Weiterentwicklung und Verbesserung einer Analysesoftware für dynamische Versuchsumgebungen

Diplomarbeit

zur Erlangung des akademischen Grades
Diplominformatiker

Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

eingereicht von:	Tobias Hampel
geboren am:	20.8.1978
in:	Eberswalde-Finow
Gutachter:	Klaus Bothe
	Hartmut Wandke
eingereicht am:	.....

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGS

AKRONYME

API Application Programming Interface

DOM Document Object Model

MVC Model View Controller

SAM Socially Augmented Microworld

MWB Mikroweltbewohner

OA Operateursarbeitsplatz

GUI Graphical User Interface

MW Mental Workload

AAF ATEO Automatik Framework

W3C World Wide Web Consortium

SAX Simple API for XML

JRE Java Runtime Environment

XML Extensible Markup Language

DTD Document Type Definition

RSS Rich Site Summary

MathML Mathematical Markup Language

XHTML Extensible HyperText Markup Language

SVG Scalable Vector Graphics

XPath XML Path Language

XSLT XSL Transformation

XPointer XML Pointer Language

XQuery XML Query Language

Jaxen Java XPath Engine

CSV Comma-Separated Values

1 EINLEITUNG

Komplexe technische Systeme sind heutzutage Bestandteil unseres täglichen Lebens. Insbesondere in industriellen Umgebungen spielen sie eine unentbehrliche Rolle und sind aus dem Arbeitsalltag nicht mehr weg zu denken. Je stärker solche Systeme eingesetzt werden, um so wichtiger ist dann eine effiziente Überwachung und Führung von Prozessen in diesen Systemen.

Um die effiziente Steuerung solcher Prozesse zu realisieren, kann man sie entweder automatisieren oder durch menschliche Beobachter regeln lassen. Hier lassen sich zwei Personengruppen ausmachen die erst einmal in Konkurrenz zueinander stehen. Auf der einen Seite gibt es Entwickler die Automatiken für die Steuerung von Prozessen konzipieren und implementieren. Dem gegenüber stehen Operateure, die solche Prozesse überwachen und direkt regeln. Beide haben die gleiche Aufgabe, nämlich die Optimierung eines komplexen dynamischen Prozesses, jedoch sehr unterschiedliche Ressourcen zur Umsetzung dieses Ziels.

Entwickler sind zeitlich sehr weit weg vom zu steuernden Prozess. Sie versuchen Ereignisse vorherzusehen und diese in das Verhalten eines Systems einzuplanen. Somit ist vor allem ihre Antizipationsfähigkeit gefordert. In der Regel haben sie ausreichend Zeit und können verschiedene Ereignisse prognostizieren und entsprechende Szenarien durchspielen. Ihnen fehlt jedoch oft die praktische Erfahrung, und ihr Wissen um die Einsatzsituation ist weitgehend abstrakt. Sie können daher niemals alle erdenklichen, insbesondere ungewöhnlichen, Situationen vorhersehen und folglich auch kein entsprechendes Verhalten in ihr System implementieren.

Operateure dagegen überwachen und regeln komplexe dynamische Prozesse (z.B. in Leitwarten für Kraftwerke) unter Verwendung von Schnittstellen zum zu überwachenden Prozess. Erst diese Schnittstellen ermöglichen es dem Operateur Eingriffe vorzunehmen, die eventuelle Unregelmässigkeiten korrigieren. Ohne solche Eingriffsmöglichkeiten würde sich die Rolle des Operateurs auf die eines reinen Beobachters reduzieren.

Operateure sitzen direkt vor dem Prozess und haben zeitlich keine nennenswerte Verzögerung für den Eingriff in diesen. Es ist also eine Frage des Erkennens von Ereignissen und der recht-

zeitigen und im besten Fall korrekten Reaktion. Zur Bewertung der Situation und zur Auswahl der richtigen Eingriffe können Operateure oft auf einen reichhaltigen Erfahrungsschatz zurückgreifen. Eine Eigenschaft die ihnen gegenüber Entwicklern einen Vorteil verschafft. Diese Erfahrung ermöglicht es ihnen in ungewöhnlichen, insbesondere kritischen Situationen angemessene Maßnahmen zu ergreifen.

Ein anderer Ansatz zur Steuerung technischer Prozesse sind Assistenzsysteme. Das Prinzip hierbei ist, Operateur und Automatik in Kooperation zusammen arbeiten zu lassen. Die Aufgabe der Entwickler ist in diesem Fall die Ausarbeitung von Konzepten in denen Automatik und Operateur zusammen diese Aufgabe lösen können. Der Grad der Automatisierung kann dabei unterschiedlich stark ausfallen. Es kommt hierbei auf die Stärken und Schwächen von Operateuren an und in wie fern hier Automatiserungssysteme helfen können.

Beim Bau komplexer Anlagen in denen es technische Prozesse zu überwachen und zu steuern gilt, ist es von großem Interesse eine effiziente Funktionsteilung zwischen Mensch und Maschine schon in der Entwurfsphase zu berücksichtigen. Die Erforschung einer solchen effizienten oder gar optimalen Funktionsaufteilung ist Gegenstand des ATEO-Projekts (ATEO - Arbeitsteilung Entwickler/Operateur). Das Ziel ist es, durch einen Vergleich der beiden Blickwinkel Erkenntnisse zu gewinnen, mit denen sich eine effiziente Funktionsteilung zwischen Mensch und Maschine erreichen lässt.

Verschiedene Untersuchungsreihen mit der ATEO Testumgebung sollen dazu beitragen, entscheiden zu können, inwieweit sich die Arbeitsteilung zwischen Entwicklern und Operateuren optimal gestalten lässt. Die Erkenntnisse daraus lassen sich im Idealfall verallgemeinern um in Zukunft eine effiziente Funktionsteilung zwischen Mensch und Maschine in komplexen Systemen zu gestalten.

Die in ATEO verwendete Versuchsumgebung nennt sich Socially Augmented Microworld (SAM) und soll die Leistungen von Entwicklern und Operateuren miteinander vergleichen. SAM beruht auf dem Konzept einer Mikrowelt. Mikrowelten sind stark reduzierte Abstrahierungen komplexer dynamischer Prozesse. Durch die Reduktion wird eine Untersuchung der Prozesse als Computersimulation ermöglicht. Die Mikrowelt von ATEO wird jedoch noch um zwei Mikroweltbewohner (MWB) bereichert um ihr eine nichtdeterministische Komponente hinzuzufügen. Ohne sie wäre die Mikrowelt durch die starke Reduktion bedingt vollständig

berechenbar und erklärbar, was nicht der Realität komplexer Prozesse entspräche. Wären sie das, käme es nie zu Störfällen oder Unregelmässigkeiten. Durch den Einsatz zweier MWB ist diese Mikrowelt nun nicht mehr vollständig berechenbar, da sich menschliches Verhalten nicht vorhersagen lässt. Barbara Gross und Jens Nachtwei bieten zum theoretischen Hintergrund von SAM weitere Informationen.[7]

Die eigentliche Aufgabe für die MWB ist nun, ein Objekt entlang eines Pfades in SAM möglichst schnell und genau vom Start ins Ziel zu steuern. Es handelt sich hierbei um eine kooperative Aufgabe, da die MWB das Fahrobjekt gemeinsam steuern und die Steuerung des Fahrobjektes zwischen beiden MWB zu gleichen Teilen verteilt ist. Während dieser Fahrt werden die MWB ständig von einem Operateur oder einer Automatik überwacht und gegebenenfalls durch Hinweise oder Eingriffe unterstützt. Die Automatiken werden von Entwicklern im Vergleich zu den Leistungen eines Operateurs konzipiert. Den Entwicklern können dabei unterschiedlich viele Informationen für die Entwicklung solcher Automatiken gegeben werden. Erste Arbeiten hierzu sind von Kai Kesselring dokumentiert.[11] Darauf aufbauende Arbeiten von Esther Fuhrmann[5] und Michael Hasselmann (Diplomarbeit in Arbeit) verfeinern diesen Ansatz noch.

Zur Bestimmung der Leistung der MWB gibt es die Indikatoren Flächenfehler, Zeitfehler und Mental Workload (MW). Der Flächenfehler ist hierbei die Abweichung der MWB von der Ideallinie. Analog dazu ist der Zeitfehler der Unterschied zwischen gefahrener Zeit und bestmöglicher Zeit. Unter MW versteht man im Zusammenhang mit ATEO das wahrgenommene Verhältnis zwischen der Menge der zur Verfügung stehenden Ressourcen zur Bewältigung einer Aufgabe und der Menge an Ressourcen die diese Aufgabe erfordert.

Das Verhalten der MWB und des Operateurs kann man durch empirische Daten beschreiben, die sich aus den Logfiles von SAM gewinnen lassen. Diese Daten machen erst einen Vergleich der Probanden möglich. Daraus lassen sich dann Gemeinsamkeiten und Unterschiede im Verhalten der Probanden in bestimmten Situationen erkennen. Es lassen sich Gruppen bilden, Schlüsse daraus ziehen und Effekte aufspüren, die von einem Versuchsleiter auf rein visueller Ebene nicht wahrgenommen werden. Die kleinste Änderung von Parametern des Experiments kann hierbei schon zu Verhaltensänderungen führen. Daraus ergibt sich die Möglichkeit künftiges Verhalten in vergleichbaren Situationen besser vorherzusagen.

Die automatisierte Verarbeitung der ATEO-Logfiles ist besonders wichtig für die am Projekt beteiligten Psychologen, da sie im Vergleich zu einer händischen Auswertung der Daten eine enorme Zeitersparnis und weniger Fehleranfälligkeit liefert. Sie bildet die Grundlage für die weiterführende statistische Analyse des Verhaltens der MWB und des Operateurs und ist daher notwendig.

1.1 VORARBEITEN

In einer Machbarkeitsstudie wurde ein Prototyp für die automatisierte Logfileauswertung entwickelt. Dieser ist Thema der vorausgegangenen Studienarbeit [8] und bildet den Ausgangspunkt für diese Diplomarbeit. In den folgenden Kapiteln wird der Begriff LFA 1.0 verwendet werden, wenn auf diesen Prototypen verwiesen wird.

LFA 1.0 bietet Basisfeatures zur Analyse der SAM-Hauptuntersuchung. In dieser Phase des Projekts wurden ausschliesslich die MWB getestet und analysiert. Es kam hierbei hauptsächlich auf die Berechnung verschiedener Leistungsindikatoren der MWB an, beispielsweise die Abweichung von der Ideallinie als Flächenmaß oder die Differenz von gefahrener Zeit und bestmöglicher Zeit auf einer bestimmten Strecke. Die Ergebnisse dieser Berechnungen wurden in einer Exceldatei zusammengefasst und bilden die Basis für eine weiterführende statistische Analyse. Der Operateursarbeitsplatz (OA) und die Automatiken befanden sich zu dieser Zeit noch in einem Entwicklungsstadium und konnten deshalb noch nicht bei der Logfileanalyse berücksichtigt werden.

1.2 MOTIVATION

Da sich die Software zur Logfileanalyse auf eine frühe Phase der ATEO-Versuche bezieht und zum Zeitpunkt ihrer Entwicklung noch nicht alle Anforderungen definiert werden konnten, zeichnete sich frühzeitig ab, dass Erweiterungen an LFA 1.0 notwendig sein werden. Beispielsweise konnte der OA noch nicht einbezogen werden. Weiterhin hat sich die Struktur und das Dateiformat des Logfiles im Laufe des Projekts mehrmals verändert. Erweiterungen in LFA 1.0 wären auch generell schwierig gewesen, da selbst bei kleinen Änderungen an relativ vielen Stellen Veränderungen im Quellcode erforderlich gewesen wären. Zudem war die Software in wenige große Module gegliedert und durch ihre Größe bedingt schwer verständlich. Ausserdem war die Logfileauswertung in LFA 1.0 ziemlich zeitintensiv. Zusätzliche Anforderungen hätten den Zeitbedarf für eine Analyse der Logfiles noch erhöht.

1.3 ZIELSTELLUNG DER ARBEIT

Das Ziel dieser Arbeit ist nun Weiterentwicklung und Verbesserung des Prototypen zur Logfileauswertung. Das bedeutet zum einen eine Erweiterung des Funktionsumfangs der Analysesoftware und zum anderen das Aufspüren und Beseitigen von Schwachstellen in der Software durch eine generelle Überarbeitung der Softwarearchitektur. Die Erweiterung des Funktionsumfangs ist eine natürliche Folge des Fortschreiten des ATEO-Projektes, da sich die konkreten Anforderungen an die Logfileanalyse aus dem aktuellen Forschungsstand des Projekts ergeben.

1.4 GLIEDERUNG

Kapitel eins stellt das Projekt ATEO vor, erläutert dessen Forschungsschwerpunkt und beschreibt die Problemstellung der Logfileanalyse, die Thema dieser Arbeit ist. Außerdem wird ein Überblick über die Vorarbeiten für diese Arbeit gegeben. Anschließend werden Aufbau und Inhalt dieser Arbeit in kurzen Sätzen vermittelt.

Das zweite Kapitel beschäftigt sich mit Analyse der Softwarearchitektur des Prototypen um dessen Schwachstellen zu identifizieren. Weiterhin werden das Thema Performance besprochen.

Im dritten Kapitel werden die gewünschten Anforderungen erhoben. Die bereits realisierten Features der Machbarkeitsstudie werden der Vollständigkeit halber noch einmal erwähnt und neue oder veränderte Anforderungen erläutert.

Die gewonnenen Erkenntnisse werden in Kapitel vier für einen verbesserten Entwurf der Architektur herangezogen. Die einzelnen Veränderungen werden besprochen um ihren Vorteil gegenüber LFA 1.0 klar zu stellen.

Das fünfte Kapitel stellt die technische Versuchsumgebung und deren einzelnen Komponenten kurz vor und beschreibt beteiligte Technologien die bei der Entwicklung der Analysesoftware eine Rolle spielen.

Kapitel sechs beschreibt die konkrete Implementierung des neuen Designs unter Verwendung der in Kapitel fünf besprochenen Technologien und Bibliotheken.

Kapitel sieben beschäftigt sich in einer kritischen Diskussion mit den vorgenommene Zielen und deren erfolgreiche Umsetzung. Es werden mögliche Erweiterungen betrachtet und eine hypothe-

tische Entwicklung eines neuen Themenschwerpunkts diskutiert, der auch starke Auswirkungen auf die Logfileauswertung haben könnte.

2 ANALYSE DES PROTOTYPEN

2.1 DER PROTOTYP LFA 1.0

Die Experimente mit der ATEO-Versuchumgebung sind in mehrere Phasen gegliedert, aus denen sich verschiedene Versuchsreihen ergeben. Eine Versuchsreihe besteht in der Regel aus etwa 25 Einzelexperimenten. In diesen Experimenten müssen die Probanden insgesamt etwa 15 Fahrten absolvieren. Die genaue Anzahl der Fahrten variiert von Versuchsreihe zu Versuchsreihe. Historisch bedingt werden die Fahrten von den Psychologen als *Step* bezeichnet. Die Analyse der Logfiles dieser Versuche erfolgte ursprünglich überwiegend händisch. Es waren hierzu mehrere Arbeitsschritte erforderlich, bei denen es nur für wenige dieser Arbeitsschritte computerunterstützte Werkzeuge gab. Diese Werkzeuge erforderten dann allerdings so hohe Ressourcen, dass ganze Rechnerpools reserviert mussten, wenn man eine Versuchsreihe auswerten wollte. Die Logfileanalyse war in den Anfangszeiten des ATEO-Projekts also besonders zeitaufwendig, Ressourcen bindend und zudem auch besonders fehleranfällig.

Vor diesem Hintergrund war die Notwendigkeit einer automatisierten Logfileanalyse, in der alle Einzelschritte vereint sind und die weniger Ressourcen erfordert, dringend gegeben. Mit der Entwicklung des Prototypen LFA 1.0 gab es erstmalig eine Software, die diese Ansprüche an die Logfileauswertung erfüllt. Er soll hier kurz vorgestellt und der Ablauf einer Logfileanalyse beschrieben werden.

LFA 1.0 ist konkret für die Auswertung von Logfiles der SAM-Hauptuntersuchung konzipiert. Diese beschäftigt sich ausschließlich mit der Analyse der Leistung der MWB in der SAM-Versuchsumgebung. Ein eventueller Operateur oder Automatiken spielen bei der SAM-Hauptuntersuchung keine Rolle.

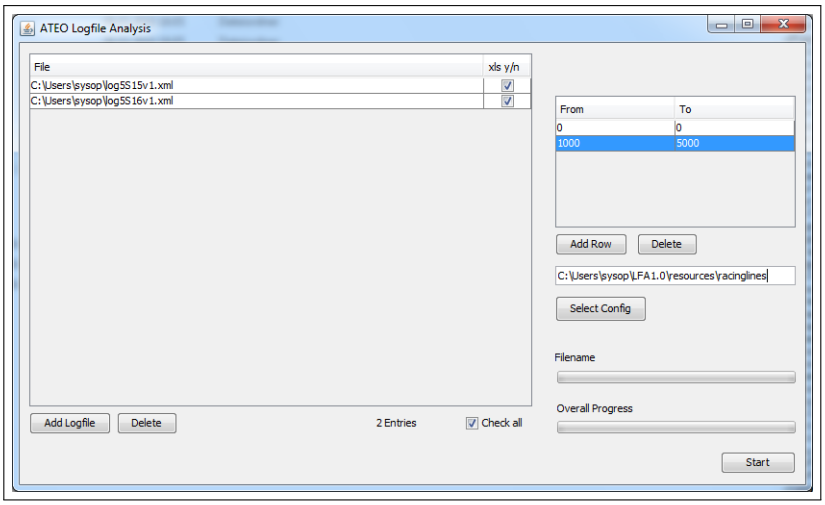


Abbildung 1: Graphical User Interface von LFA 1.0

Abbildung 1 zeigt das Graphical User Interface (GUI) für den Prototypen zur Logfileanalyse. Auf der linken Seite hat man eine Übersicht über die zu analysierenden Logfiles, auf der rechten Seite sieht man eine Tabelle mit anzuwendenden Messfenstern. Der Ablauf einer Logfileanalyse erfolgt nun nach folgendem Schema:

Zuerst werden alle relevanten Logfiles über einen entsprechenden Dateidialog ausgewählt. Üblicherweise wird dafür immer ein kompletter Satz von Logfiles verwendet. Darunter versteht man alle Logfiles einer bestimmten Fahrt in einer Versuchsreihe, also beispielsweise alle Logfiles von Fahrt 11 der SAM-Hauptuntersuchung. Nach der Auswahl der Logfiles erfolgt die Definition von so genannten Messfenstern. Dabei handelt es sich um konkrete Analyseintervalle. Bestimmte Abschnitte einer Fahrt sind für die Auswertung von größerem Interesse als andere und sollen deswegen isoliert betrachtet und analysiert werden. Die Definition eines solchen Intervalls erfolgt dabei durch Angabe von Pixelwerten für den Start- und Endpunkt eines Intervalls. Die Pixelwerte beziehen sich hierbei auf die zurückgelegte Strecke des Fahrobjekts.

Eine Analyse wird dann nur für dieses Messfenster durchgeführt. Bereiche außerhalb dieses Fensters werden ignoriert. Bei der Angabe mehrerer Fenster wird die Analyse entsprechend oft wiederholt. Die Ergebnisse der Analyse werden dann in einer Exceltabelle gespeichert und formatiert dargestellt. Für jedes Messfenster wird ein separates Excelfile erzeugt, da die Ergebnisse abhängig vom jeweiligen Messfenster sind.

Die Analyse unterteilt sich in verschiedene Features, die jeweils einen separaten Aspekt der Versuche betrachten sollen und die

Leistung der MWB bewerten. Konkrete Features sind unter anderem der Flächenfehler, der Zeitfehler, die Summe der Joystickinputs und Anzahl von Kollisionen. Unter Flächenfehler versteht man die Abweichung der gefahrenen Strecke von der Ideallinie. Analog dazu ist der Zeitfehler die Abweichung von gefahrener Zeit und bestmöglicher Zeit. Die Summe der Joystickinputs betrachtet die Intensität der Joystickbewegungen der MWB während einer Fahrt. Dabei werden horizontale (Lenkbewegungen) und vertikale Bewegungen (Beschleunigen/Bremsen) separat betrachtet. Bei der Anzahl Kollisionen wird ermittelt wie oft die MWB mit einem Hindernis zusammengestoßen sind.

Alle Analysefeatures sind unabhängig voneinander, sodass jedes Feature in LFA 1.0 separat umgesetzt wurde. Kein Feature baut auf auf dem Ergebnis eines anderen auf. Eine vollständige Liste aller Features sowie eine Beschreibung der Entwicklung des Prototypen findet man in der vorangegangenen Studienarbeit.[8]

2.2 QUALITÄTSKRITERIEN

Eine Software ist nur brauchbar, wenn sie die versprochenen Funktionen erfüllt, eine stabile Konstruktion darstellt, zuverlässig ist, vernünftige Antwortzeiten hat und leicht bedienbar ist. Hierzu werden Anforderungen an die Qualität aufgestellt, die Teil der Forderungen an das zu realisierende System sind.[6]
Wie lässt sich Qualität aber bewerten beziehungsweise was sind Qualitätsmerkmale eines Produkts? Die ISO-Norm 9126 [10] enthält Kriterien zur Bewertung der Qualität von Produkten. Nach ihr definiert sich Qualität eines Produkts als die Gesamtheit von Eigenschaften und Merkmalen eines Produkts, die sich auf dessen Eignung zur Erfüllung vorgegebener Erfordernisse bezieht. Qualität ergibt sich also aus der Differenz von Sollzustand- und Istzustand eines Produkts.[6]

Die Norm ISO/IEC 9126 stellt eins von vielen Modellen dar, um Softwarequalität sicherzustellen. Nach dem Qualitätsmodell dieser Norm teilt man die Eigenschaften von Software in sechs Kategorien ein:

- Funktionalität

  Funktionalität umfasst die Teilmerkmale Richtigkeit, Angemessenheit, Interoperabilität, Sicherheit und Konformität[1] der Funktionalität

- Effizienz

1 Konformität = Erfüllung von akzeptierten - auch informellen - Standards

Effizienz beeinhaltet Zeitverhalten, Verbrauchsverhalten und Konformität der Effizienz

- Wartbarkeit

  Zur Wartbarkeit gehören Analysierbarkeit, Änderbarkeit, Stabilität, Testbarkeit und Konformität der Wartbarkeit

- Portablität (Übertragbarkeit)

  Die Portabilität umfasst Anpassbarkeit, Installierbarkeit, Koexistenz, Austauschbarkeit und Konformität der Portabilität

- Zuverlässigkeit

  Zuverlässigkeit beinhaltet Reife, Fehlertoleranz, Wiederherstellbarkeit, Konformität der Zuverlässigkeit

- Benutzbarkeit

  Zur Benutzbarkeit gehören Verständlichkeit, Erlernbarkeit, Bedienbarkeit, Attraktivität und Konformität der Benutzbarkeit

Da Teile dieser Kriterien durchaus subjektiv empfunden werden, sind nicht alle Kriterien direkt messbar. Der Prototyp zur Logfileanalyse soll hier in diesem Kapitel hinsichtlich der ersten vier Qualitätsmerkmale untersucht werden, wobei Wartbarkeit und Effizienz im Vordergrund stehen sollen.

Funktionalität

Nach allgemeinverständlichen Kriterien steht Funktionalität für die Existenz der definierten Funktionen und das korrekte Verhalten eben dieser. In LFA 1.0 sind alle spezifizierten Funktionen korrekt umgesetzt worden und in der entsprechenden Studienarbeit dokumentiert.[8]

Portablität

Portabilität beschreibt die Eigenschaft, ein Programm auf verschiedenen Plattformen installieren zu können. Dies wird üblicherweise erreicht durch den Einsatz standardisierter Programmiersprachen. Betriebssystemspezifische Routinen sollten wenn möglich gar nicht oder zumindest sehr sparsam verwendet um ein portables System zu schaffen. Lässt sich der Einsatz solcher Routinen nicht vermeiden, sollte dies in gekapselten Komponenten geschehen um einen einfachen Austausch dieser Komponenten gewährleisten zu können. Im Allgemeinen betrachtet man ein System dann als portabel, wenn der Aufwand für die

Übertragung auf eine andere Plattform geringer ist als die Neuimplementierung des Systems.

LFA 1.0 ist von Hause aus als portabel anzusehen da es in der Programmiersprache Java umgesetzt ist. Die Software ist somit auf allen Systemen, für die es eine Java Laufzeitumgebung gibt ausführbar. Auf den Einsatz von Betriebssystem nahen Bibliotheken wurde komplett verzichtet. Einzige Bedingung für die Ausführbarkeit ist das Vorhandensein von ausreichend viel Arbeitsspeicher. Allerdings ist dies kein Kriterium hinsichtlich der Portabilität, sondern der Effizienz, worauf an späterer Stelle noch genauer eingegangen wird.

Wartbarkeit

Generell versteht man unter Wartbarkeit den Aufwand für die Weiterentwicklung einer Software und für das Beheben von Fehlern. Zwei Bewertungskriterien dafür sind die Prinzipien der Kopplung und der Kohäsion (oder auch Bindung). Unter Kopplung versteht man die Wechselwirkungen zwischen Teilsystemen eine Software. Kohäsion beschreibt das Ausmaß der Abhängigkeiten innerhalb eines Teilsystems. Im allgemeinen wird in der Softwareentwicklung angestrebt die Abhängigkeiten zwischen Subsystemen zu minimieren, was als lose Kopplung bezeichnet wird. Dies ermöglicht eine leichte Wiederverwendbarkeit der Komponenten und eine einfache Modifizierbarkeit des ganzen Softwaresystems. Der Austausch von Komponenten ist bei loser Kopplung viel einfacher. Insgesamt wird eine wesentlich einfachere Wartbarkeit erreicht. Zudem sind Klassen oder Teilsysteme besser verständlich, wenn sie wenig Bezüge zu anderen Teilsystemen haben.[19]

Neben der losen Kopplung versucht man gleichzeitig beim Entwickeln von Software die Bindung innerhalb eines Subsystems zu maximieren, was als starke Kohäsion bezeichnet wird. Dabei ist im besten Fall eine Klasse für die Lösung einer Aufgabe oder eines Problems zuständig. Klassen die mehrere Aufgaben auf einmal umsetzen entsprechen nicht dem Prinzip der starken Kohäsion und sollten vermieden werden.

Beide Prinzipien gleichermaßen gut umzusetzen erweist sich oft als schwierig, da sie sich gegenüber stehen. Eine starke Kohäsion führt häufig zu vielen Komponenten, die oft auch viele Verbindungen untereinander haben. Viele Verbindungen zwischen Komponenten stehen aber einer losen Kopplung im Wege. Bei loser Kopplung gibt es oft wenige Klassen, die für sich genommen autonom funktionieren. Dabei werden häufig innerhalb einer

Klasse mehrere Funktionalitäten realisiert, was aber dem Prinzip der starken Kohäsion entgegen steht. Man sieht also, dass es ganz besonders wichtig ist, eine angemessene Modularisierung der Software zu finden, um eine gute Balance zwischen Kopplung und Kohäsion zu gewährleisten.

Das Separation-of-Concerns-Prinzip besagt, dass man nach Möglichkeit Teilaspekte eines Problems voneinander trennen und die Teilprobleme dann isoliert betrachten soll. Das reduziert die Komplexität eines Systems und sorgt für ein besseres Verständnis. Es ist also auch unter diesem Gesichtspunkt ratsam, das Gesamtsystem in angemessene Module zu unterteilen.

Nach Betrachtung der Architektur von LFA 1.0 bleibt festzuhalten, dass sich die Software grob am Konzept des Model View Controller (MVC) orientiert. Es gibt nur sehr wenige Klassen, die zum Teil sehr groß, unübersichtlich und dadurch schwer verständlich sind. Die Klassen sind nicht in Pakete organisiert. Ihre Zugehörigkeit zu den Modulen Model, Controller oder View lässt sich daher nur aus deren Funktionalität und den Abhängigkeiten zwischen den Klassen erschließen. Die wesentlichen Klassen sind *GUIView*, *Control*, *XMLParse*, *LFA*, *Step* und *XLSWriter*.

In Abbildung 2 kann man sehen, dass die Klasse *Control* auf alle anderen Klassen zugreift und diese in irgendeiner Form verwendet. Für einen Controller der die Ablaufsteuerung eines Programms regelt ist das nicht anders zu erwarten. Die andern Klassen haben jedoch häufig Rückverbindungen zu *Control*. Diese sind in der Regel unnötig und auf schlechte Programmierung zurückzuführen. Diese Abhängigkeiten kann man durch Veränderung der Methoden in den Klassen und Parameterübergabe größtenteils eliminieren.

Die Klasse LFA implementiert alle funktionalen Features der Logfileanalyse. Dabei wird jedes Feature in einer separaten Methode umgesetzt. Hier sieht man deutlich konzeptuelle Schwachstellen. Das Separation-of-Concerns-Prinzip wurde nicht beachtet und eine starke Kohäsion lässt sich so nicht erreichen. Oberflächlich betrachtet gehören die einzelnen Methoden zwar thematisch zusammen, was eine Implementierung innerhalb einer Klasse nachvollziehbar erscheinen lässt. Im Detail handelt es sich hier jedoch um unterschiedliche Aufgaben.

Die einzelnen LFA-Methoden greifen über *Control* auf *XMLParse* zu. *XMLParse* wurde in *Control* als statische Variable definiert, um sicherzustellen, dass nie mehr als eine Instanz von *XMLParse* existiert. Der Grund dafür war der hohe Speicherbedarf der

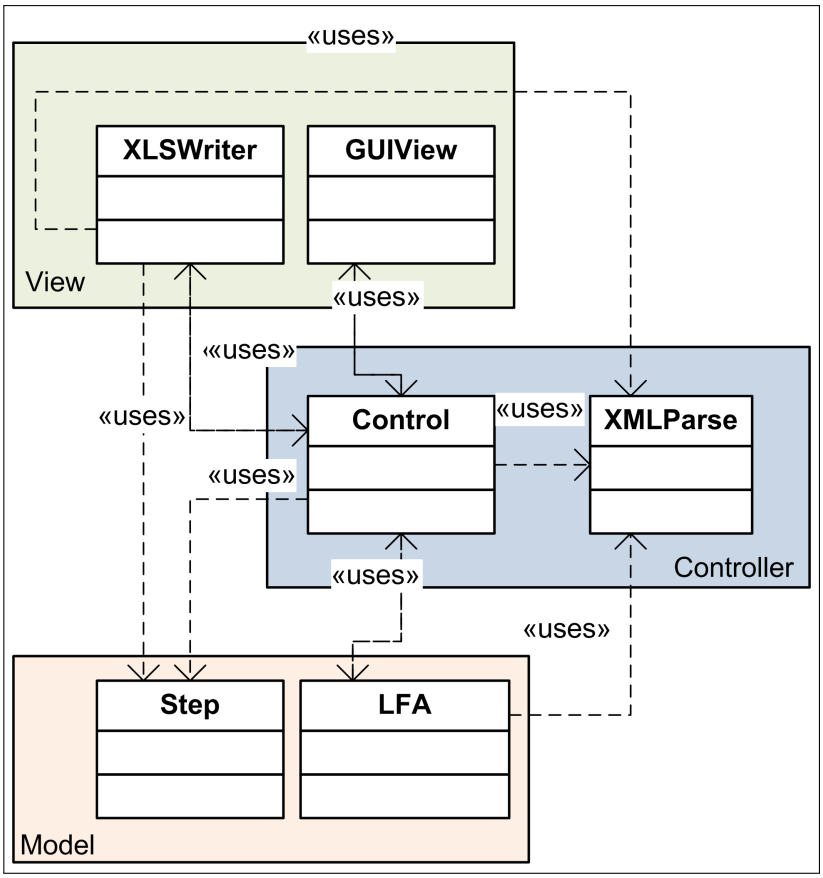


Abbildung 2: Die Klassen von LFA 1.0 und ihre Abhängigkeiten untereinander

Klasse. Es gibt jedoch auch andere Möglichkeiten, maximal eine Instanz einer Klasse zu erzeugen, beispielsweise als Singleton.

Die Zugriffe auf *XMLParse* und die gewünschten Logfilevariablen innerhalb des *XMLParse*-Objekts erfolgen zudem direkt und nicht über implementierte Schnittstellen. Diese Art der Umsetzung verletzt das Prinzip der Datenkapselung, was in der objektorientierten Programmierung ein wichtiges Konzept darstellt. Nach diesem Konzept werden Daten in einer Datenstruktur zusammengefasst, auf die nur über festgelegte Methoden zugegriffen werden kann. So erreicht man Kontrolle über den Inhalt von Klassen und verhindert, dass dieser in unerwarteter Weise von anderen Klassen verändert wird.

Abgesehen von der Verletzung des Prinzips der Datenkapselung führt diese Implementation zu einer unnötigen Abhängigkeit von *LFA* zu *Control* und zu *XMLParse*. Besser wäre es *XMLParse* als Parameter zu übergeben oder *LFA* gar nicht auf *XMLParse* zugreifen zu lassen. In diesem Falle würde der Controller dann die benötigten Daten vom Parser anfordern und sie an *LFA* wei-

tergeben.

Die Klasse *XLSWriter* greift ebenfalls direkt auf *XMLParse* zu. Das verletzt wie im vorangegangenen Fall das Prinzip der Datenkapselung und entspricht zusätzlich nicht dem Konzept des MVC. Da *XLSWriter* ein Teil des Views ist, sollte die Klasse alle zu visualisierenden Daten direkt vom Model beziehen und nicht auf den Controller zugreifen um sie von ihm zu bekommen. Der Controller sollte lediglich die Visualisierung der Daten anstoßen, aber nicht diese Daten selbst liefern. Die Daten, die sich *XLSWriter* daher von *XMLParse* holt, sollten also im Vorfeld in irgendeiner Form im Model abgelegt werden.

Die Klasse *LFA* enthält die drei Methoden *findboards*, *checkracinglines*, *createracinglines*, die keine Analysefeatures im eigentlichen Sinne sind. Vielmehr handelt es sich hier um Methoden die bestimmte Bedingungen prüfen, damit ein konkretes Feature ausgeführt werden kann. Insofern sollte man diese Methoden in den Controller verlagern, entweder als Teil von *Control* selbst oder als eigene Klassen.

Bei der Analyse des Quellcodes von LFA 1.0 wurde an vielen Stellen in der Klasse *LFA* Code dupliziert. In jeder Methode zur Umsetzung eines konkreten Features wurde ein Matchingausdruck erzeugt mit dem im *XMLParse*-Objekt nach passenden Logfilevariablen gesucht wurde. Dies lässt aber sich auch direkt in der Klasse *XMLParse* realisieren, was den duplizierten Code beseitigen und dadurch die Lesbarkeit der Features erhöhen würde. Man müsse dann durch Parameterübegabe dafür sorgen, dass der korrekte Matchingausdruck erzeugt wird.

Eine weitere unnötige Abhängigkeit der Klassen *LFA* und *Control* ist die Speicherung der Ergebnisse der einzelnen *LFA*-Methoden. *Control* ruft die einzelnen Methoden auf und speichert die Ergebnisse dann in der Klasse *Step* ab. Diese Klasse hält alle Ergebnisse der einzelnen Analysefeatures vor. Dazu werden die unterschiedlichste Datentypen der verschiedenen Features in einer Klasse zusammengefasst. Da aber gemäß dem MVC-Konzept allein das Model die Datenverarbeitung und -haltung leisten soll, kann man sich diesen Umweg sparen und diese Abhängigkeit eliminieren. Dazu müsste man eine Schnittselle schaffen die es den *LFA*-Methoden erlaubt, Ergebnisse direkt in der Klasse *Step* zu speichern.

Als Fazit lässt sich sagen, dass sich das Hinzufügen oder Bearbeiten von Features sehr schwierig gestaltet. Es wären Veränderungen an verschiedenen Stellen nötig um ein neues Feature in

die bestehende Software zu integrieren. Es ist daher notwendig, eine neue Architektur zu gestalten, die modularer aufgebaut ist und die Prinzipien Seperation-of-concern, starke Kohäsion und schwache Kopplung besser umsetzt. Außerdem muss das Zusammenspiel der Klassen *Control*, *LFA* und *XMLParse* durch Eliminierung unnötiger Abhängigkeiten verbessert werden, was letztendlich auch für eine bessere Verständlichkeit sorgen würde.

Effizienz

Unter Effizienz versteht man ein angemessenes Leistungsniveau in Bezug auf eingesetzte Ressourcen.[1] Man erwartet vom System einen gewissen Datendurchsatz und kurze Antwortzeiten. Weiterhin sollte ein System einen gewisser Ressourcenverbrauch nicht überschreiten. Effizienzbetrachtungen kann man unter den Teilaspekten Speichereffizienz und Laufzeiteffizienz führen.[15]

Effizienz ist mit dem Begriff Performance vergleichbar. Dirlewanger zitiert in seinem Buch[3] den ISO/IEC 14756 Standard[9] zur Messung der Performance bei Computersystemen und unterteilt darin die Performance in drei Klassen:

1. Die erste Klasse beschreibt Performance als die Fähigkeiten eines Computer. Dabei handelt es sich um eine Menge von Aktivitäten, die durchgeführt werden können, aber auch die Korrektheit der durchgeführten Operationen und deren Ergebnisse. Die Ergonomie, also die Benutzerfreundlichkeit (eng. Usability) des Computersystems wird ebenso zu dieser Klasse gezählt.

2. Bei der zweiten Klasse der Performance geht es um die Frage nach der Stabilität und Konsistenz des Computersystems und seiner Operationen. Gemeint ist hierbei die Zuverlässigkeit des Systems im weitesten Sinne.

3. Die dritte Klasse der Performance geht der Frage nach, wie schnell die Aufgaben durch das System ausgeführt werden. Zwei Aspekte werden dabei unterschieden: Zum einen geht es um die Zeit, die benötigt wird ein Ergebnis zu liefern; zum anderen geht es um die Anzahl von Aufgaben und Operationen, die innerhalb einer bestimmten Zeit ausgeführt werden können.

Bei der Auseinandersetzung mit LFA 1.0 soll vornehmlich die dritte Klasse herangezogen werden. Einflussfaktoren auf die Performance eines Systems sind die Implementierung der Anwendung, die verwendeten Technologien sowie das Betriebssystem. Sie spielen eine wichtige Rolle bei der Verarbeitungsgeschwindigkeit. Ineffizente Algorithmen können unnötig viele Ressourcen

hinsichtlich Speicherverbrauch oder auch CPU-Zyklen beanspruchen und so den Durchsatz enorm begrenzen. LFA 1.0 weist in dieser Hinsicht sowohl in Laufzeit- als auch in Speicherperformance einige Schwachstellen auf.

Eine Logfileanalyse eines kompletten Versuchsdurchgangs bewegt sich vom Zeitbedarf her im niedrigen zweistelligen Stundenbereich. Das ist zwar deutlich besser als die Situation ohne Analysetool (die händische Auswertung nahm in der Regel mehrere Tage in Anspruch) lässt allerdings Raum für Optimierungen offen.

Beispielsweise werden alle Features von LFA 1.0 nacheinander abgearbeitet. Da alle LFA-Funktionen unabhängig voneinander arbeiten, also nicht aufeinander aufbauen, ist das eigentlich nicht notwendig. Eine Verteilung der Rechenlast auf alle verfügbaren Kerne wäre hier ratsam. Weiterhin nimmt das XML-Parsing viel Zeit in Anspruch. Das Aufbauen des Dokumentenbaumes verbraucht Zeit, die Suche in diesem Baum ebenfalls. Eine andere Parsing-Strategie wäre in Betracht zu ziehen. Darüber hinaus nimmt das Parsen nicht nur viel Zeit in Anspruch, sondern ist auch Ursache für einen hohen Speicherbedarf der Anwendung. Der Parser baut die Dokumentenstruktur des Logfiles im Speicher nach und verbraucht dabei ein Vielfaches des eigentlichen Logfiles. Jeder Knoten des XML-Files wird dabei als Objekt mit entsprechenden Attributen angelegt, was bei der Größe der Logfiles schnell zu tausenden Objekten führt.

Die Folge war anfangs, dass beim Einlesen der Logfiles regelmäßig der der JVM zur Verfügung stehende Speicher überschritten wurde und das Programm abbrach. Die einzige Lösung war hier, den von Anfang an verfügbaren Arbeitsspeicher der JVM zu erhöhen. Dazu musste man das Programm mit besonderen Argumenten starten. Der Befehl dafür wurde in einer Batchdatei abgelegt, sodass sich bei Benutzung der Software niemand um diese Besonderheit kümmern musste. Der Nachteil dieser Lösung ist, dass die Hardware immer über ein entsprechendes Maß an Arbeitsspeicher verfügen muss.

Die Betrachtungen von LFA 1.0 geben letztendlich zu erkennen, dass die schlechte Wartbarkeit und die Performanceschwächen allein schon eine Überarbeitung der Software erfordern. Hinzu kommen noch neue Anforderungen, die sich aus dem natürlichen Projektverlauf von ATEO ergeben haben.

3

ANFORDERUNGSERMITTLUNG

Die Anforderungsermittlung ist ein wichtiger Prozess, um Erkenntnisse über die Funktionalität eines zu entwickelnden Softwaresystems zu gewinnen. In diesem Prozess werden alle wichtigen Eigenschaften ermittelt, die dann die Spezifikation der Software bilden.
Der konkrete Entwurf einer Software leitet sich direkt aus diesen Eigenschaften ab. Sie ist somit der Ausgangspunkt für alle weiteren Schritte im Softwareentwicklungsprozess. Damit eine Software als korrekt im Sinne der geforderten Aufgabe angesehen werden kann, müssen alle an sie gestellten Forderungen erfüllt sein.[18]

Zur Bestimmung der Anforderungen eines Systems können alle Informationen herangezogen werden, die dieses System auf irgendeine Art beschreiben. Diese können aus Dokumentationsmaterialien oder Spezifikationen ähnlicher Systeme stammen, zum Beispiel ältere Versionen einer Software. Sofern Quellcode verfügbar ist, lassen sich aus einer Codeinspizierung und der Analyse der einzelnen Komponenten ebenfalls Erkenntnisse gewinnen. Besonders hilfreich sind persönliche Gespräche mit den Auftraggebern und späteren Benutzern einer Software.

Es lassen sich prinzipiell zwei Arten von Anforderungen unterscheiden - funktionale und nichtfunktionale Anforderungen.[18]
Unter funktionale Anforderungen versteht man Dienstleistungen, die ein System anbieten soll, sozusagen konkrete Aufgaben die eine Software zu absolvieren hat.
Nichtfunktionale Anforderungen dagegen beziehen sich nicht auf konkrete Aufgaben oder Funktionen, sondern auf Eigenschaften die ein Softwaresystem in seiner Gesamtheit betreffen. Sie stecken den Rahmen dafür, auf welche Art und Weise eine Software ihre Aufgabe erfüllen soll. Unter solche Eigenschaften fallen zum Beispiel die Zuverlässigkeit, der Speicherbedarf oder das Zeitverhalten eines Systems. Es lassen sich hier drei Kategorien von nichtfunktionalen Anforderungen ausmachen:

Produkanforderungen

: Hier runter fallen alle Eigenschaften die das Verhalten des Systems ausmachen, beispielsweise wie schnell eine bestimmte Aufgabe erledigt wird. Ebenso fällt in diese Kategorie die Verständlichkeit oder Benutzbarkeit eines Systems,

also wie gut ein Benutzer eine Software benutzen kann ohne beispielsweise lange Einführungen oder ein Training erhalten zu haben.

Untenehmensanforderungen

Unternehmensanforderungen betreffen alle Vorschriften die an die Entwicklung und den Entwicklungsprozess einer Software geknüpft sind. Sie kommen in der Regel von der Partei, die die Software nutzen wird und den Auftrag zu seiner Entwicklung gegeben hat. Beispiele für solche Vorschriften wäre etwa die Verwendung einer spezifischen Programmiersprache oder ein bestimmtes Vorgehensmodell bei der Entwicklung.

Externe Anforderungen

Hierbei geht es um Bedingungen, die weder von Auftraggeberseite noch von den Entwicklern kommen. Das betrifft zum Beispiel die Einhaltung von Gesetzen, die im Einsatzgebiet der Software gelten oder aber auch das Befolgen von Standards.

Die Anforderungen für die neue Software zur Logfileanalyse ergeben sich aus drei verschiedenen Einflussfaktoren:

- Bisherige Anforderungen aus LFA 1.0
- Neue funktionale Anforderungen
- Designänderungen

Diese Einflussfaktoren werden jetzt im einzelnen näher betrachtet.

3.1 BISHERIGE ANFORDERUNGEN

Die Funktionalität von LFA 1.0 soll nahezu vollständig in die neue Version der Logfileanalyse übernommen werden. Lediglich die Konvertierung der XML-Logfiles in Exceldateien wurde dem Featureset entnommen und stellt jetzt keine Anforderung mehr dar. Sie wurde zum damaligen Zeitpunkt definiert um eine händische Kontrolle zu ermöglichen. Die einzelnen Funktionen von LFA 1.0 werden an dieser Stelle noch einmal kurz erwähnt jedoch nicht weiter erläutert. Sie sind in der entsprechenden Studienarbeit hinreichend spezifiziert und erläutert worden.[8]

- Parsen von XML-Logfiles
- Output als Exceldatei
- Definieren verschiedener Analyseintervalle

- Berechnung des Flächenfehlermaßes
- Berechnung der Fahrzeit
- Summenbildung verschiedener Variablen: Summe der Joystickinputs; Summe der Sensoren, die sich nicht mehr über der Fahrbahn befinden; Anzahl der Kollisionen mit Hindernissen

Alle Berechnungsfunktionen müssen hierbei anwendbar sein auf vorher definierte Analyseintervalle, von den Psychologen Messfenster genannt. Diese Messfenster sind Ausschnitte einer Versuchsfahrt. Ein Messfenster kann sich natürlich auch über die gesamte Länge einer Fahrt erstrecken, muss es jedoch nicht.

3.2 NEUE FUNKTIONALE ANFORDERUNGEN

Zum schon bestehenden Featureset aus LFA 1.0 kommen jetzt Funktionalitäten hinzu, die sich aus dem Verlauf des ATEO-Projekts ergeben haben. Sie haben sich im wesentlichen aus der Entwicklung des Operateursarbeitsplatz herausgebildet. Da der Oerateursarbeitsplatz einem agilem Entwicklungsprozess unterlag haben sich bis zu seiner endgültigen Fassung immer wieder neue Anforderungen ergeben beziehungsweise verändert. Das endgültige Featureset wird hier nun kurz definiert und erläutert.

Parsen von CSV-Logfiles

: Das Logfile hat sich im Verlauf des Projekts in seiner Struktur mehrfach geändert. Es wurden neue Variablen definiert die im Zusammenhang mit dem Operateursarbeitsplatz stehen. Dies hätte sich kaum auf die Verwendung des XML-Parsers ausgewirkt. Es wären nur minimale Modifikationen notwendig gewesen. Mit fortschreitender Entwicklung des Operateursarbeitsplatzes kam es jedoch zu einer kompletten Überarbeitung des Loggingkomponente von SAM. Es wurde die Entscheidung getroffen für den Export der Versuchsdaten CSV-Dateien zu verwenden. Für die Logfileanalyse bedeutet das die Entwicklung eines neuen Parsermoduls. In der Endversion muss die Analysesoftweare beide Logfilevarianten verarbeiten können, da es unterschiedliche Untersuchungsreihen gab die jeweils eins der beiden Dateiformate verwendeten.

Graphische Darstellung der Eingriffe des Operateurs

: Der Operateur hat während der Fahrt der MWB die Möglichkeit so genannte harte Eingriffe vorzunehmen. Solche Eingriffe sind zum Besipiel das Begrenzen der Höchstgeschwindigkeit oder das Begrenzen der Joystickinpputs. Diese Eingriffe sollen in einem Diagramme visualisiert und

als PNG-Datei exportiert werden. Der Versuchsleiter hat so die Möglichkeit, interessante Logfiles beziehungsweise interessante Stellen darin schnell auszumachen, noch bevor das Excel-Outputfile genau betrachtet wird.

Arithmetische Funktionen

Durch die zusätzlichen OA-Variablen ergab sich die Notwendigkeit, neue Berechnungen durchführen zu können. Zwar gab es das Feature der Summenbildung schon in LFA 1.0 und das auch unabhängig vom Variablennamen, allerdings war dies immer nur auf jeweils eine Variable bezogen möglich. Für die Analyse des OA war es erforderlich Summen und Mittelwerte sowohl über einzelne als auch mehrere Variablen bilden zu können. Die Klickhäufigkeit des Operateurs sei hier an erster Stelle genannt, da sie von großem Interesse ist. Diese kann man unter verschiedenen Aspekten betrachten, beispielsweise wie viele Elemente wurden vom OA insgesamt geklickt, welche Elemente wurden wie oft benutzt oder wie oft hat der OA ins Leere geklickt (so genannte Blindclicks).

Reaktionszeit

Ein wichtiges Kriterium zur Ermittlung der Leistung eines Operateurs ist das Bestimmen von Zeiten bei weichen Operateurseingriffen. Weiche Eingriffe sind hier als Hinweise auf den Streckenverlauf zu verstehen. Der Operateur hat dafür eine Reihe von Elementen zur Verfügung die den MWB entweder als Einblendung präsentiert werden oder als Ansage über Kopfhörer den MWB zugänglich gemacht werden. Beispiele solcher Hinweise sind schneller fahren oder langsamer fahren oder auch Achtung Hindernis.
Es ist wichtig zu bestimmen ob ein Hinweis (beispielsweise ein Hinweis auf ein bald erscheinendes Hindernis) die MWB rechtzeitig erreicht oder eben nicht. Dem Logfile kann man diese Information nicht direkt entnehmen, da es nur eine Aktion als solche protokolliert und nicht die Intention die dahinter steht und somit nicht klar ist ob die MWB einen Hinweis registriert haben.

Seitenwahl bei Gabelung

Eine wichtige Frage ist wie schnell sich die MWB bei einer Weggabelung auf einen Abzweig einigen können. Ein schnelles Einigen kann sich deutlich auf die Fehlermaße auswirken. Die Wahl der jeweiligen Abzweige einer Gabelung wird hier ermittelt und ein Gütekriterium berechnet, welches die Schnelligkeit der Entscheidungsfindung widerspiegelt.

3.3 DESIGNVERÄNDERUNGEN

Die hier beschriebenen Anforderungen beziehen sich auf notwendige Veränderungen der Softwarearchitektur und fallen alle in den Bereich der nichtfunktionalen Anforderungen. Sie folgen direkt aus den in Kapitel 2 aufgezeigten Schwachstellen und sind nicht alleine durch geringfügige Veränderungen im Quellcode zu bewerkstelligen.

Stärkere Modularisierung

: Die neue Software soll im Vergleich zu LFA 1.0 in mehr Module aufgeteilt sein, die dann dem Separation of Concern Prinzip entsprechen, eine starke Kohäsion aufweisen und möglichst lose gekoppelt sind. So wird es in Zukunft einfacher sein einzelne Komponenten auszutauschen oder neue hinzuzufügen. Des weiteren sind die Module dann verständlicher und leichter modifizierbar.

Zeitbedarf verringern

: Die Bearbeitung ganzer Versuchsreihen erfordert in LFA 1.0 einen Zeitraum im zweistelligen Stundenbereich. Da durch die zusätzlichen Features des OA der Aufwand einer Analyse steigt, muss eine Lösung mit besserem Zeitverhalten entworfen werden. Der Einsatz paralleler Konzepte unter Ausnutzung aller verfügbaren Prozessorenkerne ist hierfür massgebend.

Speicherverbrauch verringern

: Eine Verringerung des Speicherverbrauchs ist wünschenswert, um die Software auch auf Systemen einsetzen zu können, die mit vergleichsweise geringer Hardware ausgestattet sind. Da der hohe Speicherbedarf von LFA 1.0 im wesentlich auf die Parserkomponente zurückzuführen war, sollte der XML-Parser in der neuen Version überarbeitet werden.

Verwendung von Java

: Es war vorgegeben die Software in Java zu implementieren. Dies ist in LFA 1.0 schon so geschehen und wird auch so beibehalten. Es handelt sich somit zwar nicht um eine notwendige Designveränderung, fällt aber in den Bereich der nichtfunktionalen Anforderungen und soll hier der Vollständigkeit halber erwähnt werden.

Abschließend werden alle Anforderungen tabellarisch zusammengefasst. Sie bilden das komplette Featureset der aktuellen Software zur Logfileanalyse und sind somit Grundlage des Softwareentwurfs im folgenden Kapitel.

Parsen von xml-Files	SAM-Basisfeature (LFA 1.0)
Output als xls-datei	SAM-Basisfeature (LFA 1.0)
Definieren verschiedener Analyseintervalle	SAM-Basisfeature (LFA 1.0)
Berechnung des Flächenfehlers	SAM-Basisfeature (LFA 1.0)
Berechnung der Fahrzeit	SAM-Basisfeature (LFA 1.0)
Summenbildung verschiedener Variablen	SAM-Basisfeature (LFA 1.0)
Parsen von csv-Logfiles	OA - Anforderung
Grafiken von harten Operatuerseingriffen	OA - Anforderung
Summen und Mittelwerte des Operatuers	OA - Anforderung
Reaktionszeit der MWB	OA - Anforderung
Gabelwahl der MWB	OA - Anforderung
Modularisierung	Designveränderung
Zeitverhalten verbessern	Designveränderung
Speicherbedarf verringern	Designveränderung
Java	allgemeine Anforderung

Tabelle 1: Anforderungen

4 ENTWURF UND DESIGN

In diesem Kapitel wird der Entwurfsprozess für die Softwarearchitektur der Logfileanalyse beschrieben. Ein Überblick über die einzelnen Schritte des Vorgehens und die Erläuterung dieser Schritte führen zu einer groben Skizze der Architektur der Software.

4.1 ARCHITEKTUR

Die Architektur einer Software beschreibt den groben Rahmen eines Systems und stellt damit dessen Grundgerüst dar. Die wesentlichen Komponenten werden definiert und deren Zusammenhänge beschrieben.[18] Programmiertechnische Aspekte spielen hier erst mal keine Rolle und werden erst später bei der Detailbeschreibung des Designs betrachtet.[1]

Folgende Techniken des Softwareengineerings spielen für den Entwurf einer Softwarearchitektur eine Rolle:

Forward Engineering

> ... beschreibt den Prozess der Systementwicklung, bei dem von den abstrakten und implementationsunabhängigen Anforderungen ausgehend ein Entwurf erstellt wird, welcher in einer konkreten Implementation umgesetzt wird.[2]

Reverse Engineering

> Beim Reverse Engineering werden die Komponenten eines Systems und ihre Beziehungen untereinander analysiert und Darstellungen des Systems in einer anderen Form oder auf einem höheren Abstraktionsniveau erzeugt. Änderungen am System sind nicht Bestandteil des Reverse Engineering.[2]

Restrukturierung

> Unter Restrukturierung versteht man die Transformation der Darstellungsform unter Beibehaltung des Abstraktionsniveaus und der Funktionalität. Das Ziel ist hierbei eine Verbesserung der Softwarequalität. Beispielsweise kann unstrukturierter Code geordnet und gegliedert werden, was eine bessere Erweiterbarkeit, Wiederverwendbarkeit und Wartbarkeit zur Folge hat.[2]

Refactoring

> Unter Refactoring versteht man ebenfalls die Veränderung eines Softwaresystems mit dem Ziel der Verbesserung der internen Struktur. Das äußerliche Verhalten wird nicht verändert. Im Unterschied zur Restrukturierung zeichnet sich Refactoring durch eine inkrementelle Vorgehensweise aus, bei welcher das System in einem lauffähigen Zustand belassen wird. Gewissermaßen ist Refactoring eine spezielle Technik des Restrukturierens.[4]

Reengineering

> ... bezeichnet die Untersuchung und Modifikation eines Systems zur Verbesserung der Softwarequalität beziehungsweise zur Eliminierung von Schwachstellen. Des weiteren ist es ein Ziel des Reegineerings die Umsetzung neuer Anforderungen (Erweiterung der Funktionalität) zu ermöglichen. Dazu verwendet das Reengineering Methoden des Reverse Engineerings, der Restrukturierung, des Refactorings und des Forward Engineerings.

Abbildung 3 stellt die Zusammenhänge zwischen Reengineering, Reverse Engineerings, Restrukturierung und Forward Engineering im Software-Entwicklungsprozess grafisch dar.

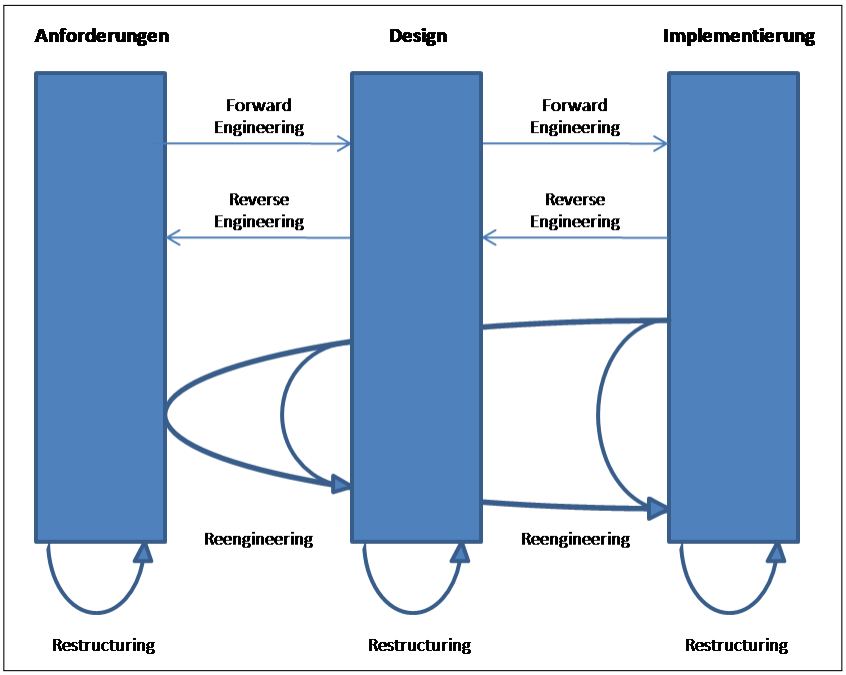


Abbildung 3: Zusammenhänge von Forward Engineering, Reverse Engineerings, Restrukturierung und Reengineering (nach Chikofsky und Cross 1990)

Das Vorgehen für den Architekturentwurf kombiniert die besprochenen Techniken des Softwareengineerings. Als erstes erfolgte

in einem Prozess des Reverse Engeneerings eine Analyse und Auswertung der bestehenden Software. Die Ergebnisse dieses Prozesses wurden in Kapitel 2 dargelegt. Im anschließenden Schritt wurden nach traditionellem Forward-Engineering-Ansatz alle funktionalen und nichtfunktionalen Anforderungen erhoben. Die nichtfunktionalen Anforderungen ergeben sich dabei aus den in Kapitel 2 beschriebenen Ergebnissen des Reverse-Engineering-Prozesses. Die funktionalen Anforderungen wurden durch die am ATEO-Projekt involvierten Psychologen definiert. Kapitel 3 liefert eine vollständige Beschreibung aller Anforderungen. Im dritten und letzten Schritt erfolgt nun die Synthese der Ergebnisse des Reverse Engineerings und des Forward Engineering, was eine umfassende Restrukturierung nach sich zog.

Grundsätzlich orientiert sich die Architektur weiterhin am Konzept des Model-View-Controllers. Das Prinzip des MVC besteht in der Entkopplung der Komponenten Model, View und Controller, wodurch eine hohe Flexibilität der einzelnen Module erreicht wird. Durch diese Entkopplung ist es mögliche einzelne Komponenten mit geringem Aufwand zu modifizieren oder auszutauschen. Der neue Architekturentwurf entspricht wie schon beschrieben dem MVC-Prinzip, bricht aber die vorhandene Struktur großer Klassen auf. Stattdessen werden alle Funktionalitäten in vielen kleinen Klassen realisiert, die in 5 Paketen organisiert sind.

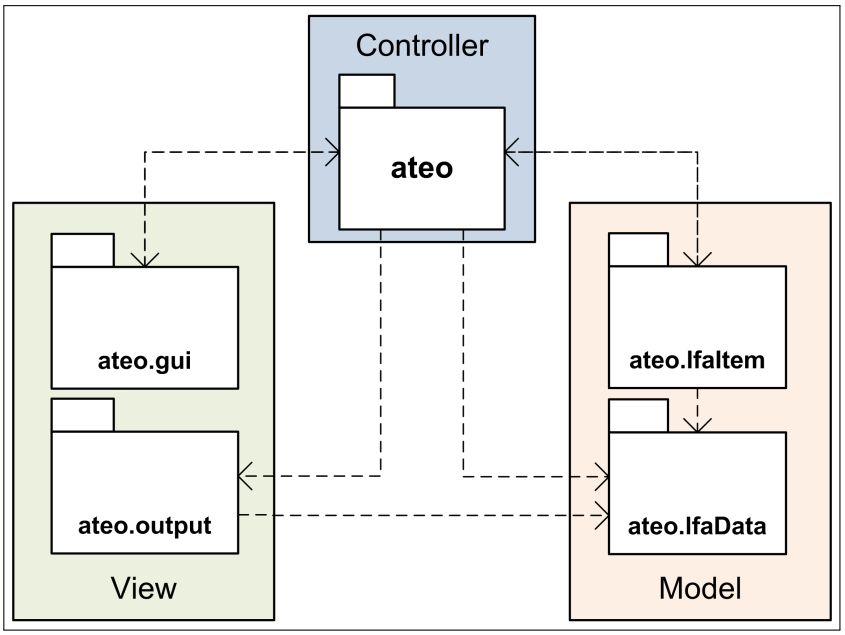


Abbildung 4: Architektur von LFA 1.8

Abbildung 4 zeigt die verschiedenen Pakete, deren Abhängigkeiten voneinander und deren Rolle im Konzept eines Model-View-

Controllers.

4.2 DESIGN

Während es bei der Architektur einer Software um ein allgemeines Grundkonzept geht, beschreibt das Design die Umsetzung der Anforderungen durch die Software.[13] Die einzelnen Teile einer Architektur werden hierfür im Detail beschrieben.

Package ateo:

Diese Paket beinhaltet alle Klassen, die für die Realisation eines Controllermoduls notwendig sind. Dazu gehören eine Klasse für die konkrete Ablaufsteuerung einer Analyse sowie verschiedene Klassen zum Parsen von Logfiles (CSV- und XML-Parser). Weiterhin gibt es eine Klasse, die alle Konfigurationsparameter bereithält (Dateinamen, Verzeichnisstrukturen von benötigten Ressourcen, Analyseintervalle) und verschiedene Hilfsklassen. Abbildung 5zeigt die Organisation der wichtigsten Klassen im Controllermodul und deren Abhängigkeiten.

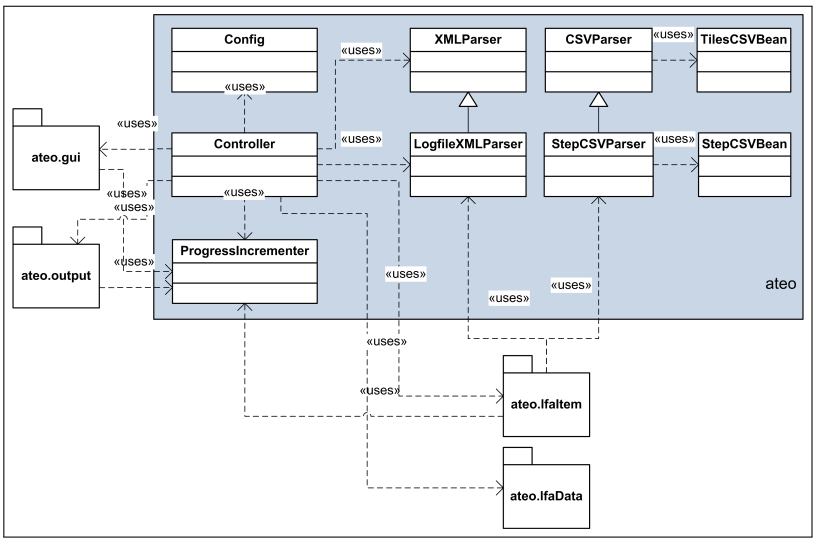


Abbildung 5: Controller

Package ateo.lfaitem:

Dieses Paket enthält die Programmlogik für die einzelnen Funktionen zur Logfileanalyse. Während in LFA 1.0 noch alle Features in einer einzigen Klasse subsumiert wurden, erhält hier nun jedes Feature seine eigene Klasse. Basis für alle Features ist die Klasse *LFAItem*. Es handelt sich hierbei um eine abstrakte Klasse, die keine konkrete Funktionalität bietet. Lediglich die Initialisierung wird hier realisiert, da sie allen Features gemein ist. Um ein konkretes Feature umzusetzen muss eine Klasse von *LFAItem* abgeleitet werden. Die Organisation der Klassen in dieser Form ermöglicht ein einfaches Verändern, Neuimplementieren oder Hinzufügen von Features.

Darüber hinaus ist es möglich eine parallele Verarbeitung der einzelnen Features zu realisieren und somit die ganze Kapazität eines Systems auszunutzen. Alle Klassen sind in ihrer Funktionalität nämlich vollkommen autonom und unabhängig von den Ergebnissen anderer Klassen. Basis der parallelen Verarbeitung ist die Verwendung der Threadingtechnologie. Jede Klasse erbt die Eigenschaften der Klasse *Thread* und kann vom Controller als Workerthread aufgerufen werden. Alle Klassen erzeugen nach Beendigung ihrer Berechnungen ein *LFASingleItemData*-Objekt, in dem die Ergebnisse gespeichert werden. Abbildung 4 zeigt alle Klassen die Features implementieren und deren Abhängigkeit zu ihrer Basisklasse.

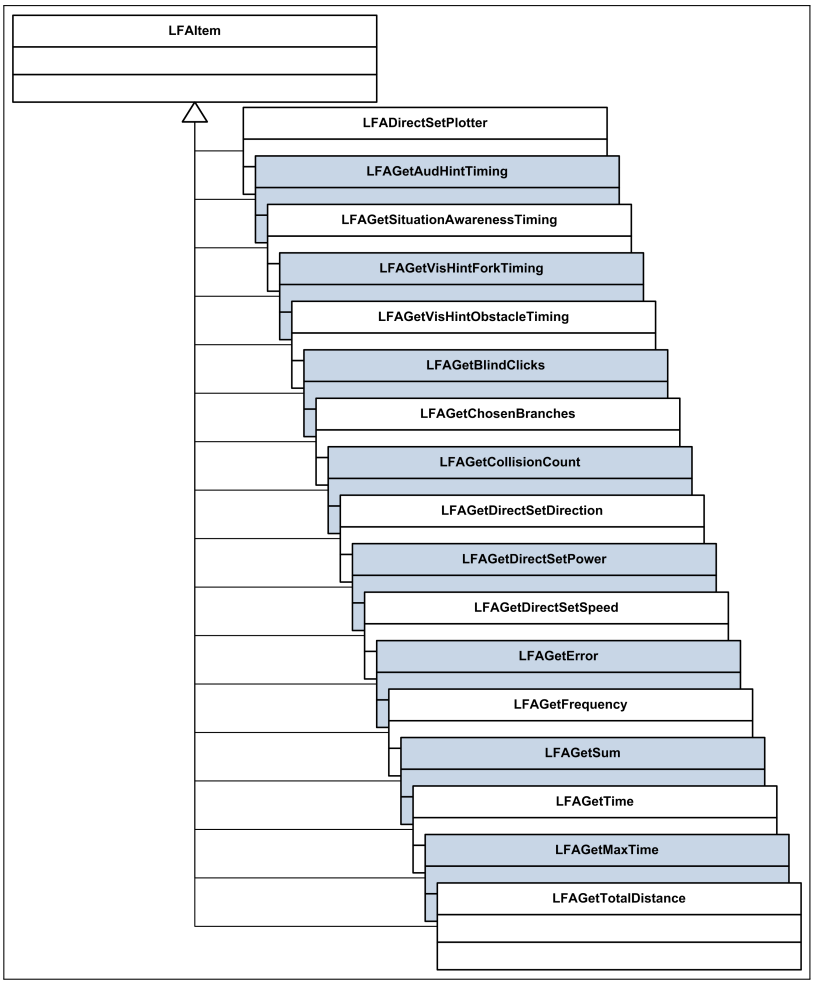


Abbildung 6: Übersicht Workerthreads

Package ateo.lfadata:

Da bei einer nebenläufigen Datenverarbeitungsstrategie wie dem Threading die Reihenfolge von eintreffenden Ergebnissen nicht im Vorfeld bestimmt werden kann, benötigt man zum Speichern der Ergebnisse eine flexible Datenstruktur. Arrays oder Listen sind hierfür schlecht geeignet da man auf deren Einträge über einen numerischen Index zugreift. Der Index für ein bestimmtes Ergebnis wäre aber bei jedem Durchlauf einer Analyse ein anderer, was den Zugriff für die Outputroutinen zur Visualisierung der Ergebnisse erschwert. Die Ergebnisse werden deshalb in einem Key-Value-Store abgelegt, da eine solche Datenstruktur erlaubt, nichtnumerische Schlüssel zur Adressierung enthaltener Elemente zu verwenden. In Java wird eine solche Datenstruktur als Hashtable oder Hasmap bezeichnet. Andere Programmiersprachen verwenden auch die Begriffe Dictionary oder assoziative Arrays. Das Paket **ateo.lfadata** enthält mehrere Klassen zur Rea-

lisierung dieser Datenstruktur.

Die kleinste Einheit bildet die Klasse *LFASingleItemData*. Es handelt sich hierbei um abgeleitete Hashmap, die Strings als Keys und Java-Objects als Values verwendet. Sie kann somit beliebige Datentypen aufnehmen und ist der Universalcontainer für alle durch *LFAItem*-Klassen erzeugten Ergebnisse.

Die nächsthöhere Ebene realisiert die Klasse *LFAStepData*. Die *LFASingleItemData*-Objekte werden hier in einer abgeleiteten Hashmap zusammengefasst. Als Keys werden wieder Strings verwendet und Values sind in dieser Ebene ausschließlich *LFASingleItemData*-Objekte. Zum besseren Verständnis sei hier noch mal erwähnt, dass die MWB während eines Versuchs mehrere Fahrten absolvieren müssen. Diese Fahrten sind nummerisch kodiert und werden historisch bedingt als **Step** bezeichnet, woraus sich der Name der Klasse ableitet. Es werden in dieser Ebene nämlich nicht alle *LFASingleItemData*-Objekte abgespeichert, sondern in jeweils ein *LFAStepData*-Objekt alle *LFASingleItemData*-Objekte, die zur gleichen Fahrt und zum gleichen Analseintervall gehören. Von Seiten der Psychologen werden anstelle des Begriffs Analyseintervall auch die Begriffe Messfenster oder Window verwendet.

Die oberste Ebene der Speicherhierarchie bildet die Klasse *LFAData* und hat als Basis wieder die Klasse *Hashmap*. Sollte eine Analyse mehrere Fahrten oder Messfenster betreffen, gibt es hier für jede Kombination von Fahrt und Messfenster einen entsprechenden Eintrag. Aus diesen drei Ebenen ergibt sich nun eine Kaskadenstruktur von Hashmaps die in Abbildung 5 dargestellt wird.

Jedes Item in jedem Fenster wird hierbei mit einem eindeutigen Key verbunden und ist so später wieder addressierbar.

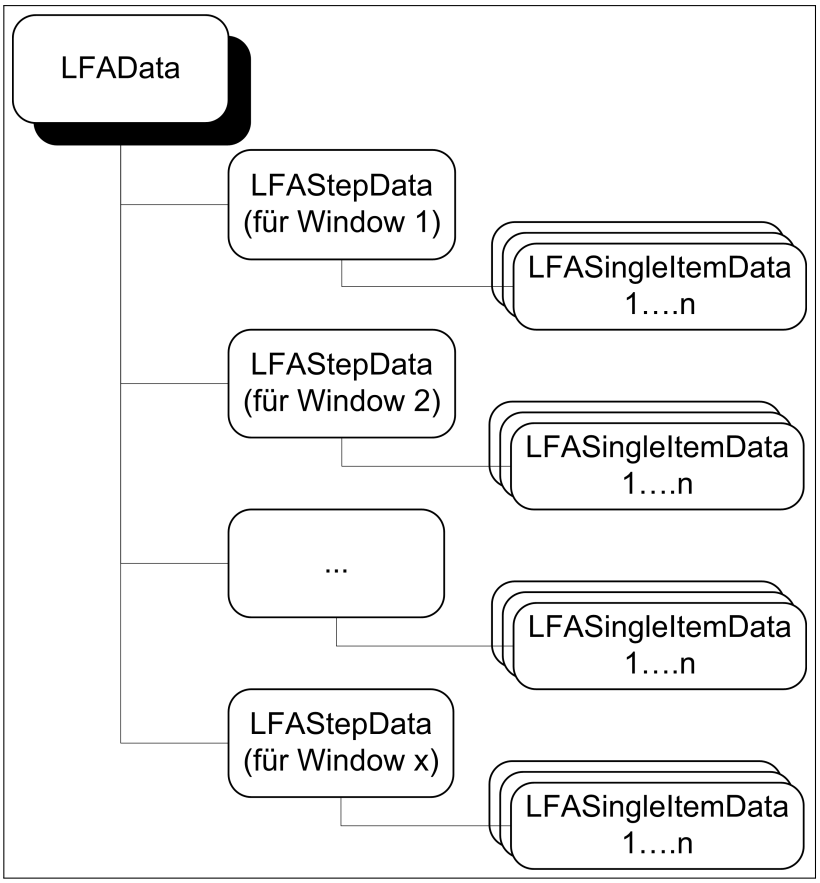


Abbildung 7: Speicherstruktur für LFA-Ergebnisse

Package ateo.gui:

Diese Paket besteht aus vier Klassen. Die Klasse *MainWindow* bietet eine Übersicht über zu analysierende Logfiles und zu verwendende Messfenster und eine Fortschrittsanzeige für den Status einer Analyse. Die Klassen *NewSlidingWindowDialog*, *FinishedDialog*, *ErrorDialog* bieten einen Dialog zum Konfigurieren von Messfenstern, einen Dialog für eine erfolgreiche Beendigung einer Analyse und einen Dialog für den Abbruch der Analyse aufgrund eines Fehlers.

Package ateo.output:

Diese Paket besteht nur aus der Klasse *Outputter*. Nachdem alle Workerthreads ihre Berechnungen geleistet haben, generiert diese Klasse eine Exceldatei, in der alle Ergebnisse zusammen getragen werden. Unterschiedliche Aspekte der Analyse (SAM-Basisdaten, Clickhäufigkeit des Operateurs, usw.) erhalten ein eigenes Sheet in dieser Datei, welches wiederum in einer separaten Methode in der Klasse implementiert wurde. *Outputter* greift auf *LFAData* direkt zu und bereitet die Ergebnisse tabellarisch auf.

5 TECHNISCHE GRUNDLAGEN

5.1 VERSUCHSUMGEBUNG VON ATEO

Der Versuchsaufbau der ATEO-Testumgebung besteht aus 4 verschiedenen Komponenten, die hier im einzelnen vorgestellt und kurz erklärt werden sollen.

1. SAM: ist eine Testumgebung bei deren Untersuchungen zwei MWB ein Fahrobjekt einen Hindernisparkour entlang steuern müssen. Jeder MWB hat zur Steuerung des Fahrobjekts einen Joystick zu seiner Verfügung. Die Eingabewerte der Joysticks werden von der Software miteinander verrechnet, so dass die MWB das Fahrzeug gemeinsam steuern. Die MWB sind angehalten während des Versuchs nicht miteinander zu sprechen, da sie mit unterschiedlichen Zielstellungen instruiert wurden. Ein MWB hat die Aufgabe die Fahrt besonders schnell zu absolvieren, der Andere soll hingegen möglichst genau fahren. Während der Fahrt stoßen die MWB immer wieder auf verschiedene Arten von Hindernissen und Weggabelungen.

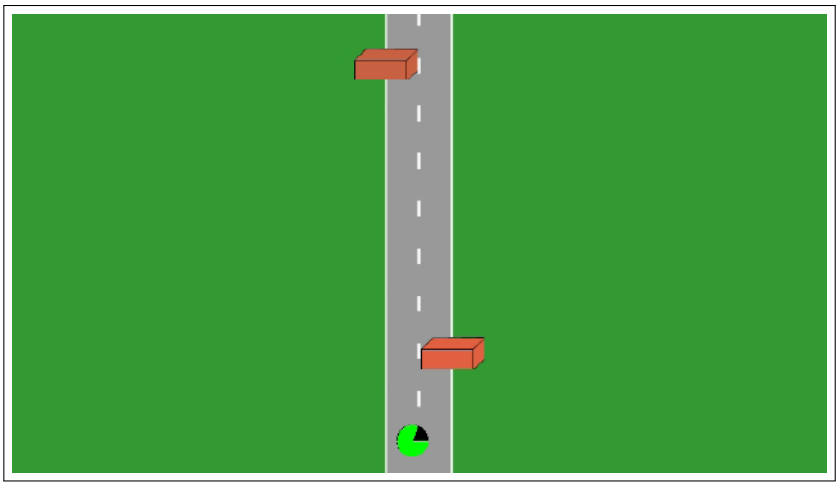


Abbildung 8: Fahrobjekt vor zwei statischen Hindernissen

In Abbildung 8 kann man das Fahrobjekt kurz vor zwei statischen Hindernissen sehen. Da beide Hindernisse die Fahrbahn zur Hälfte überdecken, sind die MWB gezwungen eine kleine S-Kurve zu fahren um diese Passage fehlerfrei zu meistern. Diese Hindernisskombination gibt es noch in einer Variante, bei der die beiden Hinternisse jeweils ein Viertel der Fahrspur bedecken. Weiterhin gibt es ein dy-

namisches Hindernis, das sobald es ins Sichtfeld der MWB gelangt von links nach rechts den Parkour kreuzt. Es ist so konzipiert, dass es mit dem Fahrobjekt kollidiert, sollten die MWB ihre aktuelle Geschwindigkeit beibehalten. Die MWB müssen also das Fahrobjekt beschleunigen oder abbremsen um eine Kollision zu vermeiden. Sollte es zu einer Kollision mit einem Hindernis kommen, so wird das Fahrobjekt von der Fahrspur entfernt und neben ihr positioniert. Darüber hinaus werden die MWB mit einer Zeitstrafe von 3 Sekunden belegt. Erst danach können sie weiterfahren. Die MWB haben also trotz ihrer unterschiedlichen Instruktionen gleichermassen das Interesse Kollisionen zu vermeiden.

Bei den Weggabelungen gibt es zwei verschiedene Varianten. Beide haben jedoch gemein, dass sie immer eine breite Abzweigung und eine schmale Alternative anbieten. Die breite Abzweigung ist allerdings länger als die schmale Alternative. Die MWB können die Schwierigkeit der Gabelungen jederzeit einschätzen, da diese weit vor Eintritt in die Abzweigung vollständig erfassbar sind.

Die unterschiedlichen Instruktionen der MWB sollen Konfliktpotential schaffen, um die Komplexität des Systems zu erhöhen. Besonders in Entscheidungssituationen wie an den Hindernissen oder Gabelungen macht sich das bemerkbar. Eine Person die dieses System beobachtet und regelt (beispielsweise ein Operateur) sieht sich in eben diesen Situationen dann veranlasst, die Rolle des Beobachters zu verlassen und in das System einzugreifen.

2. OA: Der OA ist ein grafisches Benutzerinterface für einen Operateur. Er hat durch dieses Interface die Möglichkeit die MWB bei ihrer Aufgabe zu unterstützen und sie dadurch zu einem besseren Ergebnis zu führen. Den OA gibt es in drei Ausbaustufen. Mit jeder Ausbaustufe hat der Operateur mehr Eingriffsmöglichkeiten zu seiner Verfügung. Christian Leonhard behandelt in seiner Studienarbeit die Entwicklung dieses dreistufigen OA und baut dabei auf Arbeiten von Hermann Schwarz auf.[16][12]

   Jede dieser Ausbaustufen wurde in einer eigenen Versuchsreihe getestet und analysiert. Jens Nachtwei beschreibt in seiner Dissertation die Evaluierung dieses dreistufigen Designs.[14]

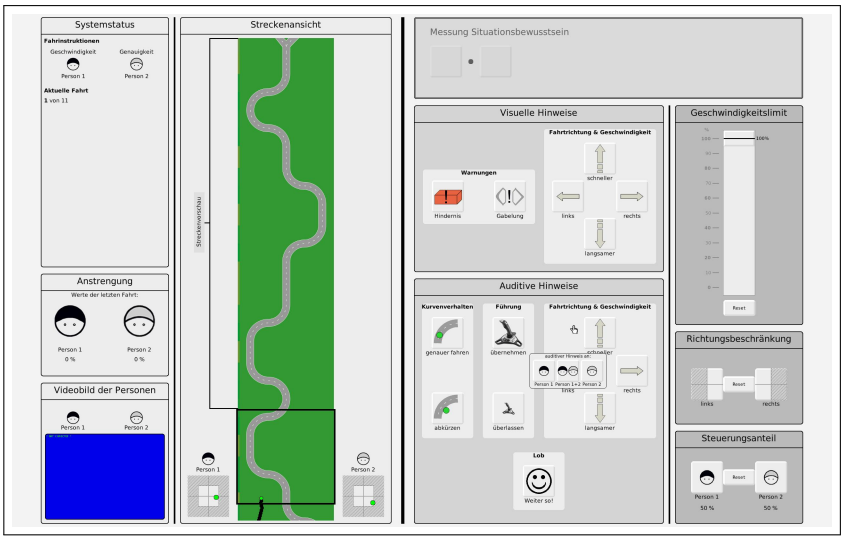


Abbildung 9: Operateursarbeitsplatz v2.8.4 (letzte Ausbaustufe)

Abbildung 9 zeigt den OA in seiner dritten und letzten Ausbaustufe. Man kann deutlich sehen, dass der OA in verschiedene Areale unterteilt ist. In der linken Hälfte befinden sich ausschließlich Elemente zur Beobachtung der MWB und zur Überwachung ihrer Aufgabe. Dazu gehören ein Videostream der MWB und eine Anzeige der individuellen Belastung der MWB. Diese können die MWB selbst auf einer Skala einstellen. Dem Operateur wird dann immer die Belastung der letzten Fahrt angezeigt. Darüber hinaus verfügt der OA über eine Streckenvorschau, die in etwa viermal so lang ist wie der Ausschnitt den die MWB zu sehen bekommen. Der Operateur kann dadurch viel eher sehen, wann es zu potentiell kritischen Situationen kommt.

Auf der rechten Seite des OA stehen dem Operateur eine Reihe von Eingriffsmöglichkeiten zur Verfügung. Zum einen verfügt er über sogenannte weiche Eingriffe. Dabei handelt es sich um auditive und visuelle Hinweise, die er den MWB auf Knopfdruck zukommen lassen kann. Das können Anweisungen zum schneller oder langsamer Fahren sein oder auch Ankündigungen über bald eintretende Ereignisse wie Gabelungen oder Hindernisse. Visuelle Hinweise werden dabei direkt auf das Display der MWB ausgegeben. Auditive Hinweise werden den MWB auf deren Kopfhörer gespielt. Dabei kann er noch entscheiden ob er nur einem oder beiden MWB diese auditiven Hinweise geben will.

Weiterhin verfügt der OA über sogenannte harte Eingriffe. Diese beeinflussen direkt das Fahrverhalten des Fahrobjekts. Beispielsweise kann er die maximale Fahrgeschwindigkeit

verändern oder das Eingabeverhältnis der Joysticks zugunsten eines der beiden MWB verlagern.

3. MW: Der MW beschreibt die empfundene mentale Anstrengung der MWB. Es handelt sich hierbei um eine von Jens Nachtwei und Saskia Kain angepasste Variante der RSME-Skala.[20] Die MWB können über ein von Christian Leonhard implementiertes grafisches Benutzerinterface ihren jeweiligen MW am Ende einer jeden Fahrt zum Ausdruck bringen. Beim Start der nächsten Fahrt sieht der Operateur diese Werte und kann entsprechend Eingriffe vornehmen. Er könnte zum Beispiel verstärkt Hinweise geben oder aber auch an bestimmten Stellen die Höchstgeschwindigkeit des Fahrzeug begrenzen.

4. AAF: Das ATEO Automatik Framework (AAF) ist eine Softwarebibliothek zur Unterstützung der MWB. Es liefert verschiedene Automatikfunktionen, die die gleiche Aufgabe wie der Operateur haben, nämlich die MWB bei ihrer Aufgabe zu unterstützen. Das AAF stellt jedoch andere Möglichkeiten zur Verfügung und arbeitet völlig autonom. Man kann über ein Konfigurationsmenü im Vorfeld eines Versuchs bestimmen, welche konkreten Funktionen eingesetzt werden sollen.

5.2 TECHNOLOGIEN

Für die beteiligten Psychologen ist es besonders wichtig auf Technologien zu setzen, die allgemein verbreitet sind. Auf diesem Wege wird die Einstiegshürde für zukünftige Projektmitglieder möglichst niedrig gehalten. Aus diesem Grunde werden in ATEO allgemein verbreitet Dateiformate verwendet. Weiterhin soll die für das Analysetool verwendete Programmiersprache Java sein. Dies wurde schon in LFA 1.0 berücksichtigt und wird auch in dieser Version fortgesetzt. Da die Logfileanalyse vollkommen unabhängig von den Versuchen mit SAM stattfindet, ist es nicht notwendig sich mit den Besonderheiten von SAM, OA und AAF im Detail auseinander zu setzen. Die Schnittstelle zwischen der Versuchsumgebung und der Logfileanalyse sind lediglich die im Logfile definierten Variablen.

Java

Die Java Technologie ist eine Sammlung von Werkzeugen, die es einem ermöglicht, Programme zu entwickeln die auf unterschiedlichen Computersystemen laufen können. Java wurde urspünglich von der Firma Sun Microsystems entwickelt, gehört aber seit der Firmenübernahme 2010 zur Oracle Corporation. Zur Java-Technologie gehören folgende Komponenten:

- die Programmiersprache Java
- Entwicklungswerkzeuge für Java Programme
- die Java Laufzeitumgebung

Java als Programmiersprache dient in erster Linie dem Formulieren von Programmen. Das Ziel bei der Entwicklung von Java war unter anderem eine einfache, objektorientierte Programmiersprache zu entwerfen, die zudem robust, architekturneutral und portabel ist. Um Java-Programme zu entwickeln, verwendet man verschiedene Werkzeuge, die in der Regel in einem Development Kit zusammengefasst sind. Üblicherweise wird hierfür das OpenJDK verwendet. Das OpenJDK ist die offizielle freie Implementierung der Java Platform, Standard Edition (Java SE). Neben der Laufzeitumgebung Java Runtime Environment (JRE) beinhaltet es einen Java-Compiler, ein Dokumentationswerkzeug, einen Archiver, ein Werkzeug zum Signieren von Anwendungen und Bibliotheken, und noch weitere Werkzeuge.

Java Programme werden nicht direkt ausgeführt sondern vom Java-Complier in Java-Bytecode übersetzt. Dieser Bytecode kann dann von einer Maschine ausgeführt werden, wobei es sich typischerweise um eine virtuelle Maschine

handelt, der JRE. Diese übersetzt dann den Bytecode in die jeweilige Maschinensprache der vorliegenden Hardware. Der Bytecode funktioniert somit nur als Zwischencode zwischen Programmiersprache und Maschinensprache. Dadurch wurde größtmögliche Plattformunabhängigkeit und Portabilität erreicht.

Die Portabilität findet dort seine Grenzen, wo es keine JRE für das Zielsystem gibt. Es gibt jedoch für sehr viele Systeme eine Laufzeitumgebung, wie zum Beispiel Microsoft Windows, Linux, Solaris, Mac OS X, AIX und viele eingebettete Systeme wie Mobiltelefone, PDAs und Smartcards. Durch die Verfügbarkeit von Laufzeitumgebungen für so viele Systeme hat Java seinen Einzug in nahezu alle Felder der Informationsverarbeitung geschafft und eine enorm große Entwicklerbasis gewonnen. Das spiegelt sich auch in der Verfügbarkeit von Frameworks und Bibliotheken wider. Es gibt praktisch für jeden erdenklichen Anwendungsfall Bibliotheken auf die man zurückgreifen kann, was nicht zuletzt auch der Grund dafür war, die Software für die Logfileanalyse in Java zu entwickeln.
Die nun folgenden Abschnitte gehen daher auf die für die Analyse relevanten Dateiformate ein und auf die für deren Verarbeitung verwendeten Java-Bibliotheken

XML

Extensible Markup Language (XML) (engl. für erweiterbare Auszeichnungssprache) ist eine Auszeichnungssprache für die Darstellung hierarchisch strukturierter Daten, die in Standard Textdateien abgespeichert werden. Die Verwendung solcher Textdateien machen die wesentliche Stärke von XML aus.
Da alle Plattformen und Systeme Textdateien verarbeiten können, ist mit Hilfe von XML ein plattform- und implementationsunabhängiger Austausch von strukturierten Daten zwischen Computersystemen möglich. Dies macht sich insbesondere in Netzwerken oder dem Internet bezahlt. Ein wichtiger Punkt ist, dass XML-Dokumente per Definition keine Binärdaten enthalten und damit menschenlesbar sind. Ein XML-Dokument besteht ausschließlich aus Textzeichen, im einfachsten Fall in ASCII-Kodierung.
Auf Basis von XML lassen sich spezielle anwendungsbezogene Sprachen definieren, indem man inhaltliche und strukturelle Restriktionen vornimmt, welche wiederum durch Schemasprachen wie Document Type Definition (DTD) oder XML-Schema ausgedrückt werden. Beispiele für XML-Sprachen sind: Rich Site Summary (RSS),

Mathematical Markup Language (MathML), Extensible HyperText Markup Language (XHTML) oder Scalable Vector Graphics (SVG). Im ATEO Projekt wurde lange Zeit das XML-Format zur Erstellung der SAM-Logfiles verwendet, die dann von der Logfileanalyse weiter verarbeitet wurden.

SAX

Simple API for XML (SAX) beschreibt eine Application Programming Interface (API) zum Parsen von XML-Daten. Die derzeit gültige Version 2.0 wurde wurde im Jahr 2000 von David Megginson veröffentlicht und ist Public Domain. Ursprünglich wurde SAX in Java entwickelt, mittlerweile gibt es aber Implementierungen in allen möglichen Programmiersprachen und hat sich als damit als Quasistandard etabliert.
Ein SAX-Parser liest XML-Daten als sequentiellen Datenstrom ein. Die Verarbeitung der Daten funktioniert auf Basis von Events. Für definierte Ereignisse werden vorgegebene Callback-Funktionen aufgerufen, die dann die eigentliche Datenverarbeitung leisten.

Ein SAX-Parser hat keine Kenntnis von der Struktur des Dokuments. Es gibt auch keine Zustände und einzelne Events stehen nicht in Verbindung zu anderen Events. Für die Datenverarbeitung ist es nicht notwendig, dass das Dokument vollständig eingelesen sein muss. Sie kann bereits mit dem Einlesen der ersten Zeilen beginnen. Zudem arbeiten SAX-Parser sehr speichereffizient, da nur die Elemente im Speicher belassen werden, für die es auch definierte Callback-Funktionen gibt. Eine Schwachstelle dieser Variante der XML-Auswertung ist, dass SAX-Ereignisse auf nicht wohlgeformte Dokumente möglich sind.

DOM

Das Document Object Model (DOM) ist eine vom World Wide Web Consortium (W3C) spezifizierte Schnittstelle für den Zugriff auf HTML- oder XML-Dokumente. Sie erlaubt die dynamische Veränderung des Inhalts, der Struktur und des Layout eines XML-Dokuments. Ein XML-Dokument wird von einem DOM-Parser komplett eingelesen und ein Dokument-Objekt erzeugt, das wie ein Stammbaum dargestellt wird. Anhand dieses Objekts kann dann mittels API-Methoden auf die Inhalte und die Struktur zugegriffen werden. Das betrifft die Navigation zwischen den einzelnen Knoten eines Dokuments, das Erzeugen, Verschieben und Löschen von Knoten sowie das Auslesen, Ändern und Löschen von Textinhalten. Alle Knoten stehen über Verwandtschafts-

beziehungen zueinander in Verbindung. Ausgehend von einem beliebigen Knoten ist jeder andere Knoten über diese Verwandtschaftsbeziehungen erreichbar.

JDOM

Bei JDOM handelt es sich Darstellung von XML-Dokumenten die speziell für Java entwickelt wurde. Wie beim DOM wird ein Dokument als Baum im Hauptspeicher repräsentiert. Allerdings werden für die einzelnen Elemente Java-Klassen verwendet. Es bietet prinzipiell die Möglichkeit den JDOM-Baum als Textdokument und somit als XML-Dokument auszugeben, was der SAX-Standard zum Beispiel nicht leisten kann.
JDOM wird in LFA verwendet um die SAM-Logfiles aus der frühen Phase des Projekts verarbeiten zu können und um aus den Grafiken mit den Ideallinien XML-Dateien mit den Koordinaten der Ideallinie zu erzeugen. Weiterhin gibt es zu jedem csv-Logfile eine config-XML-Datei, deren Daten verarbeitet werden müssen. Diese Daten sind statische Informationen über den jeweiligen Versuch wie zum Beispiel die Versuchs- oder die Probandennummer.

XPath

Die XML Path Language (XPath) ist eine vom W3C entwickelte Abfragesprache, um Teile eines XML-Dokumentes zu adressieren. Sie bildet die Grundlage für weitere Standards wie XSL Transformation (XSLT), XML Pointer Language (XPointer) und XML Query Language (XQuery). Die aktuelle Version ist XPath 2.0 und seit 2007 definiert. Ein XPath-Ausdruck adressiert Teile eines XML-Dokuments und operiert auf der logischen Dokumentenstruktur, die hierbei als Baum betrachtet wird.
XPath wird bei der Logfileanalyse verwendet um Daten schnell aus dem DOM (beziehungsweise JDOM) zu extrahieren. Ein manuelles Abtraversieren des Dokumentenbaumes ist im Vergleich zur Verwendung von XPath bedeutend langsamer.

Jaxen

Die Java XPath Engine (Jaxen) ist eine Open Source Java-Bibliothek für die Verwendung von XPath. Jaxen verwendet eine Apache-ähnliche Lizens und hat die aktuelle Versionsnummer 1.1.4. Die Jaxen-API wird bei der Logfileanalyse verwendet um XPath-Ausdrücke formulieren zu können und mit diesen dann den JDOM-Baum abzufragen.

CSV

Comma-Separated Values (CSV) beschreibt den Aufbau einer Textdatei zur Speicherung oder zum Austausch einfach strukturierter Daten. Es gibt keinen allgemeinen Standard, jedoch ist es im RFC 4180 grundlegend beschrieben.[17] In einer CSV -Datei wird eine Tabelle abgebildet beziehungsweise eine Liste unterschiedlich langer Listen. Kompliziertere, geschachtelte Datenstrukturen sind nicht umsetzbar oder müssten in verketteten CSV-Dateien gespeichert werden.

Bestimmte Zeichen haben Sonderfunktion zur Strukturierung der Daten, zum Beispiel zum Trennen von Datensätzen, Datenfeldern oder aber zum Maskieren von Sonderzeichen. Jeder Datensatz sollte laut RFC 4180 die gleiche Anzahl Spalten enthalten. Oft wird dies aber nicht eingehalten.

In den letzten Untersuchungen in ATEO wurden anstelle von XML-basierten Logfiles CSV -Dateien verwendet. Der Wechsel auf dieses Format war notwendig, da die Komplexität von SAM und dem OA im Laufe der Zeit immer stärker gewachsen ist. Das Logging im XML-Format hat dabei so viel Kapazität in Anspruch genommen, dass ein flüssiger Ablauf eines Versuchs nicht mehr gewährleistet werden konnte.

Der Wechsel auf CSV-Dateien war mit einigen Einschränkungen verbunden. Zum einen hat sich die Lesbarkeit der SAM-Logfiles deutlich verschlechtert, zum andern mussten statische Versuchsdaten in ein extra XML-File ausgelagert werden, was dem Prinzip dass ein Logfile alle relevanten Daten enthalten sollte ein wenig zuwider läuft. Dies wurde aber zugunsten eines reibungslosen Versuchsablaufs in Kauf genommen.
Es sei an dieser Stelle schon mal vorweg genommen, dass durch diesen Wechsel auch die Leistung der späteren Logfileanalyse erheblich gesteigert werden konnte, auch wenn das nicht das vordergründige Ziel dieser Maßnahme war.

Super CSV

SuperCSV ist eine Java-Bibliothek zum Verarbeiten von CSV -Dateien. Sie bietet automatisches Maskieren und Demaskieren von Sonderzeichen und ist weitgehend konfigurierbar. Das heißt man kann Trenn- und Sonderzeichen frei wählen und belegen oder auf vordefinierte Konfigurationen

zurückgreifen. SuperCSV steht unter der Apache License, Version 2.0 und hat die aktuelle Versionsnummer 2.0.0-beta-1. SuperCSV wird bei der Logfileanalyse verwendet um die neueren SAM-Logfiles und die Trackkonfigurationen verarbeiten zu können.

Excel

Excel ist ein wichtiges Werkzeug für die tägliche Arbeit der Psychologen im ATEO-Projekt. Die Ergebnisse der Logfileanalyse werden in Exceldateien strukturiert dargestellt und sind Grundlage für die weiterführende Analyse der Versuchsdaten.

Die älteren Versionen von Excel erstellten Dateien mit der Dateiendung '.xls'. Dieses Dateiformat basiert dem Binary Interchange File Format (BIFF), einem proprietären Binärformat von Microsoft. Im Jahr 2008 hat sich Microsoft dazu entschlossen eine Dokumentation der Dateistruktur ab Excel 97 zu veröffentlichen. Frühere ausführliche Analysen stammen aus der Open-Source-Gemeinde. Die neueren Excel Versionen (ab 2008) erstellen Dateien mit den Dateiendungen '.xlsx', das auf dem offenen OpenXML und damit auf XML basiert und eine Zip-Kompression nutzt.

JExcelApi

JExcelApi ist eine Java-Bibliothek zum Verarbeiten von XLS-Dateien dem alten Standardformat von Excel. Sie steht unter der GNU Library or Lesser General Public License version 2.0 (LGPLv2). Sie wird verwendet um den Output im alten XLS-Format zu generieren. Es wäre problemlos möglich gewesen, das neue auf OpenXML basierende XSLX-format zu verwenden und damit auf diese API zu verzichten. Es sind jedoch nicht immer die nötigen Mittel vorhanden um auf neuere Excelversionen zuzugreifen, sodass mitunter nur ältere Excelversionen vorhanden waren. Dies führte dann zu der Entscheidung die Ergebnisse der Logfileanalyse im alten XLS-Format abzuspeichern.

JFreeChart

JFreeChart ist ein Java-Framework für die Erstellung von Diagrammen. Es unterstützt verschiedenste Diagrammtypen sowie einen Export der Grafiken als PNG- oder JPG-Datei. Das JFreeChart Projekt wurde im Jahr 2000 von David Gilbert gestartet und ist mittlerweile eins der beliebtesten Java-Frameworks zur Diagrammerzeugung. JFreeChart wird in mehreren Open-Source- und kommerziellen Produkten verwendet, beispielsweise JBoss, StatCVS oder Net-

Beans. JFreeChart steht unter der GNU Lesser General Public Licence (LGPL) und hat die aktuelle Versionsnummer 1.0.13.
Bei der Logfileanalyse wird dieses Framework verwendet um die harten Eingriffe des Operateurs als Liniendiagramm darzustellen. Das gibt einem die Möglichkeit interessante Stellen auszumachen noch bevor man sich mit den konkreten Zahlenreihen in der Exceldatei auseinander setzt.

6 IMPLEMENTIERUNG

Dieses Kapitel geht auf die konkrete Implementierung der einzelnen Komponenten ein. Wie schon im Kapitel 4 besprochen wird grundsätzlich das Konzept des Model-View-Controllers beibehalten, wobei die einzelnen Komponenten stärker aufgegliedert werden und die Klassen in Paketen organisiert werden. Zu Beginn werden die Klassen die das GUI betreffen vorgestellt, anschließend der Controller. Dann folgt eine detaillierte Beschreibung der eigentlichen Analysekomponenten und der verwendeten Datenstruktur und abschließend der Output.

6.1 PACKAGE ATEO.GUI

Dieses Paket enthält ausschließlich Klassen die das GUI betreffen. Die enthaltenen Klassen heißen *MainWindow*, *NewSlidingWindowDialog*, *FinishedDialog* und *ErrorDialog*. Alle Klassen wurden unter Verwendung der Standard Widget Toolkit (SWT) - Bibliothek implementiert. SWT nutzt die nativen grafischen Elemente des Betriebssystems und ermöglicht somit die Erstellung von Software, die eine vergleichbare Optik aufweist wie native Programme. Die wesentliche Komponente und zugleich Einstiegspunkt der Software ist die Klasse *MainWindow*.

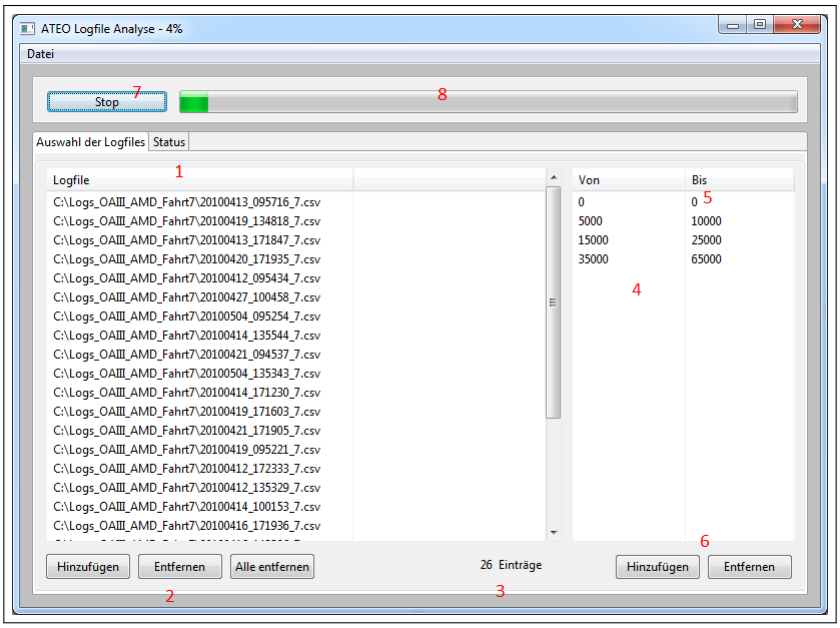


Abbildung 10: User Interface zur Logfileanalyse v1.8

In Abbildung 10 sieht man auf der linken Seite einen Bereich, in dem zu analysierende Logfiles aufgelistet werden (1). Über drei Buttons (2) werden Funktionen zur Verfügung gestellt, die das Hinzufügen oder Entfernen von Logfiles ermöglichen. Darüber hinaus wird angezeigt (3), wie viele Logfiles bei der aktuellen Analyse verarbeitet werden. Eine Checkroutine stellt sicher, dass nicht versehentlich das gleiche Logfile mehrfach der Liste hinzugefügt wird und dadurch die Analyse unnötig in die Länge gezogen wird.

Im rechten Bereich hat man die Möglichkeit verschiedene Analyseintervalle zu definieren (4). In Kapitel 2 ist schon erwähnt worden, dass sehr oft nur Teilausschnitte einer Fahrt von Interesse sind. Der erste Wert des Intervalls bestimmt den Startpunkt auf der Teststrecke, ab dem die Analyse beginnt, der zweite Wert entsprechend das Ende. Die anzugebenden Zahlen sind hierbei Pixelwerte, die das Fahrzeug entlang der Y-Achse zurücklegt.

Da es unterschiedlichen Fahrten mit unterschiedliche Strecken gibt, wurde für den Endwert des Intervalls die Konvention festgelegt, dass eine Null für das Ende einer Strecke steht. Standardmäßig ist immer das Intervall (0,0) vorgegeben (5), demzufolge geht in diesem Intervall die ganze Strecke in die Analyse ein. Das Hinzufügen und Entfernen von Intervallen erfolgt über die Verwendung zweier Buttons (6). Beim Hinzufügen wird dem Benutzer ein Dialogfenster(*NewSlidingWindowDialog*) präsentiert, über das er die Intervallgrenzen definiert.

Nachdem alle Logfiles und Analyseintervalle hinzugefügt wurden, wird beim Drücken des Start-Buttons (7) die Analyse begonnen und die Klasse *Controller* initialisiert. Der Controller regelt dann den weiteren Verlauf der Analyse. Das GUI visualisiert dann nur noch mit Hilfe des Statusbars (8) die Information über den Fortschritt der Analyse. Bei erfolgreicher Analyse oder fehlerbedingtem Abbruch wird man abschließend durch ein entsprechendes Dialogfenster(*ErrorDialog*, *FinishedDialog*) informiert.

6.2 PACKAGE ATEO

Diese Paket enthält alle Klassen, die für die Realisation des Controllermoduls relevant sind. Insgesamt gibt es in diesem Paket 12 Klassen, von denen die wichtigsten hier besprochen werden. Eine vollständige Liste aller Klassen und deren Methoden kann man dem Anhang entnehmen.

Die Klasse *Controller* ist der Dispatcher der Software und regelt den Ablauf der Logfileanalyse. *Controller* ist eine Ableitung

der Klasse *Thread*. Eine Implementierung und das Ableiten von dieser Klasse ist notwendig, da sonst keine Aktualisierung des Fortschritts in der Statusanzeige des GUIs möglich wäre. Für alle Ableitungen der Klasse *Thread* gilt, dass die Methode *run()* überschrieben werden muss.

Controller extrahiert aus der GUI die Liste der Logfiles und die Analyseintervalle. Diese Daten werden für den späteren Zugriff in der Klasse *Config* gespeichert. Anschließend werden Verzeichnisse für den Output der Analyse erstellt, die unterhalb der Verzeichnishierarchie der Analysesoftware angelegt werden. Alle Outputs werden im Unterverzeichnis *results* abgelegt. Die einzelnen Verzeichnissnamen werden nach dem Schema YYYYMMDD_HHMMSS erzeugt. Die einzelnen Stellen haben folgende Bedeutung:

- YYYY - Das aktuelle Jahr
- MM - Der aktuelle Monat
- DD - Der aktuelle Tag
- HH - Die aktuelle Stunde
- MM - Die aktuelle Minute
- SS - Die aktuelle Sekunde

Für jede Analyse wird ein eigenes Verzeichnis angelegt, dessen Bezeichnung der Zeitstempel beim Start der Analyse ist und wie beschrieben formatiert wird. Auf diese Weise wird sichergestellt, dass die Analyseoutputs strukturiert abgelegt werden und keiner versehentlich überschrieben wird. Innerhalb dieses Outputverzeichnisses gibt es ein Verzeichnis *charts* und ein Verzeichnis *xls*. Im Verzeichnis *charts* werden Grafikdateien gespeichert, die die harten Eingriffe des Operateurs visualisieren. Im Verzeichnis *xls* wird eine Exceldatei abgespeichert, die alle weiteren Analyseergebnisse tabellarisch darstellt.

Die eigentliche Logfileanalyse besteht aus zwei ineinander geschachtelten Schleifen, die über alle Messfenster und Logfiles iterieren. Die Ergebnisse jedes Analysedurchlaufs werden in einer separaten Outputdatei gespeichert. Die konkreten Analysefeatures werden aufgeteilt auf viele kleine Einzeltasks, die unabhängig voneinander ausführbar sind. Zu diesem Zweck wird ein sogenannter ThreadPoolExecutor initialisiert. Dabei handelt es sich um ein Objekt das eine Sammlung von Threads aufbaut, den Threadpool. Diese Threads übernehmen dann die Ausführung einzelner Anfragen. Die Verwaltung und Zuteilung der Einzeltasks auf freie Threads übernimmt der ThreadPoolExecutor. Die

Größe des Pools und damit die Anzahl ausführender Threads ist frei wählbar. Sie wurde so gewählt, dass sie abhängig von der Anzahl verfügbarer Prozessorkerne ist. Je nach Hardwareausstattung ist der Threadpool also unterschiedlich groß und damit können unterschiedlich viele Tasks gleichzeitig abgearbeitet werden. Der ThreadPoolExecutor hat die Lebensdauer einer Iteration. Nach Ablauf dieser Iteration wird die zugehörige Outputdatei erzeugt und mit der nächsten Stufe fortgefahren. Ein neuer Thread-Exekutor wird initialisiert und alle Tasks werden mit den aktuellen Parametern erneut ausgeführt. Listing 6.1 zeigt die Verwendung eines ThreadPoolExecutor in der Klasse *Controller*. Codeabschnitte die nichts mit ThreadPoolExecutor zu tun haben wurden wegen der Übersichtlichkeit entfernt.

```
1  package ateo;

   import java.io.File;

   // ...
6

   import java.util.concurrent.ExecutorService;
   import java.util.concurrent.Executors;

   // ...
11

   public class Controller extends Thread{

           private int windowBegin;
           private int windowEnd;
16

           // ...

           private ExecutorService executor=null;

21         // ...

           public void run(){

           // ...
26         int cpuCount=Runtime.getRuntime().availableProcessors();
           executor= Executors.newFixedThreadPool(cpuCount);

           executor.execute(new LFAGetTotalDistance(ateoLogFile, counter,
                stepData, logFileXMLParser));
           executor.execute(new LFAGetTime(windowBegin, windowEnd,ateoLogFile,
                counter, stepData, logFileXMLParser));
31         executor.execute(new LFAGetMaxTime(windowBegin, windowEnd,
                ateoLogFile, counter, stepData, logFileXMLParser));
           executor.execute(new LFAGetError(windowBegin, windowEnd, ateoLogFile
                , stepHeaderData, counter, stepData, logFileXMLParser));
           executor.execute(new LFAGetSum("NoSensorsOffTrack","
                NoSensorsOffTrack",windowBegin, windowEnd,ateoLogFile, counter,
                 stepData, logFileXMLParser));

           // ...
36         }
   }
```

Listing 6.1: Auszug aus Controller.java

Im unteren Teil des Listings sieht man wie mit dem Aufruf der Methode *execute()* Ausführungsanfragen gestellt werden. Dem Ecxecutor wird dabei ein Objekt vom Typ *Thread* übergeben und seine Ausführung in Auftrag gegeben. Alle Anfragen werden dabei in einer Queue gespeichert und zum nächstmöglichen Zeitpunkt ausgeführt.

Die Parser Klassen

Im Laufe der Entwicklung von SAM kam es mehrmals zu einer Veränderung der Loggingroutine. In der Phase der SAM-Hauptuntersuchung wurde für das Logging ein XML-Format verwendet. In der jetzt aktuellen Version verwendet SAM allerdings CSV-Dateien. Für die Logfileauswertung bedeutet das, dass die bisherige Software nicht mal mehr in der Lage gewesen wäre, aktuelle Logfiles zu parsen, geschweige denn Operationen auf den Daten auszuführen. Es mussten daher zwei unterschiedliche Module entwickelt werden, um die verschiedenen Logfileformate verwenden zu können. Prinzipiell arbeiten beide Module nach dem gleichen Prinzip. Es werden die für einen Task benötigten Variablenwerte aus dem Logfile extrahiert und als Liste an die anfragenden Einzeltasks übergeben.

Für die konkrete Realisierung der beiden Parser sind zusätzliche Bibliotheken notwendig gewesen.

XML Parser

Die Klassen *XMLParser* und *LogfileXMLParser* ermöglichen das Einlesen und Verarbeiten von XML-Dateien. *XMLParser* wird verwendet um beliebige Dateien im XML-Format einlesen zu können. So werden zum Beispiel die Koordinaten der Ideallinie einer Strecke in XM-Dateien gespeichert. Weiterhin werden die statischen Daten eines Versuchs wie zum Beispiel Versuchsnummer, Geschlecht der Probanden usw. in XML-Dateien abgelegt. Diese Daten werden zu Beginn eines Versuchs erhoben und dann nicht mehr verändert. Daher müssen sie nicht in jedem Loggingschritt erneut gespeichert werden.

LogfileXMLParser hingegen ist eine Ableitung von *XMLParser* und auf die Besonderheiten des Logfileformats zugeschnitten. Es gibt hier die Möglichkeit die Werte von allen Knoten mit dem selben Namen abzufragen. *XMLParser* bietet nur das Abfragen einzelner Knoten. Es wurden hier die Bibliotheken *JDOM* und *JAXEN(xpath)* eingesetzt, bei deren Verwendung ein Matchingausdruck gegen das Logfile gematcht wird und die passenden Elemente in einer

Liste gespeichert werden. Der Matchingausdruck entspricht dabei dem gesuchten Variablennamen. Listing 6.2 zeigt die Implementierung der Klasse *LogfileXMLParser*.

```
1  package ateo;

   import java.util.ArrayList;

   import org.jaxen.JaxenException;
6  import org.jaxen.jdom.JDOMXPath;
   import org.jdom.Element;

   public class LogfileXMLParser extends XMLParser {

11         public LogfileXMLParser(String path) {
                   super(path);
           }

           @SuppressWarnings("unchecked")
16         public String[] getElements(String node) {
                   String[] values = null;
                   try {
                           JDOMXPath myxpath = new JDOMXPath("Time/"+
                                node+"[1]");
                           ArrayList<Element> tmp = (ArrayList<Element>)
                                 myxpath.evaluate(root);
21                         values= new String[tmp.size()];
                           for (int i = 0; i < tmp.size(); i++) {
                                   Element tmp2 = (Element) tmp.get(i);
                                   values[i] = tmp2.getText();
                           }
26                 } catch (JaxenException e) {
                           // TODO Auto-generated catch block
                           e.printStackTrace();
                   }
                   return values;
31         }

           public String getAttribute(String attr){
                   String value=root.getAttributeValue(attr);
                   return value;
36         }
   }
```

Listing 6.2: Die Klasse LogfileXMLParser

CSV Parser

Hierbei handelt es sich um ein eventbasiertes Verarbeiten der Logdatei. Diese wird zeilenweise eingelesen und die relevanten Spaltenwerte jeder Zeile werden in eine Liste gespeichert. Das konkrete Parsen wurde mit der Bibliothek *supercsv* realisiert. Die Klasse *CSVParser* wird verwendet um Konfigurationsdateien im CSV-Format einzulesen. *StepCSV-Parser* ist eine Ableitung von *CSVParser* und gewährleistet den Zugriff auf CSV-Logfiles.

6.3 PACKAGE ATEO.LFAITEM

Die konkrete Umsetzung der einzelnen Features wird in den Klassen dieses Pakets realisiert. Im Vergleich zur Version 1.0 gibt es hier grundlegende Veränderungen. In der ursprünglichen Version war das LFA-Modul eine einzige Klasse. Die einzelnen Analysefeatures wurden in separaten Methoden umgesetzt und das Ergebnis der Methoden in einer simplen Variable abgespeichert, die dann an späterer Stelle verwendet wurde um eine Outputdatei im XLS-Format zu erzeugen. Alle Features wurden in einer Analyse sequentiell abgearbeitet.

```
package ateo.lfaItem;

import ateo.LogfileXMLParser;
import ateo.ProgressIncrementer;
import ateo.lfaData.LFAStepData;

public abstract class LFAItem extends Thread {

        protected String name;
        protected String context;
        protected String resultType;
        protected Object result;

        protected double start;
        protected double end;
        protected String ateoLogFile;
        protected ProgressIncrementer counter;
        protected LFAStepData stepdata;
        protected LogfileXMLParser logfileXMLParser;


        public LFAItem(double windowStart, double windowEnd, String
             ateoLogFile, ProgressIncrementer counter, LFAStepData stepdata,
              LogfileXMLParser logfileXMLParser){
                this.stepdata=stepdata;
                this.counter=counter;
                this.start=windowStart;
                this.end=windowEnd;
                this.ateoLogFile=ateoLogFile;
                this.logfileXMLParser=logfileXMLParser;
        }

        public LFAItem(String ateoLogFile, ProgressIncrementer counter,
             LFAStepData stepdata, LogfileXMLParser logfileXMLParser) {
                this.stepdata=stepdata;
                this.counter=counter;
                this.ateoLogFile=ateoLogFile;
                this.logfileXMLParser=logfileXMLParser;
        }


        public LFAItem() {
                super();
        }
}
```

Listing 6.3: Die Klasse LFAItem

In der neuen Version werden alle Features threadbasiert verarbeitet, was eine parallele Verarbeitung und ein Ausnutzen aller Prozessorkerne ermöglicht. Als Basis wurde die abstrakte Klasse *LFAItem* implementiert, die von der Klasse Thread ableitet und die Gemeinsamkeiten aller Tasks definiert. In Listing 6.3 sieht man, dass jedes Feature immer einen Namen, einen Kontext und einen Ergebnistyp hat, was in den entsprechenden Variablen als String abgespeichert wird. Weiterhin gibt es eine Variable vom Typ *Object* die für die Aufnahme des Ergebnisses der Funktion vorgesehen ist. Mit *Object* ist der Datentyp bewusst allgemein gehalten worden, da die Features unterschiedlichste Ergebnisse liefern können. Es kann sich dabei um Zahlenwerte, Felder oder auch Dictionaries handeln. Die Klasse bietet zwei Konstruktoren, so dass man Ableitungen von ihr mit oder ohne Parameterübergabe eines Messfensters instantiieren kann. Ein konkretes Feature wird in seiner eigenen Ableitung der Basisklasse implementiert, wobei der genaue Code zur Umsetzung eines Features in der Methode *run()* implementiert werden muss. Da es eine Vielzahl verschiedener Features gibt, werden diese hier nicht vollständig aufgelistet und beschrieben. Die Klasse *LFAGetSum* soll hier exemplarisch beschrieben werden, um einen Eindruck zu liefern wie alle Features implementiert worden sind. Eine Liste aller Klassen und Methoden ist dem Anhang zu entnehmen, die Umsetzung dazu findet man im beigefügten Quellcode.

```
import ateo.LogfileXMLParser;
import ateo.ProgressIncrementer;
import ateo.StepCSVParser;
import ateo.lfaData.LFASingleItemData;
import ateo.lfaData.LFAStepData;

public class LFAGetSum extends LFAItem {

        private String csvColumn;

        public LFAGetSum(String csvColumn, String name, double start, double
              end, String csvFile, ProgressIncrementer counter, LFAStepData
             stepData, LogfileXMLParser logfileXMLParser){
                super(start, end, csvFile, counter,stepData,
                     logfileXMLParser);
                this.csvColumn=csvColumn;
                this.name=name;
                this.context="SAM-Basis";
                this.resultType="value";
                this.result="";
        }

        // ...
```

Listing 6.4: Die Klasse LFAGetSum(Ausschnitt)

Die Klasse *LFAGetSum* liefert die Berechnung einer einfachen Summe für beliebige Logfilevariablen, solange es sich hierbei um Zahlenwerte handelt. Die Umsetzung diese Features ist da-

mit relativ einfach. Schließlich muss nur eine Addition über eine beliebig lange Kette von Werten realisiert werden. Da alle LFAItem-Ableitungen die selbe Struktur haben eignet sich diese Klasse gut, um ein allgemeines Schema zu verdeutlichen.

Wenn man sich den Konstruktor in Listing 6.4 betrachtet, sieh man dass dieser als erstes den Konstruktor der Basisklasse aufruft. Danach werden die in der Basisklasse definierten Variablen mit Werten belegt. Das abgeleitete LFAItem-Objekt hat mit seiner Initialisierung also schon mal einen Namen, einen Kontext und einen Ergebnistyp. Als nächstes betrachten wir den ersten Teil der Methode *run()* in Listing 6.5.

```
        public void run(){
                double sum=0;

                String[] foo=null;
                String[] foo2=null;

                if(ateoLogFile.endsWith(".xml")){
                        foo= logfileXMLParser.getElements(this.csvColumn);
                        foo2= logfileXMLParser.getElements("DistanceTotal");
                }else{
                        foo= new StepCSVParser(ateoLogFile).getStringList(
                             this.csvColumn).toArray(new String[0]);
                        foo2= new StepCSVParser(ateoLogFile).getStringList("
                             DistanceTotal").toArray(new String[0]);
                }
```

Listing 6.5: Die Klasse LFAGetSum(Ausschnitt)

Hier werden die für die Berechnung notwendigen Variablen aus dem Logfile extrahiert. Je nach dem welches Logfileformat verwendet wurde, wird dafür ein XML-Parser oder ein CSV-Parser verwendet. Dieser Parser liefert für alle notwendigen Variablen ein Stringarray zurück. In diesem Array sind alle Werte einer ganzen Fahrt für eine bestimmte Variable enthalten. Diesen Vorgang haben alle Ableitungen von *LFAItem* gemeinsam. Der einzige Unterschied besteht in den relevanten Variablen zur Umsetzung eines Features. Als nächstes wird auf die Logfilevariablen-Arrays sequentiell zugegriffen, was man in Listing 6.6 sehen kann.

```
for(int i=0;i<foo.length;i++){
                        if(!foo[i].equals("nil")){
                                if(Double.parseDouble(foo2[i])>start){
                                        if(Double.parseDouble(foo2[i])<=end)
                                             {
                                                sum+=Double.parseDouble(foo[
                                                     i]);
                                        }
                                        else break;
                                }
                        }
                }
```

Listing 6.6: Die Klasse LFAGetSum(Ausschnitt)

Die konkrete Berechnungsfunktion zur Umsetzung eines Features (in diesem Fall eine einfache Addition) wird dann für jede Iteration ausgeführt, sofern der Iterationsindex sich in den vorgegebene Intervalgrenzen eines Messfensters befindet. Dazu wird der Wert des 'DistanceTotal'-Arrays des jeweiligen Iterationsindexes mit den übergebenen Intervalgrenzen verglichen.

```
this.result=(int)sum;
                LFASingleItemData getTimeItem=new LFASingleItemData(this.
                     name, this.context, this.resultType, this.result);
                this.stepdata.addLFASingleItemData(name, getTimeItem);

                System.out.println(this.name+": "+result + " Counter: " +
                     counter.getCounter());
                counter.incCounter();
        }
}
```

Listing 6.7: Die Klasse LFAGetSum(Ausschnitt)

Listing 6.7 zeigt, dass nachdem die Schleife durchlaufen worden ist und das Ergebnis berechnet wurde, dieses in die interne Datenstruktur eingebunden wird. Dazu wird ein *LFASingleItem*-Objekt erzeugt, das das berechnete Ergebnis aufnimmt, weiterhin wird in diesem Objekt der Name, der Kontext und der Ergebnistyp gespeichert. Das erzeugte *LFASingleItem*-Objekt wird wiederum einem *StepData*-Objekt hinzugefügt. Als Key zur Adressierung wird der Name des Features verwendet. Abschließend wird der Controller durch Erhöhung eines Prozesszählers über die Fertigstellung des Features informiert. Die vollständige Implementierung der Klasse *LFAGetSum* sieht dann folgendermaßen aus.

```
1  package ateo.lfaItem;

   import ateo.LogfileXMLParser;
   import ateo.ProgressIncrementer;
   import ateo.StepCSVParser;
6  import ateo.lfaData.LFASingleItemData;
   import ateo.lfaData.LFAStepData;

   public class LFAGetSum extends LFAItem {

11         private String csvColumn;

           public LFAGetSum(String csvColumn, String name, double start, double
                end, String csvFile, ProgressIncrementer counter, LFAStepData
               stepData, LogfileXMLParser logfileXMLParser){
                   super(start, end, csvFile, counter,stepData,
                        logfileXMLParser);
                   this.csvColumn=csvColumn;
16                 this.name=name;
                   this.context="SAM-Basis";
                   this.resultType="value";
                   this.result="";
           }
21
           public void run(){
                   double sum=0;

                   String[] foo=null;
26                 String[] foo2=null;
```

```
                if(ateoLogFile.endsWith(".xml")){
                        foo= logfileXMLParser.getElements(this.csvColumn);
                        foo2= logfileXMLParser.getElements("DistanceTotal");
31              }else{
                        foo= new StepCSVParser(ateoLogFile).getStringList(
                             this.csvColumn).toArray(new String[0]);
                        foo2= new StepCSVParser(ateoLogFile).getStringList("
                             DistanceTotal").toArray(new String[0]);
                }

36              for(int i=0;i<foo.length;i++){
                        if(!foo[i].equals("nil")){
                                if(Double.parseDouble(foo2[i])>start){
                                        if(Double.parseDouble(foo2[i])<=end)
                                             {
                                                sum+=Double.parseDouble(foo[
                                                     i]);
41                                      }
                                        else break;
                                }
                        }
                }
46              this.result=(int)sum;
                LFASingleItemData getTimeItem=new LFASingleItemData(this.
                     name, this.context, this.resultType, this.result);
                this.stepdata.addLFASingleItemData(name, getTimeItem);

                System.out.println(this.name+": "+result + " Counter: " +
                     counter.getCounter());
51              counter.incCounter();
        }
}
```

Listing 6.8: Die Klasse LFAGetSum

6.4 PACKAGE ATEO.LFADATA

Die parallele Verarbeitung der Features durch den Threadpool führt dazu, dass deren Ergebnisse zu unterschiedlichen Zeiten bereit stehen und keine Reihenfolge bestimmt werden kann. Somit ist ein Abspeichern der Ergebnisse in Listen oder Arrays keine Option, da deren Zugriff über numerische Indizes erfolgt und spätere Lesezugriffe nicht mehr möglich sind. Die Lösung für dieses Problem bietet die in Kapitel 4 vorgestellte Datenstruktur. Sie ermöglicht einen wahlfreien Schreibzugriff unabhängig von der Reihenfolge der Bearbeitung. Späteres Auslesen erfolgt dann unter der Angabe eines eindeutigen Strings. Die Ergebnisse haben eine bestimmte Struktur, die von allen *lfaItem*-Klassen gleichsam implementiert wurde.

Die wichtigste Klasse ist LFASingleItemData. In Listing 6.9 kann man sehen, das sie sich von der Klasse *HashMap* ableitet. Sie verwendet Strings als Keys und kann beliebige Objekte als Werte aufnehmen. Alle *LFAItem*-Features speichern ihr Ergebnis in solch einem Objekt ab. Diese Objekte haben außer dem Ergebnis noch

die Werte Name, Kontext und Ergebnistyp. Diese Werte sind zum einen da, um ein Objekt innerhalb der Datenstruktur wiederfinden zu können und zum andern um einen strukturierten Output zu generieren.

Zur besseren Strukturierung der Datenhaltung werden die *LFASingleItemData*-Objekte in Gruppen organisiert die den verwendeten Messfenstern entsprechen. Die Klassen *StepData* und *LFAData* werden dafür verwendet und sind ebenfalls Ableitungen der Klasse *HashMap*.

```
1  package ateo.lfaData;

   import java.util.HashMap;
   import java.util.Map;

6  public class LFASingleItemData extends HashMap<String,Object>{

       public LFASingleItemData(String name, String context, String type,
            Object result){
          this.put("Name", name);
          this.put("Kontext", context);
11        this.put("Ergebnis-Typ", type);
          this.put("Ergebnis", result);
       }

       public String getItemName(){
16        return (String) this.get("Name");
       }

       public String getItemContext(){
          return (String) this.get("Kontext");
21     }

       public String getItemType(){
          return (String) this.get("Ergebnis-Typ");
       }
26
       public Object getResult(){
          return this.get("Ergebnis");
       }
   }
```

Listing 6.9: Die Klasse LFASingleItemData

6.5 PACKAGE ATEO.OUTPUT

Die Klasse Outputter generiert ein XLS-File für die weitere statistische Analyse durch die Psychologen. Dabei werden unterschiedliche Aspekte der Analyse in jeweils einem Sheet dargestellt. Die Aspekte ergeben sich aus den verwendeten Kontext-Strings in den *LFASingleItemData*-Objekten. Outputter inspiziert einmal alle Objekte, identifiziert dabei verschiedene Kontextarten und leitet daraufhin die Erzeugung eines Sheets für jeden Kontext ein. Die einzelnen Sheets werden dann sequenziell erzeugt und in einer Datei zusammengefasst.

```
package ateo.output;

        // ...

public class Outputter {

        // ...

        private void buildOutput(){

                setup();
                WritableWorkbook workbook;

                try {
                        workbook = Workbook.createWorkbook(file);

                        if(contextTypes.contains("SAM-Basis")){
                                WritableSheet samBaseItems=workbook.
                                     createSheet("SAM-Basisdaten", workbook.
                                     getNumberOfSheets());
                                fillSAMBaseItemSheet(samBaseItems, tmpdata);
                        }
                        if(contextTypes.contains("SAM-Gabel")){
                                WritableSheet samFork=workbook.createSheet("
                                     SAM-Gabelwahl", workbook.
                                     getNumberOfSheets());
                                fillBranchesSheet(samFork, tmpdata);
                        }

                        if(contextTypes.contains("OA-DirectSets")){
                                WritableSheet directSets=workbook.
                                     createSheet("OA-DirectSets", workbook.
                                     getNumberOfSheets());
                                fillDirectSetsSheet(directSets, tmpdata);
                        }

                        if(contextTypes.contains("OA-BlindClicks")){
                                WritableSheet blindClicks=workbook.
                                     createSheet("OA-BlindClicks", workbook.
                                     getNumberOfSheets());
                                fillBlindClicksSheet(blindClicks, tmpdata);
                        }
                        if(contextTypes.contains("OA-SituationAwareness")){
                                WritableSheet situationAwareness=workbook.
                                     createSheet("OA-SituationAwareness",
                                     workbook.getNumberOfSheets());
                                fillSituationAwarenessSheet(
                                     situationAwareness, tmpdata);
                        }
                        if(contextTypes.contains("OA-HinweisTiming")){
                                WritableSheet hintTiming=workbook.
                                     createSheet("OA-HinweisTiming",
                                     workbook.getNumberOfSheets());
                                fillHintTimingSheet(hintTiming, tmpdata);
                        }
                        if(contextTypes.contains("OA-HinweisHäufigkeiten")){
                                WritableSheet hintFrequencies=workbook.
                                     createSheet("OA-HinweisHäufigkeiten",
                                     workbook.getNumberOfSheets());
                                fillHintFrequenciesSheet(hintFrequencies,
                                     tmpdata);
                        }

                        CellView cv = new CellView();
                        cv.setAutosize(true);
```

```
                    for (int i = 0; i < workbook.getSheets().length; i
                         ++) {
                        for (int j = 0; j < workbook.getSheets()[i].
                             getColumns(); j++) {
                            if (workbook.getSheets()[i].
                                 getColumn(j).length!=0) {
                                workbook.getSheets()[i].
                                     setColumnView(j, cv);
                            }
                        }
                    }

                    workbook.write();
                    workbook.close();
                    counter.incCounter();
                } catch (IOException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                } catch (WriteException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
        }

        // ....
```

Listing 6.10: Auszug der Klasse Outputter

Listing 6.10 zeigt einen Ausschnitt aus der Klasse *Outputter*. Die Methode *buildOutput()* iteriert über eine Liste von Kontextarten und erzeugt dementsprechende Excelsheets. Der restliche Code zur Erzeugung der einzelnen Sheets ist größtenteils mit dem Code von GUIs vergleichbar, da es sich hierbei fast ausschließlich um Code zur Darstellung, Positionierung und Formatierung von Elementen handelt. Den konkreten Aufbau der XLS-Datei und der einzelnen Sheets kann man dem Anhang entnehmen.

7 DISKUSSION UND AUSBLICK

7.1 DISKUSSION

Der Prototyp zur automatisierten Logfileanalyse konnte in dem Maße erweitert werden, dass eine Software geschaffen wurde, die das Auswerten der Aspekte des Operateursarbeitsplatzes in angemessener Zeit ermöglicht. Dabei wurden alle funktionalen Anforderungen vollständig umgesetzt aufgezeigten Schwachstellen konnten durch eine Restrukturierung der Software überwiegend beseitigt werden. Der Zeitbedarf für eine vollständige Analyse einer Untersuchungreihe hat sich im Vergleich zum Prototypen deutlich deutlich verringert. Die Verständlichkeit der Software konnte durch Verfeinerung der Klassenstruktur stark erhöht werden und ermöglicht nun zukünftige Anpassungen ohne weite Teile der Software verändern zu müssen. Einzig eine Reduzierung des Speicherverbrauchs durch eine Überarbeitung der Parserkomponente wurde nicht umgesetzt, da diesbezüglich durch die Verwendung von CSV-Dateien keine besondere Dringlichkeit mehr gegeben war.

Insbesondere die Verteilung der konkreten Berechnungsfunktionen auf einzelne Klassen innerhalb eines Pakets ermöglicht eine einfache Erweiterung der Analysesoftware um neue Features. Es sei hier anzumerken, dass die Darstellung der Ergebnisse neuer Funktionen auch immer eine Implementierung einer zugehörigen Outputroutine erfordert. Durch die modulare Bauweise der Software bleibt hier aber die Wahl der Mittel vollkommen frei. Es kann hier im bestehenden Outputmodul eine neue Funktion erstellt werden, die ein weiteres Sheet innerhalb des Excel-Files erzeugt. Genau so ist es aber auch möglich eine beliebige andere Form der Visualiserung zu wählen ohne die Outputs der anderen Features in Mitleidenschaft zu ziehen.

Der Output für die Excelsheets erfordert maßgeschneiderten Code. Die Anforderungen an die Logfileanalyse waren zu speziell um eine generische Lösung für beliebige Outputs zu implementieren. Es macht daher vom Implementieraufwand keinen Unterschied ob man sich für die bestehende Excel-Lösung entscheidet oder eine andere Form des Outputs implementiert. Excel war jedoch bis jetzt das Mittel der Wahl, weil die Analyseergebnisse so strukturiert dargestellt werden konnten und ausserdem leicht weiter zu verwenden sind, beispielsweise als Input für eine statis-

tische Analyse in SPSS.

Die Reduzierung des Speicherverbrauchs ist immer stärker in den Hintergrund getreten und hat letztendlich wegen des Wechsels auf CSV-Logfiles an Priorität verloren. Die Ursache für den hohen Verbrauchs von LFA 1.0 lag im wesentlichen am XML-Parser, der durch das Anlegen eines DOM-Trees ein vielfaches der Größe eines Logfiles im Speicher belegte. Es war daher vorgesehen das Parser-Modul zu überarbeiten.
Eine eventbasierte Lösung auf Basis eines SAX-Parsers hätte wahrscheinlich eine deutliche Reduzierung des Speicherbedarfs bewirkt. Allerdings hätte dies einen größeren Implementieraufwand nach sich gezogen. Ein eventbasierter SAX-Parser hätte komplett neu implementiert werden müssen, insbesondere eines kompletten Satzes an Handlerfunktionen für alle auftretenden Events.

Da das alte Modul jedoch prinzipiell funktionsfähig war und durch geringfügige Anpassungen bei den Interfaces weiterhin verwendet werden konnte, wurde auf eine Neuimplementierung des Parsermoduls verzichtet. Der Einsatz von CSV-logfiles in allen zukünftigen Versuchen bedeutet, dass die Auswertung von XML-logfiles in Zukunft eigentlich keine mehr Verwendung findet. Es handelt sich somit nur noch um ein Conveniance-Feature, dass zukünftig nur noch gebraucht wird um alte Versuchsreihen noch einmal auswerten zu können, weil sich zum Beispiel neue Ansätze aus dem Projekt heraus ergeben haben. Dies ist natürlich nur sinnvoll und möglich, wenn die geloggten Variablen das hergeben.

7.2 AUSBLICK

Bei der Verwendung der Analysesoftware hat sich herausgestellt, dass häufig die selben Analyseintervalle für verschiedene Probandengruppen verwendet werden. Da es viele dieser Intervalle gibt, ist das fortwährende Eingeben sehr mühselig und zudem ein Quelle für mögliche Fehler. Ein Intervall könnte durch Unaufmerksamkeit oder Müdigkeit falsch eingeben werden und die Analyse müsste erneut mit korrekten Intervallen ausgeführt werden. Ein einmaliges Eingeben und Abspeichern von häufig verwendeten Konfigurationen wären eine enorme Arbeitserleichterung und damit ein sehr willkommenes Feature für die beteiligten Psychologen im ATEO-Projekt

Die derzeitige Software bietet Funktionen zur Auswertung hinsichtlich der MWB und des OA. Konkrete Anforderung zur Auswertung der Automatiken waren zum Zeitpunkt der Implemen-

tierung noch nicht endgültig definiert. Prinzipiell ist die Analyse der Automatiken zwar möglich, jedoch nur solange sie dieselben Eingriffsmöglichkeiten bieten wie der OA und auch die selben Logfilevariablen verwenden. Bei einem sich unterscheidenden Umfang an Eingriffsmöglichkeiten der Automatiken ist es daher erneut erforderlich die Anlysesoftware zu erweitern.

Bei den Anforderungen den OA betreffend handelt es sich teilweise nicht mehr um einfache Berechnungen. Vielmehr gehen einige davon in Richtung des Erkennens von Verhalten. Für zukünftige Untersuchungen könnte sich daher ein komplett neuer Themenschwerpunkt eröffnen. In diesem könnte man versuchen systematisch die Güte von Eingriffen zu bewerten, also die Frage welche Eingriffe und zu welchen Zeitpunkten diese Eingriffe einen Effekt auf die Leistung der MWB haben. Sollte sich herausstellen, dass bestimmte Eingriffe systematisch überhaupt keine Wirkung haben, kann man diese aus dem Fundus der Eingriffsmöglichkeiten wieder entfernen. Das verringert die Komplexität eines Assistenzsystems oder auch die des Operateursarbeitsplatzes. Bisher fehlt es allerdings komplett an Kriterien oder Metriken wie man die Güte eines Eingriffs messen und bewerten soll.

Eine einfache Wenn-Dann-Beziehung wäre ein Ansatz, den man in Betracht ziehen könnte. Man sucht dann nach eventuellen Leistungveränderungen der MWB, wenn der Operateur einen bestimmten Eingriff vornimmt. Diese Leistungsveränderungen könnte man dann Funktionen mit gewichteten Faktoren definieren. Solche Faktoren wären dann zum Beispiel die benötigte Zeit oder das Flächenfehlermaß in einem bestimmten Intervall. Ein Intervall ist hierbei sinnvoll und notwendig um einen zeitlichen Bezug zum Eingriff herstellen zu können. Zum einen braucht man ein Intervall um die Leistung vor einem bestimmten Eingriff zu bestimmen und zweites um die Leistung nach dem Eingriff zu ermitteln. Je weiter entfernt vom Eingriffszeitpunkt dann eine Leistungsänderung der MWB zu verzeichnen ist, um so unwahrscheinlicher ist der konkrete Eingriff die Ursache dafür.

Ausgehend von dieser Hypothese könnte man dann einen Satz von Eingriffen aufstellen von dem man glaubt dass sie eine Wirkung auf die MWB haben und diese dann überprüfen. Dabei kann es sich um elementare Eingriffe handeln oder auch um beliebig komplexe Kombinationen von Eingriffen. Ob solch eine Art der Untersuchung von Eingriffen jemals erfolgen wird ist ungewiss. Sollte es jedoch dazu kommen, müsste die Logfileanalyse um einige Komponenten erweitert werden.

Zunächst einmal wäre es erforderlich ein Konfigurationstool zu schaffen, das aus einem festen Set von Eingriffen Regeln für beliebig komplexe Eingriffskombinationen zu schaffen. Diese müssten dann mit entsprechenden generischen Analyseroutinen verknüpft werden, die die Bedingungen einer Regel analysieren und im Logfile nach dem Auftreten einer solchen Regel suchen. Eventuell könnte es auch von Interesse sein die Bezugsintervalle in ihrer Größe dynamisch zu verändern. Abhängig von der Größe des Intervals und der Entfernung des Ereignisses zum Eingriffszeitpunkt kann die gefundene Leistungsveränderung unterschiedlich bewertet werden. Für solche dynamischen Intervallgrenzen müßten dann ebenfalls GUI-Elemente geschaffen werden und die Analyseroutinen entsprechend angepasst werden.

Da die definierten Regeln beliebig komplex sein könnten und durch dynamisch wachsende Intervalle mit vielen Iterationen einer Routine zu rechnen ist, würde das Suchen nach solchen Regeln in den Logfiles einen enormen Rechenaufwand nach sich ziehen. Das Anwendungszenario der Logfileanalyse würde sich dann vom Berechnen und Aggregieren von Logfilevariablen hin zum Suchen, Finden und Bewerten von Verhaltensmustern entwickeln. Einige der OA-Anforderungen deuten diesen Trend ja bereits an. Eine Analyse eines kompletten Versuchsdurchgangs könnte dann durchaus die Kapazitäten herkömmlicher Computerhardware übersteigen.

Man müsste dann das Paradigma einer Ein-Computer-Software verlassen und über ein verteiltes Rechnen zur Lösung der Aufgabe nachdenken. Die Logfileanalyse müsste dann in eine Client-Server-Infrastrukur integriert werden. Dabei würden die Server über die Programmlogik zur Bearbeitung der einzelnen Tasks verfügen und im Netzwerk diese Routinen anbieten. Die Clientsoftware würde dann das GUI für Analyse beinhalten und sich dann nur noch das Verteilen der Tasks koordinieren und den Output erzeugen.

Um die Verwaltung der Serverinstanzen möglichst einfach zu gestalten könnte man auf Protokolle wie Bonjour oder UPnP zurückgreifen. Ein manuelles Verwalten der Netzwerkadressen der Server ist dann nicht nötig. Darüber hinaus müssten verschiedene Konzepte zur Robustheit der Infrastruktur erstellt und umgesetzt werden. Beispielsweise darf ein ausfallender Server nicht zur Blockade der ganzen Analyse führen.

Die Clientseite muss dementsprechend für derartige Ereignisse über geeignete Ausfalllösungen verfügen. Das Einsatz von Heartbeat-Technologien würde sich hier anbieten. Sollte nach

einem gewissen Timeout keine Reaktion vom Server erfolgen oder dieser einen Fehler melden, muss der zugeteilte Task an einen anderen Server erneut vergeben werden. Weiterhin sollte man bei der Erzeugung des Outputs über das Speichern von Zwischenergebnissen an bestimmten Checkpoints nachdenken. Ein Systemausfall würde dann nicht den vollständigen Neustart einer Analyse erfordern und würde wertvolle Zeit einsparen. Eine Analyse könnte dann einfach vom letzten Checkpoint aus wieder aufgenommen werden.

Angesichts des zu erwartenden hohen Bedarfs an Rechenleistung könnte man über die Nutzung von Clouddiensten nachdenken. Die Logfileauswertung als solche ist zwar sehr rechenintensiv, erfolgt allerdings nur sehr punktuell, nämlich in direkter Folge einer Versuchsreihe. Sollte die vorhandene Infrastruktur nicht dem Bedarf an Rechenleistung gerecht werden, könnte man einen solchen Schritt in Erwägung ziehen und einen Neukauf von Rechnerhardware mit den Kosten für die punktuelle Nutzung von Clouddiensten vergleichen. In diesen Vergleich müsste man aber auf jeden Fall die höheren Kosten für die Entwicklung von Cloudapplikationen einbeziehen. Abhängig vom Jeweiligen Anbieter eines Clouddienstes kann die Verwendung bestimmter Technologien, Programmiersprachen und Frameworks vorgeschrieben sein, was wiederum eine gewisse Einarbeitungszeit erfordert.

Zusammenfassend lässt sich sagen dass sich eine Entwicklung der Forschung in die beschriebene Richtung stark auf die Logfileanalyse auswirken würde. Sie würde eine großen Bedarf an zusätzlichen Ressourcen hervorrufen sowohl in personeller als auch materieller Hinsicht. Die Umsetzung eines derart großen und komplexen Themenschwerpunktes würde allein schon in der Konzeptionsphase enorm viele Kräfte binden. Es müssten viele verschiedene Aspekte berücksichtigt werden, die nicht nur von der Seite der Software-Entwickler betrachtet werden dürfen. Die Ausarbeitung einer Methologie zur Bewertung der Güte von Eingriffen müsste auf jeden Fall in Zusammenarbeit mit Psychologen erfolgen. Man muss jedoch davon ausgehen, dass hier viel Zeit investiert werden muss bevor eine abschließende Lösung gefunden werden kann.

Ob die beschriebenen Ideen im Rahmen des ATEO-Projekts überhaupt noch umgesetzt werden ist letzten Endes unklar. Das Projekt befindet sich mittlerweile in seiner abschließenden Phase und wird mit dem Ende dieser Phase auch nicht mehr verlängert verlängert werden. Da die jetzigen Themenschwerpunkte schon seit langem festgelegt sind, gibt es für neue Forschungsansätze nur begrenzten Spielraum. Zur Verwirklichung dieser

Ideen kommt es also bestenfalls in anderen Forschungsprojekten, die sich auf Erkenntnisse aus ATEO beziehen und die Arbeit in diesem Themenfeld fortführen möchten.

A

OUTPUTBESCHREIBUNG

Die Outputdatei der Logfileanalyse ist in einzelne Sheets unterteilt. In jedem Sheet wird ein anderer Aspekt der Untersuchung betrachtet. Die einzelnen Sheets werden an dieser Stelle vorgestellt.

A.1 SAM - BASISDATEN

In diesem Sheet werden alle Features zusammengefasst, die die Leistungsbewertung der MWB betreffen. Dazu gehören die Leistungsindikatoren Flächenfehler, Zeitfehler, Anzahl der Kollisionen usw. Die ersten Spalten erfassen zunächst einmal alle Versuchsparameter (Experimentnummer, Versuchspersonennummer, ...). Die anschließenden Spalten beeinhalten Ergebnisse der Logfileanalyse.

A.2 SAM - GABELWAHL

Dieses Sheet visualisiert das Verhalten der MWB an den Weggabelungen. Konkret wird hier erfasst, bei welchen Gabelungen sie sich für welche Abzweigung entschieden haben. Die Güte gibt dabei an, ob sie sich von Anfang an klar für eine bestimmte Alternative entschieden haben oder zwischen den Alternativen hin und her gewechselt sind. Die verschiedenen MWB-Teams werden dabei jeweils als Block zusammengefasst und alle Blöcke in der linken Hälfte des Sheets untereinander dargestellt. Auf der rechten Seite wird die durchschnittliche Leistung aller Teams tabellarisch zusammengefasst.

A.3 OA - DIRECTSETS

In diesem Sheet werden alle harten Eingriffe des Operateurs erfasst. Dabei wird festgehalten wann und wie oft welche Eingriffe vorgenommen wurden und auf welche Werte die Eingriffe eingestellt waren. Jede Eingriffsmöglichkeit jedes Operateurs wird in einem separaten Block dargestellt. Zusammenfassend werden in einer Tabelle rechts von diesen Blöcken Häufigkeiten und Durchschnittswerte aufgelistet.

A.4 OA - BLINDCLICKS

Dieses Sheet erfasst alle Blindclicks, die ein Operateur während einer Fahrt tätigt. Ein Blindclick wird hierbei als Click innerhalb des OA definiert, an dessen Stelle sich kein Button zur Auslösung eines Hinweises befindet. Auch hier werden wieder alle Daten (Koordinaten eines Clicks) eines Operateurs in einem Block zusammengefasst und alle Blöcke untereinander aufgelistet. Rechts davon werden in einer Tabelle noch einmal die Häufigkeiten aller Operateurblindclicks dargestellt.

A.5 OA - SITUATION AWARENESS

Bei der Situation Awareness wird die mentale Belastung der Operateure bewertet. Sie müssen dazu während der Versuche auf ein bestimmtes Ereignis reagieren, dass mit ihrer eigentlichen Aufgabe (Überwachung der MWB) nichts zu tun hat. Die Reaktionszeit die vergeht, bis sie auf dieses Ereignis reagieren, drückt somit ihre Belastung aus. Die Eintrittszeiten des Ereignisses und der Reaktion der Operateure werden in diesem Sheet dargestellt.

A.6 OA - HINWEISTIMING

Dieses Sheet stellt das Timingverhalten der Operateure und der von ihnen gegebenen Hinweise dar. Bei den auditiven Hinweisen geht es darum wie viel Zeit der Operateur benötigt, um nach der Auswahl eines Hinweises einen bestimmte MWB auszuwählen. Bei den visuellen Hinweisen wird ermittelt, ob ein Operateur rechtzeitig die MWB vor kommenden Hindernissen oder Gabelungen warnt. Für jede Hinweisart jedes Operateurs gibt es auch hier wieder separate Blöcke und die Durchschnittswerte jeder Hinweisart werden in einer Tabelle zusammengefasst.

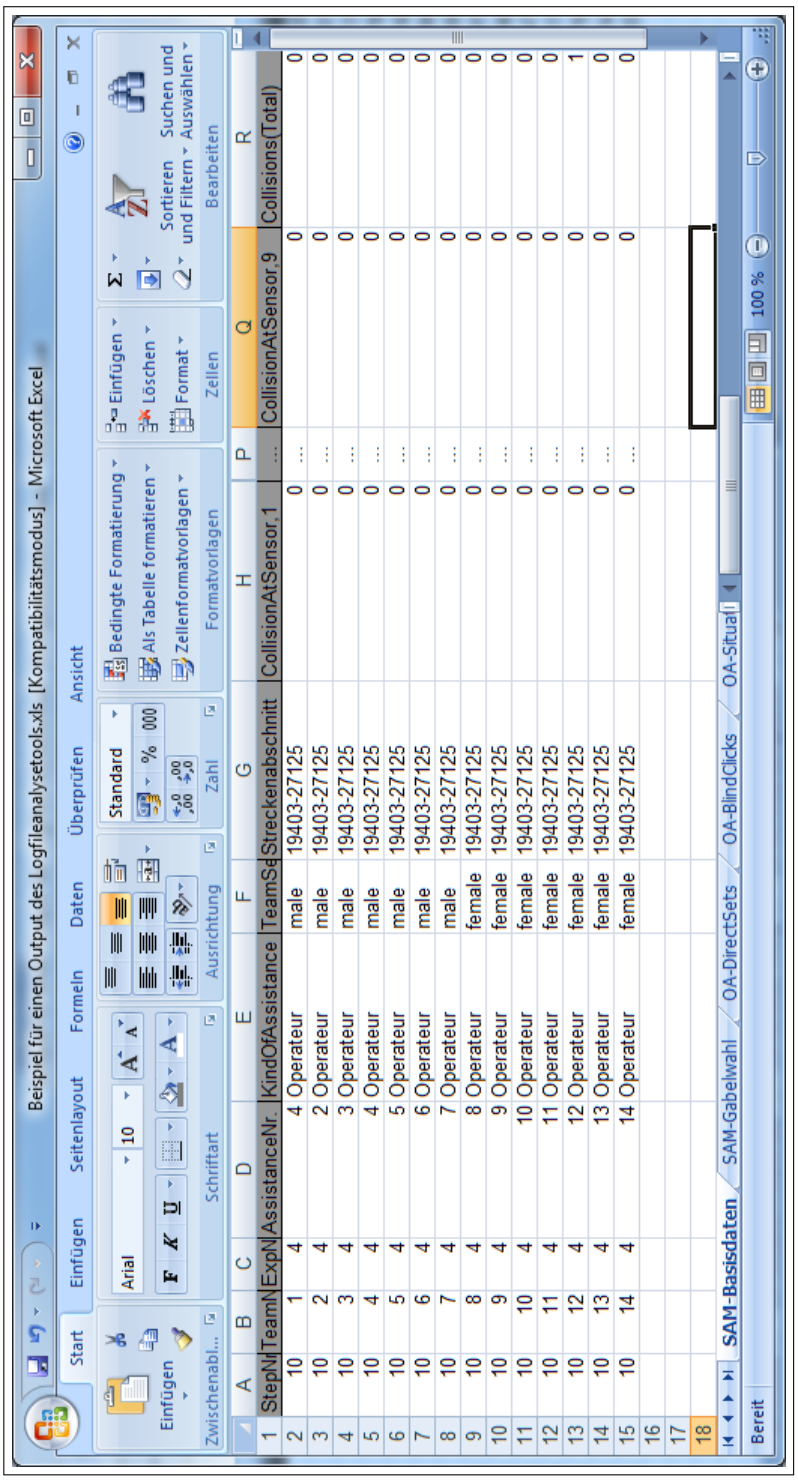
A.7 OA - HINWEISHÄUFIGKEITEN

Dieses Sheet erfasst die Anzahl aller gegebenen Hinweise der Operateure. Es werden absolute und relative Häufigkeitswerte für alle Hinweise ermittelt und aufgelistet.

A.8 OA - DIRECTSETS DIAGRAMME

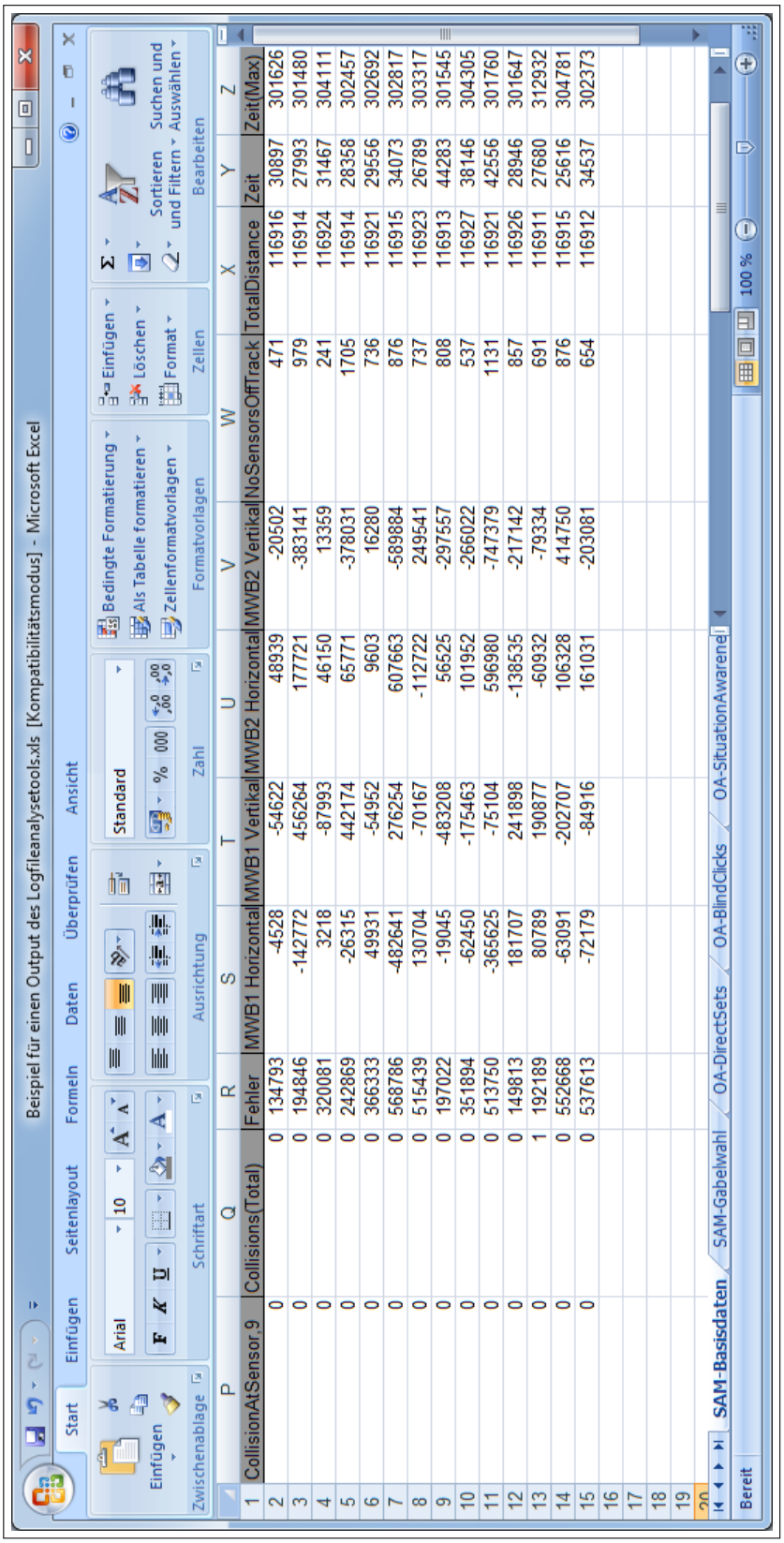
Abgesehen von der Exceldatei werden bei jeder Analyse Grafikdateien erzeugt, die die harten Eingriffe des Operateurs als Diagramm darstellen. Die Abbildungen 21 bis 23 sind beispielhafte Exemplare dafür. Die rote Linie stellt die Inputverteilung der MWB-Joysticks dar und hat zu Beginn eines Versuchs den Wert 50.

Die blaue Linie visualisiert die Höchstgeschwindigkeit des Fahrobjekts und hat einen Initialwert von 100. Die Grüne Linie steht für die Richtungsbeschränkung des Fahrobjekt. Bei Verwendung dieses Eingriffs kann das Fahrzeug nur noch in einer bestimmte Richtung fahren. Bei Betrachtung dieser Diagramme kann der Versuchsleiter leicht besonders aktive Operateure identifizieren noch bevor das Excelfile geöffnet wurde.



StepN	TeamN	ExpN	AssistanceNr.	KindOfAssistance	TeamSe	Streckenabschnitt	CollisionAtSensor_1	...	CollisionAtSensor_9	Collisions(Total)
10	1	4	4	Operateur	male	19403-27125	0	...	0	0
10	2	4	2	Operateur	male	19403-27125	0	...	0	0
10	3	4	3	Operateur	male	19403-27125	0	...	0	0
10	4	4	4	Operateur	male	19403-27125	0	...	0	0
10	5	4	5	Operateur	male	19403-27125	0	...	0	0
10	6	4	6	Operateur	male	19403-27125	0	...	0	0
10	7	4	7	Operateur	male	19403-27125	0	...	0	0
10	8	4	8	Operateur	female	19403-27125	0	...	0	0
10	9	4	9	Operateur	female	19403-27125	0	...	0	0
10	10	4	10	Operateur	female	19403-27125	0	...	0	0
10	11	4	11	Operateur	female	19403-27125	0	...	0	0
10	12	4	12	Operateur	female	19403-27125	0	...	0	1
10	13	4	13	Operateur	female	19403-27125	0	...	0	0
10	14	4	14	Operateur	female	19403-27125	0	...	0	0

Abbildung 11: Output - SAM-Basisdaten, linke Seite



	P	Q	R	S	T	U	V	W	X	Y	Z
1	CollisionAtSensor_9	Collisions(Total)	Fehler	MWB1 Horizontal	MWB1 Vertikal	MWB2 Horizontal	MWB2 Vertikal	NoSensorsOffTrack	TotalDistance	Zeit	Zeit(Max)
2	0	0	134793	-4528	-54622	48939	-20502	471	116916	30897	301626
3	0	0	194846	-142772	456264	177721	-383141	979	116914	27993	301480
4	0	0	320081	3218	-87993	46150	13359	241	116924	31467	304111
5	0	0	242869	-26315	442174	65771	-378031	1705	116914	28358	302457
6	0	0	366333	49931	-54952	9603	16280	736	116921	29556	302692
7	0	0	568786	-482641	276254	607663	-589884	876	116915	34073	302817
8	0	0	515439	130704	-70167	-112722	249541	737	116923	26789	303317
9	0	0	197022	-19045	-483208	56525	-297557	808	116913	44283	301545
10	0	0	351894	-62450	-175463	101952	-266022	537	116927	38146	304305
11	0	0	513750	-365625	-75104	596980	-747379	1131	116921	42556	301760
12	0	0	149813	181707	241898	-138535	-217142	857	116926	28946	301647
13	0	1	192189	80789	190877	-60932	-79334	691	116911	27680	312932
14	0	0	552668	-63091	-202707	106328	414750	876	116915	25616	304781
15	0	0	537613	-72179	-84916	161031	-203081	654	116912	34537	302373
16											
17											
18											
19											
20											

Abbildung 12: Output - SAM-Basisdaten, rechte Seite

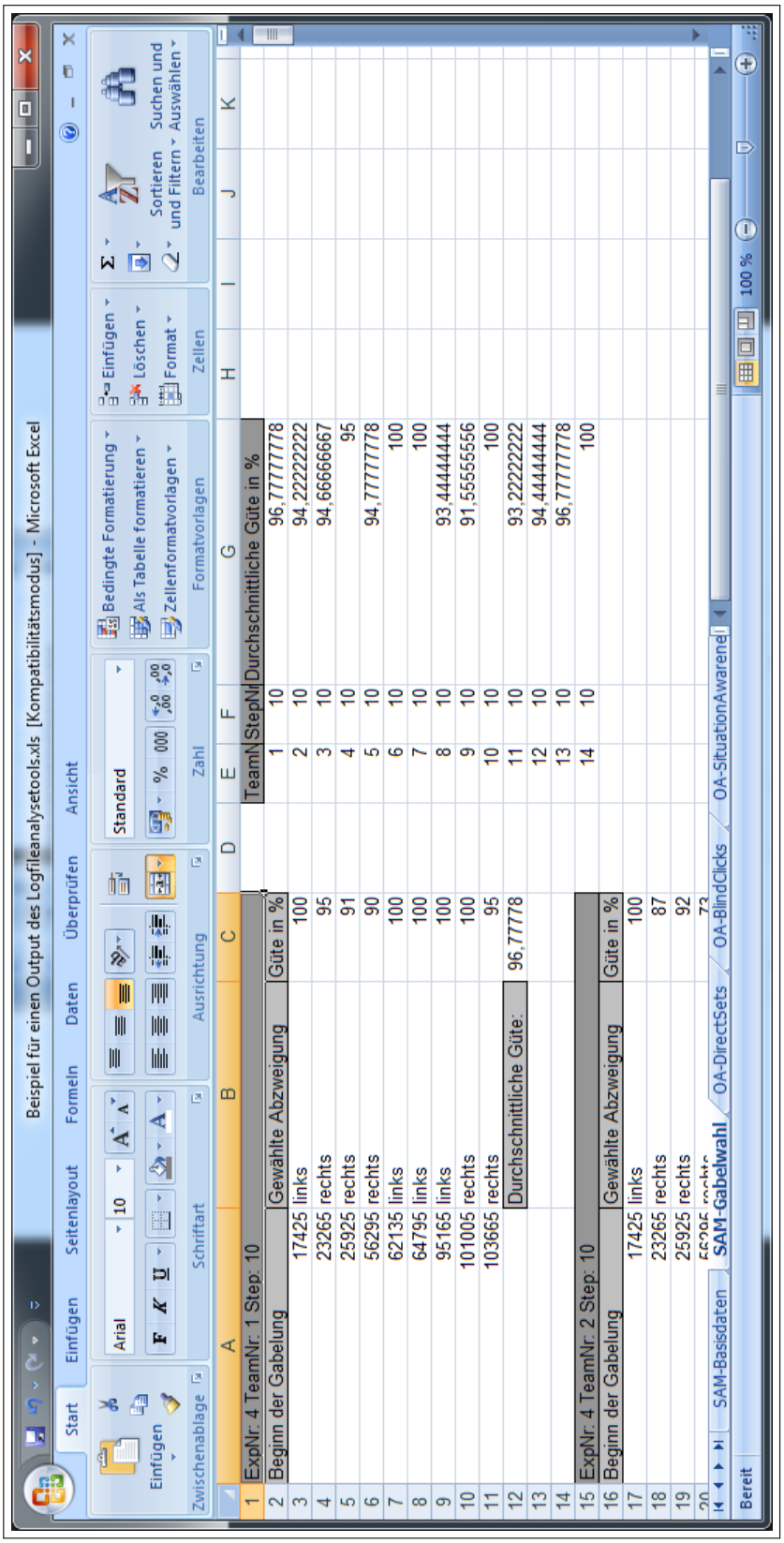


Abbildung 13: Output - MWB-Gabelwahl

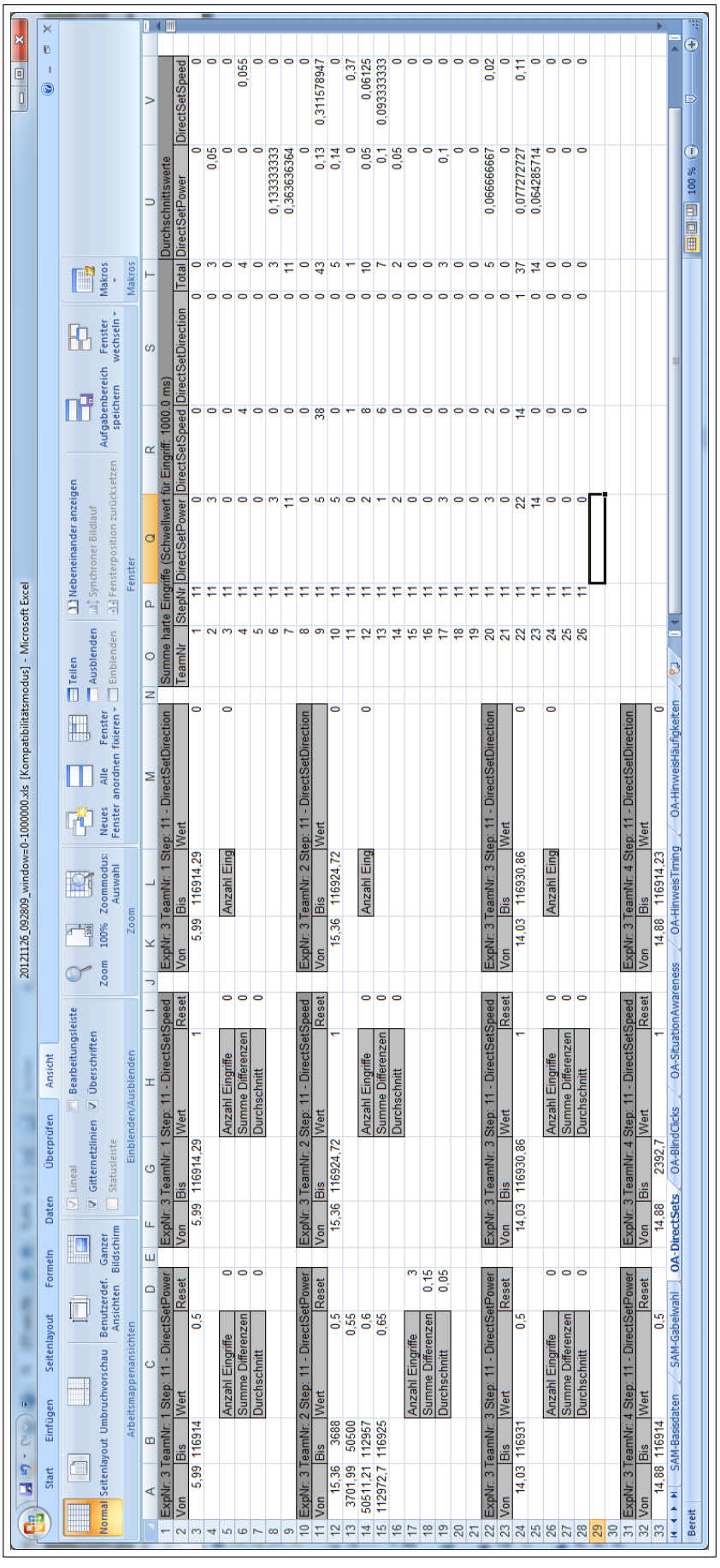


Abbildung 14: Output - Harte Eingriffe

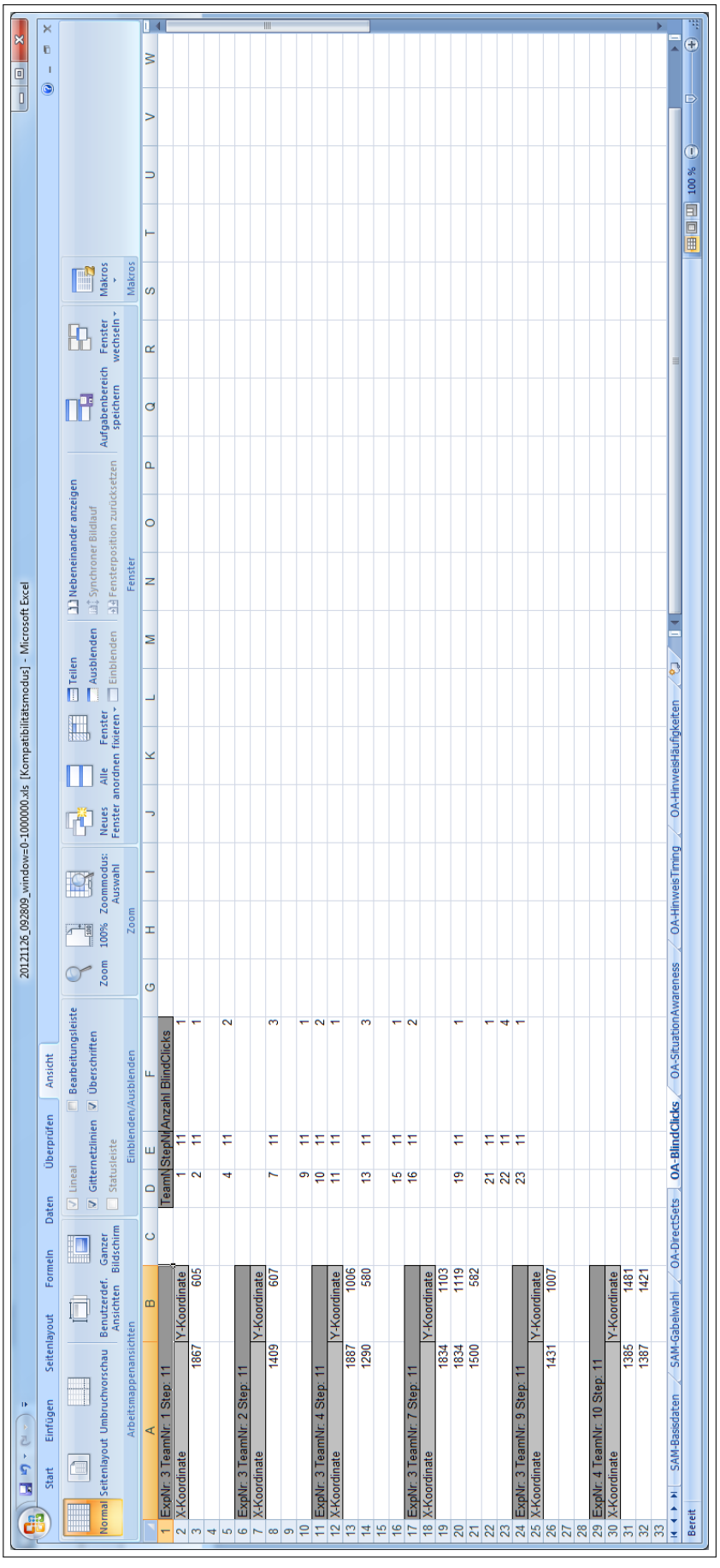


Abbildung 15: Output - Blindclicks

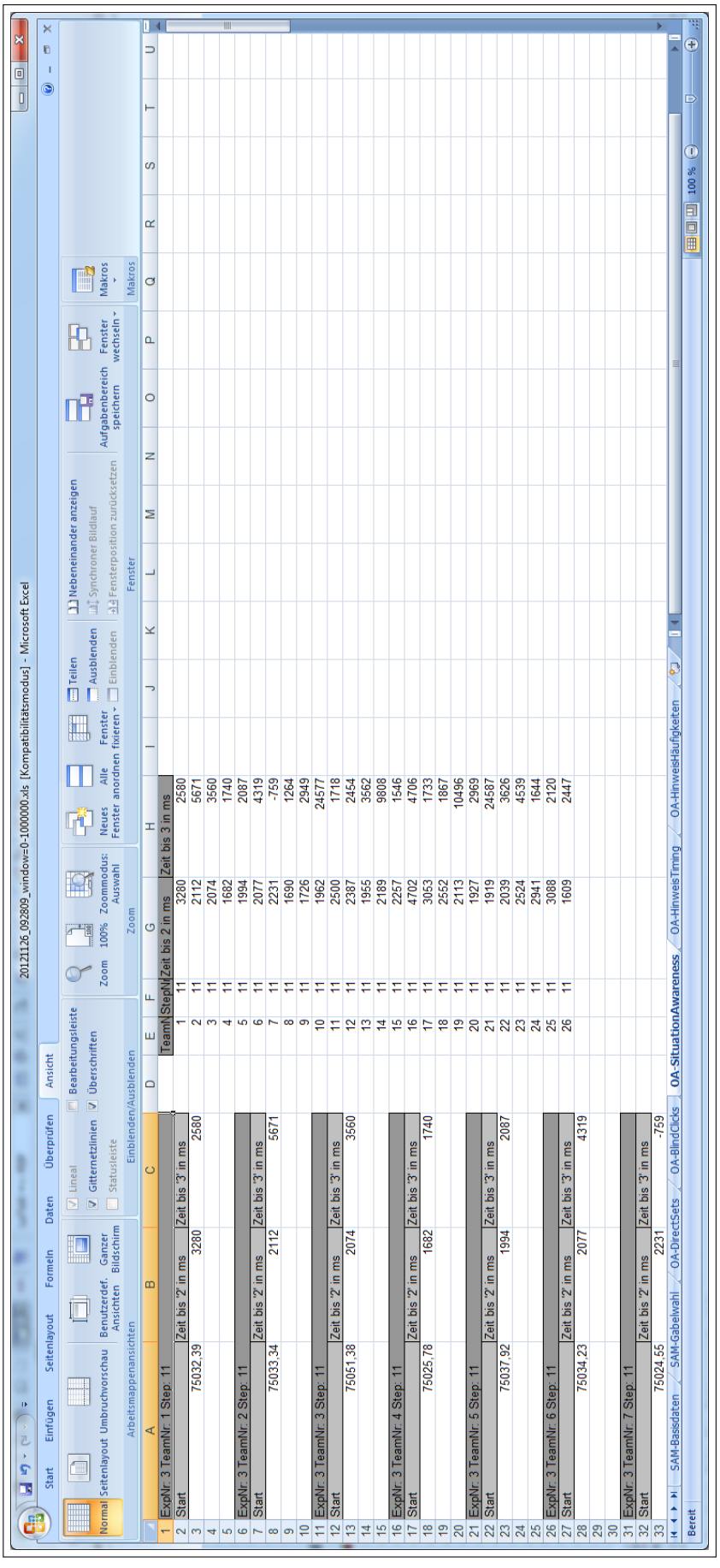


Abbildung 16: Output - Situation Awarenes

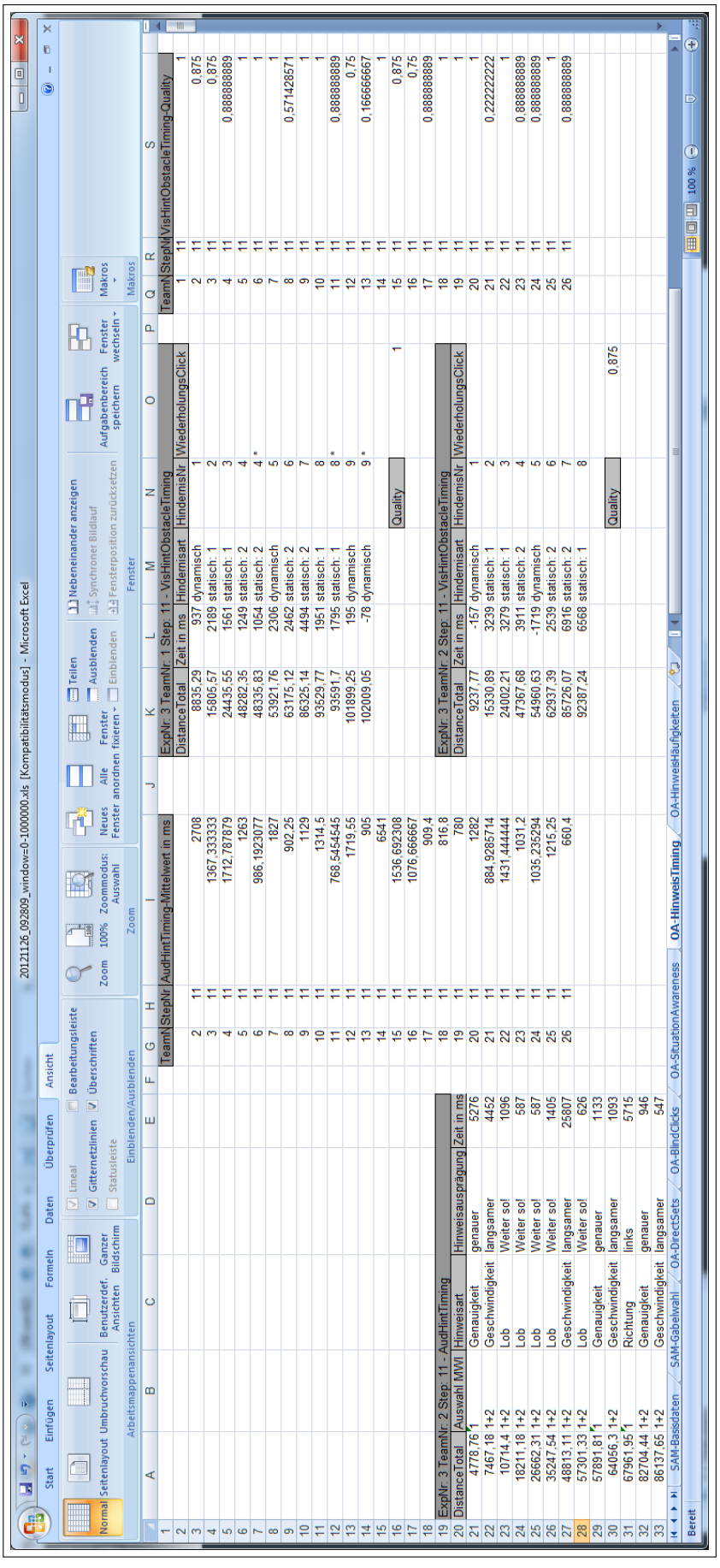


Abbildung 17: Output - Hinweistiming, linke Seite

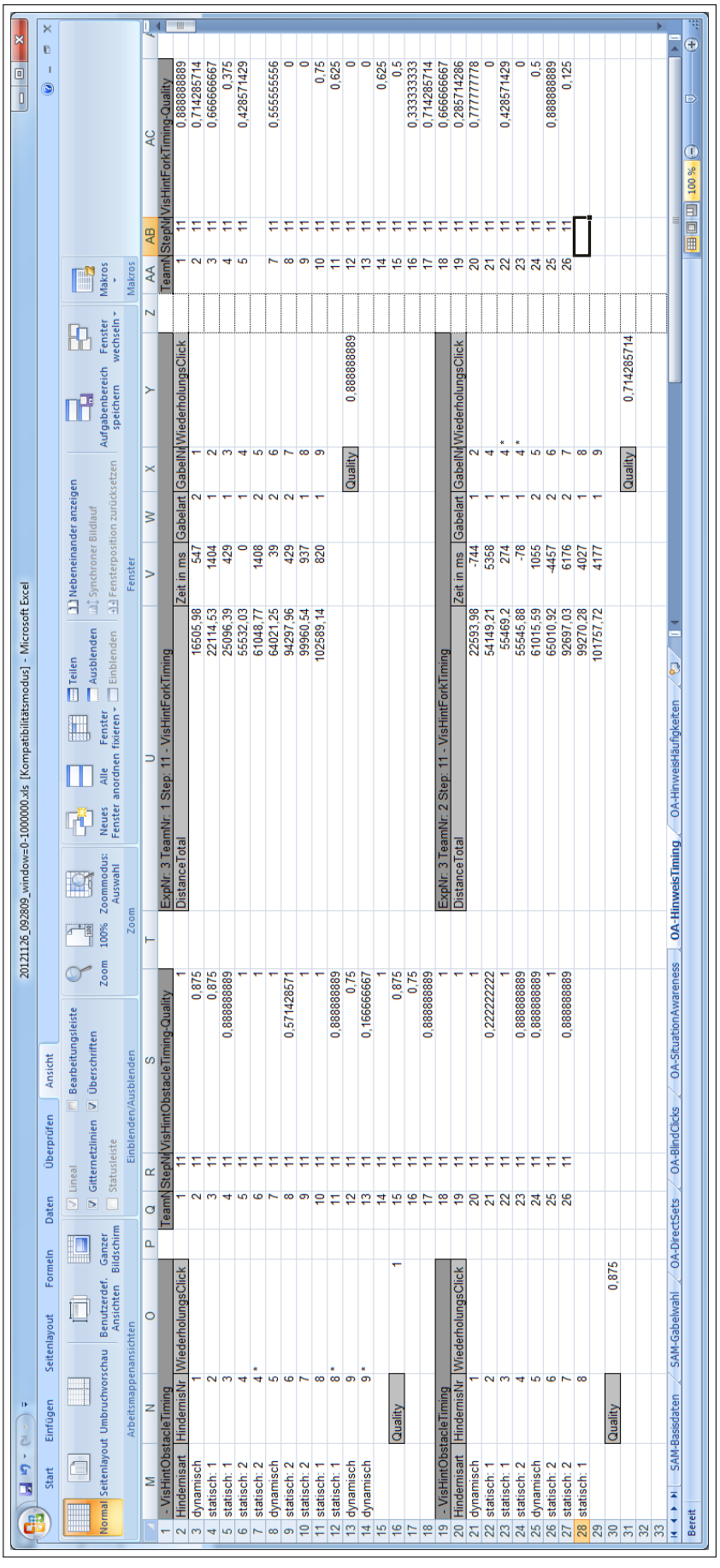


Abbildung 18: Output - Hinweistiming, rechte Seite

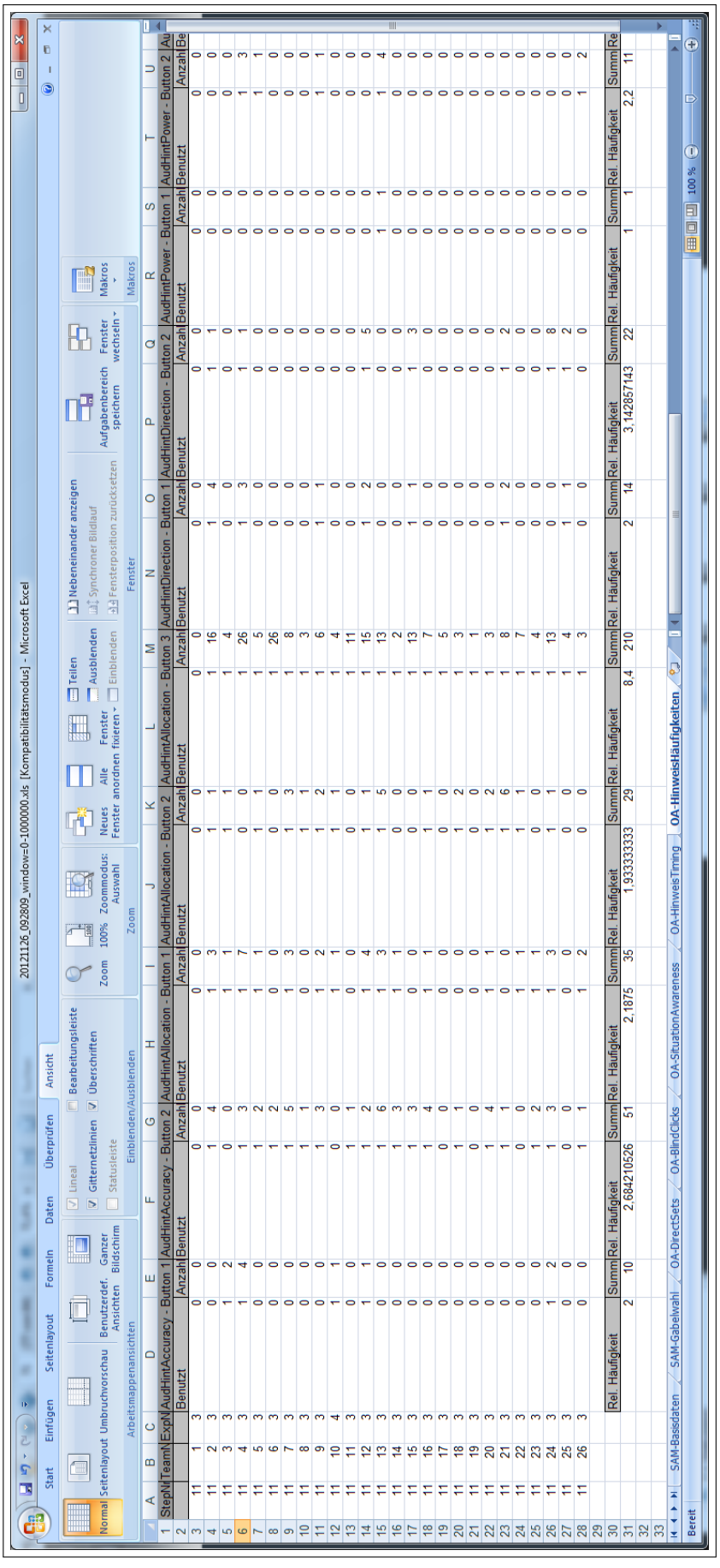


Abbildung 19: Output - Hinweishäufigkeiten linke Seite

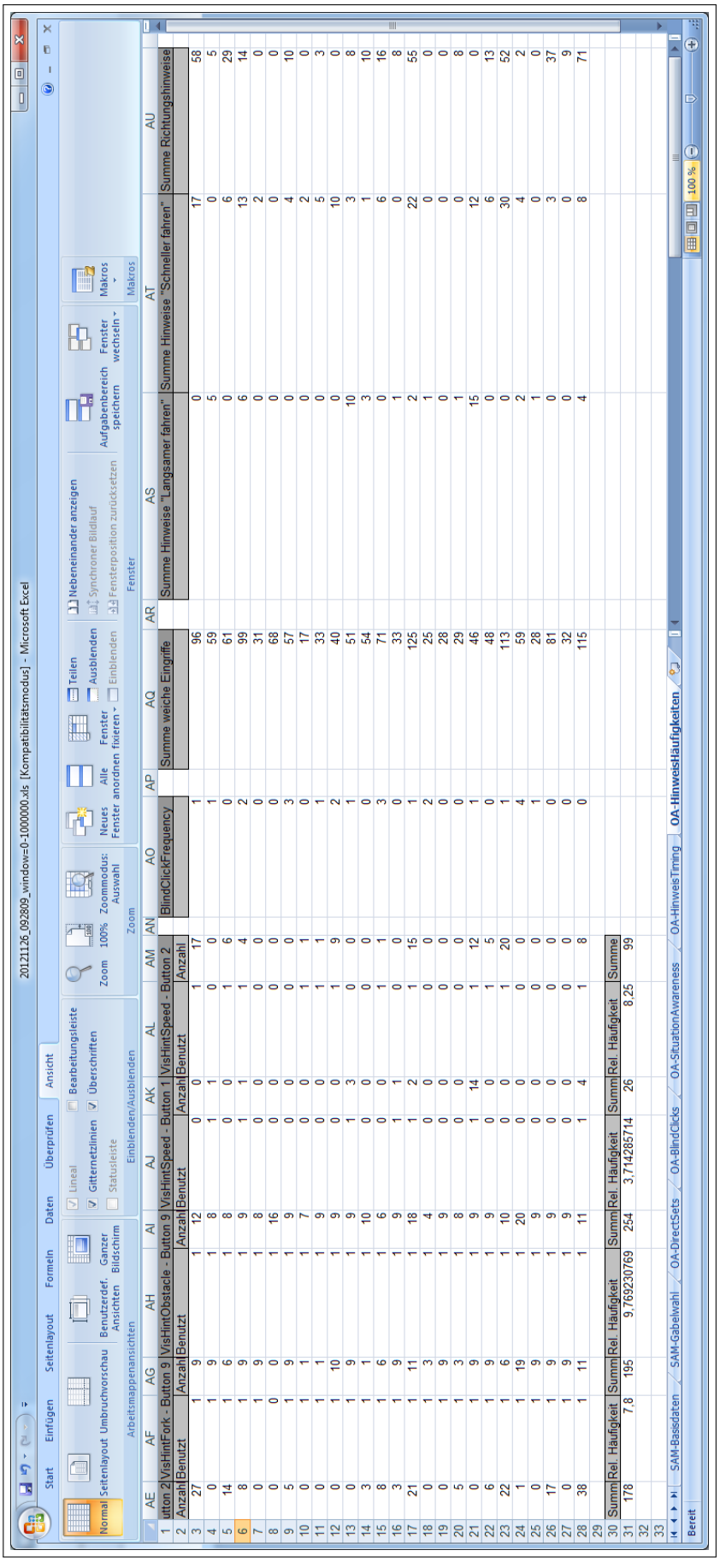


Abbildung 20: Output - Hinweishäufigkeiten - rechte Seite

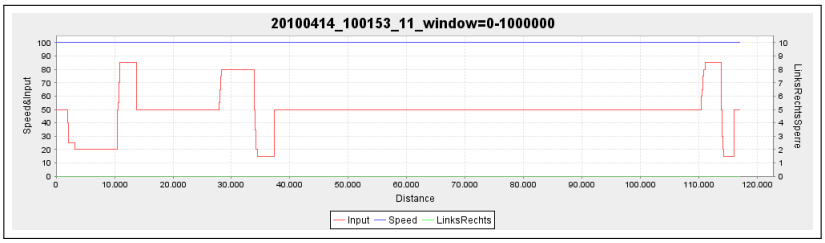


Abbildung 21: Output - Harte Eingriffe(Diagramm 1)

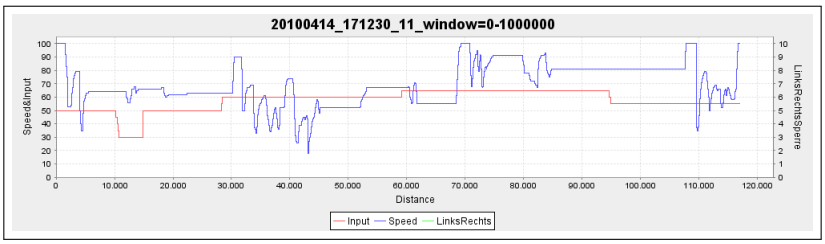


Abbildung 22: Output - Harte Eingriffe(Diagramm 2)

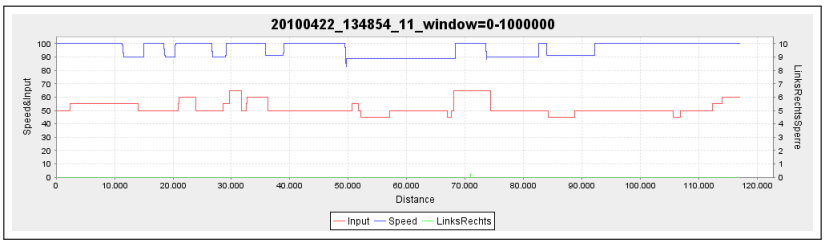


Abbildung 23: Output - Harte Eingriffe(Diagramm 3)

B

KLASSEN-/METHODENBESCHREIBUNG

In diesem Kapitel werden alle Klassen und ihre Methoden dokumentiert (abgesehen von Konstruktoren und Getter- und Setter-Methoden).

B.1 PACKAGE ATEO

B.1.1 *Controller*

Methoden:

- run()

  Typ/Rückgabewert: void

  Beschreibung: Erzeugt LFAData-Objekt und startet Analyse

- analyze()

  Typ/Rückgabewert: void

  Beschreibung: Führt Analyse durch

- getStepHeaderData()

  Typ/Rückgabewert: void

  Beschreibung: Extrahiert statische Versuchsdaten aus Logfile

- createDateString(GregorianCalendar)

  Typ/Rückgabewert: String

  Beschreibung: Generiert formatierten Datums-String aus aktuellem Zeitstempel

- createTimeString(GregorianCalendar)

  Typ/Rückgabewert: String

  Beschreibung: Generiert formatierten Zeit-String aus aktuellem Zeitstempel

- createResultsFolders(String, String)

  Typ/Rückgabewert: void

  Beschreibung: Legt Verzeichnisse für Output an

- setUpOutput()

  Typ/Rückgabewert: void

Beschreibung: Bereitet das Anlegen der Output-Verzeichnisse vor

- setup()

  Typ/Rückgabewert: void

  Beschreibung: Bereitet Analyse vor

- interrupt()

  Typ/Rückgabewert: void

  Beschreibung: Unterbricht den Threadpool-Exekutor

- setupLogFile(int)

  Typ/Rückgabewert: void

  Beschreibung: Bereitet Analyse für ein bestimmtes Logfile vor

- setupWindow(int)

  Typ/Rückgabewert: void

  Beschreibung: Bereitet Analyse für ein bestimmtes Messfenster vor

- getStepSignature(String)

  Typ/Rückgabewert: String

  Beschreibung: Extrahiert Stepnr aus Logfilenamen

- getXMLFileNameFromCSV(String)

  Typ/Rückgabewert: String

  Beschreibung: Extrahiert Konfigurationsfilenamen aus CSV-Logfilenamen

B.1.2 *CSVParser*

Methoden:

- getStringList(String)

  Typ/Rückgabewert: ArrayList<String>

  Beschreibung: Extrahiert Daten aus CSV-Datei

- cleanHeader(String[])

  Typ/Rückgabewert: String[]

  Beschreibung: Bereinigt extrahierte Daten um das letzte Element

- closeCSV()

  Typ/Rückgabewert: void

  Beschreibung: Schließt offene CSV-Datei

B.1.3 *LogfileXMLParser*

Methoden:

- getElements(String)

  Typ/Rückgabewert: String[]

  Beschreibung: Extrahiert Daten aus XML-Logfile

B.1.4 *ProgressIncrementer*

Methoden:

- incCounter()

  Typ/Rückgabewert: void

  Beschreibung: Erhöht internen Zähler und benachrichtigt angemeldete Observer

B.1.5 *RacinglineImageReader*

Methoden:

- readRacingLineImage(String)

  Typ/Rückgabewert: ArrayList<Integer>[]

  Beschreibung: Liest Strecken-Grafiken ein und extrahiert Koordinaten der Ideallinie

B.1.6 *RacinglineXMLBuilder*

Methoden:

- createRacingLineXMLFile(String)

  Typ/Rückgabewert: void

  Beschreibung: Generiert XML-Dateien mit Koordinatendaten der Ideallinie

B.1.7 *StepCSVParser*

Methoden:

- compareHeader2BeanItems()

  Typ/Rückgabewert: boolean

  Beschreibung: Überprüft ob Headervariabelen im CSV-File bekannt und definiert sind

B.1.8 *TilesXML*

Methoden:

- createXMLArray(String[])

  Typ/Rückgabewert: void

  Beschreibung: Erzeugt Liste mit Ideallinien-XML-Dokumenten

B.1.9 *XMLParser*

Methoden:

- getElement(String)

  Typ/Rückgabewert: String

  Beschreibung: Gibt Wert eines Knoten eines XML-Dokuments zurück

- getElement(String, int)

  Typ/Rückgabewert: String

  Beschreibung: Gibt Wert eines Knoten eines XML-Dokuments zurück

B.2 PACKAGE ATEO.CHECKER

B.2.1 *LFARacingLineChecker*

Methoden:

- checkRacingLineImages(String[])

  Typ/Rückgabewert: boolean

  Beschreibung: Überprüft die Existenz aller Streckengrafiken

- checkCorrespondingRacingLineXMLFile(String)

  Typ/Rückgabewert: void

  Beschreibung: Überprüft die Existenz der korrespondierender XML-Dateien zu den Streckengrafiken und erzeugt diese bei Bedarf

B.3 PACKAGE ATEO.LFAITEM

B.3.1 *LFADirectSetPlotter*

Methoden:

- run()

  Typ/Rückgabewert: void

  Beschreibung: Erzeugt Grafikdateien aus den Harten Eingriffen des Operateurs

B.3.2 *LFAGetAudHintTiming*

Methoden:

- run()

  Typ/Rückgabewert: void

  Beschreibung: Berechnet Differenzzeiten des Operateurs zwischen Auswahl eines MWB und geben eines auditiven Hinweises

B.3.3 *LFAGetAverage*

Methoden:

- run()

  Typ/Rückgabewert: void

  Beschreibung: Berechnet den Durchschnitt für eine Variable

B.3.4 *LFAGetBlindClicks*

Methoden:

- run()

  Typ/Rückgabewert: void

  Beschreibung: Bestimmt die Anzahl und die Koordinaten der Blindclicks des Operateurs

B.3.5 *LFAGetChosenBranches*

Methoden:

- run()

  Typ/Rückgabewert: void

  Beschreibung: Bestimmt die Wahl der Abzweigungen bei Gabelungen und berechnet eine Güte der Gabelwahl

- getBranchStartingPoints()

  Typ/Rückgabewert: List<Double>

Beschreibung: Berechnet die Startpunkte von Gabelungen auf der Strecke

- getTiles(LFAStepHeaderData)

  Typ/Rückgabewert: String[]

  Beschreibung: Extrahiert die verwendeten Streckengrafiken einer Fahrt aus Logfile

B.3.6 *LFAGetCollisionCount*

Methoden:

- run()

  Typ/Rückgabewert: void

  Beschreibung: Bestimmt die Anzahl der Kollisionen der MWB

B.3.7 *LFAGetDirectSetDirection*

Methoden:

- run()

  Typ/Rückgabewert: void

  Beschreibung: Bestimmt die Eingriffe des Operateurs bezüglich der Richtungsbeschränkung des Fahrobjekts

B.3.8 *LFAGetDirectSetPower*

Methoden:

- run()

  Typ/Rückgabewert: void

  Beschreibung: Bestimmt die Eingriffe des Operateurs bezüglich der Inputverteilung der MWB-Joysticks

B.3.9 *LFAGetDirectSetSpeed*

Methoden:

- run()

  Typ/Rückgabewert: void

  Beschreibung: Bestimmt die Eingriffe des Operateurs bezüglich der Höchstgeschwindigkeit des Fahrobjekts

B.3.10 *LFAGetError*

Methoden:

- run()

  Typ/Rückgabewert: void

  Beschreibung: Berechnet den Flächenfehler der MWB

- getTiles(LFAStepHeaderData)

  Typ/Rückgabewert: String[]

  Beschreibung: Ermittelt die verwendeten Streckengrafiken einer Fahrt

- calcError(double, double, double, double, double, double)

  Typ/Rückgabewert: double

  Beschreibung: Berechnet den Flächeninahlt eines Trapezes

- function1(double)

  Typ/Rückgabewert: void

  Beschreibung: Berechnet Eckpunkte eines Trapezes auf der Strecke zur Berechnung seines Flächeninhaltes

- function2(double)

  Typ/Rückgabewert: void

  Beschreibung: Berechnet Eckpunkte eines Trapezes auf der Strecke zur Berechnung seines Flächeninhaltes

B.3.11 *LFAGetFrequency*

Methoden:

- run()

  Typ/Rückgabewert: void

  Beschreibung: Ermittelt die Anzahl des Auftretens eines bestimmten Wertes einer Variable

B.3.12 *LFAGetMaxTime*

Methoden:

- run()

  Typ/Rückgabewert: void

  Beschreibung: Ermittelt die maximal mögliche Dauer für das Fahren einer Strecke

B.3.13 *LFAGetSituationAwarenessTiming*

Methoden:

- run()

  Typ/Rückgabewert: void

  Beschreibung: Ermittelt die Reaktionszeit des Operateurs zwischen Auftreten eines Situation Awarenes Events und seiner Reaktion darauf

B.3.14 *LFAGetSum*

Methoden:

- run()

  Typ/Rückgabewert: void

  Beschreibung: Berechnet die Summe von Variablenwerten

B.3.15 *LFAGetTime*

Methoden:

- run()

  Typ/Rückgabewert: void

  Beschreibung: Berechnet die gefahrene Zeit der MWB

B.3.16 *LFAGetTotalDistance*

Methoden:

- run()

  Typ/Rückgabewert: void

  Beschreibung: Ermittelt die Gesamtlänge einer Strecke

B.3.17 *LFAGetVisHintForkTiming*

Methoden:

- run()

  Typ/Rückgabewert: void

  Beschreibung: Ermittelt die Zeit zwischen dem Hinweis des Operateurs auf eine Gabelung und der Sichtbarkeit der Gabelung für die MWB

B.3.18 *LFAGetVisHintObstacleTiming*

Methoden:

- run()

  Typ/Rückgabewert: void

  Beschreibung: Ermittelt die Zeit zwischen dem Hinweis des Operateurs auf ein Hindernis und der Sichtbarkeit des Hindernis für die MWB

B.4 PACKAGE ATEO.GUI

B.4.1 *MainWindow*

Methoden:

- initGUI()

  Typ/Rückgabewert: void

  Beschreibung: Baut das GUI auf

- main(String[])

  Typ/Rückgabewert: void

  Beschreibung: Einstiegspunkt der Software

- addLogFileTableItemButtonWidgetSelected(SelectionEvent)

  Typ/Rückgabewert: void

  Beschreibung: Eventhandler für das Hinzufügen eines Logfiles

- openFileDialog()

  Typ/Rückgabewert: void

  Beschreibung: Öffnet Filedialog zur Auswahl von Logfiles

- removeSelectedLogFileTableItemsButtonWidgetSelected(SelectionEvent)

  Typ/Rückgabewert: void

  Beschreibung: Eventhandler für das Entfernen eines Logfiles

- removeAllLogFileTableItemsButtonWidgetSelected(SelectionEvent)

  Typ/Rückgabewert: void

  Beschreibung: Eventhandler für das Entfernen aller Logfiles (Löschen der ganzen Liste)

- addWindowButtonWidgetSelected(SelectionEvent)

  Typ/Rückgabewert: void

Beschreibung: Eventhandler für das Hinzufügen eines Messfensters

- removeSelectedWindowsButtonWidgetSelected(SelectionEvent)

  Typ/Rückgabewert: void

  Beschreibung: Eventhandler für das Entfernen eines Messfensters

- newLFAMenuItemWidgetSelected(SelectionEvent)

  Typ/Rückgabewert: void

  Beschreibung: Eventhandler für das Öffnen eines Analysedialogs

- statusButtonWidgetSelected(SelectionEvent)

  Typ/Rückgabewert: void

  Beschreibung: Eventhandler für das Betätigen des Statusbuttons - Startet Analyse/Bricht Analyse ab

- contentsMenuItemWidgetSelected(SelectionEvent)

  Typ/Rückgabewert: void

  Beschreibung: Eventhandler für das Abbrechen einer Analyse

- shellWidgetDisposed(DisposeEvent)

  Typ/Rückgabewert: void

  Beschreibung: Beendet die Anwendung

- progressBar1PaintControl(PaintEvent)

  Typ/Rückgabewert: void

  Beschreibung: Eventhandler - aktualisiert den Statusbar

- reset()

  Typ/Rückgabewert: void

  Beschreibung: Setzt Statusbar zurück auf 0

B.4.2 *NewSlidingWindowDialog*

Methoden:

- open()

  Typ/Rückgabewert: void

  Beschreibung: Initialisiert Dialogfenster

- cancelButtonWidgetSelected(SelectionEvent)

  Typ/Rückgabewert: void

Beschreibung: Eventhandler für das Schließen des Dialogfensters

- addSlidingWindowButtonWidgetSelected(SelectionEvent)

  Typ/Rückgabewert: void

  Beschreibung: Eventhandler für das Hinzufügen eines Messfensters zur Liste

- slidingWindowStartTextMouseDown(MouseEvent)

  Typ/Rückgabewert: void

  Beschreibung: Eventhandler - Löscht Inhalt des 'Start'-Textfeldes

- slidingWindowEndTextMouseDown(MouseEvent)

  Typ/Rückgabewert: void

  Beschreibung: Eventhandler - Löscht Inhalt des 'Ende'-Textfeldes

- slidingWindowStartTextModifyText(ModifyEvent)

  Typ/Rückgabewert: void

  Beschreibung: Eventhandler - Überprüft Inhalt des 'Start'-Textfeldes auf Integerwert

- slidingWindowEndTextModifyText(ModifyEvent)

  Typ/Rückgabewert: void

  Beschreibung: Eventhandler - Überprüft Inhalt des 'Ende'-Textfeldes auf Integerwert

B.4.3 *FinishedDialog*

Methoden:

- open()

  Typ/Rückgabewert: void

  Beschreibung: Initialisiert Dialogfenster

- okButtonWidgetSelected(SelectionEvent)

  Typ/Rückgabewert: void

  Beschreibung: Eventhandler für 'OK' - Button; Schließt Dialogfenster

- dialogShellWidgetDisposed(DisposeEvent)

  Typ/Rückgabewert: void

  Beschreibung: Eventhandler für das Schließen des Dialogfensters

B.4.4 *ErrorDialog*

Methoden:

- initGUI()

  Typ/Rückgabewert: void

  Beschreibung: Initialisiert Dialogfenster

- OKButtonActionPerformed(ActionEvent)

  Typ/Rückgabewert: void

  Beschreibung: Eventhandler für 'OK' - Button; Schließt Dialogfenster

B.5 PACKAGE ATEO.OUTPUT

B.5.1 *Outputter*

Methoden:

- setup()

  Typ/Rückgabewert: void

  Beschreibung: Bereitet Generierung eines Outputfiles vor

- buildOutput()

  Typ/Rückgabewert: void

  Beschreibung: Erzeugt nacheinander alle Sheets auf Basis einer Kontextliste und erzeugt Datei

- fillSAMBaseItemSheet(WritableSheet, LFAData)

  Typ/Rückgabewert: void

  Beschreibung: Fügt Daten in Sheet 'SAM-Basisdaten' ein

- fillHintFrequenciesHeaderColumns(WritableSheet, LFAData)

  Typ/Rückgabewert: void

  Beschreibung: Generiert Spaltennamen im Sheet 'OA-Hinweishäufigkeiten'

- fillHintFrequenciesItemColumns(WritableSheet, LFAData)

  Typ/Rückgabewert: void

  Beschreibung: Fügt Daten in Sheet 'OA-Hinweishäufigkeiten' ein

- computeSumOfColumnValues(WritableSheet, LFAData)

  Typ/Rückgabewert: void

Beschreibung: Berechnet Summe aller Spaltenwerte in Sheet 'OA-Hinweishäufigkeiten'

- computeBlindClickFrequencies(WritableSheet, LFAData)

  Typ/Rückgabewert: void

  Beschreibung: Berechnet Anzahl der Blindclicks in Sheet 'OA-Hinweishäufigkeiten'

- computeSumOfButtonClicks(WritableSheet, LFAData)

  Typ/Rückgabewert: void

  Beschreibung: Berechnet Anzahl aller weichen Eingriffe in Sheet 'OA-Hinweishäufigkeiten'

- computeSumOfSpeedHints(WritableSheet, LFAData)

  Typ/Rückgabewert: void

  Beschreibung: Berechnet Anzahl Geschwindigkeitshinweise in Sheet 'OA-Hinweishäufigkeiten'

- computeSumOfDirectionHints(WritableSheet, LFAData)

  Typ/Rückgabewert: void

  Beschreibung: Berechnet Anzahl Richtungsshinweise in Sheet 'OA-Hinweishäufigkeiten'

- fillHintFrequenciesSheet(WritableSheet, LFAData)

  Typ/Rückgabewert: void

  Beschreibung: Ruft nacheinander alle Methoden zum Befüllen 'OA-Hinweishäufigkeiten'-Sheets auf

- fillHintTimingSheet(WritableSheet, LFAData)

  Typ/Rückgabewert: void

  Beschreibung: Fügt Daten in Sheet 'OA-HinweisTiming' ein

- fillSituationAwarenessSheet(WritableSheet, LFAData)

  Typ/Rückgabewert: void

  Beschreibung: Fügt Daten in Sheet 'OA-SituationAwarenes' ein

- fillBlindClicksSheet(WritableSheet, LFAData)

  Typ/Rückgabewert: void

  Beschreibung: Fügt Daten in Sheet 'OA-Blindclicks' ein

- getSortedStepDataKeyArray(LFAData)

  Typ/Rückgabewert: String[]

  Beschreibung: Extrahiert Step-Daten aus Datenstruktur und sortiert diese

- fillDirectSetsSheet(WritableSheet, LFAData)

  Typ/Rückgabewert: void

  Beschreibung: Fügt Daten in Sheet 'OA-DirectSets' ein

- fillSAMBaseItemColumns(WritableSheet, LFAData)

  Typ/Rückgabewert: void

  Beschreibung: Fügt Feature-Daten in Sheet 'SAM-Basisdaten' ein

- createSAMHeaderColumns(WritableSheet, LFAData)

  Typ/Rückgabewert: void

  Beschreibung: Generiert Spalten für statische Versuchsdaten in Sheet 'SAM-Basisdaten'

- fillSAMHeaderColums(WritableSheet, LFAData)

  Typ/Rückgabewert: void

  Beschreibung: Fügt statische Versuchsdaten in Sheet 'SAM-Basisdaten' ein

- getContextTypes(LFAData)

  Typ/Rückgabewert: ArrayList<String>

  Beschreibung: Ermittelt Kontextarten der Features aus Datenstruktur

- filterCurrentWindow()

  Typ/Rückgabewert: LFAData

  Beschreibung: Filtert alle Ergenisse eines Messfensters aus Datenstruktur

- fillBranchesSheet(WritableSheet, LFAData)

  Typ/Rückgabewert: void

  Beschreibung: Fügt Daten in Sheet 'SAM-Gabelwahl' ein

LITERATURVERZEICHNIS

[1] Balzert, Helmut: *Lehrbuch der Software-Technik Bd.1*. Spektrum Akademischer Verlag, 2001

[2] Chikofsky, Elliot j. ; Cross, James H.: Reverse Engeneering and Design Recovery. In: *IEEE Software* 7 (1990), Nr. 1, S. 13–17

[3] Dirlewanger, Werner: *Measurement and Rating of Computer Systems Performance and of Software Efficiency: An Introduction to the ISO/IEC 14756 Method and a Guide to its Application*. Kassel University Press, 2006

[4] Fowler, Martin: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999

[5] Fuhrmann, Esther: *Entwicklung eines GUI für die Konfiguration der Software-Komponente zur Systemprozessüberwachung und -kontrolle in einer psychologischen Versuchsumgebung*. 2010. – Diplomarbeit

[6] Goll, Joachim: *Methoden und Architekturen der Softwaretechnik*. Vieweg+Teubner Verlag, 2011

[7] Gross, Barbara ; Nachtwei, Jens: Assistenzsysteme effizient entwickeln und nutzen - Die Mikrowelt als Methode zur Wissensakquisition für Entwickler und Operateure. In: Grandt, M. (Hrsg.) ; Bauch, A. (Hrsg.): *Cognitive Systems Engineering in der Fahrzeug- und Prozessführung*. 1. Deutsche Gesellschaft f. Luft- u. Raumfahrt, 2006 (DGLR-Bericht 2006/2), S. 75–88

[8] Hampel, Tobias: *Konzeption und Implementation eines Analysetools zur Auswertung von Logfiles einer komplexen, dynamischen Versuchsumgebung*. 2009. – Studienarbeit

[9] ISO/IEC 14756: *Information Technology - Measurement and Rating of Performance of computer-based Software Systems*. 1999

[10] ISO/IEC 9126: *Software Engineering - Product Quality*. 1991

[11] Kesselring, Kai: *Entwicklung einer Softwarekomponente zur Systemprozessüberwachung und -kontrolle in einer psychologischen Versuchsumgebung*. 2009. – Diplomarbeit

[12] Leonhard, Christian: *Fenster zum Prozess: Weiterentwicklung eines Operateursarbeitsplatzes im Projekt Arbeitsteilung Entwickler Operateur ATEO*. 2010. – Studienarbeit

[13] McGregor, John D. ; Sykes, David A.: *A Practical Guide to Testing Objected-Oriented Software*. Addison Wesley, 2001

[14] Nachtwei, Jens: *Design and Evaluation of a Supervisory Control Lab System for Automation Research - A Theoretical and Empirical Contribution to the Discussion on Function Allocation*. 2011. – Dissertation an der Humboldt Universität zu Berlin

[15] Pagel, Bernd-Uwe ; Six, Hans-Werner: *Software Engineering. Band 1: Die Phasen der Softwareentwicklung*. Oldenbourg Wissenschaftsverlag, 1997

[16] Schwarz, Hermann: *Fenster zum Prozess: Ein Operateursarbeitsplatz zur Überwachung und Kontrolle von kooperativem Tracking*. 2009. – Diplomarbeit

[17] Shafranovich, Y.: *RFC 4180: Common Format and MIME Type for Comma-Separated Values (CSV) Files*. October 2005

[18] Sommerville, Ian: *Software Engineering*. Achte Auflage. Pearson Studium, 2007

[19] Vogel, Oliver ; Arnold, Ingo ; Chughtai, Arif ; Ihler, Edmund ; Kehrer, Timo ; Mehlig, Uwe ; Zdun, Uwe: *Software-Architektur: Grundlagen - Konzepte - Praxis*. Zweite Auflage. Spektrum Akademischer Verlag, 2009

[20] Zijlstra, F.R.H.: *Efficiency in Work Behaviour: A Design Approach for Modern Tools*. Delft University Press, Delft, Holland, 1993

SELBSTÄNDIGKEITSERKLÄRUNG

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe. Weiterhin erkläre ich, eine Diplomarbeit in diesem Studiengebiet erstmalig einzureichen.

Berlin, den 26. November 2012 .....