Entwicklung von Komponenten einer IDE für die labAlive-App myApps Webanwendung

Bachelorarbeit

im Rahmen des Studienganges

Technische Informatik und Kommunikationstechnik

Laurence Arthur Simon

Betreuer:

Prof. Dr.-Ing. Erwin Riederer

Tag der Abgabe: 30.06.2023

Universität der Bundeswehr München
ETTI 5 - Institut für Funkkommunikation

Neubiberg, Juni 2023

Abbildung 1: Git: Leo-Labalive-www

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, insbesondere keine anderen als die angegebenen Informationen.

Der Speicherung meiner Bachelorarbeit zum Zweck der Plagiatsprüfung stimme ich zu. Ich versichere, dass die elektronische Version mit der gedruckten Version inhaltlich übereinstimmt.

Neubiberg, den 30.06.2023

Laurence Arthur Simon

Abstract

Diese Arbeit konzentriert sich darauf, eine webbasierte Entwicklungsumgebung für die LabAlive-Webanwendung zu entwickeln. LabAlive bietet ein Simulationstool, das verwendet werden kann, um eine Vielzahl von Kommunikationsschaltungen zu simulieren. Benutzer können auf der LabAlive-Webseite ein Werkzeug verwenden, um ihre eigenen Schaltungen zu entwerfen, zu beschreiben und zu herunterladen. Das ursprüngliche Werkzeug bestand nur aus einer einfachen Textbox, daher konnten Benutzer erst nach dem Herunterladen überprüfen, ob ihr Code semantisch und syntaktisch korrekt war. In dieser Arbeit wurde eine webbasierte Entwicklungsumgebung entwickelt, um die Benutzerfreundlichkeit des Werkzeugs zu verbessern, um dieses Problem zu lösen. Zu Beginn der Arbeit wurde eine umfassende Literaturrecherche durchgeführt, insbesondere in Bezug auf Programmerzeugungssysteme und Parser. Aufgrund dieser Forschung wurde das Projekt als Softwareprojekt geplant und die relevanten Anwendungsfälle für das Werkzeug wurden identifiziert und koordiniert. Auf dieser Grundlage wurde der Prototyp erstellt und die ersten Tests durchgeführt. Die Entwicklung des Prototyps, die Implementierung der Syntaxanalyse und das Syntax Highlighting gehören zu den wichtigsten Ergebnissen der Arbeit. Außerdem wurden Methoden und Algorithmen entwickelt, die die Funktionalität der Anwendung bestimmen. Ein großer Vorteil ist die dynamische Programmierung der Klassen, die es ermöglicht, neue Bauelemente in eine JSON-Datei einzubinden, ohne den bestehenden Code anzupassen. Durch die bereits entwickelten Anwendungsfälle konnte bereits eine funktionierende syntaktische Analyse implementiert werden, die den Benutzer bei der Erkennung von Syntaxfehlern unterstützt. Dies stellt einen wichtigen Fortschritt dar und ermöglicht eine verbesserte Code-Qualität und Fehlererkennung innerhalb der WebIDE. Die Entwicklung der WebIDE konnte innerhalb des vorgegebenen Zeitraums nicht fertiggestellt werden, daher sollte dies in folgenden Arbeiten berücksichtigt und angepasst werden. Es bleibt Raum für zukünftige Arbeiten, um Ergänzungen oder Verbesserungen zu entwickeln. [1][2]

Inhaltsverzeichnis

1 Einleitung . . . 13
1.1 Aufgabenstellung . . . 13

2 Grundlagen . . . 15
2.1 Software . . . 15
2.1.1 IntelliJ . . . 15
2.1.2 Miktex . . . 15
2.1.3 Visual Paradigma . . . 16
2.2 Internetseiten . . . 16
2.2.1 Overleaf . . . 16
2.2.2 JSON Formatter . . . 17
2.3 Programmiersprachen und Dateiformate . . . 17
2.3.1 Java . . . 17
2.3.2 Javascript . . . 17
2.3.3 HTML . . . 17
2.3.4 CSS . . . 18
2.4 LabAlive . . . 18
2.4.1 Die System Logik von LabAlive . . . 19
2.4.2 Der Code von LabAlive . . . 19
2.5 Datenformate . . . 20
2.5.1 JSON . . . 20
2.5.2 XML . . . 20
2.5.3 Annotation . . . 20
2.5.4 Reflection . . . 20
2.5.5 JSON vs. XML . . . 22
2.6 Compiler . . . 22
2.6.1 Interpreter . . . 22
2.6.2 Lexikalische Analyse . . . 23
2.6.3 Automat . . . 23
2.6.4 Parsing . . . 24
2.6.5 Semantische Analyse . . . 24
2.6.6 Parser . . . 24
2.6.7 Top-Down-Parser . . . 25
2.6.8 Unterschied zwischen Top-Down-Parser und Botton-Up-Parser . . . 26

3 Anforderungsspezifikation . . . 27
3.1 Dialog . . . 27
3.2 Anforderungsarten . . . 27
3.2.1 Funktionale Anforderungen . . . 27
3.2.2 Nichtfunktionale Anforderungen . . . 27

3.3 Anwendungsfälle . . . 28
3.3.1 AF-Syntax-Analyse . . . 29
3.3.2 AF-Syntax-Highlighting . . . 31
3.3.3 AF-Semantik-Analyse . . . 32
3.3.4 AF-Error Message . . . 34
3.4 Dialoge . . . 36
3.4.1 Dialog Syntax-Highlighting . . . 36

4 Implementierung . . . 37
4.1 WebIDE . . . 37
4.1.1 Oberflächenstile der IDE . . . 37
4.1.2 CSS Code für die IDE . . . 39
4.1.3 HTML Code für die IDE . . . 41
4.1.4 JS Code für die IDE . . . 42
4.2 Klasse ImportInformationFromWebsite . . . 42
4.2.1 Relevante Funktionen . . . 42
4.3 Klasse Syntax Highlighter . . . 44
4.3.1 Relevante Funktionen . . . 44
4.4 Klasse JSONFileLoader . . . 46
4.4.1 JSON Datei . . . 46
4.4.2 Relevante Funktionen . . . 48
4.5 Klasse Syntax-Analyser . . . 52
4.5.1 Grafische Darstellung des Parsing . . . 52
4.6 Funktionen . . . 53
4.7 Klasse Semantik-Analyser . . . 58
4.7.1 Segmente . . . 58
4.7.2 Grafische Darstellung der Semantikanalyse . . . 58
4.7.3 Bereits entwickelte Funktionen . . . 60

5 Fazit . . . 63
5.1 Ergebnisse . . . 63
5.2 Herausforderungen . . . 64
5.3 Zeitaufwand . . . 64
5.4 Ausblick . . . 65

6 Anhang . . . 67
6.1 Ablaufdiagramme . . . 67
6.2 QuellCode . . . 68
6.2.1 JSONFileLoader . . . 69
6.2.2 ImportInformationFromWebsite . . . 73
6.2.3 IDE.Html . . . 75
6.2.4 Syntax-Analyzer . . . 77
6.2.5 Semantik-Analyzer . . . 80

Tabellenverzeichnis . . . 85

Abbildungsverzeichnis . . . 87

Abkürzungsverzeichnis 89

Literaturverzeichnis 91

Index 93

1 Einleitung

Die vorliegende Bachelorarbeit beschäftigt sich mit der Weiterentwicklung der aktuellen LabAlive-Webanwendung und erweitert die Ergebnisse der Projektarbeit mit dem Thema „Anforderungen und Architektur einer IDE für die LabAlive-MyApps-Webanwendung“. Zum Zeitpunkt der Bearbeitung dieser Arbeit besteht diese aus bloßen Textfeldern, die als Eingabefelder fungieren. Mit den derzeitigen Eingabefeldern können Benutzer mithilfe der von LabAlive definierten Sprache Kommunikationsschaltungssimulationen erstellen, speichern und herunterladen. Das Ziel dieser Arbeit ist es, eine umfassende WebIDE zu entwickeln und testen, um die Benutzerfreundlichkeit der Anwendung zu erhöhen. Um dies zu erreichen, sollen funktionale und nichtfunktionale Anforderungen definiert und eine Reihe von Algorithmen und Methoden entwickelt werden, um die gewünschte Funktionalität der WebIDE zu ermöglichen. Im Einführungsteil der Arbeit wird erklärt, welche Programme und Technologien für die Erstellung der WebIDE verwendet wurden. Daran anschließend werden die Anforderungen diskutiert und deren Entwicklung im Laufe der Bachelorarbeit beschrieben. Im Abschnitt zur Implementierung werden die bereits abgeschlossenen und noch nicht abgeschlossenen Komponenten der WebIDE behandelt. Die Implementierung verschiedener Anwendungsfälle wird anhand von Codebeispielen erläutert. Es wird weiterhin demonstriert, wie die WebIDE die Anforderungen erfüllt und welche Funktionalitäten bereits erfolgreich genutzt wurden. Abschließend wird eine Schlussfolgerung gezogen, die die erreichten Leistungen, die Schwierigkeiten, den Zeitaufwand und einen Ausblick auf zukünftige Arbeiten umfasst. Es wird berücksichtigt, ob die gesteckten Ziele erreicht wurden und welche Verbesserungen oder Erweiterungen für die WebIDE möglich sind.

1.1 Aufgabenstellung

Im Rahmen dieser Bachelorarbeit wird die LabAlive-MyApps-Webanwendung als Ausgangspunkt für die Weiterentwicklung verwendet. Zurzeit besteht die Webanwendung hauptsächlich aus zwei unkomplizierten Eingabefeldern, um die Anwendung zu beschreiben. Das System ist jedoch noch nicht interaktiv genug, um dem Benutzer das bestmögliche Benutzererlebnis zu bieten. Daher soll die Webanwendung um ein spezielles Eingabefeld für Programmiercode und eine Konsolenausgabe erweitert werden, um dem Benutzer vor dem Herunterladen des Codes eine unmittelbare Rückmeldung über die Validität des Codes zu geben. Die Hauptaufgabe dieser Arbeit besteht darin, den Benutzern die Möglichkeit zu geben, Eingabefehler zu erkennen und zu korrigieren, indem die Webanwendung eine interaktive Validierung des Codes durchführt. Wenn der Benutzer beispielsweise Syntaxfehler macht, werden sie erkannt und in der Konsole ausgegeben. Es ermöglicht dem Benutzer, Fehler frühzeitig zu erkennen und zu beheben, ohne den Code herunterzuladen. Dies führt zu einer effizienteren Entwicklungsumgebung und erleichtert die Fehleranalyse und -behebung. Um die Benutzerfreundlichkeit weiter zu verbessern, sollen im Laufe der Arbeit geeignete Funktionalitäten zur Validierung und Syntaxhervorhebung entwickelt werden. Des Weiteren werden Erweiterungen wie Textergänzung eingeführt, um dem Benutzer beim Schreiben des Codes zu helfen. Es umfasst unter anderem die automatische Vervollständigung von Codefragmenten und die Bereitstellung von kontextbezogenen Hinweisen. Dadurch wird der Entwicklungsprozess effektiver und einfacher. Die Arbeit von Daniel Hiriliman stellt eine Grundlage für die Lösung dieser Aufgabe dar. Seine Arbeit liefert nützliche Informationen und

Ergebnisse, die als Grundlage für die Weiterentwicklung der Webanwendung dienen.

2 Grundlagen

Dieses Kapitel gibt einen Überblick über die Forschung und konzentriert sich hauptsächlich auf den Softwareentwicklungsprozess, insbesondere auf die Entwicklung eines Parsers. Es wird der aktuelle Stand anhand der Prozessentwicklung und der notwendigen Werkzeuge beschrieben.

2.1 Software

2.1.1 IntelliJ

IntelliJ IDEA Ultimate ist eine IDE für die Softwareentwicklung. Sie bietet eine Vielzahl von Funktionen und Werkzeugen, die Entwicklern dabei helfen, Codes zu testen, Fehler zu beheben und effektiv zu programmieren. Verschiedene Programmiersprachen wie Java, Kotlin, JavaScript und andere werden von IntelliJ IDEA Ultimate unterstützt. Die IDE verfügt über eine Vielzahl von Funktionen, darunter einen Code-Editor mit intelligenten Code-Vervollständigungsfunktionen, automatischer Refaktorisierung, einer integrierten Versionskontrolle, Debugger, Test- und Profiling-Tools. Entwickler können mit IntelliJ IDEA Ultimate auf eine große Auswahl an Plugins zugreifen, um die IDE an ihre spezifischen Bedürfnisse anzupassen.

Abbildung 2.1: IntelliJ IDEA Ultimate Logo

IntelliJ IDEA wurde als Hauptprogramm für die Entwicklung und Strukturierung des Projekts in der Bachelorarbeit verwendet. Es fungierte als zentrale Plattform für die Verwaltung und Programmierung des Projektcodes.[3]

2.1.2 Miktex

MiKTeX ist eine Software-Distribution, die speziell für die Anforderungen der akademischen und wissenschaftlichen Gemeinschaft im Bereich der Satz- und Dokumentenlayouts entwickelt wurde. Es bietet eine große Auswahl an Tools und Paketen für die Arbeit mit LaTeX, einem weitverbreiteten System zur Erstellung hochwertiger technischer und wissenschaftlicher Dokumente.
Für die Bachelorarbeit wurde dieses Tool hauptsächlich zur Dokumentation und zum Schreiben der Bachelorarbeit genutzt.[4]

Abbildung 2.2: MikeTex Logo

2.1.3 Visual Paradigma

Visual Paradigm ist ein umfassendes UML-Werkzeug, das für die Softwaremodellierung entwickelt wurde. Es ist eine Desktop-Software, die auf Windows, Linux und macOS läuft.Dazu gehören Funktionen wie Diagrammerstellung, Codegenerierung, Reverse Engineering, Versionskontrolle und Dokumentationserstellung. Die Benutzer können mit Visual Paradigm eine Vielzahl von UML-Diagrammtypen wie Klassendiagramme, Aktivitätsdiagramme, Zustandsdiagramme und Sequenzdiagramme erstellen und bearbeiten.

Abbildung 2.3: Visual Paradigm Logo

Visual Paradigm wurde innerhalb des Entwicklungsprozesse für die vollständige Erstellung aller Anwendungsfälle der Anfoderungsspezifikation genutzt.[5]

2.2 Internetseiten

2.2.1 Overleaf

Overleaf ist eine webbasierte Plattform, mit der Benutzer bei der Erstellung und Bearbeitung von LaTeX-Dokumenten zusammenarbeiten können. LaTeX-Dateien können online erstellt und mit anderen zusammen bearbeitet werden. Mit Overleaf können Benutzer ihre LaTeX-Projekte in der Cloud speichern und von verschiedenen Geräten aus darauf zugreifen. Mit integrierten LaTeX-Tools wie Syntaxhervorhebung, Vorlagen, automatischer Kompilierung und Fehlerprüfung bietet es eine benutzerfreundliche Oberfläche.

Abbildung 2.4: Overleaf Logo

2.2.2 JSON Formatter

JSON Formatter ist ein kostenloses Tool, das einem dabei hilft, JSON-Daten zu validieren, zu speichern, zu teilen und zu formatieren. Benutzer können ihre JSON-Daten einfach organisieren und überprüfen, ob sie den JSON -Syntaxregeln entsprechen. Außerdem können formatierte JSON-Daten mit dem Tool gespeichert und geteilt werden. Benutzer können mit JSON-Formattern ihre JSON -Daten übersichtlich gestalten und damit effektiv arbeiten.

Abbildung 2.5: JSON formatter Logo

2.3 Programmiersprachen und Dateiformate

2.3.1 Java

Die Java-Technologie, auch bekannt als Java Technology, ist eine Zusammenstellung von Spezifikationen, die ursprünglich von Sun Microsystems (heute Oracle Corporation) entwickelt wurden. Sie umfasst die Programmiersprache Java sowie verschiedene Laufzeitumgebungen für die Ausführung von Java-Programmen.[6]
Die relevanten Komponenten sind:

- Die Java-Programmiersprache ermöglicht die Formulierung von Programmen in Java.
- Das Java Development Kit (JDK) ist ein Entwicklungsframework, das grundlegende Werkzeuge wie einen Compiler und Bibliotheken enthält.
- Die Java-Laufzeitumgebung (JRE) ist eine standardisierte Softwareplattform, die es ermöglicht, entwickelte Java-Programme auszuführen.

2.3.2 Javascript

JavaScript, auch JS genannt, ist eine weit verbreitete Skriptsprache, die hauptsächlich verwendet wird, um Webanwendungen zu erstellen. Ursprünglich wurde sie entwickelt, um Webbrowsern die Möglichkeit zu geben, dynamisches HTML zu verwenden und die Interaktivität von Websites zu verbessern. JavaScript ermöglicht Entwicklern, Benutzerinteraktionen auf Webseiten zu bewerten und darauf zu reagieren. Es bietet die Möglichkeit, beispielsweise Formulardaten zu überprüfen, Inhalte dynamisch zu aktualisieren oder Benutzeraktionen wie Klicks oder Mausbewegungen zu erkennen. [7]

2.3.3 HTML

Hypertext Markup Language (HTML) ist die grundlegende Struktursprache für das Erstellen von Webseiten im Internet. Es verwendet Markup-Tags, um den Inhalt einer Webseite zu kennzeichnen und zu organisieren. Die Elemente, die auf einer Website angezeigt werden, wie Überschriften, Absätze,

Bilder, Links und Tabellen, werden durch HTML definiert.HTML bietet Entwicklern die Möglichkeit, den Text auf einer Website anzupassen, Bilder hinzuzufügen, Links zu anderen Webseiten zu erstellen und Multimedia-Inhalte wie Videos und Audiodateien einzubetten. [8]

DOM

Das Document Object Model, auch bekannt als DOM, ist eine Programmierschnittstelle, die HTML und XML-Dokumente in Form von strukturierten Bäumen darstellt. Jede DOM-Komponente in einem HTML-Dokument wird als DOM-Element dargestellt. Ein DOM-Element ist eine Kopie von HTML Tags oder einer anderen XML-Markup-Sprache und verfügt über Eigenschaften und Methoden, die es ermöglichen, auf das Element zuzugreifen und es zu ändern.

DIV

Ein häufig verwendetes XML-Element ist ein <div>-Element, das als Container für andere Elemente fungiert. Es erzeugt im XML-Dokument einen rechteckigen Bereich, der angepasst und positioniert werden kann. Auch wenn ein „div“-Element keine spezifische semantische Bedeutung hat, wird es verwendet, um Abschnitte zu gruppieren und das Layout einer Webseite zu organisieren. Es ist in der Lage, mit CSS-Stilen zu gestalten und eine Vielzahl von Eigenschaften wie Hintergrundfarbe, Rand, Abstände und Positionierung zu haben. Es kann auch JavaScript-Code verwenden, um das Verhalten der Webseite zu kontrollieren.

2.3.4 CSS

Cascading Style Sheets (CSS) ist eine Sprache, die zur Gestaltung und Formatierung von Webseiten verwendet wird. Entwickler können das Äußere einer Webseite mit CSS anpassen, indem sie verschiedene Stile und Eigenschaften auf HTML-Elemente anwenden. CSS verwendet Selektoren, um bestimmte Elemente und Stile für eine Webseiten auszuwählen. Funktionen wie Farben, Schriftarten, Abstände, Ausrichtungen und Hintergrundbilder sind dabei enthalten. [9]

2.4 LabAlive

Im Rahmen des LabAlive-Projekts [10] werden Simulationsumgebungen für Kommunikationstechnik entwickelt. Im Folgenden wird eine detaillierte Erklärung der Logik und Funktionalität dieser Umgebungen gegeben. Es gibt zwei Teile: die reine Logik und die programmatische Implementierung dieser Logik in Java. Die Simulationsumgebungen im LabAlive-Projekt basieren auf detaillierter Logik, die sowohl fachlich definiert als auch programmatisch in Java umgesetzt wird, um ihre Funktionalität zu gewährleisten. Dadurch können realistische und präzise Simulationen von Schaltungen der Kommunikationstechnik erstellt werden.

2.4.1 Die System Logik von LabAlive

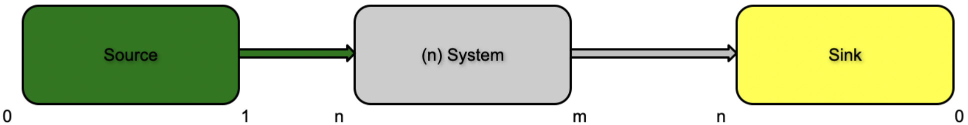


Abbildung 2.6: labAlive Code-Aufbau Systeme

Systeme können durch Connectoren („-“) miteinander verbunden werden, um die Schaltungen umzusetzen. Abbildung 2.6 zeigt eine schematische Darstellung einer solchen Schaltung. Startend wird ein System verwendet, das keine Eingänge hat, sondern lediglich einen oder mehrere Ausgänge (wie eine Quelle). Systeme mit Eingängen und Ausgängen befinden sich in der Mitte der Schaltung. Es ist möglich, am Ende der Schaltung entweder ein System ohne Ausgang zu haben oder automatisch eine Senke zu verwenden.
Die syntaktische Überprüfung der angegebenen Systeme ist der erste Schritt bei der Erstellung der Schaltung. Im zweiten Schritt erfolgt eine semantische Überprüfung, bei der die Bedeutung und Funktionalität der Systeme überprüft werden. Ein System besteht aus einer Klasse und ihren Konstruktoren oder Methoden, die weitere Details und Funktionalität bereitstellen. In den folgenden Abschnitten wird darauf näher eingegangen. Diese Methode stellt sicher, dass die Schaltungen sowohl syntaktisch als auch semantisch korrekt aufgebaut sind und ihre gewünschten Funktionen erfüllen.

2.4.2 Der Code von LabAlive

Der Code der LabAlive-Anwendung folgt einem spezifischen Format und einer festgelegten Struktur, die in den folgenden Abschnitten detailliert beschrieben wird. Dies umfasst die Verwendung bestimmter Klassen, Methoden und Parameter, um die gewünschten Funktionen der Anwendung zu implementieren.

```
1        SineGenerator Amplitude(2.0) - lowpass - sink
```

Diese Zeile beschreibt einen Signal Generator, der ein Sinus Signal mit der Amplitude von 2 V erstellt. Am Anfang steht die Klasse, gefolgt von Methoden, welchen Werte übergeben werden. Zum Verbinden von zwei Systemen wird ein “Wiring” genutzt, hierfür wird an den SineGenerator ein Filter angeschlossen um das Signal zu filtern und die Darstellung zu verändern. Als letztes Element folgt „sink“, dieses wird aber vom System, falls nicht vorhanden, selbständig ergänzt. Um eine Direktanzeige zu bekommen, benötigt man noch ein „scope show“. Als einfaches Beispiel dient folgender Code:

```
1        SineGenerator Amplitude(2.0) - lowpass - sink
2        scope show
```

Man kann aber auch etwas komplexere Schaltungen erzeugen und modifizieren. Folgendes Beispiel zeigt eine Möglichkeit der Modifizierung:

```
1                sine-gain(1)-lowpass(4e3)-sink1
2                sine-gain2(1)-lowpass2(4e3)-sink2
```

Innerhalb dieses Codes wird eine Signalverarbeitung auf zwei Ebenen gezeigt. Sie werden beide getrennt verstärkt durch „gain“ und durch eine Tiepassfilterung „lowpass“ bearbeitet. Zum Schluss gehen sie in zwei verschiedene sinks.

Wie innerhalb von der Systemlogik der LabAlive-Anwendung beschrieben, besteht ein System wie in Abbildung 2.7 aus einer Klasse und Konstruktoren oder Methoden, die von den entsprechenden Parametern aufgerufen werden.

Abbildung 2.7: LabAlive Code-Aufbau Klassen

2.5 Datenformate

2.5.1 JSON

Die Abkürzung JSON bezieht sich auf „JavaScript Object Notation“ und ist ein kompaktes Datenformat, das verwendet wird, um strukturierte Daten darzustellen. Es ist leicht zu lesen und zu schreiben und wird hauptsächlich für den Datenaustausch in Webanwendungen verwendet. JSON verwendet eine einfache Syntax, die aus Paaren von Schlüsselwerten besteht. Es kann Daten in Form von Objekten, Arrays und einfachen Datentypen wie Strings, Zahlen und Booleschen Werten darstellen.[11]

2.5.2 XML

XML ist eine weit verbreitete Auszeichnungssprache zur Darstellung und Strukturierung von Daten und steht für „Extensible Markup Language“. XML ermöglicht eine einheitliche Organisation und Formatierung von Daten in einer hierarchischen Struktur. XML ermöglicht die Darstellung von Daten auf eine plattformunabhängige und maschinenlesbare Weise. [12]

2.5.3 Annotation

Mithilfe von Annotationen werden beispielsweise Methoden annotiert. Das einfachste Beispiel ist „Override“. Man kann im Code ein Interface für eine individuelle Annotation erstellen. Die Funktionen füllen immer eine gleichnamige Variable aus, aus der diese Informationen später geladen werden können. Die eigene Annotation kann durch das Einstellen von Zielen (@Target) angepasst werden, sodass sie beispielsweise nur auf Methoden gesetzt werden kann. Man kann mit Retention festlegen, zu welchem Zeitpunkt die Anmerkung geladen werden soll. Es ist erlaubt, diese nur als Zeichen zu verwenden. Da auf die Laufzeitannotation zugegriffen wird, benötigen wir sie in diesem Projekt. Es wird die Speicherung auf die Dauer besetzt. [13]

2.5.4 Reflection

Java Reflection[14] ist eine Funktionalität, die seit Java Version 8 stabil integriert ist und es ermöglicht, während der Laufzeit auf Klassen des Packages oder den angegebenen Pfad zuzugreifen, ohne diese als eigenständige Objekte zu erstellen. Ein klarer Vorteil besteht darin, dass zur Laufzeit weniger Speicherplatz benötigt wird. Auch wenn nur wenige Objekte gleichzeitig vorhanden sind, wäre der

Speicherbedarf insgesamt (Stand 23.06.23) um die 1.700 Klassen höher. In diesem Projekt wird Java-Reflection allerdings nicht hauptsächlich wegen dieser Punkten genutzt. Beginnen wir mit den negativen Argumenten vor dem Hauptargument für Reflection: Der erste Aspekt ist der Grund, warum Reflection für viele Anwendungen nicht verwendet wird. Da Reflection während der Laufzeit zusätzlichen Code nachladen kann, ist eine Optimierung durch den Compiler nicht mehr durch Java Reflection möglich. In bedeutenden Programmen führt dies zu einer Verringerung der Leistung. Der zweite negative Aspekt betrifft die Cybersicherheit. Hier wird das Prinzip des „Information Hiding“ gebrochen. Dadurch kann auf jede Klassen-Variable oder Methode zugegriffen werden, auch wenn sie „private“ ist. Nachdem die Vor- und Nachteile berücksichtigt wurden, wurde die Entscheidung hauptsächlich aufgrund der Funktionalität von Java Reflection getroffen. Mit Java Reflections können die Methodennamen, ihre Attribute und der Rückgabetyp problemlos als String gespeichert werden. Indem die Funktionalität vom Server abgekoppelt wird, werden die negativen Punkte hier abgeschwächt. Der Administrator führt den Code händisch aus da der Code nicht im Source Code enthalten ist, der beim Starten des Servers geladen wird. Es gibt keine funktionalen Verbindungen vom Server zu diesem Code mit Java Reflections, da die Ergebnisse über eine JSON-Datei an den User gesendet werden. Da das Projekt komplett auf dem Server liegt und nur diejenigen darauf zugreifen können, die bereits in dem Projekt programmieren, ist das Fehlen des „Information Hiding“ und somit der unbegrenzte Zugang zu allen Methoden und Attributen ebenfalls kein Problem. Außerdem sind in der Klasse die Verfahren markiert, die dem Benutzer über die Datei übermittelt werden. Daher erhält der Benutzer nicht den vollständigen Zugang zu allen verfügbaren Methoden, sondern nur die, die zuvor ausgewählt wurden.

2.5.5 JSON vs. XML

Im Vergleich der Dateiformate sind verschiedene Aspekte für unseren Anwendungsfall relevant. Der erste wichtige Punkt für uns ist die Lesbarkeit und die Möglichkeit, die Datei auch ohne vorgeschriebene Funktionen einfach zu bearbeiten und zu ergänzen. JSON bietet hierbei eine einfache und lesbare Struktur durch eine klare und kompakte Syntax, die leicht verständlich und lesbar ist. Hingegen basiert XML auf umfangreichen Tags und Verschachtelungen und ist dadurch komplexer. JSON ermöglicht darüber hinaus eine direkte Integration mit JavaScript, was bedeutet, dass die Entwicklung mit JSON in JavaScript das Einbinden in Skripte vereinfacht und effizienter gestaltet.

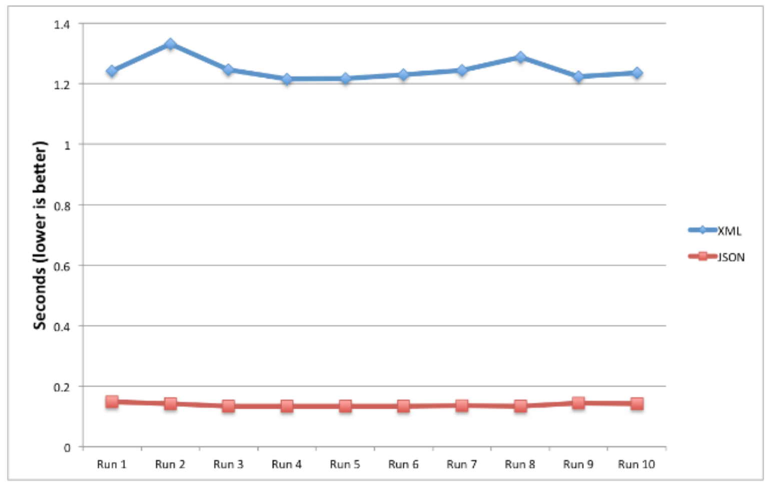


Abbildung 2.8: JSON APIs compared to XML APIs [15]

Dadurch ist auch die Datenübertragung effizienter gestaltbar, da das JSON-Format kompakt ist und dadurch die Datenübertragungszeiten und der Speicherbedarf geringer ausfallen als bei XML. XML hat aufgrund seiner umfangreicheren Struktur Tags und dadurch eine größere Datenübertragungszeit hat. JSON ist auch innerhalb der modernen Webtechnologie und apis gut unterstützt. Es ist daher das bevorzugte Format für den Austausch von Daten zwischen Client und Server in vielen Webanwendungen.

2.6 Compiler

2.6.1 Interpreter

Ein Interpreter setzt die Anweisungen im Quelltext direkt um, im Gegensatz zu einem Compiler erzeugt er keinen Zieltext. Oftmals wird der Quelltext ähnlich wie bei einem Compiler aufbereitet. Im Gegensatz zur Zielcode-Erzeugung unterscheiden sich die Phasen eines Interpreters, wie in Abbildung 2.9 dargestellt, lediglich in der Interpretation. Der Interpreter stellt auch in der Regel eine Laufzeitumgebung zur Verfügung.
"Der Vorteil von Interpretern ist die direkte Ausführung eines Quelltextes und die damit verbundenen einfacheren Testmöglichkeiten. Darüber hinaus sind Interpreter unabhängigkeit von der Maschine: Sofern es Interpreter für unterschiedliche Plattformen gibt, können Programme plattform-übergreifend genutzt werden."[16, S. 6]. Um den Code schrittweise auszuführen, arbeitet der Interpreter dann mit dem abstrakten Syntaxbaum. Die Anweisungen werden interpretiert und in Maschinenbefehle umgewandelt, die der Computer ausführen kann. Der Interpreter liest und führt den Code Zeile für Zeile in Echtzeit aus.

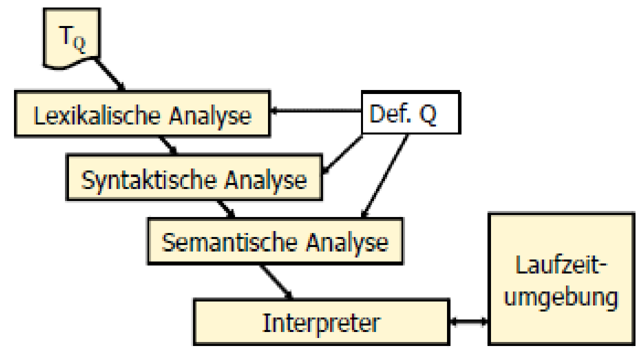


Abbildung 2.9: Die Phase eines Interpreters [16, S.6]

[17] [1]

2.6.2 Lexikalische Analyse

Die lexikalische Analyse ist der Schritt im Compiler- und Interpreterprozess. Er konzentriert sich darauf, den Quellcode in einzelne Token oder lexikalische Einheiten zu zerlegen. In der Regel wird dies mit einem Lexer, einem lexikalischen Scanner, durchgeführt. Ein Programmteil, der für die lexikalische Analyse verantwortlich ist, wird als Lexer oder Scanner bezeichnet. Er findet die Token im Quellcode und sortiert sie nach ihrer Art. Darüber hinaus werden unnötige Leerzeichen, Kommentare und andere unerlässliche Informationen übersprungen. Diese Token können verschiedene wichtige Codesbausteine wie Schlüsselwörter, Identifikatoren, Operatoren, Zahlen und Symbole darstellen. In der Regel umfasst der lexikalische Analyseprozess das Lesen von Zeichen im Quellcode und die Gruppierung der gelesenen Zeichen in Token gemäß festgelegten Regeln. Diese Regeln werden als reguläre Ausdrücke oder endgültige Automaten dargestellt. Da die lexikalische Analyse den Grundstein für die weitere Analyse und Verarbeitung des Quellcodes legt, spielt sie insgesamt eine wichtige Rolle im Compiler und Interpreterprozess. Sie ermöglicht eine effektive Verarbeitung großer Codebasen und hilft dabei, Fehler und Probleme im Quellcode frühzeitig zu erkennen.

2.6.3 Automat

Ein Automat ist eine Vorrichtung oder ein abstraktes Modell, das Zustände und Übergänge darstellt. Er wird verwendet, um die Funktionalität oder das Verhalten eines Systems zu beschreiben. Systeme, die Zustände durchlaufen und auf Eingaben reagieren, werden als endliche Automaten bezeichnet. Sie bestehten aus einem Startzustand, einer Menge von Endzuständen, einer Menge von Zuständen und einer Übergangsfunktion, die den Übergang zwischen Zuständen auf der Grundlage der Eingabezeichen und vorangegangenen Zustand steuert. Ein endlicher Automat kann in zwei Kategorien unterteilt werden: deterministisch und nichtdeterministisch. Ein deterministischer endlicher Automat hat einen genauen Übergang für jeden Zustand und jedes Eingabezeichen. Im Gegensatz dazu kann es bei einem nichtdeterministischen endlichen Automaten mehrere mögliche Übergänge geben, was zu einer gewissen Unsicherheit führt. Der abstrakte Automat befindet sich innerhalb der Endautomaten, die die Modellierung und Analyse komplexer Systeme und Prozesse ermöglichen. Sie können zusätzliche Funktionen und Eigenschaften haben, die über die grundlegenden Merkmale eines endlichen Automaten hinausgehen, wie Speicher oder eine erweiterte Ausdrucksfähigkeit. Viele Bereiche der Informatik und Mathematik verwenden abstrakte Automaten, darunter Sprachverarbeitung und Compilerbau.

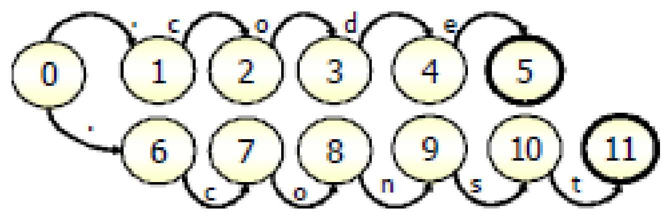


Abbildung 2.10: Beispiel eines endlichen Automaten [16, S.36]

2.6.4 Parsing

Parsing, auch bekannt als syntaktische Analyse, bezeichnet den Prozess der Analyse einer Zeichenkette oder eines Textes gemäß bestimmten Grammatik- oder Syntaxregeln. Um die Struktur und Bedeutung der Eingabe zu verstehen, muss sie in kleinere, nützlichere Einheiten aufgeteilt werden. Beim Parsing wird die Eingabe, die normalerweise eine Abfolge von Zeichen oder Token ist, in eine Datenstruktur umgewandelt, die als Parsebaum oder Syntaxbaum bekannt ist. Der Parsebaum zeigt die hierarchische Struktur der Eingabe gemäß den vorgegebenen Grammatikregeln. Der Parser kann Syntaxfehler im Quellcode während des Parsingprozesses finden. In diesem Fall wird eine Fehlermeldung erstellt, um den Entwickler darauf hinzuweisen, dass ein Fehler vorhanden ist und um Informationen über die Position des Fehlers im Quellcode bereitzustellen. Es gibt verschiedene Methoden zum Parsing, einschließlich Top-Down- und Bottom-Up-Parsing. Wenn es um Parsing geht, ist ein Interpreter normalerweise mit einem Parser verbunden. Der Parser untersucht den Quellcode und erzeugt eine interne Darstellung. Diese wird vom Interpreter genutzt, um die Anweisungen auszuführen. [17]

2.6.5 Semantische Analyse

Die syntaktische Analyse überprüft die korrekte Struktur des Quellcodes, während die semantische Analyse die Bedeutung und logische Konsistenz des Codes untersucht. Das Hauptziel der semantischen Analyse ist sicherzustellen, dass der Code die semantischen Regeln und Bedingungen der Programmiersprache erfüllt. Dies umfasst die Überprüfung von Typen, die Behandlung von Deklarationen und die Sicherstellung, dass Ausdrücke und Anweisungen korrekt sind. Der abstrakte Syntaxbaum (AST) aus der vorherigen syntaktischen Analyse wird während der semantischen Analyse verwendet. Die Struktur und der Kontext des Codes sind im AST enthalten. Diese Informationen werden vom semantischen Analyseprozess verwendet, um sicherzustellen, dass Variablen, Funktionen und andere Elemente des Codes korrekt verwendet werden. Ein wesentlicher Bestandteil der semantischen Analyse ist die Typanalyse. Es wird garantiert, dass Ausdrücke und Operationen kompatiblen Typen zugeordnet werden und dass keine inkonsistenten oder undefinierten Verwendungen von Variablen oder Funktionen vorliegen. Die semantische Analyse kann auch Fehler wie unbelegte Variablen, nicht erreichbarer Code oder unzulässige Typumwandlungen finden. Insgesamt ist die semantische Analyse ein wichtiger Schritt, um sicherzustellen, dass der Code logische Regeln und richtige Bedeutungen erfüllt. Sie hilft dabei, Fehler und Unstimmigkeiten zu erkennen und eine Grundlage für die effiziente Ausführung des Codes zu schaffen.

2.6.6 Parser

Der Parser ist ein wesentlicher Bestandteil des Interpreters. Der Parser untersucht den Quellcode und erzeugt eine interne Repräsentation, die der Interpreter verstehen kann. Der Parser untersucht

die Syntax des Codes und schafft eine Struktur namens abstrakter Syntaxbaum. Der Interpreter kann den Code mithilfe des abstrakten Syntaxbaums in seine einzelnen Bestandteile aufschlüsseln und die entsprechenden Aktionen ausführen. Um den Code schrittweise auszuführen, arbeitet der Interpreter dann mit dem abstrakten Syntaxbaum. Die Anweisungen werden interpretiert und in Maschinenbefehle umgewandelt, die der Computer ausführen kann. Der Interpreter liest und führt den Code Zeile für Zeile in Echtzeit aus.

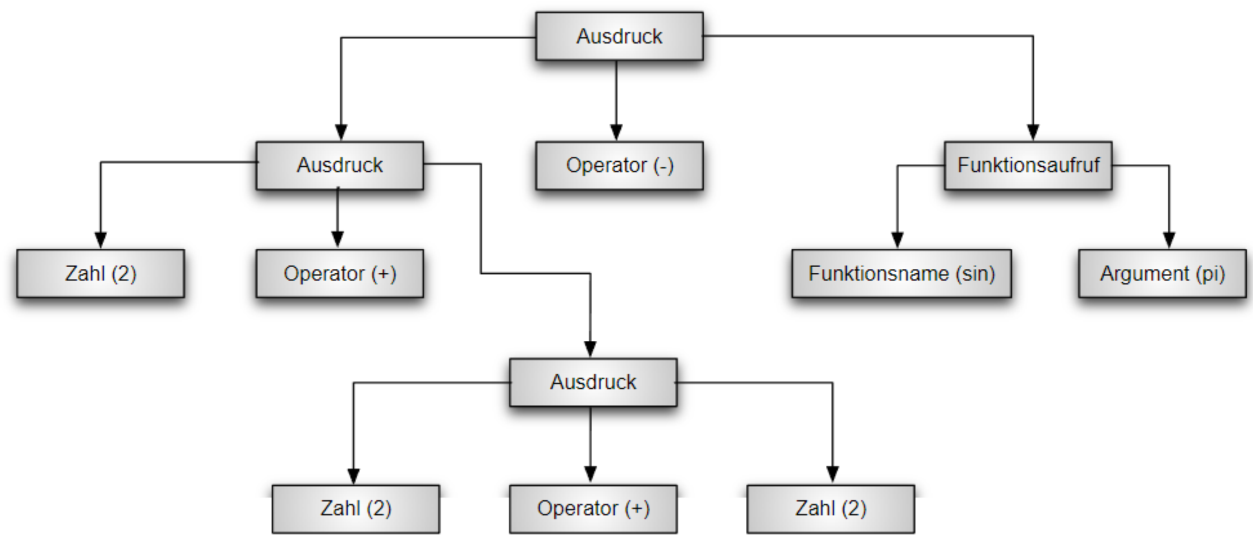


Abbildung 2.11: Parser Organigram [2]

In der Abbildung 2.11 wird Schritt für Schritt gezeigt, wie ein Parser aussehen kann. In der Mitte des ersten Schritts wird der Lexer verwendet, um die Eingabedaten, die als einfache Zeichenkette erscheinen, in Token zu zerlegen. Da die Zerlegung in Token einer regulären Grammatik folgt, wird in der Regel ein endlicher Automat verwendet. In der zweiten Ebene kann man das Parsing oder die Syntaxanalyse erkennen. In der Regel wird dies durch einen abstrakten Automaten durchgeführt, der sich um die Grammatik der Eingabe kümmert. Er führt eine syntaktische Analyse der Eingangsdaten durch und erzeugt einen Ableitungsbaum oder auch Parserbaum. Diese unterstützen die semantische Analyse in der vorletzten Ebene. Er bestimmt die Bedeutung der Eingabe. Die Ausführung ist der letzte Schritt. Dies führt entweder zur Erstellung von Code in einem Compiler oder zur Ausführung durch eine Interpreter.

2.6.7 Top-Down-Parser

Ein Top-Down-Parser ist eine Art Parser, der verwendet wird, um eine Eingabesequenz von Token oder Symbolen gemäß einer bestimmten Grammatik zu analysieren. Der Parser verwendet rekursive Grammatikregeln, um die Eingabe in der vorgegebenen Reihenfolge zu untersuchen, indem er mit dem Startsymbol der Grammatik beginnt. Der Top-Down-Parser verwendet die Vorwärtsanalyse, um die Eingabe durch Ableitungen zu erzeugen, die von oben (Startsymbol) nach unten (Eingabe) verlaufen. Dazu wählt der Parser auf der Grundlage des aktuellen Eingabesymbols eine geeignete Regel aus und versucht, die Regelproduktion zu verwenden, um die Eingabe zu analysieren. Wenn der Parser ein Nichtterminalsymbol entdeckt, erweitert er es, bis er schließlich Terminalsymbole entdeckt, die der Eingabe entsprechen. Der Hauptvorteil eines Top-Down-Parsers besteht darin, dass er

einfacher zu lesen ist und der Struktur der Grammatik natürlicher folgt. Auch für LL(k)-Grammatiken (Links-Nach-Rechts, Linksableitung mit k-Symbolen Vorausschau) funktioniert es gut. Jedoch kann er Schwierigkeiten mit mehrdeutigen oder linksrekursiven Grammatiken haben.[1]

```
expression = expression "+" term | term.
```

Abbildung 2.12: LL(k) und Syntaxbaum eines Top-Down-Parsers [1, S.54]

2.6.8 Unterschied zwischen Top-Down-Parser und Botton-Up-Parser

Ein Top-Down-Parser beginnt mit dem Startsymbol der Grammatik und versucht, die Eingabe von oben nach unten zu analysieren, indem er die Grammatikregeln rekursiv anwendet. Ein Bottom-Up-Parser hingegen beginnt mit der Eingabe und versucht, sie von unten nach oben in die Grammatikregeln zu reduzieren. Teilsequenzen der Eingabe, die zu Nichtterminalsymbolen der Grammatik passen, werden vom Bottom-Up-Parser erkannt und durch das entsprechende Nichtterminalsymbol ersetzt. Bis das Startsymbol erscheint, wird der Vorgang wiederholt.

3 Anforderungsspezifikation

Innerhalb der Anforderungsspezfikation wird auf einen der wichtigsten Teile der Softwareentwicklung eingegangen und zwar die Anforderungen an ein System klar und präzise festzulegen. Indem sie die gewünschten Funktionalitäten, Leistungsmerkmale und Einschränkungen definiert, dient sie als Grundlage für die Entwicklung und Umsetzung eines Projekts. Eine Anforderungsspezifikation beschreibt, was das System erreichen oder bereitstellen soll, und legt die Kriterien fest, anhand derer der Projekterfolg gemessen wird. Die Grundlage für eine erfolgreiche Entwicklung und Implementierung wird durch eine klare und umfassende Anforderungsspezifikation geschaffen.

3.1 Dialog

Der Dialog im Rahmen der Anforderungsspezifikation ist ein wesentlicher Schritt, um sicherzustellen, dass die Anforderungen klar, umsetzbar und verständlich sind. Es ist die Darstellung der zugrunde liegenden Regeln und Konzepte. Dieses ist in grafischer Form umsetzbar.

3.2 Anforderungsarten

3.2.1 Funktionale Anforderungen

"Beschreiben die Funktionalitäten des Systems bzw. der Dienste, die das System leisten sollte" [18, S.10]. Es werden bestimmte Handlungen oder Prozesse beschrieben, die das System ausführen soll, wie beispielsweise die Verarbeitung von Eingaben, Berechnungen oder die Anzeige von Informationen. Die gewünschten Ergebnisse oder Outputs, die das System liefern soll, werden durch funktionale Anforderungen definiert.

3.2.2 Nichtfunktionale Anforderungen

Nichtfunktionale Anforderungen sind Qualitäts- und Eigenschaften des Systems, die nicht direkt mit der Funktionalität selbst zusammenhängen. Sie bestimmen, wie das System bestimmte Anforderungen erfüllen oder bestimmte Eigenschaften haben soll. Nichtfunktionale Anforderungen können verschiedene Aspekte umfassen, wie beispielsweise:

- Sicherheitsanforderungen an Zugriffskontrolle, Datenschutz, Verschlüsselung oder Widerstandsfähigkeit gegenüber Angriffen.

  [18, S.11].

- Wartbarkeit: Modularität, Erweiterbarkeit, Testbarkeit oder Systemdokumentation, um zukünftige Änderungen oder Aktualisierungen zu erleichtern.

  [18, S.13].

3.3 Anwendungsfälle

Ein Anwendungsfall, auch als Use Case bezeichnet, ist eine Beschreibung einer bestimmten Interaktion zwischen einem Benutzer und einem System. Er beschreibt normalerweise, wie der Benutzer das System in bestimmten Situationen nutzt, um ein bestimmtes Ziel zu erreichen. Daher bezeichnet ein Anwendungsfall eine spezifische Funktionalität oder einen spezifischen Ablauf, den das System unterstützt. Anwendungsfälle gehören zu den Funktionalen Anforderungen und bestehen normalerweise aus verschiedenen Komponenten, die in einer prägnanten und strukturierten Weise dokumentiert werden. Das erste Element eines AF ist der Name, der eine aussagekräftige Bezeichnung haben sollte, um den Zweck oder das Ziel des AF zu beschreiben. Wer mit dem System interagiert, wird als Akteur / Akteure bezeichnet. Die Vorbedingung beschreibt die Voraussetzungen, die erfüllt sein müssen, bevor der Anwendungsfall überhaupt erst ausgeführt werden kann. Darauf folgt der Auslöser, der beschreibt welche Aktion oder welches Ergebnis zur Ausführung des Anwendungsfall führt. Als letztes Element ist der Ablauf, eine detaillierte Schritt für Schritt Beschreibung der Interaktion zwischen dem Akteur und dem System, um das „Ziel“ des AF zu erreichen. Zum Abschluss wird eine Nachbedingung definiert, die die Zustände oder Ergebnisse beschreibt, die nach erfolgreicher Durchführung des Anwendungsfalls erreicht werden. Zusammenfassend sollen Anwendungsfälle verwendet werden, um die Funktionalität und das Verhalten eines Systems zu definieren und zu verstehen. Sie unterstützen die Entwickler bei der Betrachtung des Systems aus der Sicht des Benutzers und stellen sicher, dass alle wichtigen Interaktionen und Funktionalitäten abgedeckt sind.
Die folgende Tabelle zeigt eine Übersicht der entwickelten Anwendungsfälle für die Web-IDE sowie deren Prioritäten. Die Priorität ist umgekehrt proportional zum Wert, hier nach entspricht ein niedriger Wert einer höheren Priorität und ein hoher Wert einer niedrigeren Priorität.

Nr	Anwendungsfall	Priorität
06_01	Debugging	3
06_02	Syntax-Analyse	1
06_03	Semantik-Analyse	1
06_04	Fehlerhinweis	2
06_05	Korrekturvorschlag	3
06_06	Autovervollständigung	2
06_07	Informationshinweis	3
06_08	Syntax-Hervorhebung	1
06_09	Hilfe / Aufbauhilfe	4

Tabelle 3.1: Anwendungsfälle

3.3.1 AF-Syntax-Analyse

Akteure: Der Benutzer oder das System, das den Code eingibt.
Kurze Beschreibung: Die Syntaxanalyse ist der Prozess, bei dem der eingegebene Code auf seine syntaktische Korrektheit überprüft wird. Dabei werden die grammatikalischen Regeln und Strukturen der Programmiersprache angewendet, um sicherzustellen, dass der Code die richtige Syntax aufweist.
Vorbedingungen: Der Benutzer hat den Code eingegeben oder stellt den Code bereit, der einer Syntaxanalyse unterzogen werden soll.
Auslöser: Der Benutzer initiiert die Codeanalyse, indem er einen entsprechenden Button betätigt, oder das System startet die Analyse automatisch nach einem bestimmten Zeitintervall.

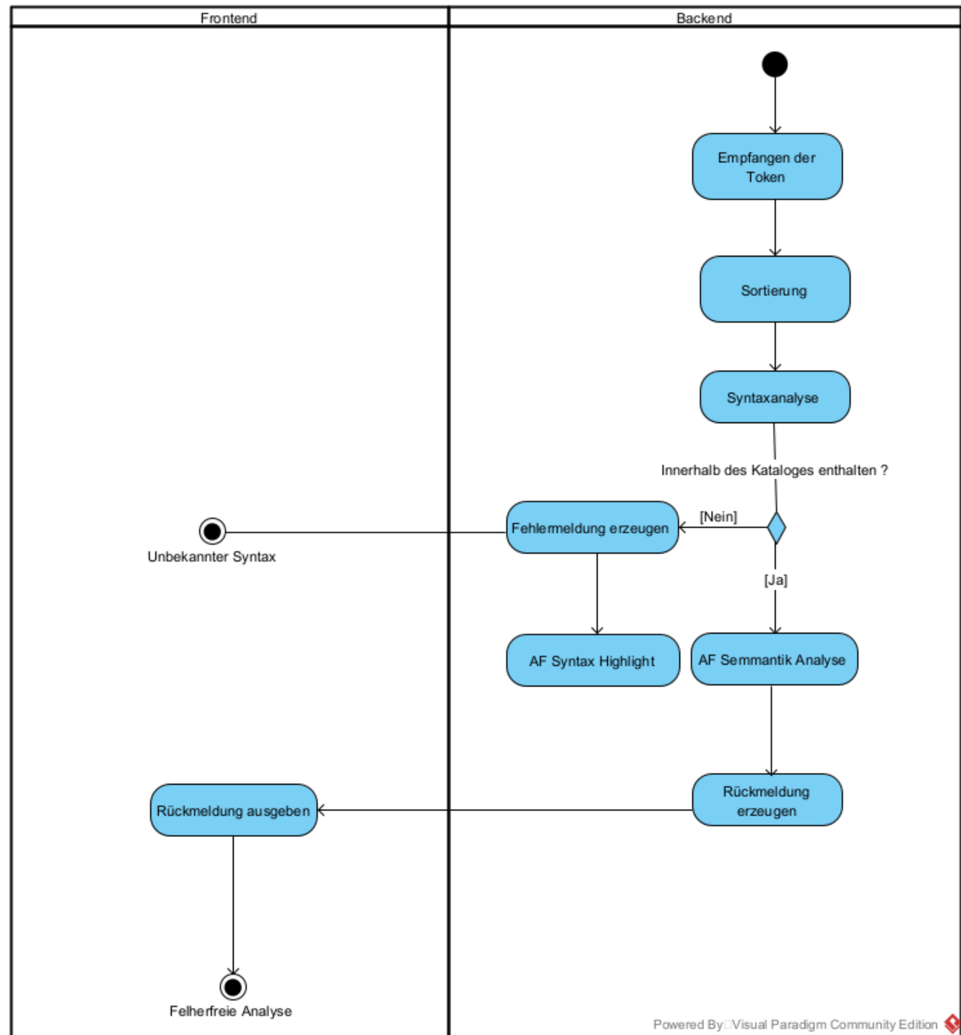


Abbildung 3.1: Anwendungsfall Syntax Analyse

Innerhalb des ersten Schrittes werden die betragenden Token des AF-Syntax Highlight **empfangen**. Danach werden sie entsprechend nach Postion **sortiert** und eingeordnet.
Die **Syntaxanalyse** vergleicht mithilfe des Katalogs die übergebenen Token auf ihre syntaktische Korrektheit.
Ist es nicht innerhalb des Katalog wird eine **Fehlermeldung erzeugt**.
Danach werden einmal die Fehlerstellen an das AF-**Syntax Highlight** zurückgeben und anschließend damit weiter gearbeitet. Zusätzlich wird noch an das Frontend für den Benutzer zurückgeben, dass eine **unbekannte Syntax** vorliegt.

Wenn die Syntaxanalyse korrekt durchgeführt wurde und keine Unterschiede festgestellt wurden, wird eine positive Rückmeldung erzeugt, um an das Frontend zurückgegeben zu werden. Diese Rückmeldung signalisiert, dass der Ablauf korrekt verlaufen ist und der Code syntaktisch korrekt ist. Als abschließender Schritt wird die Rückmeldung innerhalb des Frontends ausgegeben. Das Frontend zeigt dem Benutzer eine entsprechende **Rückmeldung** an, um zu signalisieren, dass die Syntaxanalyse erfolgreich war und der Code syntaktisch korrekt ist. Diese Rückmeldung bestätigt dem Benutzer, dass der eingegebene Code den Syntaxanforderungen entspricht und keine Fehler aufweist.
Nachbedingungen:

- Erfolgreiche Syntaxanalyse: Der Code wird als syntaktisch korrekt befunden und kann in den nächsten Schritten der Codeverarbeitung verwendet werden.

- Fehlerhafte Syntax: Wenn der Code syntaktische Fehler enthält, werden diese erkannt und entsprechende Fehlermeldungen oder Hinweise werden dem Benutzer zur Verfügung gestellt.

Alternative Abläufe: Fehlerbehandlung: Wenn syntaktische Fehler gefunden werden, kann der Benutzer sie korrigieren und erneut eine Syntaxanalyse durchführen. Codeverwerfung: In einigen Fällen kann der Benutzer den fehlerhaften Code verwerfen und einen neuen Code eingeben oder bereitstellen. Auslöser: Der Auslöser für die Syntaxanalyse ist die Bereitstellung oder Eingabe des Codes durch den Benutzer.

3.3.2 AF-Syntax-Highlighting

Der Akteur kann in diesem AF der Benutzer selber sowie das System sein. In diesem Anwendungsfall ist der Auslöser entweder der Benutzer, der einen Button betätigt und somit den AF startet, oder die Entwicklungsumgebung selbst, die nach einiger Zeit mithilfe einer Clock oder einer Callback den AF startet.

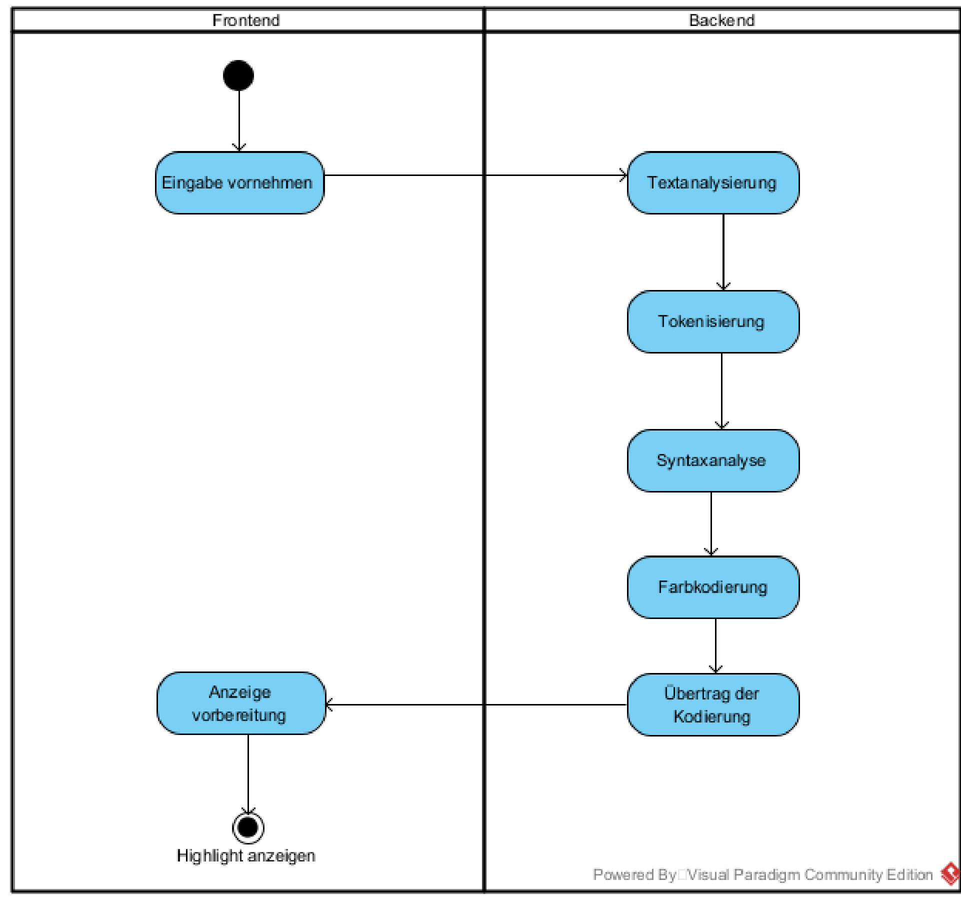


Abbildung 3.2: Anwendungsfall Syntax-Highlighting

Innerhalb des Frontends befindet sich die IDE, in der der Benutzer eine Eingabe vornehmen kann. Diese Eingabe wird dann zur Textanalyse in das Backend übergeben, wo der Code in einzelne Token aufgeteilt wird. Innerhalb dieses Schritts werden beispielsweise Kommentare und Whitespaces entfernt. Im nächsten Teil handelt es sich um die Syntaxanalyse, die ihre eigene AF-Syntaxanalyse startet. Nachdem dieser Durchlauf abgeschlossen ist, erhält man Informationen über mögliche Fehlerstellen, um diese im nächsten Schritt mithilfe der Token durch Farbkodierung kenntlich zu machen. Hierfür wird auf der Backend-Seite die Codierung für den Code vorgenommen und im vorletzten Schritt an das Frontend übertragen. Die Übertragung wird so angepasst, dass die Daten in einem für Webseiten verarbeitbaren HTML-Format vorliegen. Im Frontend wird nun die neue Codierung eingepflegt und für den Benutzer vorbereitet. Der Ablauf endet damit, dass die übertragene Codierung dem Benutzer als farblich hervorgehobener Code in der Entwicklungsumgebung angezeigt wird, wodurch eine klarere visuelle Darstellung des Codes ermöglicht wird.
Nachbedingungen: Der Code wurde erfolgreich analysiert und farblich hervorgehoben. Der farblich hervorgehobene Code wird dem Benutzer angezeigt.
Dialoge: Innerhalb des Dialoges 3.5 finden sie die Nachbedingung dargestellt.

3.3.3 AF-Semantik-Analyse

Akteure: Der Benutzer oder das System, das den Code eingibt.
Kurze Beschreibung: Ein Beispiel für eine Anwendung, die in der integrierten Entwicklungsumgebung (IDE) durchgeführt wird, ist die Semantikanalyse. Sie bezieht sich auf die Prüfung des Codes auf semantische Korrektheit, was bedeutet, dass der Code eine logische Struktur und die beabsichtigte Bedeutung hat. Um sicherzustellen, dass der Code keine logischen Fehler oder inkonsistenten Ausdrücke enthält, werden bei der Semantik-Analyse Regeln und Konventionen der Programmiersprache angewendet.
Vorbedingung: Der AF-Syntax Analyse wurde erfolgreich abgeschlossen und die AF Semantikanalyse wird gestartet.
Auslöser: Nachdem die Syntaxanalyse erfolgreich abgeschlossen wurde, wird die Semantikanalyse gestartet.

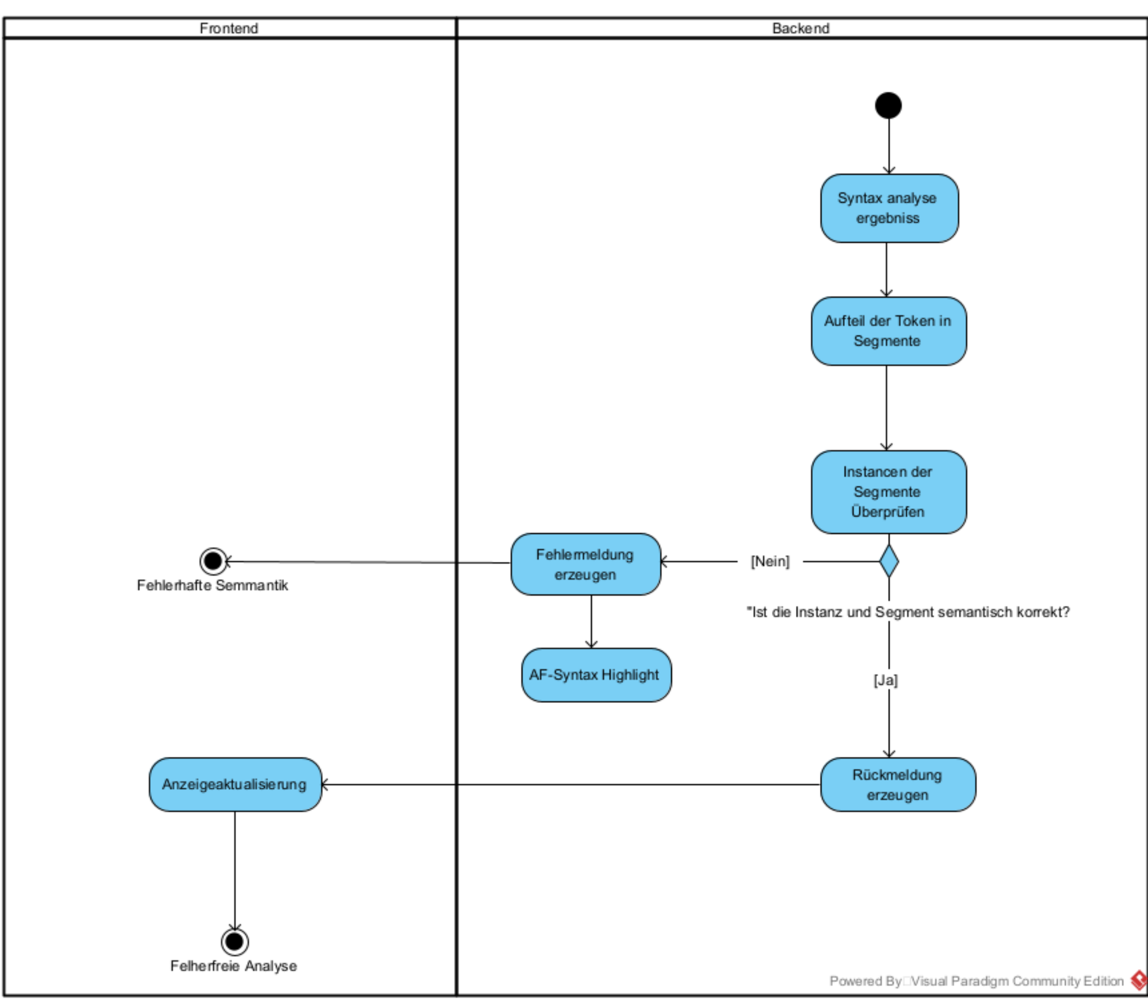


Abbildung 3.3: Anwendungsfall Semantik Analyse

Nachdem die **Syntaxanalyse** abgeschlossen ist und möglicherweise Fehler oder Unstimmigkeiten im Code gefunden wurden, werden diese Informationen an den AF der Semantikanalyse übergeben. Dadurch kann die Semantikanalyse auf den bereits erkannten Fehlern aufbauen und diese weiter analysieren.
Um eine Übersicht darüber zu erhalten, welche Systeme oder Methoden im aktuellen Code vorhanden sind, werden die **Token in Segmente unterteilt**. Diese Segmente stellen logische Einheiten im Code

dar, wie Bauelemente mit Methoden oder Connectoren.
Nachdem die Token segmentiert wurden, werden die Instanzen der **Segmente nun überprüft**, um sicherzustellen, dass jedes Segment semantisch korrekt ist. Anhand von festgelegten Regeln und Bedingungen wird die semantische Korrektheit der einzelnen Segmente überprüft. Es erfolgt eine Überprüfung der Variablentypen, der Anwendung von Funktionen und der korrekten Verwendung von Operatoren. Nachdem die semantische Analyse der einzelnen Segmente durchgeführt wurde, wird die semantische Analyse zwischen den Segmenten durchgeführt. Um sicherzustellen, dass die Semantik des gesamten Codes konsistent ist, werden die Beziehungen und Abhängigkeiten zwischen den Segmenten untersucht.
Falls während der semantischen Analyse ein Fehler auftritt, wird eine entsprechende Fehlermeldung erzeugt. Diese Fehlermeldung beschreibt den aufgetretenen Fehler und gibt Informationen darüber, was konkret schief gelaufen ist. Zum Beispiel kann die Fehlermeldung anzeigen, dass ein falscher Datentyp verwendet wurde oder dass der Ausgang eines bestimmten Bauelements nicht zum Eingang eines anderen Bauelements passt.
Nach der Erkennung von Fehlern in der semantischen Analyse werden die entsprechenden Fehlerstellen an das Syntax-Highlighting weitergegeben, um sie optisch für den Nutzer sichtbar zu machen.
Wenn während der semantischen Analyse kein Fehler auftritt, wird eine **Rückmeldung erzeugt** und die **Anzeige aktualisiert**. Die Rückmeldung informiert den Benutzer darüber, dass die semantische Analyse erfolgreich abgeschlossen wurde und keine Fehler im Code gefunden wurden.
Nachbedingung:

- Die Semantikanalyse wird erfolgreich abgeschlossen, ohne dass schwerwiegende Fehler oder Inkompatibilitäten gefunden wurden.

- Die ermittelten Ergebnisse und Informationen der Semantikanalyse werden für weitere Schritte oder Ausgaben verwendet, z. B. für die Generierung von Vorschlägen, Warnungen oder Fehlermeldungen.

3.3.4 AF-Error Message

Akteure: Das System ist in diesem Fall das Ausführende doch die Anzeige wird vom User aufgerufen.
Kurze Beschreibung: Der Anwendungsfall Error Message tritt auf, wenn ein Fehler in der Syntax oder Semantik des Codes festgestellt wurde. Es wird eine Fehlermeldung erstellt und dem Benutzer gegeben, um den Fehler zu identifizieren und zu beheben.
Vorbedingung: Innerhalb der AF-Syntax-Analyse oder AF-Semmantik-Analyse ist ein Fehler festgestellt worden.
Auslöser: Das Auftreten eines Fehlers in der Syntax oder Semantik des Codes während der Ausführung oder Analyse ist der Auslöser für die Fehlermeldung des Anwendungsfalls.

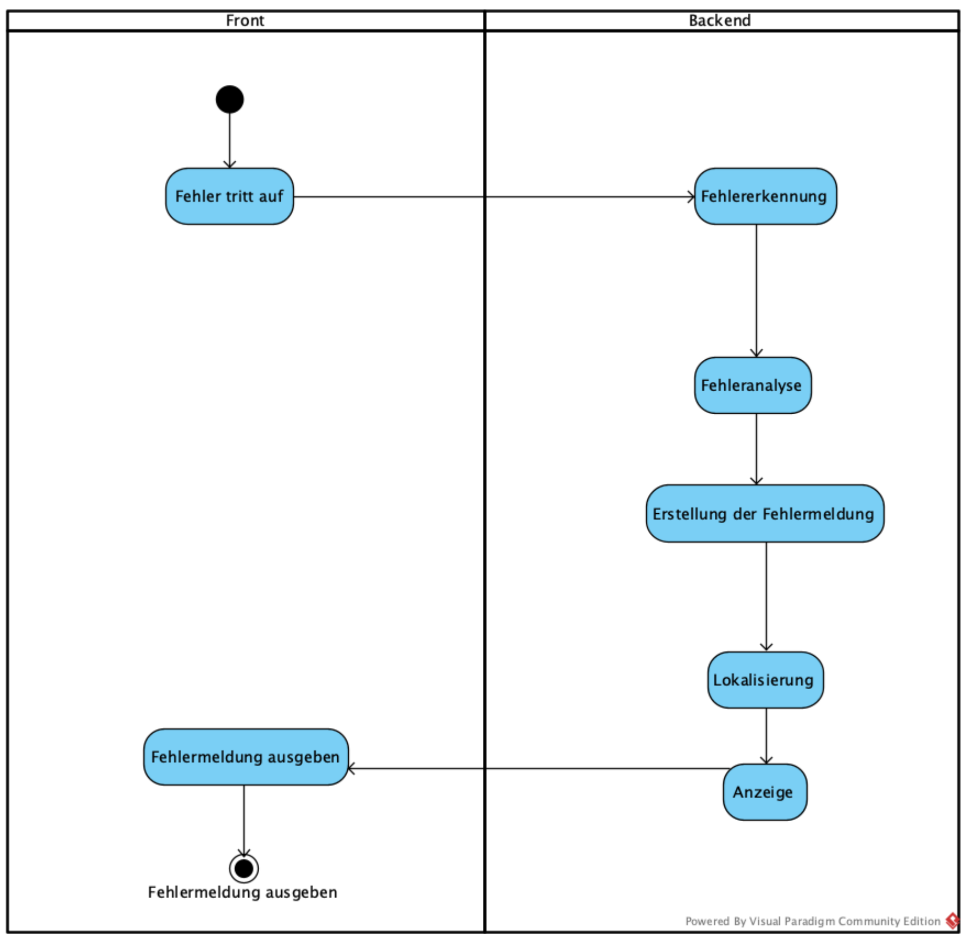


Abbildung 3.4: Anwendungsfall Error Message

Wenn ein **Fehler auftritt**, entweder durch eine automatische Systemüberwachung oder durch die Benutzerinteraktion mit dem Button, beginnt der Prozess.
Wenn ein **Fehler erkannt** wird, beginnt der Fehlerbehandlungsprozess. Das kann durch das Abfangen von Ausnahmen oder die Bewertung von Rückgabewerten von Funktionen erreicht werden.
Innerhalb der **Fehleranalyse** wird der Fehler analysiert, um die Ursache des Fehlers zu identifizieren. Dies kann durch die Überprüfung von der Syntax oder auch der Semmantik auftretten. Die Überprüfung bezieht sich sowohl auf fehlerhafte Codezeilen als auch auf falsche Parameter und Logik.
Mit der **Lokalisierung** wird die Fehlerstelle innerhalb der IDE ermittelt, um die Fehlermeldung präzise sichtbar zu machen. Dies beinhaltet die genaue Bestimmung der Position im Code, an der der Fehler aufgetreten ist, z.B. die betroffene Zeile oder Funktion.

Die **Anzeige** ist verantwortlich für die Darstellung der Fehlermeldung und die grafische Gestaltung. Sie präsentiert dem Benutzer Informationen über den Fehler, einschließlich seiner Art, seines genauen Auftretens und möglicher Lösungsvorschläge. Die Fehlermeldung wird in einer formatierten und benutzerfreundlichen Weise präsentiert, um eine klare und verständliche Darstellung zu gewährleisten. Die Anzeige soll folgende Aspekte umfassen:

1. Fehlerart: Die Fehlermeldung enthält Informationen darüber, um welche Art von Fehler es sich handelt, z. B. Syntaxfehler, Semantikfehler oder Parameterfehler.

2. Fehlerbeschreibung: Die Fehlermeldung enthält eine präzise Beschreibung des Fehlers, um dem Benutzer eine klare Vorstellung von der Art des Problems zu vermitteln. Die Beschreibung kann Details über die fehlerhafte Codezeile, ungültige Eingabewerte oder unerwartete Verhaltensweisen enthalten.

3. Fehlerort: Der genaue Ort des Fehlers im Code wird in der Fehlermeldung angezeigt. Es kann sich um die entsprechende Codezeile, Funktion oder das Modul handeln. Die genaue Lokalisierung ermöglicht dem Benutzer, den fehlerhaften Codeabschnitt gezielt zu lokalisieren.

4. Lösungsvorschläge: Die Fehlermeldung kann dem Benutzer Vorschläge zur Lösung oder zur Behebung des Fehlers geben. Es kann beispielsweise darauf hindeuten, dass es ähnliche Funktionen gibt, dass eine bestimmte Logik verwendet wird oder dass bekannte Problemlösungen existieren.

Als letzter Punkt wird nun die **Fehlermeldung** für den Benutzer **ausgegeben** und dargestellt. **Nachbedingungen**:

- Fehlermeldung: Der Benutzer erhält eine spezifische Fehlermeldung, die ihm zur Verfügung gestellt wird, um den Fehler zu verstehen und zu beheben.

- Benutzerinteraktion: Der Benutzer kann auf die Fehlermeldung reagieren und geeignete Maßnahmen ergreifen.

3.4 Dialoge

3.4.1 Dialog Syntax-Highlighting

Innerhalb dieses Dialogs wird grafisch dargestellt wie eine Fehlerhafte Syntax für den Benutzer kenntlich gemacht werden soll. Wichtig hier bei ist, dass nur der Token unterschrichen wird, der syntaktisch nicht korrekt ist.

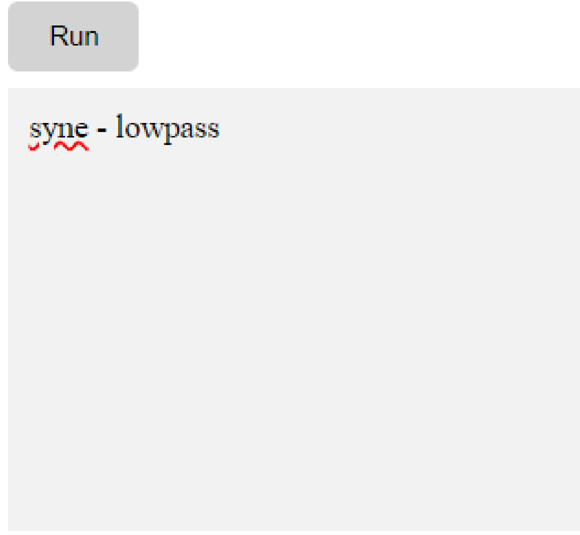


Abbildung 3.5: Dialog Syntax-Highlighting

4 Implementierung

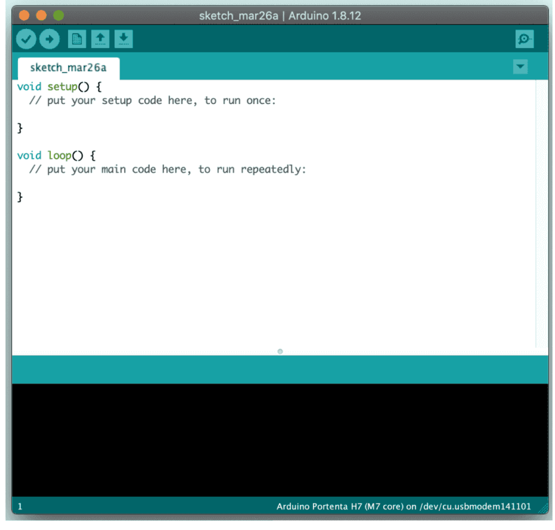
Das Implementierungskapitel beschreibt die praktische Umsetzung der Konzepte und Methoden in das System. Es beinhaltet eine detaillierte Beschreibung der verwendeten Programmiersprachen, Tools und Technologien sowie der Systemarchitektur. Es werden auch wichtige Algorithmen, Datenstrukturen und Schnittstellen erklärt. Das Kapitel behandelt auch Herausforderungen, Lösungsansätze und Designentscheidungen, die während der Implementierungsphase aufgetreten sind. Das Ziel ist es, dem Leser einen umfassenden Einblick in die technische Umsetzung der entwickelten Lösung zu bieten.

4.1 WebIDE

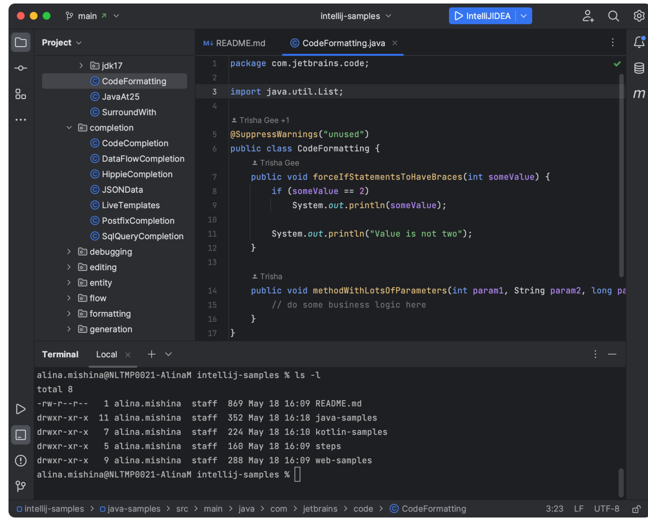
4.1.1 Oberflächenstile der IDE

Erstentwurf

Im ersten Abschnitt dieses Kapitels beschäftigen wir uns mit der Struktur der Entwicklungsumgebung. Im Folgenden wird die Programmierung, die ursprüngliche Benutzeroberflächenkonzeption und die Prototypen gezeigt, die zur Erstellung der Funktionen verwendet wurde. Die Programmierung der Oberfläche konzentrierte sich darauf, eine Umgebung zu schaffen, die einfach zu bedienen und einfach zu verstehen ist. Um eine konsistente und ansprechende Benutzererfahrung zu gewährleisten, wurden bewährte Designprinzipien und Standards berücksichtigt. Durch die Untersuchung anderer Web-IDEs und der damit verbundenen Anforderungen und Bedürfnisse der Entwickler wurde die erste Idee für die Oberfläche entwickelt. Um eine effektive und benutzerfreundliche Arbeitsumgebung zu schaffen, wurden verschiedene Aspekte berücksichtigt.



(a) Arduino [19]



(b) intelliJ IDEA [20]

Abbildung 4.1

Diese wurden als Grundlage für die Entwicklung des ersten Oberflächenkonzepts verwendet, um den Entwicklern ein vertrautes und einfaches Umfeld zu bieten. Um eine übersichtliche Arbeitsumgebung zu schaffen, wurde die Oberfläche der IDE bewusst einfach gehalten. Einige grundlegende Werkzeuge wie das Laden und Speichern von bestehenden Projekten sind bereits vorhanden und können wie in anderen Entwicklungsumgebungen über den „File“-Reiter aufgerufen werden. Das Ziel war es, die Benutzeroberfläche der IDE auf das Wesentliche zu reduzieren, damit sie für den Nutzer einfach zu verstehen und intuitiv zu bedienen ist. Um eine Überfüllung zu vermeiden und die Übersichtlichkeit zu gewährleisten, wurden bewusst nur bestimmte Werkzeuge und Funktionen sichtbar gemacht. Die IDE wird aufgrund dieser bewussten Entscheidung nicht zu kompliziert und ermöglicht es den Entwicklern, sich auf ihre Hauptaufgabe zu konzentrieren: das Entwickeln von Code. Allerdings sind die grundlegenden Werkzeuge vorhanden, um effektiv und produktiv zu sein.

Abbildung 4.2: Erste Idee der IDE

Der Entwurf besteht aus einem Button und zwei Feldern. Die Entwickler geben den Code für die Simulationsschaltung in das linke Feld ein, das als Editor-Fenster fungiert. Sie verbringen hier die meiste Zeit und haben die Möglichkeit, den Code nach ihren Bedürfnissen anzupassen. Die Konsole im rechten Feld zeigt während der Ausführung des Codes wichtige Informationen und Rückmeldungen an. Hier können die Entwickler Fehlermeldungen, Debugging-Informationen und andere Informationen sehen. Der Button ermöglicht es den Entwicklern, die Simulation zu starten, den Code zu kompilieren oder andere Aufgaben auszuführen. Insgesamt bietet die IDE eine benutzerfreundliche und effektive Benutzeroberfläche, die Entwicklern beim Entwerfen und Testen der Simulationsschaltung unterstützt.

Prototyp

Die Implementierung der Oberflächenfunktionen begann mit dem Prototypen. Er half, grundlegende Ideen und Designs zu testen und zu verbessern. Der Prototyp ermöglichte es, Probleme frühzeitig zu identifizieren und Lösungsansätze zu entwickeln. Die Oberfläche konnte durch den iterativen Entwicklungsprozess schrittweise verbessert werden, um eine benutzerfreundliche und effektive Arbeitsumgebung zu schaffen.
Im weiteren Verlauf wird die eigentliche IDE-Form weiterentwickelt und verbessert. In dieser Version können die Anwendungsfälle und deren Hintergrundfunktionen durch den Compile-Button gestartet werden. Zukünftig ist es jedoch geplant, die Möglichkeit einzuführen, den Code kontinuierlich zu überprüfen, indem entweder ein Callback oder ein Timer verwendet wird, um Änderungen sofort zu erkennen.

Abbildung 4.3: Finale Form des Prototyps

4.1.2 CSS Code für die IDE

Der Stil jedes div-Elements wird mit CSS gestaltet. Der folgende Code betrachtet verschiedene Elemente, um grundlegende Designentscheidungen zu erläutern.

Container

```
1        .container {
2            display: flex;
3            height: 100vh;
4            width: 100vh;
5            justify-content: center;
6            align-items: flex-start;
7            flex-direction: column;
8            background-color: white;
9        }
```

Listing 4.1: Code Container

Die Klasse „container“ stellt die Gesamtstruktur der Webseite dar. Hier werden die Positionierung und das allgemeine Layout für die IDE festgelegt. Wichtig war dabei, von Anfang an eine Gestaltung zu wählen, die es ermöglicht, die Hauptseite später einfach einzufügen.

Toolbar

```
1       .toolbar {
2            display: flex;
3            justify-content: space-between;
4            align-items: center;
5            width: 80%;
6            margin: 0 10%;
7            height: 50px;
8        }
```

Listing 4.2: Code Toolbar

Die Klasse „toolbar“ definiert den Stil und die Positionierung der Toolbar, die beispielsweise Buttons für bestimmte Funktionen enthält. Im späteren Verlauf soll die Toolbar erweitert werden, um zweischichtige Dropdown-Menüs für Funktionen wie beispielsweise Hilfetexte zu ermöglichen.

Textarea-container

```
1          .textarea-container {
2              display: flex;
3              flex: 1;
4              width: 100%;
5          }
```

Listing 4.3: Code textarea-container

Die Klasse "textarea-container"kümmert sich um grobe Struktur für die Textbereiche wie In- und Output. Es ist möglich, bestimmte Einstellungen wie Hintergrundfarbe, Schriftgröße und Rahmenstil zu ändern.

Textareas und output

```
1  .textarea {
2              flex: 1;
3              width: 100%;
4              color: gray;
5              padding: 10px;
6              box-sizing: border-box;
7              resize: none;
8              background-color: #f2f2f2;
9              border: none;
10             outline: none;
11             overflow-y: auto;
12         }
13
14 .output {
15             width: 100%;
16             background-color: #d9d9d9;
17             color: black;
18             padding: 10px;
19             overflow-y: auto;
20         }
```

Listing 4.4: Code textareas

In dem Element mit der Klasse „textarea“ wird das grundlegende Erscheinungsbild der selbst geschriebenen Textareas festgelegt. Zusätzlich dazu gibt es das Element „output“ , das speziell für die Konsolenausgabe vorgesehen ist. Es hat eine eigene Farbgebung und Schriftfarbe, um sich visuell von den anderen Textareas zu unterscheiden.

Linenumbers

```
1          .linenumbers {
2              display: inline-block;
3              width: 40px;
4              text-align: right;
5              padding-right: 5px;
```

```
6             margin-top: 10px;
7             color: gray;
8             white-space: pre-line;
9             pointer-events: none;
10        }
```

Listing 4.5: Code textareas

Die Klasse „Zeilennummern“ definiert den Bereich, in dem die Zeilennummern des Codes angezeigt werden. Diese sind neben dem div-Container mit der ID „input“ angeordnet.

4.1.3 HTML Code für die IDE

Aufbau der IDE Struktur mit HTML

HTML, CSS und Javascript wurden verwendet, um die Webseiten der IDE zu entwickeln und darzustellen. Im folgenden Abschnitt wird sich genauer mit der Struktur der IDE auseinandersetzen, die mit HTML erstellt wurde. Die Grundstruktur der IDE besteht aus dieser HTML-Struktur, die es dem Benutzer ermöglicht, den Code einzugeben, Aktionen auszuführen und die Ausgabe zu betrachten.

```
1  <div class="container">
2      <div class="toolbar">
3          <button id="compileButton" class="play-button" ↩
              onclick="mainSemmantik()">Compile</button>
4          <button id="duplicate-button" class="play-button" ↩
              onclick="main()">Download</button>
5      </div>
6      <div class="textarea-container">
7          <div class="linenumbers"></div>
8          <div class="textareas">
9              <div class="textarea" contenteditable="true" ↩
                  spellcheck="false" id="input"></div>
10             <div class="textarea output" contenteditable="true" ↩
                  spellcheck="false" id="output" disabled></div>
11         </div>
12     </div>
13 </div>
```

Listing 4.6: Code der Update Funktion

Die HTML-Struktur der IDE wurde komplett überarbeitet und mit CSS angepasst. Dabei wurden alle Elemente als divs realisiert. Die Gesamtstruktur und Anordnung der einzelnen Felder werden durch Container-divs mit der Klasse „Container“ gewährleistet. Das erste Element ist die Toolbar. Im Prototypen gibt es zwei Buttons: „Compile“ zum Starten der Anwendungsfälle und zur manuellen Überprüfung des Codes und „Download“ zum späteren Herunterladen der Anwendung.

Im nächsten Schritt wird die Struktur für die beiden benötigten Textboxen „Input“ und „Output“ mithilfe der Klasse „textarea-container“ erstellt. Da herkömmliche Textareale zu beschränkt in ihrer Funktionalität für unsere Anwendungsfälle sind, wurden sie als divs umgesetzt. Das Input-Feld ist für die Eingabe von Codes gedacht, während das Output-Feld als Konsole für spätere Ausgaben dient. Die Verwendung von divs ermöglicht eine umfassendere Anpassung und Erweiterung der Funktionalität der Textboxen für unseren spezifischen Anwendungsfall.

4.1.4 JS Code für die IDE

Die Oberfläche der IDE wurde mithilfe von speziell für diesen Zweck entwickelten Skripten erstellt. In diesem Abschnitt werden einige der Skripte vorgestellt, die ausschließlich für die Gestaltung der Oberfläche geschrieben wurden.

Zeilennummer

```
1  <script>
2      const input = document.getElementById('input');
3      const linenumbersDiv = document.querySelector('.linenumbers');
4      input.addEventListener('input', updateLineNumbers);
5
6      function updateLineNumbers() {
7          const lines = input.innerText.split('\n');
8          const lineNumbers = [];
9          for (let i = 0; i < lines.length; i++) {
10             lineNumbers.push(i + 1);
11         }
12         linenumbersDiv.innerText = lineNumbers.join('\n');
13     }
14 </script>
```

Listing 4.7: Code der Update Funktion

Die Funktionalität zur Zeilennummerierung wird in diesem Skript verwendet. Zunächst wird das HTML-Element mit der id „input“ sowie das Element mit der Klasse „linenumbers“ abgerufen, um den aktuellen Zeilenstand zu bestimmen. Da das Skript automatisch auf der Webseite ausgeführt wird, ist kein expliziter Aufruf im HTML-Code notwendig. Sobald eine Änderung im „input“-Element vorliegt, wird der Eventlistener die Funktion „updateLineNumbers()“ ausführen. Dies ermöglicht es auch in Zukunft Callback-Funktionen zu integrieren. Die Funktion „updateLineNumbers()“ zählt die Anzahl der Zeilenumbrüche im Text des „input“-Elements mittels der „split“-Methode. Die entsprechenden Zeilennummern werden dann durch eine Schleife generiert. Der Inhalt des Elements wird schließlich mit der Klasse „lineNumbers“ mit den aktualisierten lineNumbers aktualisiert.

4.2 Klasse ImportInformationFromWebsite

Eine grundlegende Funktionalität für verschiedene Anwendungsfälle wird durch die Klasse „ImportInformationFromWebsite“ gemäß den Anforderungsspezifikationen bereitgestellt. Sie wird verwendet, um die Eingaben des Benutzers abzurufen und für weitere Verarbeitungsschritte bereitzustellen. Daher bildet diese Klasse die Grundlage für die Verarbeitung der Benutzereingaben und ermöglicht es, sie für zusätzliche Funktionen und Anwendungsfälle vorzubereiten und zu verwenden. Im Folgenden werden weitere Details zur spezifischen Funktionalität und Verwendung der importierten Daten gegeben.

4.2.1 Relevante Funktionen

GetInformationFromWebsite

```
1  async function GetInformationFromWebsite() {
2      let textareaValue = document.getElementById("input");
3      let userInput = textareaValue.innerText;
```

```
4     return SplitInformationFromWebsite(userInput);
5 }
```

Listing 4.8: Code der GetInformationFromWebsite Funktion

Die Klasse startet mit der Funktion „GetInformationFromWebsite()“, die alle Inhalte des Editor-Feldes der WebIDE abruft.
Dies beinhaltet nicht nur den vom Benutzer eingegebenen Code, sondern auch Formatierungsinformationen wie Leerzeichen und Zeilenumbrüche. Diese Funktion ermöglicht es, alle wichtigen Informationen aus dem Editor-Feld zu extrahieren und für weitere Verarbeitungsschritte zu verwenden. Es wird die Methode „innerText“ verwendet, um die relevanten Informationen aus den HTML-Tags des div-Elements zu extrahieren. Dadurch kann man nur die notwendigen Daten auswählen und den Rest ignorieren. Anschließend wird der Wert, der extrahiert wurde, an die nächste Funktion weitergegeben, um ihn zu verarbeiten. Durch diese Methode können wir die gewünschten und unerwünschten Informationen effektiv unterscheiden und gezielt verwenden.

SplitInformationFromWebsite

```
1 function SplitInformationFromWebsite(userInput) {
2     const splittedInputArray = userInput.split(/\r?\n/).flatMap(line ↩
         => line || "\n");
3     return SeparateInputIntoLines(splittedInputArray);}
```

Listing 4.9: Code der SplitInformationFromWebsite Funktion

Die Funktion „SplitInformationFromWebsite()“ ist ein wichtiger Teil für spätere Funktionen, obwohl sie sehr kurz ist. Sie nimmt den userInput, den man aus der Funktion 4.8 erhält, und teilt ihn an den Stellen auf, an denen ein Zeilenumbruch (\n) oder eine neue Zeile(„\r“) auftritt. Der reguläre Ausdruck „/r?\n/“ ermöglicht es, sowohl Zeilenumbrüche als auch Wagenrückläufe als Trennzeichen zu erkennen. Der Teil .flatMap(line => line || "\n") des Codes wendet eine Funktion auf jede Zeile des geteilten Arrays an. Die Funktion überprüft, ob eine Zeile einen Wert enthält. Wenn eine Zeile einen Wert hat, wird sie unverändert beibehalten. Wenn eine Zeile jedoch keinen Wert hat (z.B. eine leere Zeile), wird sie durch einen Zeilenumbruch ("\n") ersetzt. Das Ergebnis ist ein Array, das alle Zeilen des Eingabetexts enthält, wobei Zeilenumbrüche leere Zeilen ersetzen. Zusammenfassend teilt die Funktion „SplitInformationFromWebsite()“ den Benutzerinput in separate Zeilen auf und ersetzt leere Zeilen durch Zeilenumbrüche, um eine einheitliche Darstellung des Textes sicherzustellen.

SeparateInputIntoLines

```
1 function SeparateInputIntoLines(splittedInputArray) {
2     return splittedInputArray.map(str => str.split(' '));
3 }
```

Listing 4.10: Code der SeparateInputIntoLines Funktion

Die Funktion „SeparateInputIntoLines“ liefert ein Array, das die zuvor aufgeteilten Eingaben enthält. Die „map“-Funktion wird verwendet, um über jedes Element des Arrays zu iterieren und eine Aufteilung in Zeilen durchzuführen. Das Ergebnis ist ein Array mit zwei Dimensionen, bei dem jede Zeile des ersten Inputs in einem anderen Array gespeichert wird. Dies ermöglicht eine spätere Analyse und Bearbeitung der Eingabezeilen in einer Zeile.

4.3 Klasse Syntax Highlighter

Die Aufgabe der Klasse „SyntaxHighlighter" besteht darin, die Formatierung für Fehler in der Syntax und Semantikanalyse zu erfassen. Die Funktionen „TransformInnerHTML()" und „ArrayIntoHtmlFormat()" ermöglichen die Umwandlung der Fehlerinformationen in das HTML-Format. Die Abbildung 3.2 zeigt den Anwendungsfall des Syntax-Highlighting. Um die Fehlerdarstellung angemessen zu formatieren, werden Daten aus den Anwendungsfällen der Semantik 3.3- und Syntaxanalyse 3.1 verwendet.

4.3.1 Relevante Funktionen

TransformInnerHTML

Ein Auszug des Codes der Funktion wird im folgenden Abschnitt angezeigt. Dieser Code-Auszug zeigt die grundlegende Funktionalität des Syntax-Highlighters, der in der Syntaxanalyse verwendet wird, um Fehler visuell hervorzuheben. Der Anhang enthält zusätzliche Informationen und den vollständigen Code.

```
1
2   const userInput = document.getElementById("input");
3     const consoleOutput = document.getElementById("output");
4     const textInput = await GetInformationFromWebsite();
5     const textOutput = ↩
        consoleOutput.innerText.replace(consoleOutput.innerText, "");
6     let newContent = textInput;
7     let newConsoleFail = textOutput;
8     userInput.innerHTML = userInput.innerText;
```

Listing 4.11: Code der TransformInnerHTML Funktion

Innerhalb dieses Abschnitts der Funktion „TransformInnerHTML" werden die beiden div-Textareas der WebIDE geladen. Zunächst werden beide Felder abgerufen. Im nächsten Schritt wird das Konsolenfeld geleert, um Platz für die neue Fehlerausgabe zu schaffen. Anschließend wird das „userInput"-Feld (Editor-Feld) zum ersten Mal aktualisiert. Das bedeutet, dass der aktuelle Textinhalt des Feldes abgerufen wird, um ihn weiterzuverarbeiten.

```
1     for (let lineIndex = 0; lineIndex < textInput.length; ↩
        lineIndex++) {
2         let flag = false;
3         for (let i = 0; i <= misstakesPostionArray[lineIndex].length ↩
            - 1; i++) {
4             const designatedProblematicWord = ↩
                mistakeWordArray[lineIndex][i];
5             newContent[lineIndex][misstakesPostionArray[lineIndex][i]] ↩
                = `<u style="text-decoration-color: red; ↩
                text-decoration-style: wavy" ↩
                >${designatedProblematicWord}</u>`;
6             if(designatedProblematicWord == "\n" || ↩
                designatedProblematicWord == ""){
7                 flag = true;
8                 break;
9             }
10            if (misstakesPostionArray.length != 0) {
11                newConsoleFail = newConsoleFail.replace("", function ↩
                    () {
```

```
12                      return "Diese eingabe ist der Syntax nicht ↩
                           bekannt: " + '<span style="color: #ff0000;  ↩
                           font-weight: bold;">' + ↩
                           designatedProblematicWord + '</span><br>';
13                  });
14              }
15          }
16          if(flag == false) {
17              newContent[lineIndex] = ↩
                    newContent[lineIndex].concat('<br>');
18          }else{
19              console.log("error empty line error")
20          }
21      }
```

Listing 4.12: Code der TransformInnerHTML Funktion

Eine verschachtelte For-Schleife bildet den Mittelpunkt dieser Funktion. Zunächst wird das Array, das übergeben wurde, in verschiedene Zeilen aufgeteilt. Die äußere Schleife iteriert über jede Zeile, wobei eine Flag verwendet wird, um einen unerwünschten Zeilenumbruch durch das div-Element zu beseitigen. Dies ist von entscheidender Bedeutung, um sicherzustellen, dass die Darstellung des Editor-Felds nur durch die Syntax der einzelnen Instanzen verändert wird und nicht durch die Anordnung oder Reihenfolge der Instanzen. Zunächst wird nach fehlerhaften Instanzen in der inneren Schleife gesucht. Die „newContent“-Variable wird verwendet, um eine fehlerhafte Instanz in die korrekte HTML-Form zu bringen. Hier wird ein neuer Style beschrieben, der die rote Unterstreichung erzeugt, dass aus anderen IDEs bekannt ist. Anschließend wird festgestellt, ob ein neuer Absatz durch einen Zeilenumbruch hinzugefügt wurde. Wenn dies der Fall ist, wird die Schleife abgebrochen und die entsprechende Flag gesetzt. Wenn dies nicht der Fall ist, wird die Variable „newConsoleFail“ verwendet, um die Fehlerausgabe zu generieren. Später sollte diese Ausgabe so angepasst werden, dass genaue Informationen ausgegeben werden und nicht nur das Fehlerwort und die Zeilennummer. Wenn die Flag auf „false“ steht, wird der neue Inhalt in einem verschachtelten Array gespeichert und für die weitere Verarbeitung vorbereitet. Der genaue Grund für diese Verarbeitung wird in der folgenden Funktion erklärt.

```
1   userInput.innerHTML = ArrayIntoHtmlFormat(newContent);
2   consoleOutput.innerHTML = newConsoleFail;
```

Listing 4.13: Code der TransformInnerHTML Funktion

Innerhalb dieses Teils der Funktion „TransformInnerHTML“ werden der Inhalt des Editor-Felds und der Konsole aktualisiert. Hierfür wird abschließend die Funktion „ArrayIntoHtmlFormat“ aufgerufen für das Editor-Feld.

ArrayIntoHtmlFormat

Die Funktion „ArrayIntoHtmlFormat“ ist recht einfach und dient dazu, die gelöschten Zeilenabstände automatisch einzufügen. Dies ist notwendig, da gelegentlich Zeilenumbrüche übergeben werden, die jedoch nicht vom Benutzer eingegeben wurden. Daher wurden sie zuvor von der Funktion 4.9 entfernt und werden nun manuell eingefügt. Dies gewährleistet eine korrekte Darstellung des Codes im HTML-Format, bei dem die Zeilenumbrüche an den richtigen Stellen platziert werden.

```
1  function ArrayIntoHtmlFormat(array){
2      let newContent = "";
3      for (let lineIndex = 0; lineIndex < array.length; lineIndex++) {
4          let flag = false;
5          for (let i = 0; i <= array[lineIndex].length-1; i++) {
6              if(i < array[lineIndex].length-1){
7                  newContent = newContent + array[lineIndex][i] + " ";
8              }else{
9                  newContent = newContent + array[lineIndex][i];
10             }
11         }
12     }
13     return newContent;
14 }
```

Listing 4.14: Code der ArrayIntoHtmlFormat Funktion

Die Funktion ist relativ einfach aufgebaut. Sie durchläuft das übergebene Array und kombiniert die Elemente zu einem neuen Inhalt. Es wird festgestellt, ob jede Zeile des Arrays das letzte Element ist. Um sicherzustellen, dass der Inhalt korrekt dargestellt wird, wird nach dem aktuellen Element ein Leerzeichen hinzugefügt, wenn es nicht das letzte Element ist. Am Ende wird der frische Inhalt wiederhergestellt. Die Funktion ermöglicht die Konvertierung des Arrays in das gewünschte HTML Format, indem die Elemente angemessen formatiert werden.

4.4 Klasse JSONFileLoader

Die Klasse „JsonFileLoader" stellt eine grundlegende Funktionalität für verschiedene Anwendungsfälle bereit. Sie dient dazu, eine JSON-Datei zu laden, die von Herrn Hirlimann [21] im Rahmen seiner Bachelorarbeit zur Verfügung gestellt wurde. Diese JSON-Datei enthält wichtige Informationen über die Bauelemente, die für verschiedene Funktionen, wie zum Beispiel die Syntaxanalyse, benötigt werden. Die Klasse „JsonFileLoader" ermöglicht es mir, auf diese Informationen zuzugreifen und sie in meiner Anwendung zu verwenden. Die Inhalte dieser JSON Datei werde in folgende Grafik dargestellt in 4.15 und die Inhalte in 4.1 näher erklärt.

4.4.1 JSON Datei

Auschnitt aus der JSON datei

```
1    "Generator": {
2      "in": {
3        "count": "0",
4        "types": [
5          "?"
6        ]
7      },
8      "out": {
9        "count": "1",
10       "types": [
11         "?"
12       ]
13     },
14     "superclass": "SignalSource",
15     "classtype": "source",
```

```
16      "methods": [
17        {
18          "name": "frequency",
19          "parameters": [
20            "double"
21          ],
22          "returntype": "class de.labAlive.baseSystem.Generator"
23        },
24        {
25          "name": "amplitude",
26          "parameters": [
27            "double"
28          ],
29          "returntype": "class de.labAlive.baseSystem.Generator"
30        }
31      ],
32      "constructors": []
33    },
34    ...
```

Listing 4.15: Code der Update Funktion

Im zweiten Format (Listing 4.15) besteht das vollständige JSON aus einer Vielzahl von (ca. 400) Key-Value-Paaren. Nachdem die Klassennamen eindeutig sind, wurden sie jeweils als Schlüssel verwendet. Als Wert wurde erneut ein eigenes JSON verwendet. In diesem ersten verschachtelten JSON wurden die Key-Value-Paare gemäß der Tabelle 4.1 gesetzt. Die JSON-Objekte, die als Wert in der ersten Tabelle gesetzt wurden, werden in den folgenden beiden Tabellen (Tabelle 4.2 für die Schlüssel „in“ und „out“ und Tabelle 4.3 für die Schlüssel („methods“ und „constructors“) genauer beschrieben.

Key	Bedeutung
classnameAsString	Name des Bauteils
in	Eingang des Bauteils
out	Ausgang des Bauteils
superclass	Elternklasse der Klasse
classtype	Klassentyp
methods	Methoden der Klasse
constructors	Konstruktoren der Klasse

Tabelle 4.1: JSON File

Key	Bedeutung	Valuetype
count	Anzahl der Ein- Ausgänge	String
types	Mögliche Signaltypen der Ein- Ausgänge	Array aus Strings

Tabelle 4.2: In & Out JSON

Key	Bedeutung	Valuetype
name	Methoden- Konstruktorname	String
parameters	Klassen der übergebnene Parameter	String
returntype	Klasse des Rückgabewerts	Array aus Strings

Tabelle 4.3: Methods & Constructors JSON

Die folgenden Funktionen wurden im Laufe der Zeit umprogrammiert oder sogar vollständig neu geschrieben, da sich die Anforderungen und das Format der JSON verändert haben. Diese Anpassungen waren erforderlich, um den Anforderungen gerecht zu werden und die Daten in einem aktualisierten Format zu verarbeiten. Durch diese Änderungen wurden die Funktionen verbessert und an die neuen Bedingungen angepasst.

4.4.2 Relevante Funktionen

Die Aufgabe des JSON-Loaders besteht darin, eine JSON-Datei zu laden, die grundlegende Informationen über die Elemente des LabAlive-Projekts enthält. Die Klassennamen, Ein- und Ausgänge, Superklassen, Methoden und Konstruktoren sind Teil dieser Informationen. Die Syntax 3.1- und Semantikanalysen 3.3 der AF basieren auf diesen Daten. Die Error Message 3.4 nutzt sie auch, um Vorschläge und Informationen über die Funktionsmöglichkeiten zu geben. Daher ist der JSON-Loader entscheidend für die Bereitstellung der Informationen, die für die weitere Verarbeitung und Analyse der LabAlive-Anwendung erforderlich sind.

LoadJsonContent

```
1  function loadJsonContent() {
2      return fetch("Data.json")
3          .then(response => response.json())
4          .then(data => {
5              if (!data) {
6                  throw new Error("Ungueltiges JSON-Format.");
7              }
8              const result = [];
9              for (const classNames in data) {
10                 const classData = data[classNames];
11                     const inCount = classData.in?.count || "";
12                     const inTypes = classData.in?.types || [];
13                     const outCount = classData.out?.count || "";
14                     const outTypes = classData.out?.types || [];
15                     const superClass = classData.superclass || "";
16                     const methods = classData.methods || [];
17                     const constructors = classData.constructors || [];
18                     const waveForms = classData.fields;
19                     result.push({
20                         classNames: classNames,superClass,
21                         inCount, inTypes,
22                         outCount, outTypes,
23                         methods,
24                         waveForms,
25                         constructors
26                     });...
```

Listing 4.16: Code der loadJsonContent Funktion

Mithilfe der Fetch-api kann die Funktion „loadJsonContent()“ das Laden und Zwischenspeichern der JSON-Datei „Data.json“ ermöglichen. Die JSON-Datei wird durch die Verwendung von „.then(response => response.json())“ in ein Javascript-Objekt umgewandelt. Anschließend wird überprüft, ob die geladenen Informationen vorhanden sind. Wenn nicht, wird eine Warnung ausgelöst, da ungültige oder fehlende Daten problematisch sein können. Nach diesem Schritt werden die Daten aus dem Javascript-Objekt weiter verarbeitet. Es werden wichtige Daten wie Eingänge, Ausgänge, Klassen, Superklassen, Methoden, Konstruktoren und Wellenformen extrahiert und in einer übersichtlicheren Struktur gespeichert.
Diese Verarbeitung wird in ein Array namens „result“ gespeichert, in dem jedes Element ein Objekt darstellt. Jedes Objekt enthält die relevanten Informationen für eine Klasse. Die Anwendungen können dieses Array verwenden, um auf die gespeicherten Daten zuzugreifen und sie entsprechend zu nutzen. Da die Daten einmal geladen und zwischengespeichert werden, um Zeit zu sparen und das Wiederladen der Datei zu vermeiden, wird die Verwendung der Funktion „loadJsonContent()“ effektiver.

ImportClassFromJsonFile

```
1  async function ImportClassFromJsonFile() {
2      let classesJson = [];
3      await LoadJsonContent().then(result => {
4          result.forEach(item => {
5              classesJson = classesJson.concat(item.classNames);
6              return classesJson;
7          });
8      }).catch(error => {
9          console.log("Fehler beim Verarbeiten der JSON-Daten fuer ↩
               Classnames: " + error.message);
10     });
11     return classesJson;
12 }
```

Listing 4.17: Code der ImportClassFromJsonFile Funktion

Die Funktion „ImportClassFromJsonFile“ teilt das Array, das aus der Klasse „JsonFileLoader“ stammt, auf und fokussiert sich dabei auf die Extraktion der Klassennamen. Sie wird in der Syntaxanalyse verwendet. Die Funktion extrahiert die Klassennamen aus dem Array und speichert sie in einer geeigneten Datenstruktur. Dies ermöglicht es anderen Funktionen, auf die Klassennamen zuzugreifen und sie für weitere Verarbeitungsschritte in den Analysen zu verwenden. Die Funktion „ImportClassFromJsonFile“ spielt eine wichtige Rolle bei der Bereitstellung der Grundlage für die Syntaxanalyse, da sie die Klassennamen aus der JSON-Datei extrahiert und für die folgenden Analyseverarbeitungsschritte zur Verfügung stellt.

ImportInAndOutPutFromJsonFile

```
1  ...
2              const classNames = item.classNames;
3              let inCount  = item.inCount;
4              let inTypes = item.inTypes;
5              let outCount = item.outCount;
6              let outTypes = item.outTypes;
7              const ComponentObject = {
8                  name: classNames,
9                  inCount: inNr,
```

```
10                  inTypes: inTyp,
11                  outCount:outNr,
12                  outTypes:outTyp
13              };
14              ComponentInAndOut.push(ComponentObject);
15  ...
16  }
```

Listing 4.18: Auschnitte der ImportInAndOutPutFromJsonFile Funktion

Neben dem Klassennamen sind für die Semantikanalyse auch andere Informationen wichtig, wie die Anzahl der Ein- und Ausgänge sowie deren Typisierung. Um Zeit zu sparen und die Programmierung übersichtlicher zu machen, wird ein eigenes Array erstellt, das diese Informationen enthält. Dadurch können wichtige Informationen schnell abgerufen werden, ohne auf die ursprüngliche Datenquelle zurückzugreifen.

ImportMethodsFromJson

```
1  async function ImportMethodsFromJsonFile() {
2      let methodsJson = [];
3      let methodsName = [];
4      await LoadJsonContent().then(result => {
5          result.forEach(item => {
6              methodsJson = methodsJson.concat(item.methods);
7              methodsJson.forEach(method => {
8                  let names = method.name;
9                  methodsName.push(names);
10             })
11             return methodsName;
12         });
13     }).catch(error => {
14         console.log("Fehler beim Verarbeiten der JSON-Daten fuer ↩
               Classnames: " + error.message);
15     });
16     return methodsName;
17 }
```

Listing 4.19: Code der ImportMethodsFromJson Funktion

Da sich die Funktion „ImportMethodsFromJson“ nur auf die Extraktion und Speicherung der Methodennamen konzentriert, wird sie nur für die Syntaxanalyse verwendet. Es werden keine zusätzlichen Daten wie Parameter oder Arten der Rückgabe gespeichert. Das hat den Grund, dass diese Analyseebene sich auf die formale Struktur und Syntax der Programmiersprache konzentriert. Hierbei wird überprüft, ob die verwendeten Methoden korrekt in Bezug auf die Syntax der Programmiersprache sind, ohne dabei die Logik oder Semantik der Methoden zu berücksichtigen. Im Gegensatz dazu wird für die Semantikanalyse eine separate Funktion namens „ImportMethodsWithParameter“4.20 verwendet, die alle Methodeninformationen einschließlich Parameter und Rückgabetypen verarbeitet.

ImportMethodsWithParameter

```
1
2          result.forEach(item => {
3              const classNames = item.classNames;
4              const methods = item.methods;
5              methods.forEach(methods=>{
```

```
6                    const methodename = methods.name;
7                    const parameters = methods.parameters;
8                    const allParameters = [];
9                    parameters.forEach(parameters => {
10                       let type = parameters;
11                       if (parameters.includes('.')) {
12                           let shortenedType = parameters.includes('.') ↩
                                 ? parameters.split('.').pop() : parameters;
13                           type = shortenedType;
14                       }
15                       const typeObj = {
16                           typ: type
17                       };
18                       allParameters.push(typeObj);
19                   });
20                   const numberOfParameters = parameters.length;
21                   const returntyp = methods.returntype;
22                   const MethodObject = {
23                       classname: classNames,
24                       methodenames : methodename,
25                       parameter : allParameters,
26                       lenghts : numberOfParameters,
27                       returntyp : returntyp
28                   };
29                   MethodsWithParameter.push(MethodObject);}...
```

Listing 4.20: Ausschnitte der ImportMethodsWithParameter Funktion

Die Funktion „ImportMethodsWithParameter“ führt einen entscheidenden Schritt zur Vorbereitung der semantischen Analyse durch. Dadurch werden den Verfahren die notwendigen Daten hinzugefügt. Es werden sowohl der Klassenname der Methode als auch die Parameter und deren Anzahl festgelegt. Außerdem wird der Rückgabetyp angegeben. Diese Informationen dienen dazu, sicherzustellen, dass die Methoden korrekt verwendet und ausgeführt werden, was für die spätere semantische Analyse von entscheidender Bedeutung ist.

ImportConstructorsFromJsonFile

```
1  async function ImportConstructorsFromJsonFile() {
2  ...
3                    const numberOfParameters = parameters.length;
4                    const constructorObj = {
5                        name: constructorName,
6                        parameters: allParameters,
7                        lengths: numberOfParameters
8                    };
9
10                   allConstructors.push(constructorObj);
11               });
12           });...
```

Listing 4.21: Auschnitte der ImportConstructorsFromJsonFile Funktion

Die Funktion „ImportConstructorsFromJsonFile“ funktioniert ähnlich. Diese Funktion verwaltet die aktuellen Konstruktoren innerhalb der Klassen. Der Name des Konstruktors, die Anzahl der Parameter und die Typen der Parameter sind in dem erstellten Array enthalten. Falls der Benutzer einen Konstruktor verwendet, sind diese Informationen für die semantische Analyse relevant.

sortByMethodenParameterCount

```
1  async function sortByMethodenParameterCount(methods) {
2      methods.sort((a,b) => b.lenghts - a.lenghts);
3      console.log(methods);
4      return methods;
5  }
```

Listing 4.22: Auschnitte der sortByMethodenParameterCount Funktion

Sort Funktionen gibt es zwei, hier wird die Methode oder Constructor mit den meisten Parameter im Array nach oben sortiert. Das Umsortieren wird gemacht, wenn innerhalb der Semantik-Überprüfung eine Methode oder Consctructor auftritt und zuerst mit der höchstmöglichen Parameter Anzahl gerechnet wird. Hier durch wird ausgeschlossen, dass die Semantikanalyse schon abbricht, wenn man zu viele Parameter hat, obwohl es Funktionen gibt, die mit demselben Namen mehr Parameter benötigen.

ImportSuperclassWithClassname

```
1
2  async function ImportSuperclassWithClassname(){
3      const data =  await LoadJsonContent();
4      const classNamesWithSuperClass = data.map(item => {
5          return { classNames: item.classNames.toLowerCase(), ↩
              superClass: item.superClass || "" , method: item.methods};
6      });
7      return classNamesWithSuperClass;
8  }
```

Listing 4.23: Code der ImportSuperclassWithClassname Funktion

Innerhalb dieser Klasse werden die Informationen für die Semantikanalyse speziell für das Segment „SystemInit“ vorbereitet. Neben den Klassennamen werden auch die Superklassen und die Methodennamen der Superklassen benötigt. Mit diesen Informationen ist es möglich, die Semantik des Segments zu überprüfen und sicherzustellen, dass die Verwendung der Klassen, Superklassen und Methoden korrekt ist. Die Superklassen liefern wichtige Referenzen und Funktionalitäten auf, die im Segment „SystemInit“ zugegriffen wird. Durch die Vorbereitung der relevanten Informationen wird eine effektive und genaue Semantikanalyse ermöglicht.

4.5 Klasse Syntax-Analyser

Das Parsing in der Klasse „Syntax-Analyser“ behandelt den Anwendungsfall der Syntax-Analyse, wie im Anforderungsfall 3.1 beschrieben. Der SyntaxAnalyser durchläuft verschiedene Schritte, um sicherzustellen, dass der Quellcode syntaktisch korrekt ist. Dabei analysiert er den Quellcode Zeichen für Zeichen oder Token für Token und überprüft, ob die Verwendung den syntaktischen Regeln der Programmiersprache entspricht. Für die Entwicklung des Syntax-Analysers wurden verschiedene Entscheidungen getroffen, Algorithmen implementiert und Codes geschrieben. Im Folgenden werden diese näher erläutert.

4.5.1 Grafische Darstellung des Parsing

Die Grafik zeigt, wie das Parsen innerhalb der Syntax-Klasse mithilfe anderer Klassen realisiert wurde. Im ersten Schritt des Algorithmus werden die Informationen von der Webseite abgerufen. Im nächsten

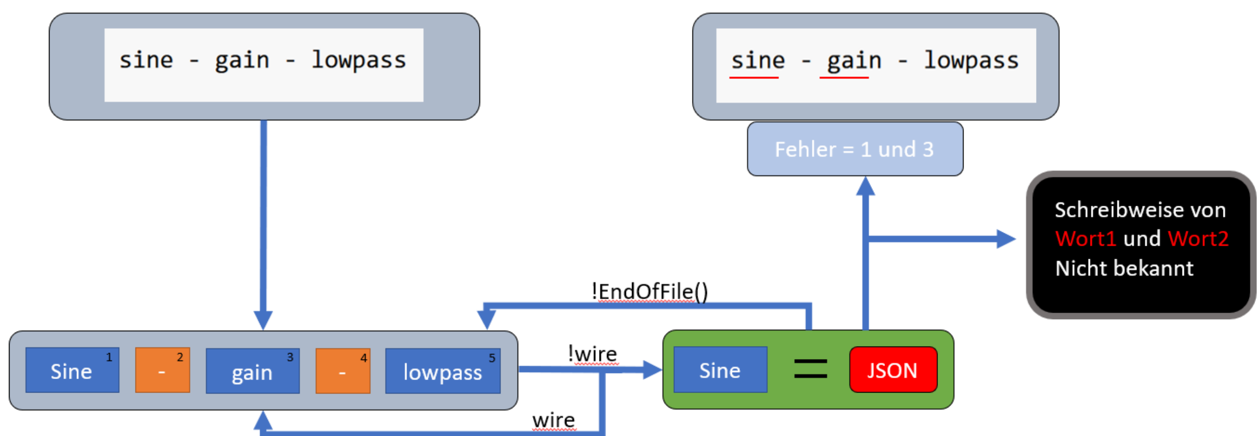


Abbildung 4.4: Grafische Darstellung des Parsing

Schritt erfolgt die Tokenisierung, bei der die einzelnen Elemente und Tokens erkannt und in ein Array übertragen werden. Anschließend wird jedes Element, sofern es sich nicht um einen Connector (Wire) handelt, mithilfe der geladenen JSON-Dateien auf seine syntaktische Korrektheit überprüft. Am Ende des Algorithmus wird überprüft, ob Fehler aufgetreten sind. Wenn Fehler vorliegen, wird das Syntax-Highlighting durchgeführt und eine Ausgabe in der Konsole erzeugt.

4.6 Funktionen

Im folgenden Abschnitt wird fachlich erläutert, wie die Syntaxanalyse mithilfe der verschiedenen Funktionen umgesetzt wird. Dazu wird der Code der Funktion „CheckvalidityOfSyntaxFromJson()“ in einzelne Teile aufteilt und näher erklärt. Dabei wird auch auf die Verbindung zu vorherigen Funktionen eingegangen.

```
1 async function CheckValidityOfSyntaxFromJSON() {
2     const userInput = await GetInformationFromWebsite();
3     const errorPostion = [];
4     const errorWordArray = [];
```

Listing 4.24: Auschnitte der CheckValidityOfSyntaxFromJSON Funktion

Im ersten Teil der Funktion werden zwei Arrays erstellt. Das erste Array ist das „Errorposition“-Array, in dem die Position des Fehlers im Code festgehalten wird. Dies umfasst die Position im Satz sowie die Zeilennummer, in der der Fehler auftritt. Das zweite Array ist das „errorWordArray“, in dem das fehlerhafte Wort selbst gespeichert wird. Dies hat den Vorteil, dass während des Syntax Highlightings nicht jedes Mal das Wort im Eingangsarray gesucht werden muss, sondern bereits im „errorWordArray“ vorhanden ist und direkt markiert werden kann. Das erste Element des Arrays wird durch den Aufruf der Funktion „GetInformationFromWebsite()“ 4.8 aus dem JSON Loader erhalten. Diese Funktion übernimmt die Tokensierung des Codes und teilt ihn in einzelne Zeilen auf.

```
1     for (let lineIndex = 0; lineIndex < userInput.length; ↩
          lineIndex++) {
2         let error = [];
3         let errorWord = [];
4         for (let numberOfInstance = 0; numberOfInstance < ↩
              userInput[lineIndex].length; numberOfInstance++) {
```

```
5                       switch (true) {
```

Listing 4.25: Auschnitte der CheckValidityOfSyntaxFromJSON Funktion

Innerhalb dieses Abschnitts lässt sich die Struktur in zwei Ebenen erkennen. Auf der ersten Ebene wird mit dem Parameter „lineIndex“ die Zeilenposition betrachtet. Hierbei werden die Positionierung jeder Zeile sowie deren Anzahl festgelegt. Zudem werden zwischen den Arrays die entsprechenden Beziehungen zwischen den Zeilen und den Instanzen hergestellt, wie zuvor beschrieben. Auf der zweiten Ebene wird dann mit dem Parameter „numberOfInstance“ über alle Instanzen innerhalb einer Zeile iteriert. Hierbei werden die spezifischen Eigenschaften und Informationen der Instanzen behandelt.

```
1                   case ↩
                        userInput[lineIndex][numberOfInstance].includes("-"):
2                        console.log("Es Handelt sich um ein Wire");
3                        break;
4                   case ↩
                        userInput[lineIndex][numberOfInstance].includes("label"):
5                        for (i = numberOfInstance; i < ↩
                            userInput[lineIndex].length; i++) {
6                            if (userInput[lineIndex][i].endsWith('"')) {
7                                numberOfInstance = i;
8                                break;}}
9                        console.log("Es Handelt sich um ein Label");
10                       break;
```

Listing 4.26: Auschnitte der CheckValidityOfSyntaxFromJSON Funktion

Der Algorithmus wird mithilfe einer switch-case-Abfrage umgesetzt. Dabei wird Token für Token durchgegangen und auf verschiedene Fälle überprüft. Im ersten Fall wird geprüft, ob es sich bei der aktuellen Instanz um ein Wire handelt. Falls dies nicht der Fall ist, wird zum nächsten Fall übergegangen. In diesem Fall wird überprüft, ob es sich um ein Label handelt. Hierbei wird auch auf den Anfang und das Ende des Labels geachtet. Anschließend wird die aktuelle Position der „numberOfInstance“ gesetzt, um sicherzustellen, dass die nachfolgenden Instanzen nicht doppelt überprüft werden, da sie für die Semantik irrelevant sind.

```
1                   case ↩
                        (userInput[lineIndex][numberOfInstance].includes("<") ↩
                        || ↩
                        userInput[lineIndex][numberOfInstance].includes(">") ↩
                        || ↩
                        userInput[lineIndex][numberOfInstance].includes("^") ↩
                        || ↩
                        userInput[lineIndex][numberOfInstance].includes("v"))&& ↩
                        userInput[lineIndex][numberOfInstance].length:
2                        console.log("Es ist ein Layout");
3                        break;
4                   case ↩
                        (userInput[lineIndex][numberOfInstance].includes("a") ↩
                        || ↩
                        userInput[lineIndex][numberOfInstance].includes("d") ↩
                        || ↩
                        userInput[lineIndex][numberOfInstance].includes("s") ↩
                        || ↩
                        userInput[lineIndex][numberOfInstance].includes("w")) ↩
                        && userInput[lineIndex][numberOfInstance].length ↩
                        <= 1:
```

```
5                       console.log("Es ist ein Layout");
6                       break;
```

Listing 4.27: Auschnitte der CheckValidityOfSyntaxFromJSON Funktion

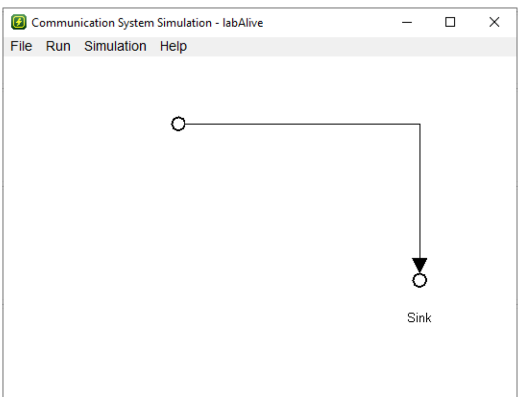


Abbildung 4.5: Grafische Darstellung von layout "1 >v 2"

Innerhalb der ersten Abfrage wird überprüft, ob benutzerspezifische Layout-Bedingungen für die Schaltung festgelegt sind. Ein solches Layout könnte beispielsweise wie in Abbildung 4.5 aussehen.

```
1                   case (typeof userInput[lineIndex][numberOfInstance] ↩
                        === 'string'):
2                       const returnvalue = await ↩
                            CheckSyntaxInWaveForms(userInput, error, ↩
                            lineIndex, numberOfInstance);
3                       if (returnvalue == true)
4                           break;
5                       const methodenFehler = await ↩
                            CheckSyntaxInMethods(userInput, error, ↩
                            lineIndex, numberOfInstance);
6                       const classnamesFehler = await ↩
                            CheckSyntaxInClassnames(userInput, error, ↩
                            lineIndex, numberOfInstance);
7                       error = methodenFehler.filter(number => ↩
                            classnamesFehler.includes(number));
8                       if (error.length === methodenFehler.length && ↩
                            error.length === classnamesFehler.length) {
9                           errorWord = ↩
                                errorWord.concat(userInput[lineIndex][numberOfInstance]);
10                          break;
11                      }
12                      break;
13                  default:
14                      console.log("Eingabe nicht bekannt" + ↩
                            userInput[lineIndex][numberOfInstance]);
15                      break;
```

Listing 4.28: Auschnitte der CheckValidityOfSyntaxFromJSON Funktion

Der nachfolgende Code führt eine Analyse der Klassennamen und Methodennamen durch, bei der die syntaktische Logik der Programmiersprache überprüft wird. Dabei werden mögliche Fehler in diesen Elementen identifiziert, um sicherzustellen, dass der Code den syntaktischen Regeln der Programmiersprache entspricht. Zunächst wird hierfür überprüft, ob die übergebene Instanz eine Waveform ist. Dafür wird die aktuelle Instanz an die Funktion 4.29 „CheckSyntaxInWaveForms()“ übergeben.

Wenn diese Funktion zurückgibt, dass es sich bei der Instanz um eine Waveform handelt, endet die Überprüfung für diese Instanz und es wird mit der nächsten fortgefahren. Andernfalls wird mit der Überprüfung der Methodennamen fortgefahren. Innerhalb der Methodenüberprüfung wird die aktuelle Instanz an die Funktion 4.30 „CheckSyntaxInMethods“ weitergeleitet, in der überprüft wird, ob die Instanz eine existierende Methode ist. Wenn dies nicht der Fall ist, wird die Fehlerposition zurückgegeben. Wenn jedoch innerhalb der Klassennamenüberprüfung mit der Funktion 4.31 „CheckSyntaxInClassnames“ ein Fehler auftritt mit der aktuellen Instanz, die übergeben wurde, wird dieser Fehler dem „errorPosition“-Array hinzugefügt. Wenn nur eine der beiden Funktionen einen Fehler ergibt, wird kein Fehler gemeldet.

```
1
2  async function CheckSyntaxInWaveForms(userInput, fehler, lineIndex, ↩
       numberOfInstance) {
3      const Waveforms = ["SINE",
4          "COSINE",
5          "TRIANGLE",
6          "SAWTOOTH",
7          "SQUARE",
8          "DC",
9          "RANDOM_SQUARE",
10         "LAPLACE",
11         "DIRAC_DELTA",
12         "SAMPLES_LOCAL_SELECT",
13         "SAMPLES_SERVER_SELECT",
14         "SELECTED_SAMPLES_FROM_SERVER"];
15     for (i = 0; i < Waveforms.length - 1; i++) {
16         if (Waveforms[i].toLowerCase() == ↩
               userInput[lineIndex][numberOfInstance].toLowerCase())
17             return true;
18     }
19     return false;
20 }
```

Listing 4.29: Code der CheckSyntaxInWaveForms Funktion

Überprüfung der aktuellen Instanz darauf, ob es sich um eine Waveform handelt. Falls dies der Fall ist, wird true zurückgegeben.

```
1  async function CheckSyntaxInMethods(userInput, fehler, lineIndex, ↩
       numberOfInstance) {
2      const methods = await ImportMethodsFromJsonFile();
3      let similarity = 0.0;
4      for (let i = 0; i < methods.length; i++) {
5          similarity = ↩
               CalculateSimilarity(userInput[lineIndex][numberOfInstance], ↩
               methods[i]);
6          if (similarity >= similarityIndex) {
7              break;
8          }
9      }
10     if (similarity <= similarityIndex) {
11         fehler = fehler.concat(numberOfInstance);
12     }
13     return fehler;
14 }
```

Listing 4.30: Code der CheckSyntaxInMethods Funktion

Mithilfe dieser Funktion wird überprüft, ob die aktuelle Instanz eine Methode ist. Dafür werden zunächst alle Methoden aus der JSON-Datei eingebunden. Anschließend wird für jede vorhandene Methode die Ähnlichkeit mit der aktuellen Instanz berechnet. Diese Ähnlichkeit wird mithilfe der Funktion „CalculateSimilarity“ ermittelt, welche zurückgibt, wie ähnlich eine Eingabe dem Methodennamen in der JSON-Datei ist. Dies dient dazu, dass der Benutzer innerhalb der IDE auch abgekürzte Versionen von Bauelementen und Methoden verwenden kann. Falls die Ähnlichkeit höher ist als ein festgelegter Wert, existiert das Bauelement bzw. die Methode.

```
1  async function CheckSyntaxInClassnames(userInput, fehler, lineIndex, ↩
       numberOfInstance) {
2      const classNames = await ImportClassFromJsonFile();
3      let similarity = 0.0;
4      for (let i = 0; i < classNames.length; i++) {
5          similarity = ↩
               CalculateSimilarity(userInput[lineIndex][numberOfInstance], ↩
               classNames[i]);
6          if (similarity >= similarityIndex) {
7              break;
8          }
9      }
10     if (similarity <= similarityIndex) {
11         fehler = fehler.concat(numberOfInstance);
12     }
13     return fehler;
14 }
```

Listing 4.31: Code der CheckSyntaxInClassnames Funktion

Ähnlich wie bei der Funktion „CheckSyntaxInMethods“ wird auch hier die aktuelle Instanz übergeben und alle Klassennamen aus der JSON-Datei verwendet, um die Ähnlichkeit zu berechnen. Wenn die berechnete Ähnlichkeit größer oder gleich der Mindestähnlichkeit ist, wird kein Fehler angezeigt. Falls jedoch die Ähnlichkeit zu niedrig ist, wird eine Fehlerposition gesetzt.

```
1  function CalculateSimilarity(word1, word2) {
2      const length = Math.max(word1.length, word2.length);
3      let matches = 0;
4      for (let i = 0; i < length; i++) {
5          if (word1[i] === word2[i]) {
6              matches++;
7          }
8      }
9      return matches / length;
10 }
```

Listing 4.32: Code der CalculateSimilarity Funktion

In dieser Funktion wird die übergebene Instanz mit dem gegebenen Klassennamen oder Methodennamen verglichen, und basierend darauf wird die Ähnlichkeit berechnet und zurückgegeben.

```
1              }
2          }
3          errorPostion.push(error);
4          errorWordArray.push(errorWord);
5      }
6      console.log(errorPostion);
7      return [errorPostion, errorWordArray];
8  }
```

Listing 4.33: Auschnitte der CheckValidityOfSyntaxFromJSON Funktion

Zum Schluss der „CheckValidityOfSyntaxFromJSON“ werden die Errorpostion und Errorwordarray mit den aktuellen Zeilen ergänzt und zurückgegeben.

4.7 Klasse Semantik-Analyser

Innerhalb der Semantik-Analyse wird, wie bereits in den Anwendungsfällen beschrieben, die semantische Logik der Programmiersprache überprüft. Hierbei werden verschiedene Funktionen aus den vorherigen Schritten aufgerufen und verwendet, um dieses Ziel zu erreichen. Die Semantik-Analyse stellt sicher, dass der Code in Bezug auf die Bedeutung und Logik der Programmiersprache korrekt ist. Durch die semantische Analyse können potenzielle Fehler und Inkonsistenzen im Code frühzeitig erkannt werden, um eine fehlerfreie und sinnvolle Ausführung des Programms sicherzustellen.

4.7.1 Segmente

Segmente sind verschiedene Teile des Codes, der zusammen gehört. Beispiele hier für ist das „User defined Label“ oder die „SystemInit“. Innerhalb der folgende Tabelle 4.4 werde die verschiede Segmente aufgelistet:

Segmentname	**Beispiel**
System Init	sine Lowpass
Connection	- "1 >v 2"
Signal Visualization	waveform triangle 1V 500Hz
User defined label	label "Hello World!"
Display Settings	scope show
Variablen	Test = ParameterTyp

Tabelle 4.4: In & Out JSON

Die Segmente werde im spätern Verlauf bei der Entwicklung des SemantikAnalyser benötigt.

4.7.2 Grafische Darstellung der Semantikanalyse

Ähnlich wie bei der Syntaxanalyse wird auch bei der Semantikanalyse auf das JSON-Dateiformat zugegriffen. Allerdings liegt bei der Semantikanalyse der Fokus auf der Logik des Codes und detaillierten Informationen über die Klassen aus der JSON-Datei. Nachdem die Segmente erkannt wurden, wird die Semantikanalyse zunächst auf Ebene der einzelnen Instanzen innerhalb der Segmente durchgeführt. Dabei wird jede einzelne Instanz auf ihre semantische Korrektheit überprüft. Sobald die Überprüfung der einzelnen Instanzen abgeschlossen ist, erfolgt die Analyse des gesamten Segments und anschließend die Überprüfung der Semantik zwischen den Segmenten.
Die Semantik zwischen den Bauelementen bezieht sich darauf, dass viele Bauelemente unterschiedliche Ein- und Ausgänge haben können. Dies betrifft sowohl die Anzahl der Ein- und Ausgänge als auch deren Typisierung. Nicht jeder Ausgangstyp ist kompatibel mit jedem Eingangstyp. Innerhalb der Segmentprüfung sind insbesondere Methoden und Konstruktoren relevant. Wird der Parameter mit dem korrekten Typ angegeben? Ist die Anzahl der Parameter korrekt angegeben? Existiert möglicherweise eine gleichnamige Methode mit mehr Parametern?
Die Semantikanalyse stellt sicher, dass die Logik des Codes korrekt ist und dass alle Bauelemente und deren Verbindungen sinnvoll und fehlerfrei sind. Durch die Überprüfung der semantischen Korrektheit

wird sichergestellt, dass das Programm die beabsichtigten Aktionen ausführt und keine logischen Fehler enthält.

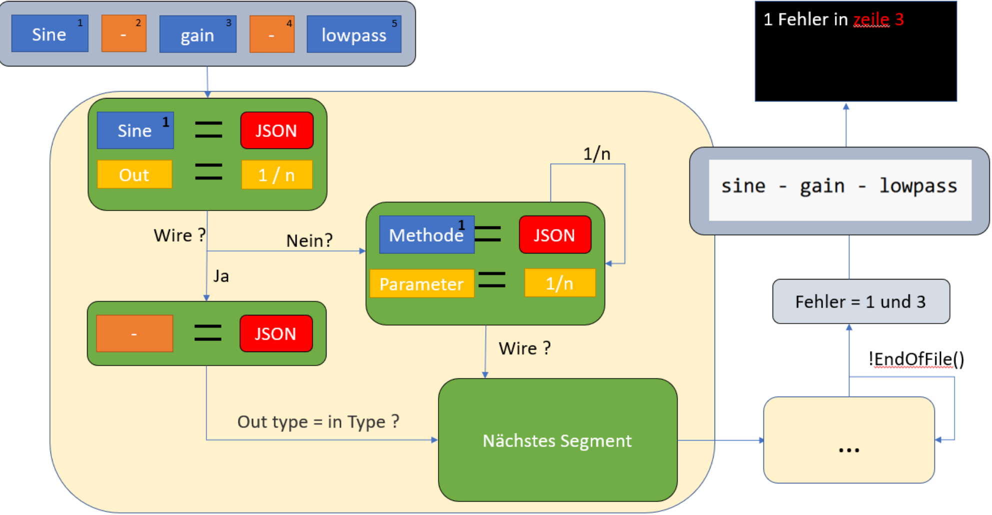


Abbildung 4.6: Grafische Darstellung der Semantikanalyse

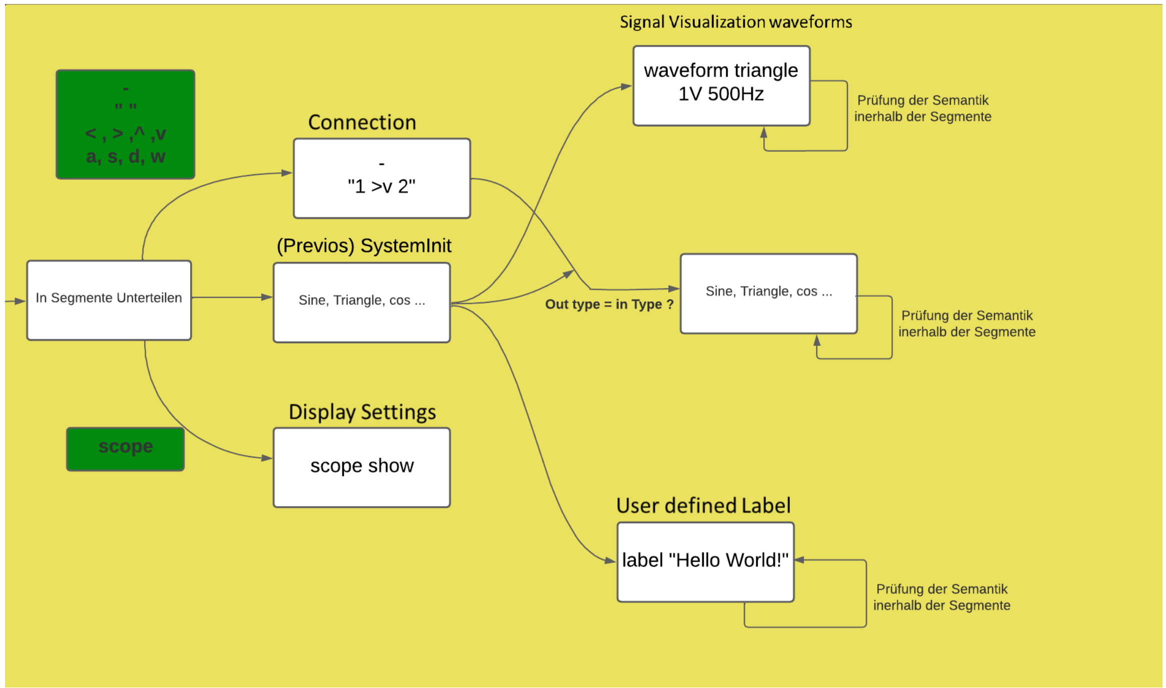


Abbildung 4.7: Detaillierte Darstellung der Segmente

4.7.3 Bereits entwickelte Funktionen

```
1  async function SperateLineArrayIntoSegments(userInputInCurrentLine){
2      for (let numberOfInstance = 0; numberOfInstance < ↩
           userInputInCurrentLine.length; numberOfInstance++) {
3          switch (true) {
4              case ↩
                  userInputInCurrentLine[numberOfInstance].toLowerCase().includes("label"):
5                  const status = ↩
                      CheckValidityOfLabel(userInputInCurrentLine,numberOfInstance);
6                  if(status[0] == true){
7                      numberOfInstance = status[1];
8                      break;
9                  }
```

Listing 4.34: Aktueller der SperateLineArrayIntoSegments Funktion

Ähnlich wie bei der Syntaxanalyse wird auch bei der Semantikanalyse schrittweise vorgegangen, indem zuerst die Zeilen und dann die einzelnen Instanzen analysiert werden. Während der Semantikanalyse werden die einzelnen Elemente noch weiter in Segmente unterteilt. Um diese Unterteilung vornehmen zu können, greifen wir auf bestimmte Schlüsselwörter zurück. Ein Beispiel für ein solches Schlüsselwort könnte das Wort „label“ sein. Sobald ein solches Schlüsselwort erkannt wird, wird die Funktion „CheckValidityOfLabel()“ aufgerufen. Innerhalb dieser Funktion wird die Semantik des übergebenen Labels überprüft.

```
1  function  CheckValidityOfLabel(userInputCurrentLine,i){
2      let valueLabel = [];
3      if(userInputCurrentLine[i+1].toLowerCase().includes('"')){
4          valueLabel.push(userInputCurrentLine[i]);
5          valueLabel.push(userInputCurrentLine[i+1]);
6          for(j =i+2; j < userInputCurrentLine.length ;j++){
7             if(!userInputCurrentLine[j].endsWith('"')){
8                 valueLabel.push(userInputCurrentLine[j]);
9             }else{
10                valueLabel.push(userInputCurrentLine[j]);
11            break;
12            }
13         }
14     }else{
15         return false;
16     }
17     objSegments._currentLabel = valueLabel;
18     return [true,j-1];
19 }
```

Listing 4.35: Code der CheckValidityOfLabel Funktion

Für ein benutzerdefiniertes Label ist es wichtig, dass nach dem Label ein ‚"‘ (Anführungszeichen) folgt und mit ‚"‘ wieder geschlossen wird. Innerhalb meines Codes wird überprüft, ob an der Stelle nach dem Label das erste Mal das Anführungszeichen auftritt. Wenn dies der Fall ist, durchläuft der Code eine Schleife, um das letzte Auftreten des Anführungszeichens zu finden. Sobald dies erfolgreich ist, werden alle relevanten Informationen für das benutzerdefinierte Label im globalen Objekt und in der entsprechenden Variable für das Label, „_ currentLabel“, gespeichert. Dieser Vorgang gewährleistet, dass das benutzerdefinierte Label korrekt formatiert ist und die benötigten Informationen für die weitere Verarbeitung zur Verfügung stehen. Dadurch wird sichergestellt, dass das Label in der semantischen Analyse korrekt verwendet werden kann.

```
1             case ↩
                  (CheckValidityOfConnector(userInputInCurrentLine[numberOfInstance])== ↩
                  true) :
2                 objSegments._Connection = ↩
                      [userInputInCurrentLine[numberOfInstance-1], ↩
                      userInputInCurrentLine[numberOfInstance], ↩
                      userInputInCurrentLine[numberOfInstance+1]];
3                 break; break;
4                 }
```

Listing 4.36: Aktueller der SperateLineArrayIntoSegments Funktion

Nun wird das Segment „Connection“ oder auch der Connector überprüft. Hierfür wird die aktuelle Instanz an eine Funktion namens CheckValidityOfConnector() übergeben, die true zurückgibt, falls die aktuelle Instanz ein Connector ist. Wenn dies der Fall ist, wird auch hier ein globales Objekt namens „_ Connection“ gesetzt. Zusätzlich wird erfasst, welche Segmente sich links und rechts vom Connector befinden. Dadurch können während der Überprüfung von Segmenten Informationen einfach über die Connectoren abgerufen werden. Diese Überprüfung gewährleistet, dass die Verbindungen zwischen den Segmenten korrekt sind und die erforderlichen Informationen für die semantische Analyse zur Verfügung stehen. Durch die Verwendung von Connectoren können Informationen zwischen verschiedenen Segmenten ausgetauscht und verarbeitet werden.

```
1  function CheckValidityOfConnector(userinput){
2      if(userinput.includes("-")  || userinput.includes("w")||
3          userinput.includes("v")  || userinput.includes("a")||
4          userinput.includes("^")  ||  userinput.includes("s")||
5          userinput.includes(">")  ||  userinput.includes("d")||
6          userinput.includes("<") ){
7          console.log("connector")
8          return true;}
9
10     return false;
11 }
```

Listing 4.37: Code der CheckValidityOfConnector Funktion

Es wird überprüft, ob die aktuelle Instanz ein Connector ist, indem sie mit allen möglichen Connectoren verglichen wird.

```
1             case (typeof userInputInCurrentLine[numberOfInstance] ↩
                  === 'string'):
2                 if(await ↩
                      CheckIfInputIsClass(userInputInCurrentLine[numberOfInstance])){
3                      objSegments._currentSystemInit = ↩
                           userInputInCurrentLine[numberOfInstance];
4                      CheckValidityOfSystemInit(userInputInCurrentLine,numberOfInstance);
5             }
6                 break;
```

Listing 4.38: Aktueller der SperateLineArrayIntoSegments Funktion

In dieser Segmentprüfung beziehen wir uns auf das SystemInit-Segment. Das SystemInit-Segment umfasst das aktuelle System, bestehend aus einem Bauelement und optionalen Methoden mit Parametern. Das SystemInit-Segment dient als Ausgangspunkt für andere Segmente, wie in Abbildung 4.7 dargestellt. Hierfür wird zuerst mit der Funktion „CheckIfInputIsClass“ geprüft, ob es eine Klasse ist.

```
1  async function CheckIfInputIsClass(input){
2      const classNames = await ImportClassFromJsonFile();
3      let similarity = 0.0;
4      for (let i = 0; i < classNames.length; i++) {
5          similarity = CalculateSimilarity(input, classNames[i]);
6          if (similarity >= similarityIndex) {
7              return true;
8          }
```

Listing 4.39: Aktueller der SperateLineArrayIntoSegments Funktion

Die Funktion ist eine angepasste Version von CalculateSimilarity() (siehe 4.32). Dabei wird die aktuelle Instanz mit allen möglichen Klassennamen und Waveforms verglichen. Bei der Klassenprüfung wird überprüft, ob die Ähnlichkeit einen bestimmten Schwellenwert überschreitet, und in diesem Fall wird true zurückgegeben. Wenn die Ähnlichkeit jedoch unterhalb des Schwellenwerts liegt, handelt es sich nicht um ein SystemInit-Segment, sondern möglicherweise um einen falsch geschriebenen Eintrag oder eine andere Art von Segment. Dieser Fall wird jedoch bereits während der Syntaxanalyse erkannt.

```
1
2  async function CheckValidityOfSystemInit(input,numberOfInstance){
3      const classnameWithSuperclass = await ↩
          ImportSuperclassWithClassname();
4      const lastSuperClass = ↩
          findLastSuperClass(classnameWithSuperclass,input[numberOfInstance].toLowerCase());
5      const currentClassArray = classnameWithSuperclass.find(item => ↩
          item.classNames.toLowerCase() === ↩
          lastSuperClass.toLowerCase());
6      const methode = currentClassArray.method;
7      console.log(methode);
8  ...
9  }
```

Listing 4.40: Aktueller Stand der CheckValidityOfSystemInit Funktion

Abschließend widmen wir uns der Implementierung der Funktion „CheckValidityOfSystemInit“. Diese Funktion hat die Aufgabe, das Segment „CheckValidityOfSystemInit“ sowie alle darin enthaltenen Instanzen zu überprüfen. Es ist jedoch wichtig zu beachten, dass diese Funktion aufgrund des zeitlichen Rahmens dieser Arbeit nicht vollständig fertiggestellt wurde und dementsprechend noch nicht getestet werden konnte.
Zu Beginn werden alle Klassen mit ihren Superklassen ermittelt. Diese Informationen werden dann an die Funktion „findLastSuperClass()“ übergeben, die die letzte Superklasse für die aktuelle Klasse ermittelt. Dieser Schritt ist notwendig, da in der letzten Superklasse in der Regel alle benötigten Methoden und Konstruktoren enthalten sind, falls diese in der aktuellen Klasse verwendet werden. Dazu greifen wir auf das oben erwähnte Array „classnameWithSuperclass“ zurück und extrahieren vorab die Methoden und Konstruktoren für die weitere Verarbeitung.

5 Fazit

5.1 Ergebnisse

Zu Beginn dieser Arbeit war die Text2App-Anwendung, auch bekannt als MyApps, lediglich ein einfaches Textfeld. Es gab keine Fehleranalyse und der einzige Weg für die Benutzer, um herauszufinden, ob ihr Code syntaktisch und semantisch korrekt war, bestand darin, ihn herunterzuladen und zu hoffen, dass die Anwendung startete. Um dieses Problem zu beheben, wurde das Textfeld durch zwei DIV-Elemente ersetzt, die mithilfe von CSS-Styling das Aussehen einer Mini-IDE erhielten. Zusätzlich wurden Zeilennummern für das Editor-Feld implementiert, um die Benutzerfreundlichkeit und Übersichtlichkeit zu verbessern. Während der Bachelorarbeit wurde hauptsächlich mit der Entwicklung von Parsern und ihren lexikalischen und semantischen Analysen gearbeitet. Im Gegensatz zur Entwicklung von IDEs für bekannte Programmiersprachen, bei denen häufig bereits fertige Datensätze der Entwickler vorhanden sind, musste der Parser für die Text2App von Grund auf neu entwickelt werden. Der Parser für die Text2App verhielt sich jedoch anders als ein herkömmlicher Parser. Statt statisch definierte Token mit der Eingabe zu vergleichen, musste die syntaktische und semantische Analyse komplett neu entwickelt werden. Hierfür wurden im Laufe der Bachelorarbeit Anwendungsfälle entwickelt und kontinuierlich verbessert. Um die beschriebenen Systeme, die den Anwendungsfällen zugrunde liegen, umzusetzen, wurden zwei Algorithmen entwickelt. Einer davon konzentriert sich ausschließlich auf die Syntaxüberprüfung, während sich der andere mit der Logik und Semantik befasst. Aufgrund des zeitlichen Rahmens der Arbeit konnte die semantische Analyse jedoch nicht vollständig umgesetzt werden. In zukünftigen Arbeiten müssen noch einige Teile der Semantikanalyse vervollständigt werden, wie z. B. die Überprüfung der Logik zwischen den Segmenten, die Prüfung der Kompatibilität von Instanzen innerhalb der Segmente (Klassen und Methoden) und die Überprüfung der Methodenlogik. Dennoch wurde die Grundlage für alle Anwendungsfälle erfolgreich entwickelt. Alle Klassen wurden so entwickelt, dass sie sich ausschließlich an der JSON-Datei orientieren. Dadurch ist nichts statisch im Code festgelegt und kann dynamisch über die Datei ergänzt und erweitert werden. Die Analyseprozesse arbeiten beide im Millisekundenbereich und sind daher für den Benutzer nicht spürbar. Dadurch wird es möglich sein, in zukünftigen Anwendungsfällen wie der Code-Ergänzung und der Anzeige von Informationen diese schnell und unbemerkt zu verarbeiten. Die Grundlage für das Einlesen und Verarbeiten des vom Benutzer eingegebenen Codes wurde vollständig entwickelt. Dieser Teil wurde intensiv getestet und Fehler wurden behandelt oder abgefangen, um eine robuste Code-Verarbeitung zu gewährleisten. Zusätzlich wurden die Anwendungsfälle auf theoretischer Ebene geplant und durch die Implementierung die Grundlage dafür geschaffen, auf der weitere Arbeiten aufbauen können. Vor Beginn der Bachelorarbeit wurde in der Projektarbeit ein Zeitplan erstellt (Abbildung 5.1), der jedoch aus zeitlichen Gründen nicht vollständig eingehalten werden konnte. Die tatsächliche zeitliche Abfolge des Projekts ist in Abbildung 5.2 dargestellt.

5.2 Herausforderungen

Es ist häufig schwierig, große Projekte wie eine webbasierte integrierte Entwicklungsumgebung umzusetzen, ohne einen umfassenden Überblick über die erforderlichen Schritte und den Gesamtaufbau des Projekts zu haben. Der Zeitaufwand für die einzelnen Arbeitsphasen während des Projekts wird in der Grafik 5.3 detailliert dargestellt. Es wurden während der Bachelorarbeit einige der geplanten Anwendungsfälle umgesetzt. Allerdings wurden aufgrund der fortlaufenden Anpassungen an der Code-Analyse, sowohl der semantischen Analyse als auch der syntaktischen Analyse, immer wieder grundlegende Änderungen an der JSON-Datei vorgenommen. Dadurch war die Entwicklung der lexikalischen Analyse und syntaktische Analyse erschwert. Statt den üblichen Parsern, bei denen feste Token verglichen werden, wurde in diesem Fall die JSON-Datei von Herrn Hirlimann verwendet. Dadurch mussten die beiden Analysen auf neue Art und Weise entwickelt und getestet werden. Zudem werden übliche Web-IDEs mit vorgefertigten Datensätzen betrieben, was in unserem Fall während der Entwicklung ebenfalls umgesetzt werden musste. Eine weitere Herausforderung bestand darin, dass im Verlauf der Arbeit keine mit JavaScript entwickelte IDE gefunden wurde, an der man sich hätte orientieren können. Die Planung der Anwendungsfälle und Funktionen, die die Web-IDE bieten sollte, sollte für zukünftige Projekte dementsprechend angepasst werden. Der Zeitrahmen von drei Monaten für die Entwicklung einer vollständigen Web-IDE mit neun Anwendungsfällen war dafür zu knapp bemessen, da innerhalb der Entwicklung einige Themen aufgetreten sind, wie unter anderem die Übertragung von nicht erkennbarem Whitespace sowie das Setzen von nicht geschriebenen Zeilenumbrüchen oder doppelte Fehler wurden nicht erkannt. Die Probleme wurden alle gelöst, kosteten aber entsprechend Zeit in der Entwicklungsphase. Diese Erkenntnisse können ebenfalls in zukünfte Arbeiten einbezogen werden.

5.3 Zeitaufwand

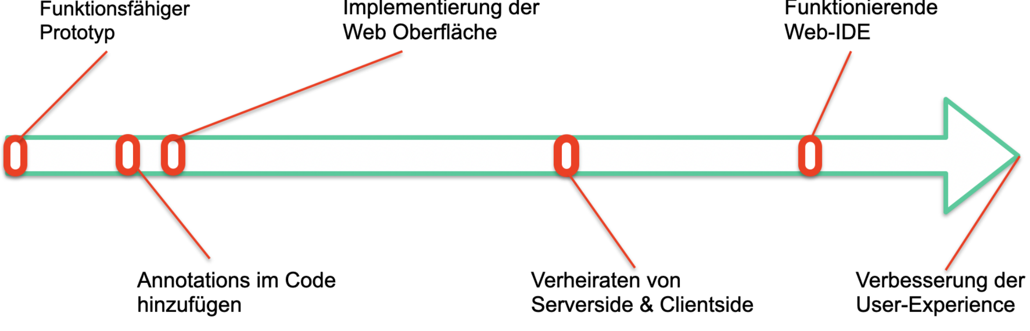


Abbildung 5.1: Ausblick Projektarbeit geplant

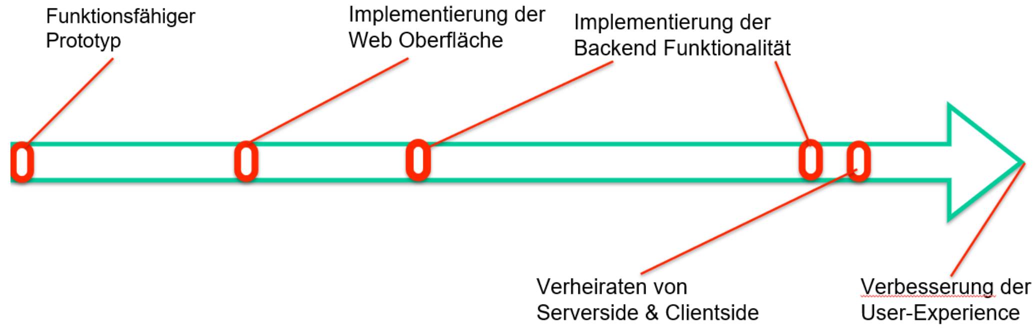


Abbildung 5.2: Ausblick Projektarbeit tatsächlich

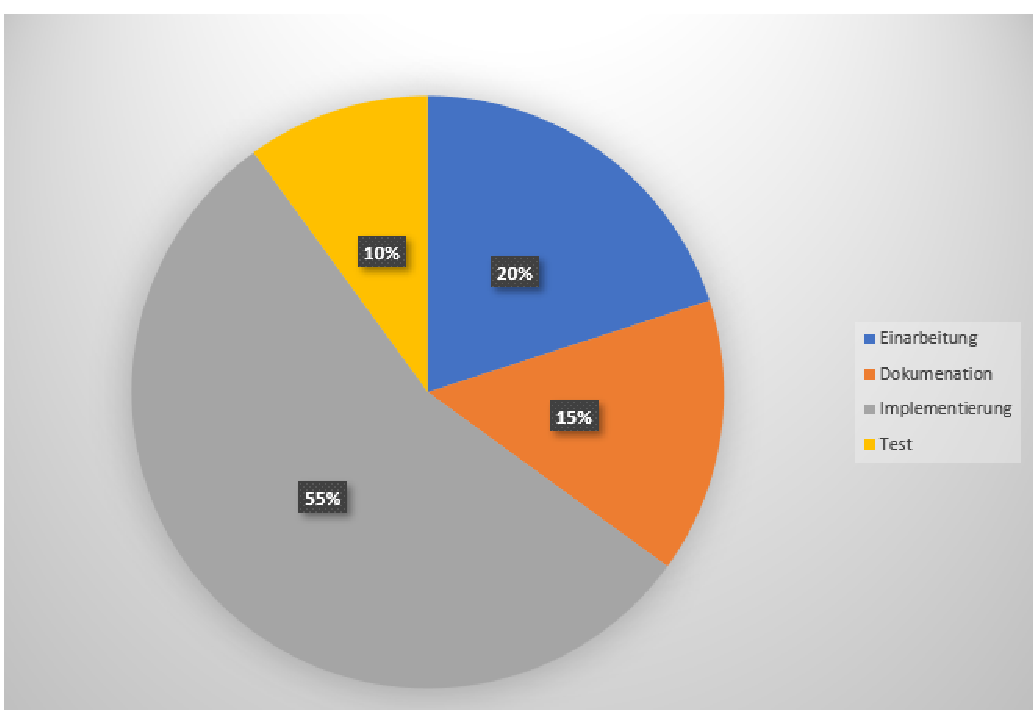


Abbildung 5.3: Zeitaufwand während der Abschlussarbeit

5.4 Ausblick

Auf die entwickelte Funktionalität können die restlichen Anwendungsfälle in Form weiterer studentischer Arbeiten als Grundlagen dienen. Ideen hier für wären untere anderem:

- Die Fertigstellung der Semantik-Analyse, Error-Message, Auto-Compliton und Informationnote
- Die Entwicklung eines benutzerfreundlichen Frontends mit zusätzlichen Funktionen für die Implementierung
- Entwicklung eines Drag-and-drop System zum Einfügen von Systemen aus einer vorgeschrieben Bibliothek mit Beispielen
- Die allgemeine Weiterentwicklung der WebIDE und Ergänzung, Beendigung laufender Entwicklung der offenen Anwendungsfälle

- Das Einbinden von „Neuronalen Netzen “, um Benutzungsfreundlichkeit zu erhöhen.

6 Anhang

6.1 Ablaufdiagramme

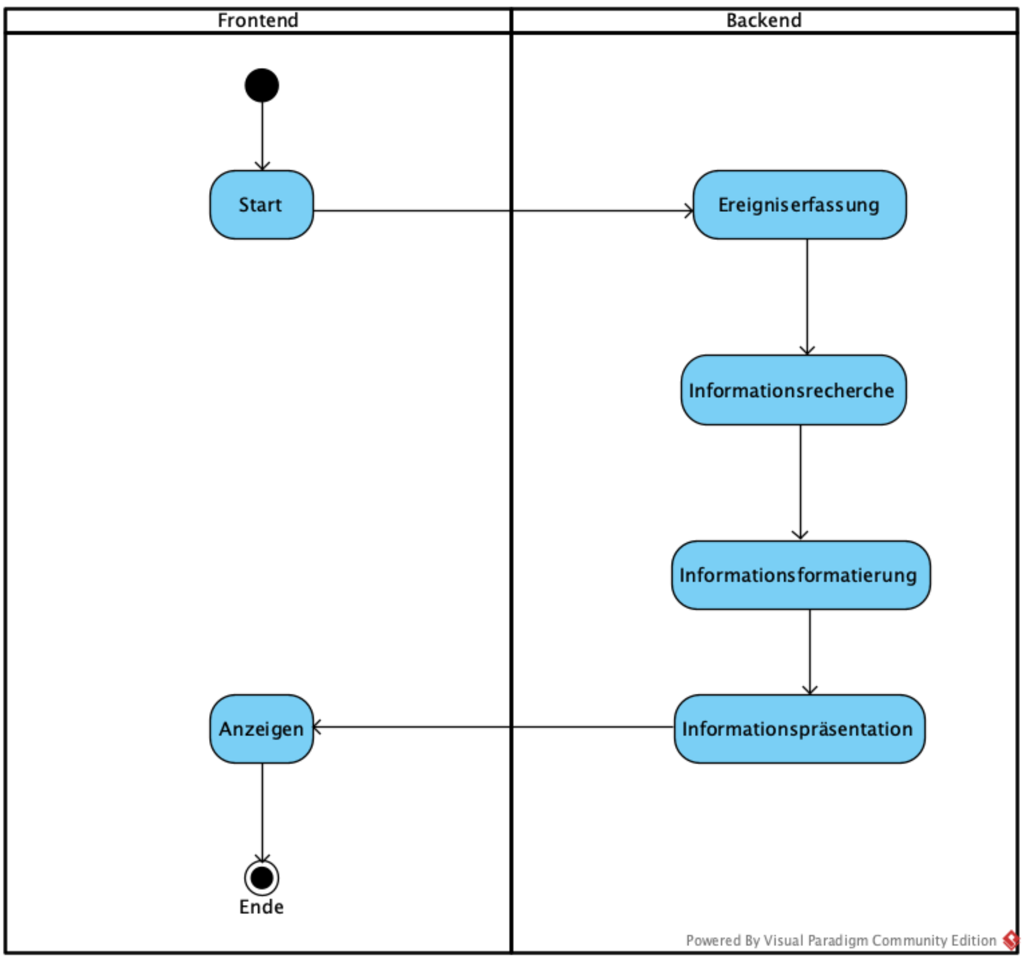


Abbildung 6.1: Ablaufdiagramme AF Information-Note

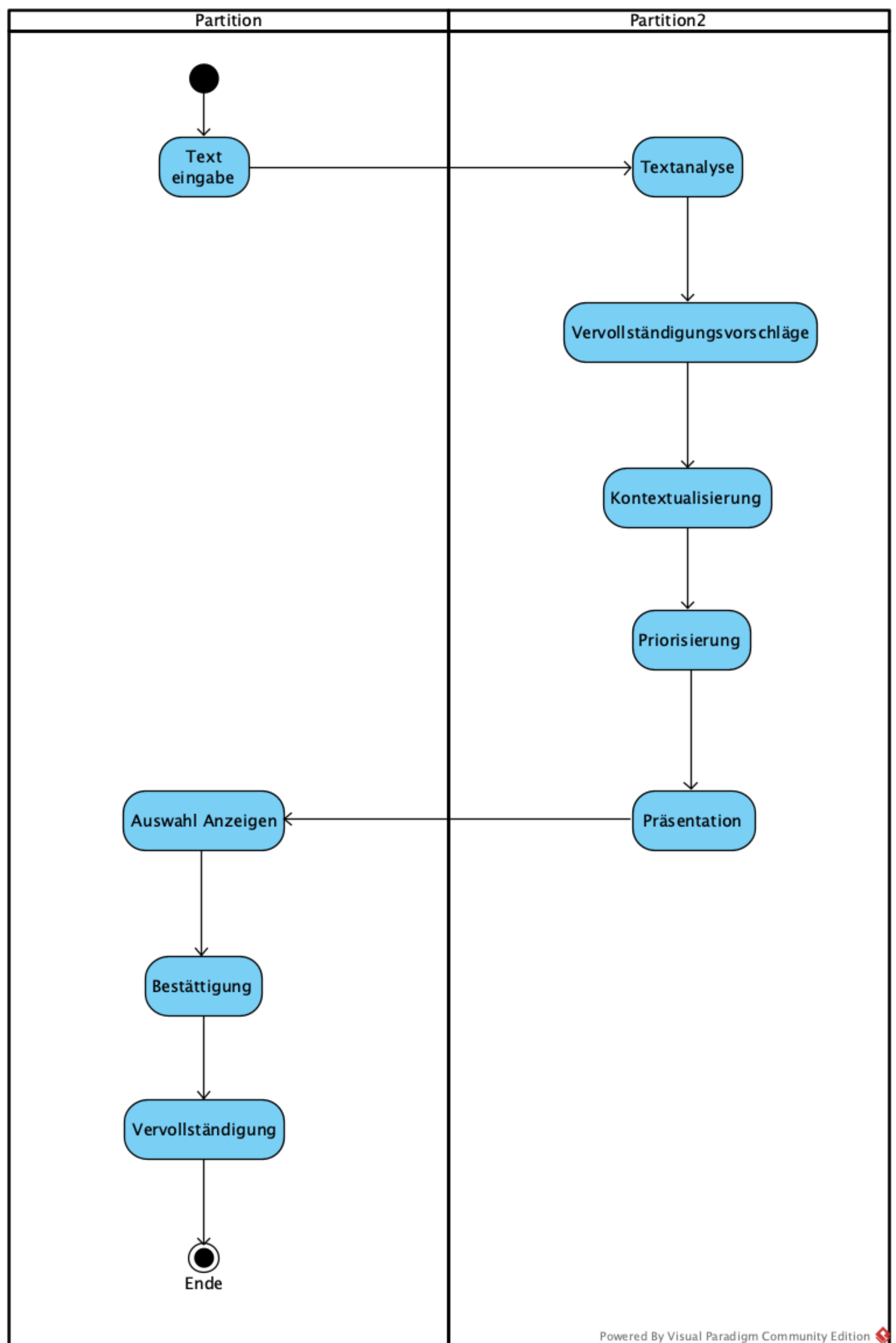


Abbildung 6.2: Ablaufdiagramme AF Auto-Compliton

6.2 QuellCode

Hier wird der Code, der einzelnen Klassen, sowie der Webside eingefügt.

6.2.1 JSONFileLoader

```
function LoadJsonContent() {
    return fetch("Data.json")
        .then(response => response.json())
        .then(data => {
            if (!data) {
                throw new Error("Ung ltiges JSON-Format.");
            }
            const result = [];
            for (const classNames in data) {
                const classData = data[classNames];
                    const inCount = classData.in?.count || "";
                    const inTypes = classData.in?.types || [];
                    const outCount = classData.out?.count || "";
                    const outTypes = classData.out?.types || [];
                    const superClass = classData.superclass || "";
                    const methods = classData.methods || [];
                    const constructors = classData.constructors || [];
                    const waveForms = classData.fields;
                    result.push({
                        classNames: classNames,
                        inCount,
                        inTypes,
                        outCount,
                        outTypes,
                        superClass,
                        methods,
                        waveForms,
                        constructors
                    });

            }

            return result;
        })
        .catch(error => {
            console.log("Fehler beim Laden der JSON-Daten: " + error.message);
            return [];
        });
}
async function allMethodsFromArray(){
    LoadJsonContent().then(result => {
        var allMethods = [];
        result.forEach(item => {
            allMethods = allMethods.concat(item.methods);
            return allMethods;
        });
        console.log(allMethods);
    }).catch(error => {
        console.log("Fehler beim Verarbeiten der JSON-Daten f r Methods: " + error.message);
    });
}

async function ImportConstructorsFromJson() {
```

```
54      try {
55          const result = await LoadJsonContent();
56          const allConstructors = [];
57          result.forEach(item => {
58              const constructors = item.constructors;
59              const allParameters = [];
60              constructors.forEach(constructor => {
61                  const constructorName = constructor.name;
62                  const parameters = constructor.parameters;
63                  parameters.forEach(parameters => {
64                      let type = parameters;
65                      if (parameters.includes('.')) {
66                          let shortenedType = parameters.includes('.') ↩
                             ? parameters.split('.').pop() : parameters;
67                          type = shortenedType;
68                      }
69                      const typeObj = {
70                          typ: type
71                      };
72                      allParameters.push(typeObj);
73                  });
74                  const numberOfParameters = parameters.length;
75                  const constructorObj = {
76                      name: constructorName,
77                      parameters: allParameters,
78                      lengths: numberOfParameters
79                  };
80
81                  allConstructors.push(constructorObj);
82              });
83          });
84
85          console.log(allConstructors);
86      } catch (error) {
87          console.log("Fehler beim Verarbeiten der JSON-Daten f r ↩
             Constructors: " + error.message);
88      }
89  }
90  async function allClassnamesFromArray() {
91      try {
92          const result = await LoadJsonContent();
93
94          const allClassnames = result.flatMap(item => item.classNames);
95
96          console.log(allClassnames);
97
98          return allClassnames;
99      } catch (error) {
100         console.log("Fehler beim Verarbeiten der JSON-Daten f r ↩
             Classnames: " + error.message);
101         return [];
102     }
103 }
104 async function ImportClassFromJsonFile() {
105     let classesJson = [];
106     await LoadJsonContent().then(result => {
107         result.forEach(item => {
```

```
108                 classesJson = classesJson.concat(item.classNames);
109                 return classesJson;
110             });
111         }).catch(error => {
112             console.log("Fehler beim Verarbeiten der JSON-Daten f r ↩
                    Classnames: " + error.message);
113         });
114         return classesJson;
115 }
116 async function ImportMethodsFromJsonFile() {
117         let methodsJson = [];
118         let methodsName = [];
119         await LoadJsonContent().then(result => {
120             result.forEach(item => {
121                 methodsJson = methodsJson.concat(item.methods);
122                 methodsJson.forEach(method => {
123                     let names = method.name;
124                     methodsName.push(names);
125                 })
126                 return methodsName;
127             });
128         }).catch(error => {
129             console.log("Fehler beim Verarbeiten der JSON-Daten f r ↩
                    Classnames: " + error.message);
130         });
131         return methodsName;
132 }
133 async function ImportSuperclassWithClassname(){
134         const data =  await LoadJsonContent();
135         const classNamesWithSuperClass = data.map(item => {
136             return { classNames: item.classNames.toLowerCase(), ↩
                    superClass: item.superClass || "" , method: item.methods};
137         });
138         return classNamesWithSuperClass;
139 }
140 
141 async function ImportInAndOutPutFromJsonFile() {
142         let ComponentInAndOut = [];
143         try {
144             const result = await LoadJsonContent();
145             result.forEach(item => {
146                 const classNames = item.classNames;
147                 let inNr  = item.inCount;
148                 let inTyp = item.inTypes;
149                 let outNr = item.outCount;
150                 let outTyp = item.outTypes;
151                 const ComponentObject = {
152                     name: classNames,
153                     inCount: inNr,
154                     inTypes: inTyp,
155                     outCount:outNr,
156                     outTypes:outTyp
157                 };
158                 ComponentInAndOut.push(ComponentObject);
159             });
160         } catch (error) {
```

```
161            console.log("Error processing JSON data for class names: " + ↩
                   error.message);
162        }
163        console.log(ComponentInAndOut );
164        return ComponentInAndOut;
165    }
166    async function ImportConstructorsWithParameter() {
167        try {
168            const result = await LoadJsonContent();
169            const allConstructors = [];
170            result.forEach(item => {
171                const constructors = item.constructors;
172                const allParameters = [];
173                constructors.forEach(constructor => {
174                    const constructorName = constructor.name;
175                    const parameters = constructor.parameters;
176                    parameters.forEach(parameters => {
177                        let type = parameters;
178                        if (parameters.includes('.')) {
179                            let shortenedType = parameters.includes('.') ↩
                                   ? parameters.split('.').pop() : parameters;
180                            type = shortenedType;
181                        }
182                        const typeObj = {
183                            typ: type
184                        };
185                        allParameters.push(typeObj);
186                    });
187                    const numberOfParameters = parameters.length;
188                    const constructorObj = {
189                        name: constructorName,
190                        parameters: allParameters,
191                        lenghts: numberOfParameters
192                    };
193
194                    allConstructors.push(constructorObj);
195                });
196            });
197
198            console.log(allConstructors);
199        } catch (error) {
200            console.log("Fehler beim Verarbeiten der JSON-Daten  f r  ↩
                   Constructors: " + error.message);
201        }
202    }
203    async function ImportMethodsWithParameter() {
204        let MethodsWithParameter = [];
205        try {
206            const result = await LoadJsonContent();
207            result.forEach(item => {
208                const classNames = item.classNames;
209                const methods = item.methods;
210                methods.forEach(methods=>{
211                    const methodename = methods.name;
212                    const parameters = methods.parameters;
213                    const allParameters = [];
214                    parameters.forEach(parameters => {
```

```
215                         let type = parameters;
216                         if (parameters.includes('.')) {
217                             let shortenedType = parameters.includes('.') ↩
                                  ? parameters.split('.').pop() : parameters;
218                             type = shortenedType;
219                         }
220                         const typeObj = {
221                             typ: type
222                         };
223                         allParameters.push(typeObj);
224                     });
225                     const numberOfParameters = parameters.length;
226                     const returntyp = methods.returntype;
227                     const MethodObject = {
228                         classname: classNames,
229                         methodenames : methodename,
230                         parameter : allParameters,
231                         lenghts : numberOfParameters,
232                         returntyp : returntyp
233                     };
234                     MethodsWithParameter.push(MethodObject);
235                 })
236             });
237     } catch (error) {
238         console.log("Error processing JSON data for class names: " + ↩
                error.message);
239     }
240     console.log(MethodsWithParameter );
241     return MethodsWithParameter;
242 }
243 async function sortByMethodenParameterCount(methods) {
244     methods.sort((a,b) => b.lenghts - a.lenghts);
245     console.log(methods);
246     return methods;
247 }
248 async function sortByConstructorsParameterCount(constructors) {
249     constructors.sort((a,b) => b.lenghts - a.lenghts);
250     console.log(constructors);
251     return constructors;
252 }
```

Listing 6.1: Funtkonen der JSONFileLoader Klasse

6.2.2 ImportInformationFromWebsite

```
1
2  async function GetInformationFromWebsite() {
3      let textareaValue = document.getElementById("input");
4      let userInput = textareaValue.innerText;
5      return SplitInformationFromWebsite(userInput);
6  }
7
8  function SplitInformationFromWebsite(userInput) {
9      const splittedInputArray = userInput.split(/\r?\n/).flatMap(line ↩
          => line || "\n");
10     return SeparateInputIntoLines(splittedInputArray);
```

```
11 }
12
13 function SeparateInputIntoLines(splittedInputArray) {
14     return splittedInputArray.map(str => str.split(' '));
15 }
16
17
18 \subsection{SyntaxHighlighter}
19 \begin{lstlisting}[language=JavaScript, label=lst:SyntaxHighlighter, ↩
       caption={Funktionen  der SyntaxHighlighter Klasse}, linewidth=15cm]
20 function ArrayIntoHtmlFormat(array){
21     let newContent = "";
22     for (let lineIndex = 0; lineIndex < array.length; lineIndex++) {
23         let flag = false;
24         for (let i = 0; i <= array[lineIndex].length-1; i++) {
25             if(i < array[lineIndex].length-1){
26                 newContent = newContent + array[lineIndex][i] + " ";
27             }else{
28                 newContent = newContent + array[lineIndex][i];
29             }
30         }
31     }
32     return newContent;
33 }
34
35 async function TransformInnerHTML(misstakesPostionArray, ↩
       mistakeWordArray) {
36     const userInput = document.getElementById("input");
37     const consoleOutput = document.getElementById("output");
38     const textInput = await GetInformationFromWebsite();
39     const textOutput = ↩
           consoleOutput.innerText.replace(consoleOutput.innerText, "");
40     let newContent = textInput;
41     let newConsoleFail = textOutput;
42     userInput.innerHTML = userInput.innerText;
43     for (let lineIndex = 0; lineIndex < textInput.length; ↩
           lineIndex++) {
44         let flag = false;
45         for (let i = 0; i <= misstakesPostionArray[lineIndex].length ↩
               - 1; i++) {
46             const designatedProblematicWord = ↩
                   mistakeWordArray[lineIndex][i];
47             newContent[lineIndex][misstakesPostionArray[lineIndex][i]] ↩
                   = `<u style="text-decoration-color: red; ↩
                   text-decoration-style: wavy" ↩
                   >${designatedProblematicWord}</u>`;
48             if(designatedProblematicWord == "\n" || ↩
                   designatedProblematicWord == ""){
49                 flag = true;
50                 break;
51             }
52             if (misstakesPostionArray.length != 0) {
53                 newConsoleFail = newConsoleFail.replace("", function ↩
                       () {
54                     return "Diese eingabe ist der Syntax nicht ↩
                           bekannt: " + '<span style="color: #ff0000;  ↩
                           font-weight: bold;">' + ↩
```

```
                            designatedProblematicWord + '</span><br>';
55                      });
56                  }
57              }
58              if(flag == false) {
59                  newContent[lineIndex] = ↩
                        newContent[lineIndex].concat('<br>');
60              }else{
61                  console.log("error empty line error")
62              }
63          }
64          userInput.innerHTML = ArrayIntoHtmlFormat(newContent);
65          consoleOutput.innerHTML = newConsoleFail;
66  }
```

Listing 6.2: Funktionen der ImportInformationFromWebsite Klasse

6.2.3 IDE.Html

```
1
2   <!DOCTYPE html>
3   <html>
4   <head>
5       <script src="SyntaxAnalyzer.js"></script>
6       <script src="ImportInformationFromWebsite.js"></script>
7       <script src="SyntaxHighlighter.js"></script>
8       <script src="SemmantikAnalyzer.js"></script>
9       <script src="JsonFileLoader.js"></script>
10      <style>
11          .container {
12              display: flex;
13              height: 100vh;
14              width: 100vh;
15              justify-content: center;
16              align-items: flex-start;
17              flex-direction: column;
18              background-color: white;
19          }
20
21          .toolbar {
22              display: flex;
23              justify-content: space-between;
24              align-items: center;
25              width: 80%;
26              margin: 0 10%;
27              height: 50px;
28          }
29
30          .textarea-container {
31              display: flex;
32              flex: 1;
33              width: 100%;
34          }
35
36          .linenumbers {
37              display: inline-block;
```

```
38            width: 40px;
39            text-align: right;
40            padding-right: 5px;
41            margin-top: 10px;
42            color: gray;
43            white-space: pre-line;
44            pointer-events: none;
45        }
46
47
48        .textareas {
49            display: flex;
50            flex: 1;
51            width: 100%;
52            max-height: 300px;
53        }
54
55        .textarea {
56            flex: 1;
57            width: 100%;
58            color: gray;
59            padding: 10px;
60            box-sizing: border-box;
61            resize: none;
62            background-color: #f2f2f2;
63            border: none;
64            outline: none;
65            overflow-y: auto;
66        }
67
68        .output {
69            width: 100%;
70            background-color: #d9d9d9;
71            color: black;
72            padding: 10px;
73            overflow-y: auto;
74        }
75
76        button {
77            margin-top: 0px;
78            padding: 10px 20px;
79            background-color: lightgray;
80            border: none;
81            border-radius: 5px;
82            cursor: pointer;
83        }
84
85        .play-button {
86        }
87
88        .play-button:hover {
89            background-color: gray;
90            color: white;
91        }
92
93        .disabled {
94            opacity: 0.6;
```

```
95              cursor: not-allowed;
96          }
97
98
99      </style>
100 </head>
101 <body>
102
103 <div class="container">
104     <div class="toolbar">
105         <button id="compileButton" class="play-button" ↩
                onclick="mainSemmantik()">Compile</button>
106         <button id="duplicate-button" class="play-button" ↩
                onclick="main()">Download</button>
107     </div>
108     <div class="textarea-container">
109         <div class="linenumbers"></div>
110         <div class="textareas">
111             <div class="textarea" contenteditable="true" ↩
                    spellcheck="false" id="input"></div>
112             <div class="textarea output" contenteditable="true" ↩
                    spellcheck="false" id="output" disabled></div>
113         </div>
114     </div>
115 </div>
116 </body>
117 </html>
118 <script>
119     const input = document.getElementById('input');
120     const linenumbersDiv = document.querySelector('.linenumbers');
121     input.addEventListener('input', updateLineNumbers);
122
123     function updateLineNumbers() {
124         const lines = input.innerText.split('\n');
125         const lineNumbers = [];
126         for (let i = 0; i < lines.length; i++) {
127             lineNumbers.push(i + 1);
128         }
129         linenumbersDiv.innerText = lineNumbers.join('\n');
130     }
131 </script>
```

Listing 6.3: Funktionen der IDE Klasse

6.2.4 Syntax-Analyzer

```
1 let similarityIndex = 0.5;//entsprechend anpassen
2 async function mainSyntax() {
3     await CheckValidityOfSyntaxFromJSON();
4 }
5
6 async function CheckValidityOfSyntaxFromJSON() {
7     const userInput = await GetInformationFromWebsite();
8     const errorPostion = [];
9     const errorWordArray = [];
```

```
10      for (let lineIndex = 0; lineIndex < userInput.length; ↩
            lineIndex++) {
11          let error = [];
12          let errorWord =[];
13          for (let numberOfInstance = 0; numberOfInstance < ↩
                userInput[lineIndex].length; numberOfInstance++) {
14              switch (true) {
15                  case ↩
                        userInput[lineIndex][numberOfInstance].includes("-"):
16                      console.log("Es Handelt sich um ein Wire");
17                      break;
18                  case ↩
                        userInput[lineIndex][numberOfInstance].includes("label"):
19                      for(i = numberOfInstance; i ↩
                            <userInput[lineIndex].length; i++){
20                          if(userInput[lineIndex][i].endsWith('"')){
21                              numberOfInstance = i;
22                              break;
23                          }
24                      }
25                      console.log("Es Handelt sich um ein Label");
26                      break;
27
28                  case ↩
                        userInput[lineIndex][numberOfInstance].includes("<") ↩
                        || ↩
                        userInput[lineIndex][numberOfInstance].includes(">") ↩
                        || ↩
                        userInput[lineIndex][numberOfInstance].includes("^") ↩
                        || ↩
                        userInput[lineIndex][numberOfInstance].includes("v"):
29                      console.log("Es ist ein Layout");
30                      break;
31                  case ↩
                        (userInput[lineIndex][numberOfInstance].includes("a") ↩
                        || ↩
                        userInput[lineIndex][numberOfInstance].includes("d") ↩
                        || ↩
                        userInput[lineIndex][numberOfInstance].includes("s") ↩
                        || ↩
                        userInput[lineIndex][numberOfInstance].includes("w")) ↩
                        && userInput[lineIndex][numberOfInstance].length ↩
                        <=1:
32                      console.log("Es ist ein Layout");
33                      break;
34                  case ↩
                        userInput[lineIndex][numberOfInstance].includes("(") ↩
                        || ↩
                        userInput[lineIndex][numberOfInstance].includes(")"):
35                      console.log("Es handelt sich um Klammern")
36                      break;
37                  case ↩
                        userInput[lineIndex][numberOfInstance].includes("kHz") ↩
                        || ↩
                        userInput[lineIndex][numberOfInstance].includes("M") ↩
                        || ↩
                        userInput[lineIndex][numberOfInstance].includes("V"):
```

```
38                          break;
39                      case (typeof userInput[lineIndex][numberOfInstance] ↩
                            === 'string'):
40                          const returnvalue = await ↩
                                CheckSyntaxInWaveForms(userInput, ↩
                                error,lineIndex ,numberOfInstance);
41                          if (returnvalue == true)
42                              break;
43                          const methodenFehler = await ↩
                                CheckSyntaxInMethods(userInput, ↩
                                error,lineIndex ,numberOfInstance);
44                          const classnamesFehler = await ↩
                                CheckSyntaxInClassnames(userInput, ↩
                                error,lineIndex, numberOfInstance);
45                          error = methodenFehler.filter(number => ↩
                                classnamesFehler.includes(number));
46                          if (error.length === methodenFehler.length && ↩
                                error.length  === classnamesFehler.length) {
47                              errorWord = ↩
                                    errorWord.concat(userInput[lineIndex][numberOfInstance]);
48                              break;
49                          }
50                          break;
51                      default:
52                          console.log("Eingabe nicht bekannt" + ↩
                                userInput[lineIndex][numberOfInstance]);
53                          break;
54                  }
55              }
56              errorPostion.push(error);
57              errorWordArray.push(errorWord);
58          }
59          console.log(errorPostion);
60          return [errorPostion,errorWordArray];
61      }
62
63      async function CheckSyntaxInWaveForms(userInput, fehler, lineIndex, ↩
            numberOfInstance){
64          const Waveforms =       ["SINE",
65              "COSINE",
66              "TRIANGLE",
67              "SAWTOOTH",
68              "SQUARE",
69              "DC",
70              "RANDOM_SQUARE",
71              "LAPLACE",
72              "DIRAC_DELTA",
73              "SAMPLES_LOCAL_SELECT",
74              "SAMPLES_SERVER_SELECT",
75              "SELECTED_SAMPLES_FROM_SERVER"];
76          for(i=0; i < Waveforms.length-1;i++){
77          if(Waveforms[i].toLowerCase() == ↩
                userInput[lineIndex][numberOfInstance].toLowerCase())
78              return true;
79          }
80          return false;
81      }
```

```
82
83 async function CheckSyntaxInClassnames(userInput, fehler, lineIndex, ↩
       numberOfInstance) {
84     const classNames = await ImportClassFromJsonFile();
85     let similarity = 0.0;
86     for (let i = 0; i < classNames.length; i++) {
87         similarity = ↩
               CalculateSimilarity(userInput[lineIndex][numberOfInstance], ↩
               classNames[i]);
88         if (similarity >= similarityIndex) {
89             break;
90         }
91     }
92     if (similarity <= similarityIndex) {
93         fehler = fehler.concat(numberOfInstance);
94     }
95     return fehler;
96 }
97
98 async function CheckSyntaxInMethods(userInput, fehler, lineIndex, ↩
       numberOfInstance) {
99     const methods = await ImportMethodsFromJsonFile();
100    let similarity = 0.0;
101    for (let i = 0; i < methods.length; i++) {
102        similarity = ↩
               CalculateSimilarity(userInput[lineIndex][numberOfInstance], ↩
               methods[i]);
103        if (similarity >= similarityIndex) {
104            break;
105        }
106    }
107    if (similarity <= similarityIndex) {
108        fehler = fehler.concat(numberOfInstance);
109    }
110    return fehler;
111 }
112
113 function CalculateSimilarity(word1, word2) {
114     const length = Math.max(word1.length, word2.length);
115     let matches = 0;
116     for (let i = 0; i < length; i++) {
117         if (word1[i] === word2[i]) {
118             matches++;
119         }
120     }
121     return matches / length;
122 }
```

Listing 6.4: Funktionen der SyntaxAnalyzer Klasse

6.2.5 Semantik-Analyzer

```
1 async function mainSemmantik() {
2     await CheckValidityOfSemantic();
3     transformInnerHTMLforSemmantik();
4 }
```

```
5
6  const objSegments= {
7      _currentSystemInit: "",
8      _currentMethode: "",
9      _currentConstructors:"",
10     _currentLabel:"",
11     _Connectionl:"",
12     set currentSystem(system){
13         this._currentSystemInit = system;
14     },
15     get currentSystem(){
16         return this._currentSystemInit;
17     },
18     set currentMethode(methode){
19         this._currentMethode = methode;
20     },
21     get currentMethode(){
22         return this._currentMethode;
23     },
24     set currentConstructors(cunstructors){
25         this._currentConstructors = cunstructors;
26     },
27     get currentConstructors(){
28         return this._currentConstructors;
29     },
30     set currentLabel(label){
31         this._currentLabel = label;
32     },
33     get currentLabel(){
34         return this._currentLabel;
35     },
36     set currentConnection(connection){
37         this._Connection = connection;
38     },
39     get currentConnection(){
40         return this._Connection;
41     }
42
43 }
44
45 async function CheckValidityOfSemantic() {
46     const userInput = await GetInformationFromWebsite();
47     const segmentsInformation = SperateArrayIntoSegments(userInput);
48
49 }
50
51 function CheckValidityOfMethode(){
52     return [true,i];
53 }
54 function CheckValidityOfConstructor(){
55 }
56 function findLastSuperClass(data, searchclassName) {
57    const classData = data.find(item => item.classNames === ↩
         searchclassName);
58     if (classData && classData.superClass) {
59         return findLastSuperClass(data, classData.superClass);
60     }
```

```
61      return searchclassName;
62
63  }
64  async function CheckValidityOfSystemInit(input,numberOfInstance){
65      const classnameWithSuperclass = await ↩
           ImportSuperclassWithClassname();
66      //  berprfen   welche methoden es gibt  f  r  die funktion
67      const lastSuperClass = ↩
           findLastSuperClass(classnameWithSuperclass,input[numberOfInstance].toLowerCase());
68      const currentClassArray = classnameWithSuperclass.find(item => ↩
           item.classNames.toLowerCase() === ↩
           lastSuperClass.toLowerCase());
69      const methode = currentClassArray.method;
70      console.log(methode);
71      //  berprfem   bis wo das array geht
72
73
74
75  }
76
77  function  CheckValidityOfLabel(userInputCurrentLine,i){
78      let valueLabel = [];
79      if(userInputCurrentLine[i+1].toLowerCase().includes('"')){
80          valueLabel.push(userInputCurrentLine[i]);
81          valueLabel.push(userInputCurrentLine[i+1]);
82          for(j =i+2; j < userInputCurrentLine.length ;j++){
83             if(!userInputCurrentLine[j].endsWith('"')){
84                 valueLabel.push(userInputCurrentLine[j]);
85             }else{
86                 valueLabel.push(userInputCurrentLine[j]);
87             break;
88             }
89           }
90      }else{
91          return false;
92      }
93      objSegments._currentLabel = valueLabel;
94      return [true,j-1];
95
96  }
97  async function CheckIfInputIsClass(input){
98      const classNames = await ImportClassFromJsonFile();
99      let similarity = 0.0;
100     for (let i = 0; i < classNames.length; i++) {
101         similarity = CalculateSimilarity(input, classNames[i]);
102         if (similarity >= similarityIndex) {
103              return true;
104         }
105
106 }return false;}
107 function CheckValidityOfConnector(userinput){
108     if(userinput.includes("-") || userinput.includes("w")||
109         userinput.includes("v") || userinput.includes("a")||
110         userinput.includes("^") ||  userinput.includes("s")||
111         userinput.includes(">") ||  userinput.includes("d")||
112         userinput.includes("<") ){
113         console.log("connector")
```

```
114             return true;}
115
116         return false;
117 }
118 function SperateArrayIntoSegments(userInput){
119         const segmentsArray = [];
120         for (let lineIndex = 0; lineIndex < userInput.length; ↩
                lineIndex++) {
121           const lineSegments =  ↩
                  SperateLineArrayIntoSegments(userInput[lineIndex]);
122         }
123
124
125 }
126 async function SperateLineArrayIntoSegments(userInputInCurrentLine){
127         for (let numberOfInstance = 0; numberOfInstance < ↩
                userInputInCurrentLine.length; numberOfInstance++) {
128             switch (true) {
129                 case ↩
                        userInputInCurrentLine[numberOfInstance].toLowerCase().includes("label")
130                     const status = ↩
                            CheckValidityOfLabel(userInputInCurrentLine,numberOfInstance);
131                     if(status[0] == true){
132                         numberOfInstance = status[1];
133                         break;
134                     }
135                     break;
136                 case ↩
                        (CheckValidityOfConnector(userInputInCurrentLine[numberOfInstance])== ↩
                        true)  :
137                     objSegments._Connection = ↩
                            [userInputInCurrentLine[numberOfInstance-1], ↩
                            userInputInCurrentLine[numberOfInstance], ↩
                            userInputInCurrentLine[numberOfInstance+1]];
138                     break;
139                 case (typeof userInputInCurrentLine[numberOfInstance] ↩
                        === 'string'):
140                     if(await ↩
                            CheckIfInputIsClass(userInputInCurrentLine[numberOfInstance])){
141                         objSegments._currentSystemInit = ↩
                                userInputInCurrentLine[numberOfInstance];
142                         CheckValidityOfSystemInit(userInputInCurrentLine,numberOfInstance);
143                 }
144                     break;
145                 default:
146                     console.log("unbekanntes Segment: Name = " + ↩
                            userInputInCurrentLine[numberOfInstance]);
147                     break;
148             }
149         }
150
151 }
152 async function transformInnerHTMLforSemmantik(errors){
153         let syntaxErrors = await CheckValidityOfSyntaxFromJSON();
154         let semmantikErrors = await CheckValidityOfSemantic();
155         let  test = await LoadJsonContent();
156         let syntaxAndSemmantikErrorPostions = syntaxErrors[0];
```

```
157     let syntaxAndSemmantikErrorWords = syntaxErrors[1];
158     TransformInnerHTML(syntaxAndSemmantikErrorPostions,syntaxAndSemmantikErrorWords);
159 }
```

Listing 6.5: Funktionen der SemantikAnalyzer Klasse

Tabellenverzeichnis

3.1 Anwendungsfälle 28

4.1 JSON File 47
4.2 In & Out JSON 47
4.3 Methods & Constructors JSON 48
4.4 In & Out JSON 58

Abbildungsverzeichnis

1 Git: Leo-Labalive-www . . . 3

2.1 IntelliJ IDEA Ultimate Logo . . . 15
2.2 MikeTex Logo . . . 16
2.3 Visual Paradigm Logo . . . 16
2.4 Overleaf Logo . . . 16
2.5 JSON formatter Logo . . . 17
2.6 labAlive Code-Aufbau Systeme . . . 19
2.7 LabAlive Code-Aufbau Klassen . . . 20
2.8 JSON APIs compared to XML APIs [15] . . . 22
2.9 Die Phase eines Interpreters [16, S.6] . . . 23
2.10 Beispiel eines endlichen Automaten [16, S.36] . . . 24
2.11 Parser Organigram [2] . . . 25
2.12 LL(k) und Syntaxbaum eines Top-Down-Parsers [1, S.54] . . . 26

3.1 Anwendungsfall Syntax Analyse . . . 29
3.2 Anwendungsfall Syntax-Highlighting . . . 31
3.3 Anwendungsfall Semantik Analyse . . . 32
3.4 Anwendungsfall Error Message . . . 34
3.5 Dialog Syntax-Highlighting . . . 36

4.1 . . . 37
4.2 Erste Idee der IDE . . . 38
4.3 Finale Form des Prototyps . . . 39
4.4 Grafische Darstellung des Parsing . . . 53
4.5 Grafische Darstellung von layout "1 >v 2" . . . 55
4.6 Grafische Darstellung der Semantikanalyse . . . 59
4.7 Detaillierte Darstellung der Segmente . . . 59

5.1 Ausblick Projektarbeit geplant . . . 64
5.2 Ausblick Projektarbeit tatsächlich . . . 65
5.3 Zeitaufwand während der Abschlussarbeit . . . 65

6.1 Ablaufdiagramme AF Information-Note . . . 67
6.2 Ablaufdiagramme AF Auto-Compliton . . . 68

Abkürzungsverzeichnis

IDE Integrated Development Environment

UML Unified Modeling Language

LaTeX Lamport TeX

JSON JavaScript Object Notation

JDK Java Development Kit

JRE Very High Speed Integrated Circuit Hardware Description Language

JS Java Script

HTML Hypertext Markup Language

XML Extensible Markup Languag

API Application Programming Interface

CSS Cascading Style Sheets

Lexer lexikalischer Scanner

AST Abstrakten Syntaxbaum

LL Links-Nach-Rechts, Linksableitung

AF Anwendungsfall

II Initiation Interval

LUT Look-Up Table

IDFT Inverse Diskrete Fourier-Transformation

IFFT Inverse Fast Fourier Transform

DOM DOMDocument Object Model

id IDIdentifikationsnummer

div DIVDivision

api APIApplication Programming Interface

Literaturverzeichnis

[1] Prof. Dr.Ing. Dieter R. Pawelczak. *Secure Software Engineering*. [Online; accessed 15. Jun. 2023]. München: Instituts für System Engineering. URL: https://ilias.unibw.de/ilias.php?ref_id=495591&cmd=view&cmdClass=ilrepositorygui&cmdNode=wp&baseClass=ilrepositorygui (siehe Seiten 7, 23, 26).

[2] Rafael Zink. *Parser organigram*. [Online; accessed 16. Jun. 2023]. Dez. 2003. URL: https://de.wikipedia.org/w/index.php?title=Parser&oldid=225991691 (siehe Seiten 7, 25).

[3] *IntelliJ IDEA*. [Online; accessed 21. Jun. 2023]. Juni 2023. URL: https://www.jetbrains.com/de-de/idea (siehe Seite 15).

[4] *MikeTexHome*. [Online; accessed 21. Jun. 2023]. Juni 2023. URL: https://miktex.org (siehe Seite 15).

[5] *Ideal Modeling & Diagramming Tool for Agile Team Collaboration*. [Online; accessed 21. Jun. 2023]. Juni 2023. URL: https://www.visual-paradigm.com (siehe Seite 16).

[6] *Wikipedia Java Techonolgie*. 14.06.2023. URL: https://de.wikipedia.org/wiki/Cascading_Style_Sheets?Language=English&CategoryNo=218&No=1021&PartNo=2#contents (siehe Seite 17).

[7] Autoren der Wikimedia-Projekte. *JavaScript Wikipedia*. [Online; accessed 14. Jun. 2023]. Apr. 2002. URL: https://de.wikipedia.org/w/index.php?title=JavaScript&oldid=233952785 (siehe Seite 17).

[8] Autoren der Wikimedia-Projekte. *Hypertext Markup Language Wikipedia*. [Online; accessed 14. Jun. 2023]. Juni 2002. URL: https://de.wikipedia.org/w/index.php?title=Hypertext_Markup_Language&oldid=234244753 (siehe Seite 18).

[9] . *Cascading Style Sheets Wikipedia*. [Online; accessed 14. Jun. 2023]. Mai 2002. URL: https://de.wikipedia.org/w/index.php?title=Cascading_Style_Sheets&oldid=229201184 (siehe Seite 18).

[10] *Virtual Communications Lab - labAlive*. [Online; accessed 20. Jun. 2023]. Juni 2023. URL: https://www.etti.unibw.de/labalive (siehe Seite 18).

[11] Autoren der Wikimedia-Projekte. *JavaScript Object Notation Wikipedia*. [Online; accessed 14. Jun. 2023]. Juni 2005. URL: https://de.wikipedia.org/w/index.php?title=JavaScript_Object_Notation&oldid=234371070 (siehe Seite 20).

[12] Autoren der Wikimedia-Projekte. *Extensible Markup Language Wikipedia*. [Online; accessed 14. Jun. 2023]. Juni 2002. URL: https://de.wikipedia.org/w/index.php?title=Extensible_Markup_Language&oldid=230377215 (siehe Seite 20).

[13] *Lesson: Annotations (The Java Tutorials Learning the Java Language)*. [Online; accessed 20. Jun. 2023]. Juni 2023. URL: https://docs.oracle.com/javase/tutorial/java/annotations/index.html (siehe Seite 20).

[14] *Using Java Reflection*. [Online; accessed 20. Jun. 2023]. Juni 2023. URL: https://www.oracle.com/technical-resources/articles/java/javareflection.html (siehe Seite 20).

[15] Torrey Betts. *Mobile Performance Testing - JSON vs XML Infragistics Blog*. [Online; accessed 15. Jun. 2023]. Juni 2023. URL: https://www.infragistics.com/community/blogs/b/torrey-betts/posts/mobile-performance-testing-json-vs-xml (siehe Seite 22).

[16] Prof. Dr.Ing. Dieter R. Pawelczak. "Programmerzeugungssysteme". In: (), Seite 6. URL: https://ilias.unibw.de/ilias.php?ref_id=495591&cmd=view&cmdClass=ilrepositorygui&cmdNode=wp&baseClass=ilrepositorygui (siehe Seiten 22–24).

[17] Hagen Langer Dr.Sven Neumann. *Eine Einführung in die maschinelle Analyse natürlicher Sprache*. Uni Trier, 1994. URL: https://www.uni-trier.de/fileadmin/fb2/LDV/Naumann/BUCH1-1.pdf (siehe Seiten 23, 24).

[18] Prof. Dr. Andrea Baumann. *Programmerzeugungssysteme*. [Online; accessed 15. Jun. 2023]. München: Instituts für System Engineering. URL: https://ilias.unibw.de/ilias.php?ref_id=430563&cmd=view&cmdClass=ilrepositorygui&cmdNode=wp&baseClass=ilRepositoryGUI (siehe Seite 27).

[19] *AEK-CH2-SC2.1-ARDUINO-IDE.png*. [Online; accessed 23. Jun. 2023]. Juni 2023. URL: https://docs.arduino.cc/static/4106ba9a36bb5b73bc95520a96f785ea/29114/AEK-CH2-SC2.1-ARDUINO-IDE.png (siehe Seite 37).

[20] *IntelliJ IDEA*. [Online; accessed 23. Jun. 2023]. Juni 2023. URL: https://www.jetbrains.com/de-de/idea (siehe Seite 37).

[21] Daniel Hirlimann. "Entwicklung einer Code-Validierung für labAlive myApps aus Sichtweise der Cybersecurity". Juni 2023 (siehe Seite 46).

Index

A

Ablaufdiagramme 67
abstrake Automaten 23
AF-Syntax Analyse 29
Anhang 67
Annotation 20
Anwendungsfälle 28
Ausblick 65
automat 23

B

BottonUpParser 26

C

CSS 18

D

div 18
DOM 18

E

Einleitung 13
endlicher Automat 23
Ergebnisse 63
Error Message 34

F

Fazit 63

G

Grafik Parsing 52, 59
Grafik Semantikanalyse1 59

H

Herausforderungen 64
Html 17

I

Implementierung 37
ImportClassFromJsonFile 49
ImportConstructorsFromJsonFile 51
ImportInAndOutPutFromJsonFile 49
ImportInformationFromWebsite 42
ImportMethodsFromJson 50
ImportMethodsWithParameter 50
ImportSuperclassWithClassname 52
intellij-IDEA . . . . . 15 f
Internetseiten 16
Interpreter 22

J

Java 17
Javascript 17
JSON 20
JSON Formatter 17
Json Loader 48
JSON Values 46
JSONFileLoader 46

L

LabAlive 7
labalive 18
labAlive Code 19
labAlive Logik 19
Layout Beispiel 55
Lexikalische Analyse 23
LoadJsonContent 48

M

Methoden 15
Miktex 15

O

Overleaf 16

P

Parser 24
Parsing 22, 24
Programmiersprachen und Dateiformate . . . 17
Prototyp 38

Q

QuellCode 68

R

Reflection 20

S

Segmente 58
Semantik Analyse 32
Semantik Funktionen 60
SemantikAnalyser 58
Semantische Analyse 24
sortByMethodenParameterCount 52
Syntaktische Analyse 24
Syntax Highlighting 31
SyntaxAnalyser 52
SyntaxHighlighter 44

T

TopDownParser . . . . . 25 f

W

WebIDE 7, 37

X

XML 20

Z

Zeitaufwand 64