich hier Code hinzugefügt habe, der das Programm an der Stelle etwas langsamer macht.
In allen Aufgaben ist dieser Performance–Bug relativ einfach zu erkennen und dürfte auch Auffallen, obwohl die Anwendung ansonsten völlig unbekannt ist. In der Einführungsaufgabe hier, werden wir nun ungefähr sehen, wie kompliziert diese Aufgaben sind. Die Aufgabe endet damit, dass der Bug gefunden und benannt wird. Es ist nicht notwendig – beziehungsweise es ist nicht gewünscht – den Code zu verändern oder den Bug zu beheben.
Im Nachhinein kann in der Umfrage [sic] noch angegeben werden, wie der Bug behoben werden könnte. Aber das ist optional.

Wir sehen hier nun die Netbeans Entwicklungsumgebung, welche den Java–Profiler "VisualVM" integriert hat. Mit diesem Profiler wurde im Voraus bereits ein Snapshot der Anwendung gemacht. Das heißt, wir haben über einen kurzen Zeitraum die Anwendung getestet und dabei die Auslastung der einzelnen Methoden gemessen.
Die aufgezeichneten Performancedaten werden in dieser Ansicht als tabellarischer Aufrufbaum dargestellt. Den sehen wir hier.
Dieser Baum kann schrittweise auf– und zugeklappt werden. Dann können wir die einzelnen Methoden in dem Aufrufbaum uns anschauen.
Zur Erklärung nochmal: Diese Methode hat diese Methode aufgerufen.
Mit einem Rechtsklick kann der Quell–Code der Methode aufgerufen werden.
Und im Quell–Code können wir dann auch hier die Methode – den Methodenaufruf – getElements sehen, welcher diesem Aufruf hier entspricht.
Rechts neben den Methoden können wir jetzt hier eine Heatmap sehen.
Diese gibt an in welchen Methoden sich das Programm am häufigsten aufgehalten hat.
Das generelle Vorgehen um den Performance–Bug zu finden, ist es nun sich die langsamsten Methoden anzuschauen, das sind die, die hier die größten roten Balken haben und zu schauen, ob der Performance–Bug gefunden werden kann.
Hierfür hangelt man sich am besten dann durch den Aufrufbaum so ein bisschen durch und schaut sich für einzelne Methoden den Quellcode an.
Sobald der Performance–Bug erkannt wurde, soll bitte Bescheid gegeben werden, da ich die Bearbeitungszeit manuell mit einer Stoppuhr stoppe.
Es gibt alledings in Netbeans noch eine zweite Ansicht, welche mindestens genau so wichtig ist, wie die erste mit diesem Baum hier:
Und zwar die sogenannte Hot–Spots Ansicht. Die können wir erreichen, indem wir hier auf den Button klicken.
Wir können, wenn wir sie anzeigen die andere wieder ausblenden, indem wir dort auch nochmal auf den Button klicken und hier Filtern, so wie ich das in dem anderem auch schon gemacht habe. Und hier aber ist ganz besonders wichtig, dass wir nach der Total Time sortieren, zumindest für diese Aufgabe, die wir hier haben. Und da haben wir dann auch sortiert nochmal, nach den einzelnen Aufrufzeiten der einzelnen Objekte – äh – Methoden.
Diese Ansicht sollte man sich im Hinterkopf behalten, weil sie kann auch für die Aufgabe eventuell nützlich werden. Einige Methoden kann man mit dieser Ansicht zum Beispiel deutlich einfacher erkennen.
Der Nachteil dieser Ansicht ist allerdings, dass man den Kontext nicht mehr sieht und wenn zwar diese Methode hier oben die Langsamste ist, der Grund dafür, dass sie die Langsamste ist, aber in einer anderen Methode liegt, können wir das in dieser Ansicht nicht genau sehen.
Hilfreich ist hier dafür dann noch die Option, dass man mit dem Rechtsklick in den Forward–Calls wieder suchen kann. Das heißt in der Ansicht, die wir uns zuerst angeschaut haben. Dann springt man hier zu der Methode und dann kann man sich den näheren Kontext nochmal genau anschauen.

Vor Start der richtigen Anwendung, ist nur aber noch genügend Zeit um die Anwendung in Ruhe auszuprobieren und dabei weitere Fragen zu stellen.

Text A.2: Transkription des Instruktionsvideos für Netbeans

A.3 Änderungsprotokoll des Experiments

Testperson	Version	Änderung/Besonderheit
TP1	Pilot	Pilot hatte noch keine keine Videos
TP2	Videos	Ab TP2 gab es Einführungsvideos
TP3	NoDiscuss	Ab TP3 wurden eventuelle Fehler erst besprochen, nachdem die Testperson den entsprechenden Fragebogen-Abschnitt ausgefüllt hat.
TP4	BetterDialog	Optimierung des Auswahldialogs bei mehreren Aufrufzielen.
TP5	WrongEdges	Neu: Frage nach Studiengang, Frage nach Erfahrungen mit Netbeans/VisualVM Neuer Bug: Ausgehende Kanten nur mit „UnknownOut“ beschriftet
TP6	WrongEdges	
TP14	WrongEdges	
TP7	NoColor	Fix der „UnknownOut“-Kanten. Neuer Bug: Stattdessen keine farbigen Methoden-Blöcke.
TP8	NoColor	
TP15	NoColor	
TP16	NoColor	
TP18	NoColor	
TP9	ColoredAgain	Methodenblöcke wieder eingefärbt.
TP25	ColoredAgain	
TP13	ColoredAgain	
TP17	ColoredAgain	
TP11	ColoredAgain	

Tabelle A.1: Änderungen und Besonderheiten bei einzelnen Testpersonen. Geordnet nach Durchfühungsreihenfolge.

A.4 Fragebogen

Die folgenden Fragen wurden auf dem Fragebogen der Studie gestellt.

#	**Frage / Antwortmöglichkeiten**
1	Studienfortschritt • aktuell im Bachelor • aktuell im Master • aktuell in der Ausbildung • abgeschlossenes Studium • abgeschlossene Ausbildung • ...
2	Aktueller oder abgeschlossener Studiengang/Ausbildung (Mehrfachauswahl möglich) • Informatik • Digitale Medien • Fachinformatiker Systemintegration • Fachinformatiker Anwendungsentwicklung • ...
3	Ich arbeite zur Zeit in der Softwareentwicklung oder im Softwaretest • in der Softwareentwicklung • im Softwaretest • Beides • keine Antwort • ...
4	Erfahrungen im Bereich Software-Entwicklung • 1 (keine Erfahrung) bis 5 (viel Erfahrung)
5	Erfahrungen im Bereich Software-Test • 1 (keine Erfahrung) bis 5 (viel Erfahrung)
6	Erfahrungen im Bereich Performance-Analyse und Profiling • 1 (keine Erfahrung) bis 5 (viel Erfahrung)
7a, 7b	Haben Sie das Performanceproblem gefunden? • Ja • Nein • Ich bin mir nicht sicher

8a, 8b	Hätten Sie einen Ansatz, wie das Performanceproblem nun behoben werden kann? • Ja • Nein • Ich weiß wo ich genauer suchen muss • Ich bin mir unsicher • keine Angabe
9a, 9b	Beschreiben Sie kurz wie Sie glauben, dass die Programm-Performanz optimiert werden kann. • ...
10a, 10b	Hat es Ihnen Spaß gemacht die Anwendung zu benutzen? • 1 (kaum Spaß) bis 5 (viel Spaß)
11a, 11b	Glauben sie, dass diese Darstellung in einem Softwareprojekt erfolgreich eingesetzt werden kann? • nein • eher nicht • neutral • eher ja • ja
12a, 12b	Standard Usability Score • Ich kann mir sehr gut vorstellen, das System regelmäßig zu nutzen. • Ich empfinde das System als unnötig komplex. • Ich empfinde das System als einfach zu nutzen. • Ich denke, dass ich technischen Support brauchen würde, um das System zu nutzen. • Ich finde, dass die verschiedenen Funktionen des Systems gut integriert sind. • Ich finde, dass es im System zu viele Inkonsistenzen gibt. • Ich kann mir vorstellen, dass die meisten Leute das System schnell zu beherrschen lernen. • Ich empfinde die Bedienung als sehr umständlich. • Ich habe mich bei der Nutzung des Systems sehr sicher gefühlt. • Ich musste eine Menge Dinge lernen, bevor ich mit dem System arbeiten konnte. **Antworten:** • überhaupt nicht • eher nicht • neutral • stimme eher zu • stimme voll zu
13	Welche Funktionen waren hilfreich für Sie?

	• (1) 3D-Aufrufkanten zwischen den Methoden • (2) hierarchische 3D-Darstellung des Softwareprojektes • (3) Einblendung des Programmcodes • (4) Vor- und Zurückbutton • (5) Methodennamen-Tooltip • (6) Hervorgehobene Zeilen **Antworten:** • nicht hilfreich • eher nicht hilfreich • neutral • eher hilfreich • sehr hilfreich • (nicht benutzt)
14a, 14b	Sonstige Anmerkungen • ...
15	Welche Funktionen waren hilfreich für Sie? • (1) Vorwärtsaufrufe • (2) Hot Spots • (3) Hierarchische/Geschachtelte Darstellung des Call-Graphs. • Programmcode anzeigen **Antworten:** • nicht hilfreich • eher nicht hilfreich • neutral • eher hilfreich • sehr hilfreich • (nicht benutzt)
16	Welche Darstellung hat Ihnen besser gefallen? • 2D Ansicht in Netbeans • 3D Ansicht in SEE • Gleich gut • Unsicher
17	In welcher Darstellung kamen Sie (gefühlt) leichter zum Ziel? • 2D Ansicht in Netbeans • 3D Ansicht in SEE • kein großer Unterschied • Ich weiß es nicht
18	Würden Sie Tools für die Performance-Analyse von Software gerne zukünftig einsetzen?

	• Ja • Nein • Vielleicht
19	Wie vertraut waren Sie vor der Studie Sie mit Profiling Software? Beispielsweise VisualVM oder direkt in die IDE-Integrierte Tools? • gar nicht • eher nicht • neutral • eher vertraut • sehr vertraut
20	Falls Sie andere Darstellungsformen kennen, empfinden Sie diese als besser/intuitiver als die 3D-Darstellung in SEE? • Ja • Unterscheiden sich nur kaum • Nein • Ich kenne keine vergleichbaren Tools • ...
21	Falls Sie vorherige Frage mit "Ja"beantwortet haben, welche Tools/Darstellungen finden Sie besser/intuitiver? • ...
22	Glauben Sie, dass Sie mit mehr Übung diese Tools effektiv bzw. effektiver einsetzen könnten? • Ja • Nein • Unsicher
23	Sonstige Anmerkungen • ...

A.5 Bildschirmfotos des Fragebogen

Der Fragebogen wurde über eine Web-Oberfläche ausgefüllt. Eine kompakte Repräsentation der Webseite ist in den folgenden Bildern zu sehen. Die Spaltenweise Anordnung ist nachträglich eingearbeitet worden. In der Oberfläche sahen ist immer nur eine der Spalten sichtbar und die ausfüllende Person wechselt nacheinander durch die einzelnen Abschnitte.

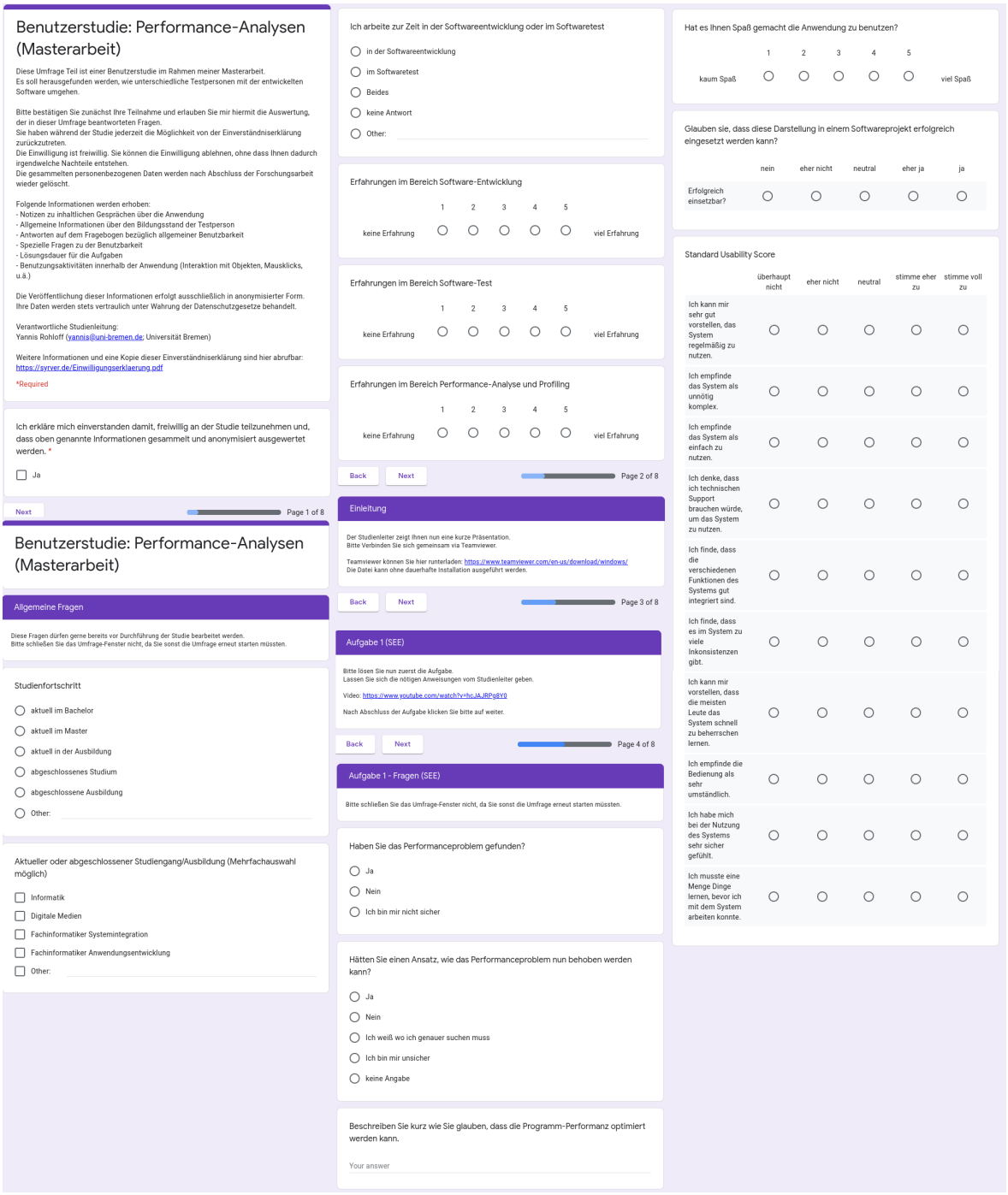


Abbildung A.1: Komposition der Seiten der Online-Umfrage. Spaltenweise zu lesen.

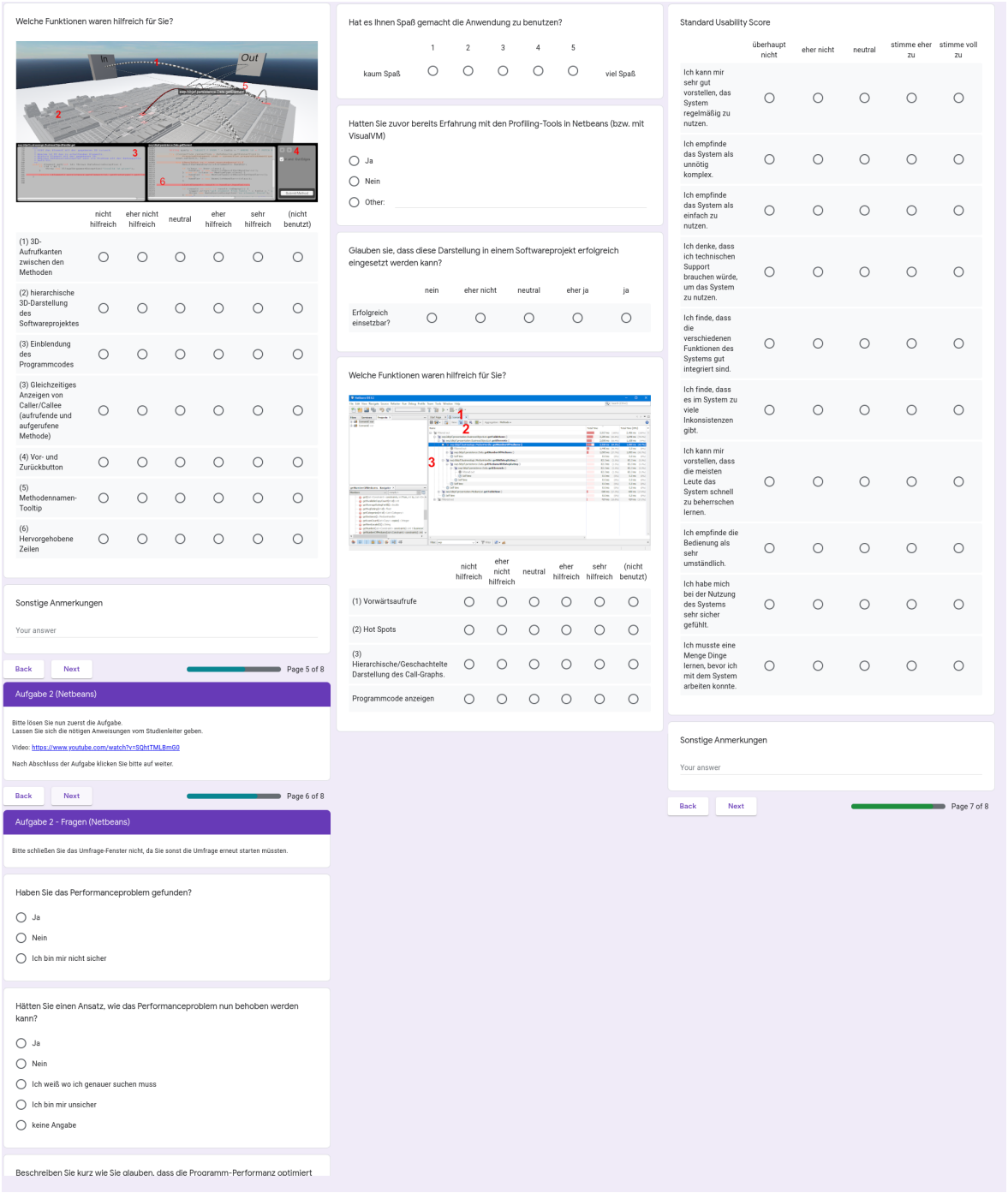


Abbildung A.2: Komposition der Seiten der Online-Umfrage. Spaltenweise zu lesen.

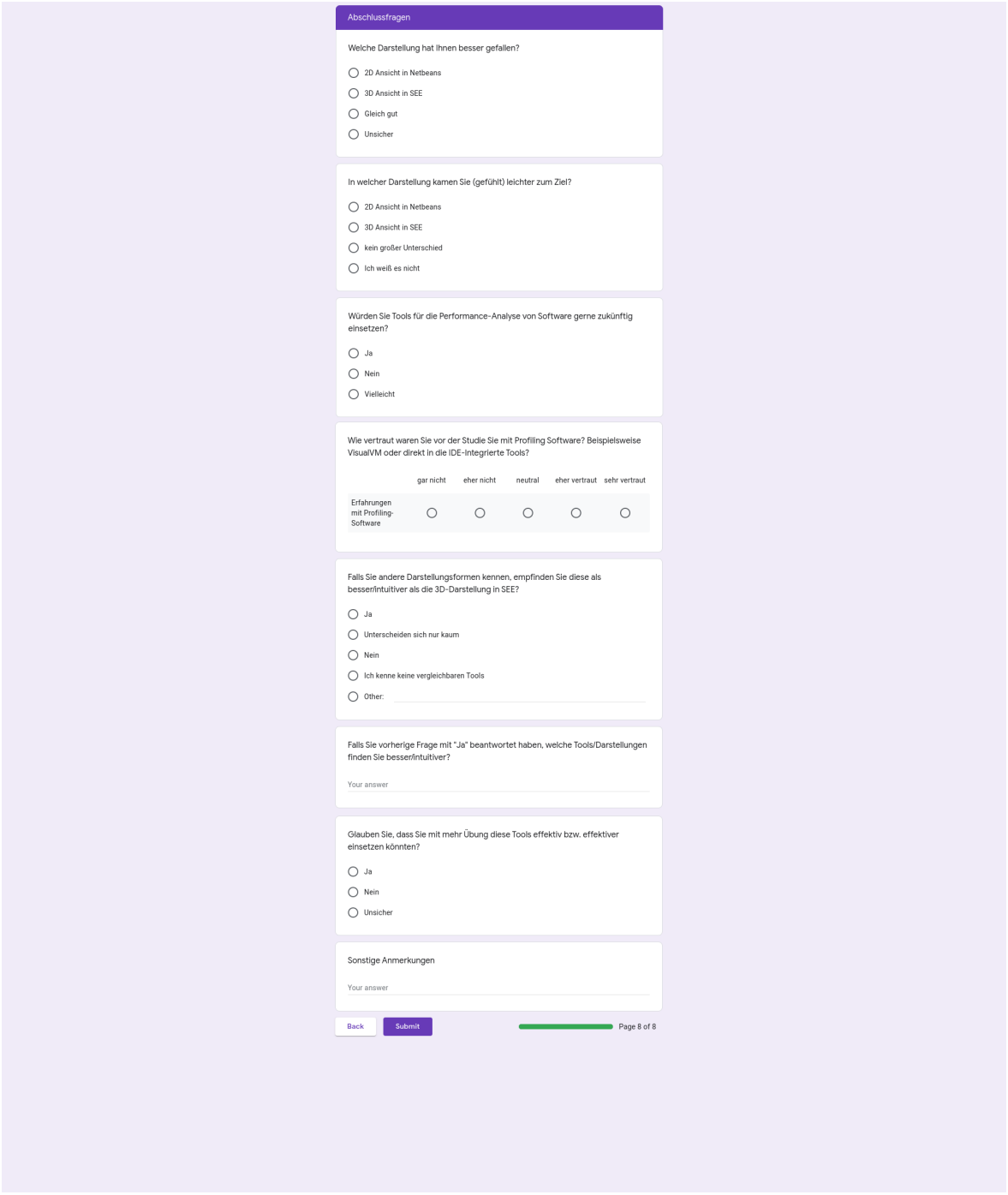


Abbildung A.3: Komposition der Seiten der Online-Umfrage. Spaltenweise zu lesen.

A.6 Präsentationsfolien der Studie

Die folgenden Präsentationsfolien wurden in der Benutzerstudie verwendet um den Testpersonen einen Überblick über die kommenden Aufgaben zu geben. Hauptsächlich getragen wurde die Studie jedoch durch die Instruktionsvideos.

Benutzerstudie: Performance-Analysen

Vergleich von 2D und 3D Darstellung von Laufzeit-Heatmaps

Masterarbeit Yannis Rohloff

Willkommen!

1. Vorbereitung
2. Einleitende Folien
3. Aufgabe 1
   a. Einführung Software A
   b. Aufgabe Software A
   c. Fragebogen
4. Aufgabe 2
   a. Einführung Software B
   b. Aufgabe Software B
   c. Fragebogen
5. Abschließender Fragebogen

Vorbereitung

1. Einverständniserklärung bestätigen.
2. Verbindung mit TeamViewer aufbauen.
3. Allgemeiner Abschnitt des Fragebogens.

Einführung

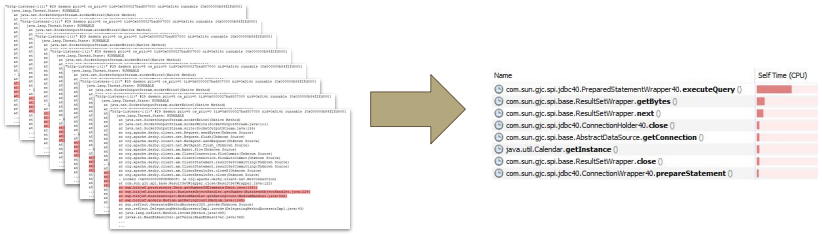
Einführung

- Programm in Laufzeit befindet sich immer an einem Punkt.
- Dieser Punkt kann durch einen Stack-Trace ausgedrückt werden.
- Ein Stack-Trace zeigt den Stapel der Methodenaufrufe an, welche zu der aktuellen Methode führen.

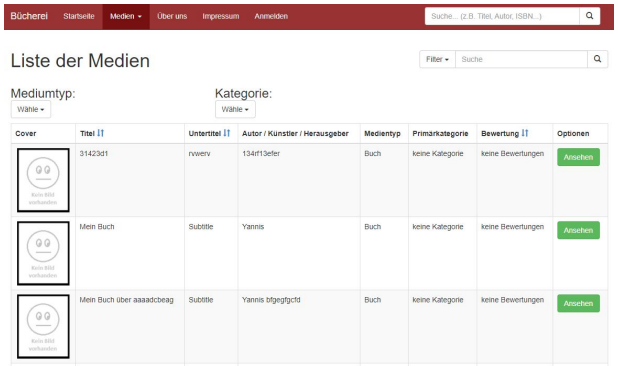
```
"http-listener-1(1)" #29 daemon prio=5 os_prio=0 tid=0x0000027bad807000 nid=0x314c runnable [0x000000b84f2fd000]
   java.lang.Thread.State: RUNNABLE
    at java.net.SocketOutputStream.socketWrite0(Native Method)
    at java.net.SocketOutputStream.socketWrite(SocketOutputStream.java:111)
    at java.net.SocketOutputStream.write(SocketOutputStream.java:155)
    at org.apache.derby.client.net.Request.sendBytes(Unknown Source)
    at org.apache.derby.client.net.Request.flush(Unknown Source)
    at org.apache.derby.client.net.NetAgent.sendRequest(Unknown Source)
    at org.apache.derby.client.net.NetAgent.flush_(Unknown Source)
    at org.apache.derby.client.am.Agent.flow(Unknown Source)
    at org.apache.derby.client.am.ClientConnection.flowCommit(Unknown Source)
    at org.apache.derby.client.am.ClientConnection.flowAutoCommit(Unknown Source)
    at org.apache.derby.client.am.ClientStatement.resultSetCommitting(Unknown Source)
    at org.apache.derby.client.am.ClientStatement.resultSetCommitting(Unknown Source)
    at org.apache.derby.client.am.ClientResultSet.closeX(Unknown Source)
    at org.apache.derby.client.am.ClientResultSet.close(Unknown Source)
    - locked <0x00000000fcb8bb00> (a org.apache.derby.client.net.NetConnection)
    at com.sun.gjc.spi.base.ResultSetWrapper.close(ResultSetWrapper.java:122)
    at swp.bibjsf.persistence.Data.getNumberOfElements(Data.java:1581)
    at swp.bibjsf.businesslogic.BusinessObjectHandler.getNumber(BusinessObjectHandler.java:229)
    at swp.bibjsf.businesslogic.MediumHandler.getRatingCount(MediumHandler.java:532)
    at swp.bibjsf.models.Medium.getRatingCount(Medium.java:1085)
    at sun.reflect.GeneratedMethodAccessor328.invoke(Unknown Source)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at javax.el.BeanELResolver.getValue(BeanELResolver.java:362)
    ...
    ...
```

Einführung

- Die Idee: Sammeln von ganz vielen Stack-Traces um eine Heatmap zu berechnen.

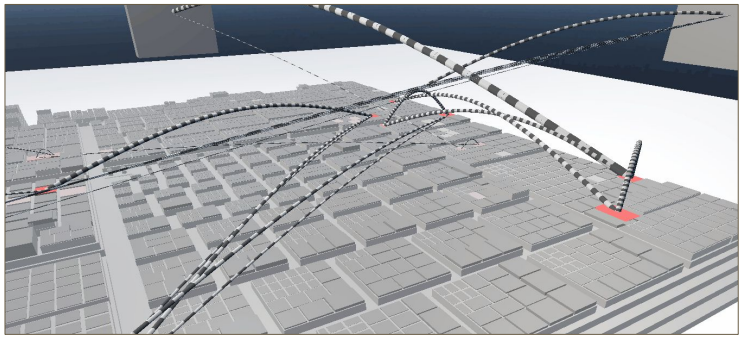


libAwesome



SEE

SEE 3D-Ansicht



Aufgabe - Übung

1. Lassen Sie sich die Bedienung der Software vom Studienleiter zeigen und machen sie sich mit der Software vertraut.

   Erklärvideo: https://www.youtube.com/watch?v=hcJAlRPg8Y0

Aufgabe

1. Finden Sie das Performance-Problem
   a. Es wurde absichtlich eine Stelle in der Software extra verlangsamt.
      Diese Stelle ist in der Ansicht recht deutlich erkennbar und sollte im Programmcode ohne längere Analysen erkennbar sein.

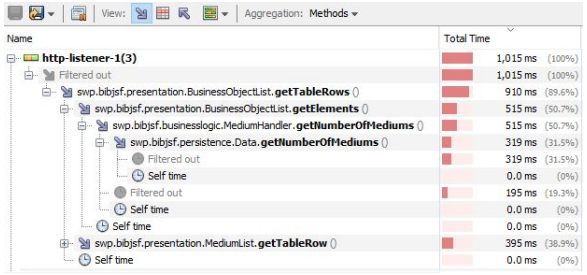
2. Klicken sie auf den Abgabe-Button, sobald Sie das Problem ausgewählt haben und es im Code-Fenster angezeigt wird.
   a. Sie können sich die Software danach in Ruhe noch weiter anschauen und die ihre Antwort korrigieren.

Fragebogen

- Fragebogen Abschnitt 2 ausfüllen.

Netbeans / VisualVM

Call-Graph



Aufgabe - Übung

1. Lassen Sie sich die Bedienung der Software vom Studienleiter zeigen und machen sie sich mit der Software vertraut.

   Erklärvideo: https://www.youtube.com/watch?v=ElI8GaOwQcU

Aufgabe

1. Finden Sie das Performance-Problem
   a. Es wurde absichtlich eine Stelle in der Software extra verlangsamt.
   Diese Stelle ist in der Ansicht recht deutlich erkennbar und sollte im Programmcode ohne längere Analysen erkennbar sein.

2. Geben Sie Bescheid, wenn sie glauben, dass Sie das Problem gefunden haben.
   a. Sie können sich die Software danach in Ruhe noch weiter anschauen und die ihre Antwort korrigieren.

Fragebogen

- Fragebogen Abschnitt 3 ausfüllen.

Abschluss

Vielen Dank!

Bitte füllen Sie nun noch die letzten Fragen im Fragebogen aus.

Danke sehr!

Bitte füllen Sie nun noch die
restlichen Fragen im Fragebogen aus.

A.7 Einwilligungserklärung

Es folgt die Einwilligungserklärung, welche den Studienteilnehmern zur Archivierung angeboten wurde. Unterschrieben wurde dieses Erklärung nur digital durch das setzen eines Hakens in dem Fragebogen, da die Studie online stattgefunden hat.

Einwilligungserklärung zur Erhebung und Verarbeitung personenbezogener Daten für Forschungszwecke

Version vom 05.04.2021

1 Gegenstand des Forschungsprojekts

1.1 Forschungsprojekt

Masterarbeit von Yannis Rohloff mit dem Arbeitstitel "Visualisierung von Performance-Informationen in 3D-Softwarestädten"

Beschreibung: Untersuchung der des Nutzerverhaltens innerhalb von zwei unterschiedlichen Darstellungsarten von Informationen.

1.2 Durchführende Person

Universität Bremen
Verantwortlicher: Yannis Rohloff
Interviewer: Yannis Rohloff

1.3 Art der personenbezogenen Daten des Betroffenen (der interviewten Person)

- Notizen zu inhaltlichen Gesprächen über die Anwendung
- Allgemeine Informationen über den Bildungsstand der Testperson

Weitere nicht personenbezogene Daten sind:

- Allgemeine und spezielle Fragen zu der Benutzbarkeit der Software
- Lösungsdauer für die Aufgaben
- Benutzungsaktivitäten innerhalb der Anwendung (Interaktion mit Objekten, Mausklicks, u.ä.)

2 Einwilligungserklärung

Die Einwilligungserklärung zu diesem Dokument geschieht über den in der Studie verwendeten digitalen Fragebogen. Der dort zusätzlich beiliegende Text ist:

> Diese Umfrage Teil ist einer Benutzerstudie im Rahmen meiner Masterarbeit.
> Es soll herausgefunden werden, wie unterschiedliche Testpersonen mit der entwickelten Software umgehen.
>
> Bitte bestätigen Sie zunächst Ihre Teilnahme und erlauben Sie mir hiermit die Auswertung, der in dieser Umfrage beantworteten Fragen.
> Sie haben während der Studie jederzeit die Möglichkeit von der Einverständniserklärung zurückzutreten.
> Die Einwilligung ist freiwillig. Sie können die Einwilligung ablehnen, ohne dass Ihnen dadurch irgendwelche Nachteile entstehen.
> Die gesammelten personenbezogenen Daten werden nach Abschluss der Forschungsarbeit wieder gelöscht.
>
> Folgende Informationen werden erhoben:
> – Notizen zu inhaltlichen Gesprächen über die Anwendung
> – Allgemeine Informationen über den Bildungsstand der Testperson
> – Antworten auf dem Fragebogen bezüglich allgemeiner Benutzbarkeit
> – Spezielle Fragen zu der Benutzbarkeit
> – Lösungsdauer für die Aufgaben
> – Benutzungsaktivitäten innerhalb der Anwendung (Interaktion mit Objekten, Mausklicks, u.ä.)
>
> Die Veröffentlichung dieser Informationen erfolgt ausschließlich in anonymisierter Form.
> Ihre Daten werden stets vertraulich unter Wahrung der Datenschutzgesetze behandelt.
>
> Verantwortliche Studienleitung:
> Yannis Rohloff (yannis@uni−bremen.de; Universität Bremen)
>
> Weitere Informationen und eine Kopie dieser Einverständniserklärung sind hier abrufbar: [Downloadlink entfernt]

> Ich erkläre mich einverstanden damit, freiwillig an der Studie teilzunehmen und, dass oben genannte Informationen gesammelt und anonymisiert ausgewertet werden.

Ihre Einwilligung ist freiwillig. Sie können die Einwilligung ablehnen, ohne dass Ihnen dadurch irgendwelche Nachteile entstehen.

Ihre Einwilligung können Sie jederzeit gegenüber der Studienleitung widerrufen, mit der Folge, dass die Verarbeitung Ihrer personenbezogenen Daten, nach Maßgabe Ihrer Widerrufserklärung, durch diesen für die Zukunft unzulässig wird. Dies berührt die Rechtmäßigkeit der aufgrund der Einwilligung bis zum Widerruf erfolgten Verarbeitung jedoch nicht.

Relevante Definitionen der verwendeten datenschutzrechtlichen Begriffe sind in der Anlage Begriffsbestimmungen enthalten.

2.1 Zweck der Datenverarbeitung / Ziel des Projekts

Die gesammelten Daten dienen der Evaluation von zwei Darstellungsformen für Metriken aus der Performance-Analyse.

2.2 Kontaktdaten

Yannis Rohloff (yannis@uni-bremen.de)

2.3 Rechtsgrundlage

Die von Ihnen erhobenen personenbezogene Daten werden auf Basis Ihrer Einwilligung gemäß Art. 6 Abs. 1 S. 1 lit. a DSGVO verarbeitet. Sofern besondere Kategorien personenbezogener Daten betroffen sind, verarbeitet die Arbeitsgruppe die von Ihnen erhobenen personenbezogenen Daten auf Basis Ihrer Einwilligung gemäß Art. 9 Abs. 2 lit. a DSGVO.

2.4 Empfänger oder Kategorien von Empfängern / Drittstaatenübermittlung

An folgende Empfänger oder Kategorien von Empfängern werden Ihre personenbezogenen Daten durch die Arbeitsgruppe übermittelt oder können übermittelt werden:

> weitere Arbeitsgruppen und Mitarbeiter der Universität Bremen, sofern dies für die Erfassung, Auswertung oder Speicherung der Daten notwendig oder sinnvoll ist

2.5 Dauer, für die die personenbezogenen Daten gespeichert werden / Kriterien für die Festlegung der Dauer

Bis zum Abschluss der Analysearbeiten, längstens jedoch für ein Jahr

2.6 Ihre Rechte

Im Rahmen der gesetzlichen Vorgaben haben Sie gegenüber der Arbeitsgruppe grundsätzlich Anspruch auf:

> Bestätigung, ob Sie betreffende personenbezogenen Daten durch die Arbeitsgruppe verarbeitet werden, Auskunft über diese Daten und die Umstände der Verarbeitung, Berichtigung, soweit diese Daten unrichtig sind, Löschung, soweit für die Verarbeitung keine Rechtfertigung und keine Pflicht zur Aufbewahrung (mehr) besteht, Einschränkung der Verarbeitung in besonderen gesetzlich bestimmten Fällen und Übermittlung Ihrer personenbezogenen Daten – soweit Sie diese bereitgestellt haben – an Sie oder einen Dritten in einem strukturierten, gängigen und maschinenlesbaren Format.

Darüber hinaus haben Sie das Recht, Ihre Einwilligung jederzeit gegenüber der Arbeitsgruppe zu widerrufen, mit der Folge, dass die Verarbeitung Ihrer personenbezogenen Daten, nach Maßgabe Ihrer Widerrufserklärung, durch diesen für die Zukunft unzulässig wird. Dies berührt die Rechtmäßigkeit der aufgrund der Einwilligung bis zum Widerruf erfolgten Verarbeitung jedoch nicht. Schließlich möchten wir Sie auf Ihr Beschwerderecht bei der zuständigen Aufsichtsbehörde hinweisen.

2.7 Keine automatisierte Entscheidungsfindung (inklusive Profiling)

Eine Verarbeitung Ihrer personenbezogenen Daten zum Zweck einer automatisierten Entscheidungsfindung (einschließlich Profiling) gemäß Art. 22 Abs. 1 und Abs. 4 DSGVO findet nicht statt.

3 Begriffsbestimmungen nach DSGVO

Im Sinne dieser Verordnung bezeichnet der Ausdruck...

Nach Artikel 4 Nr. 1 DSGVO [1]:

> "personenbezogene Daten" alle Informationen, die sich auf eine identifizierte oder identifizierbare natürliche Person (im Folgenden "betroffene Person") beziehen; als identifizierbar wird eine natürliche Person angesehen, die direkt oder indirekt, insbesondere mittels Zuordnung zu einer Kennung wie einem Namen, zu einer Kennnummer, zu Standortdaten, zu einer Online-Kennung oder zu einem oder mehreren besonderen Merkmalen identifiziert werden kann, die Ausdruck der physischen, physiologischen, genetischen, psychischen, wirtschaftlichen, kulturellen oder sozialen Identität dieser natürlichen Person sind;

Nach Artikel 4 Nr. 1 DSGVO [1]:

> "Verarbeitung" jeden mit oder ohne Hilfe automatisierter Verfahren ausgeführten Vorgang oder jede solche Vorgangsreihe im Zusammenhang mit personenbezogenen Daten wie das Erheben, das Erfassen, die Organisation, das Ordnen, die Speicherung, die Anpassung oder Veränderung, das Auslesen, das Abfragen, die Verwendung, die Offenlegung durch Übermittlung, Verbreitung oder eine andere Form der Bereitstellung, den Abgleich oder die Verknüpfung, die Einschränkung, das Löschen oder die Vernichtung;

4 Quellenangaben

1 EU-Datenschutz-Grundverordnung (Stand 27.04.2016), abgerufen von https://www.datenschutz-grundverordnung.eu/grundverordnung/art-4-ds-gvo/ (am 05.04.2021)

2 Inhalt und Struktur dieses Dokumentes basieren auf einer DSGVO-Einwilligungserklärung von www.audiotranskription.de (CC BY 3.0 DE)