Entwicklung einer Code-Validierung für labAlive myApps aus Sichtweise der Cybersecurity

Bachelorarbeit

im Rahmen des Studienganges

Technische Informatik und Kommunikationstechnik

Daniel Hirlimann

Betreuer:

Prof. Dr.-Ing. Erwin Riederer

Tag der Abgabe: 26.6.2023

Universität der Bundeswehr München
Fakultät für Elektrotechnik und Technische Informatik
Institut für Funkkomunikation

Neubiberg, Juni 2023

Abbildung 1: labAlive

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 26.6.2023

Daniel Hirlimann

Abstract

Das Thema dieser Arbeit ist das Erstellen und Implementieren einer Backend - Grundlage für die Code-Validierung in LabAlive. Zu Beginn der Arbeit wird die Aufgabenstellung mit dem derzeitigen Sachstand erklärt. Im darauffolgenden Teil werden die Grundlagen vorgestellt. Darin wird sowohl auf die ausgewählte Software, als auch auf die Funktionalität von Labalive eingegangen. Ein Blick auf das Projekt aus der Sichtweise der Cybersicherheit findet bereits hier statt. Im Hauptteil der Arbeit erfolgt eine Beschreibung der Ausführung und Implementierung. Hierbei wird zuerst auf die Theorie der Umsetzung eingegangen und dann jede Klasse mit Funktionsweise und Aufbau dargestellt. Zuletzt werden die Ergebnisse und Probleme präsentiert, sowie ein Ausblick auf mögliche zukünftige Projekte gegeben.

Inhaltsverzeichnis

1 Einleitung

labAlive ist eine Simulationsumgebung für kommunikationstechnische Schaltungen. In den verschiedenen Modulen wird, von den Kernanforderungen (starten einer Simulationsumgebung weiterhin App genannt), bis hin zur Bereitstellung eines Servers alles eingebettet. Auf der Webseite kann sich der User einen Account erstellen und seine Apps Verwalten. Grundlage für die Apps ist ein geschriebener Code, welcher durch das Auflösen in die Java Klassen des labAlive Projekts als ausführbare .jnlp Datei heruntergeladen werden kann. labAlive ist weltweit zugänglich und aufgrund seiner Sicherheit und Effizienz gut für die Lehre geeignet. Besonders an dieser Bachelorthesis ist, das Arbeiten in einem großen Projekt, während weitere wissenschaftliche Arbeiten in diesem Projekt zeitgleich entstehen. Deswegen sind regelmäßige Treffen, sowie eine ausgezeichnete Teaminterne Kommunikation Grundsteine für das erreichen der Ziele dieser Bachelorthesis.
Das Frontend der Clientseite wird parallel zu dieser Thesis in einer anderen Bachelorarbeit von Herrn Lt. Laurence Simon erläutert[17].

1.1 Aufgabenstellung und Sachstand

Die Syntax und Semantik in labAlive ist eine komplett eigene, wobei auf Effizienz und Sicherheit geachtet wurde. Für diese gibt es dementsprechend noch keinen eigenen Validierungs- und Ergänzungsservice. Für die User Nutzung auf der Webseite, sowie das Erstellen des .jnlp ist es natürlich von Vorteil, wenn eine Art Compiler vor dem Erstellen bzw. Abspeichern der Apps die Syntax und Semantik überprüft. Bei der Erstellung der Apps wird einfach über “Try and Error” getestet, ob der Code eine lauffähige App erstellt. Hier eröffnet sich ein weiterer Verwendungszweck um das Erstellen der Apps noch effizienter zu gestalten.

1.2 Motivation

In meinem ersten Modul, in welchem ich mit labAlive in Berührung gekommen bin, war die Aufgabe, eine Datenbank inkl. Struktur zu erstellen und dies dann an die Webseite anzubinden. Hieraus entstand die Möglichkeit, dass der User sich anmeldet und seine Apps speichern kann. Darauf folgte das zweite Modul in welchem die Zielsetzung vom Server weg ging, hin zu den Apps, also simulierten kommunikationstechnischen Schaltungen. Es wurden einerseits reine App - Funktionalitäten wie “Copy&Paste” umgesetzt, andererseits auch Funktionalitäten in denen die App mit dem Server kommuniziert um die erstellte Schaltung abzuspeichern. Im dritten Modul mit labAlive blieb der Fokus weiterhin auf der App, deswegen konnten hier viele neue Funktionalitäten umgesetzt werden. Der Schwerpunkt lag aber diesmal mehr auf der Zusammenarbeit der Systeme (z.B. Oszilloskop Signale) in dieser Schaltung. Basierend auf meinen Erfahrungen und Kenntnissen im labAlive Projekt, sowie der zuvor durchgeführten Projektarbeit “Anforderungen und Architektur für die Validierung und Fehlerbehandlung von labAlive Text2App Code” innerhalb des Instituts für Funkkommunikation habe ich mich für eine Bachelorarbeit im labAlive Projekt entschieden. Nachdem der fehlende Validierungs- und Ergänzungsservice eine

deutliche Verbesserung der “Usability” mit sich bringt, wird im Rahmen dieser Arbeit das Backend für eine Clientseitige Code-Validierung und Ergänzung umgesetzt.

2 Theorie und Grundlagen

2.1 Benutzte Software

2.1.1 Git

Abbildung 2.1: Git Logo

Git ist eine open-source Lösung zur Versionskontrolle [21]. Aufgrund der Größe und Komplexität des gesamten labAlive Projekts sind studentische Arbeiten in diesem Umfang nur noch mit einer Versionskontrolle wie Git möglich.
Version: 2.37.1

2.1.2 IntelliJ

Abbildung 2.2: IntelliJ Logo

IntelliJ ist eine von jetBrains entwickelte IDE für Java [22]. Als zusätzliches Plugin wurde “Smart Tomcat” Version 4.3.8 genutzt. Dies ist nicht zwingend erforderlich erleichtert aber das Erstellen des Tomcat Apache Servers.
Die Einbindung des labAlive Projekts fand nach Anleitung von mir statt.
Version: 2022.2.4 Community Edition
Runtime version: 17.0.5+7-b469.71 aarch64

2.1.3 Json Editor

Abbildung 2.3: Json formatter Logo

JSON-Editor ist ein kostenloses online Tool, welches JSON-Daten formatiert, validiert und dem Nutzer beim Abspeichern oder Teilen dieser hilft. [1]

2.1.4 MacTeX

Abbildung 2.4: MacTeX Logo

MacTeX ist ein Bündel an Software, welches genutzt wird um ein LateX Dokument zu erstellen [26]. Hauptsächlich wird das Programm “TeXShop”, zum bearbeiten der .tex Dateien, “makeindex” für das Erstellen des Index und “biber” für das Hinzufügen eines Literaturverzeichnis aus dem Bundle genutzt.
Version: 5.12

2.1.5 OpenJDK

Abbildung 2.5: OpenJDK Logo

OpenJDK ist eine open-source Implementierung der Java Platform, Standard Edition und ausgewählten Projekten [27]. Aus Sichtweise der Cybersicherheit ist eine open-source Lösung v.a. eine grundlegende mit mehreren millionen Nutzern zu bevorzugen.
Version: 17.0.2
Runtime Environment (build 17.0.2)

2.1.6 PyCharm

Abbildung 2.6: PyCharm Logo

PyCharm ist eine von jetBrains entwickelte IDE für Python [28]. Es wurden keine zusätzlichen Plugins genutzt.
Version: 2022.3 Professional Edition
Runtime version: 17.0.5+1-b653.14 aarch64

2.1.7 VisualParadigm

Abbildung 2.7: VisualParadigm Logo

Visual Paradigm ist eine Software, welche mehrere Projektmanagement Tools beinhaltet [2]. Hier wurden hauptsächlich Tools zur Erstellung von UML Diagrammen genutzt.
Version: 17.0 (Community)

2.2 Java

Abbildung 2.8: Java Logo

Java ist eine weit verbreitete objektorientierte Programmiersprache [23]. Als “Language Level” wurde LTS 17 in IntelliJ eingestellt. Zusätzlich wurde noch der Apache Tomcat Server Version 8.5.81 genutzt [13].

2.2.1 Annotation

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target({ElementType.TYPE})
3 public @interface Annotation {
4     String in();
5     Boolean bool();
6     int count();
7 }
```

Listing 2.1: Annotation Beispielcode

Mit Annotations werden z.B. Methoden annotiert [19]. Das einfachste Beispiel ist hierbei “Override”. Im Code erstellt man, wie in Listing 2.1 dargestellt, für eine eigene Annotation ein Interface. In dieses kann man auch Methoden einfügen. Die Funktionen füllen immer eine gleichnamige Variable, aus der später diese Daten auch geladen werden können. Durch das Einstellen von Zielen (@Target) kann die eigene Annotation angepasst werden, so dass diese z.B. nur auf Methoden gesetzt werden kann. Mit @Retention stellt man ein, zu welchem Zeitpunkt die Annotation geladen werden soll. So ist es möglich, diese nur als eine Art Markierung zu nutzten. In diesem Projekt benötigen wir die Annotation zur Laufzeit, da während der Ausführung darauf zugegriffen wird. Hierfür wird die Retention auf Runtime gesetzt.

2.2.2 Reflection

Bei Java Reflection handelt es sich um eine, ab Java Version 8 stabil eingebaute Funktionalität, um zur Laufzeit auf Klassen des Packages bzw. angegebenen Pfad zugreifen zu können, ohne diese als eigene Objekte erstellen zu müssen [30]. Ein Vorteil liegt hier klar auf der Hand, es wird zur Laufzeit nicht so viel Speicherplatz benötigt. Bei insgesamt (stand heute) um die 1700 Klassen wäre der Speicherbedarf höher, auch wenn immer nur wenige Objekte gleichzeitig existieren. Zudem ist es aus Sicht der Cybersicherheit erfreulich, wenn man auf alles in einer Klasse zugreifen kann, ohne dass diese selber ausgeführt werden muss. So würde es für diese Funktionalität kein großes Problem darstellen, wenn Schadcode in eine Klasse eingeschleust werden würde. In diesem Projekt wird Java-Reflection allerdings hauptsächlich nicht wegen dieser Punkten genutzt. Bevor wir zum Hauptargument für Reflection kommen, gehen wir auf die negativen Punkte ein. Gerade der erste Punkt ist für viele Anwendungen bereits der Grund warum Reflection nicht angewendet wird. Eine Optimierung wie sie der Compiler normalerweise durchführt ist mit Java Reflection nicht mehr möglich, weil Reflection zur Laufzeit weiteren Code nachladen kann. Dies führt in großen Programmen zu einer Leistungsreduktion. Beim zweiten negativen Punkt geht es um die Cybersicherheit. Hier wird das Prinzip “Information Hiding” ausgehebelt. So kann auf jede Variable oder Methode der Klasse Zugegriffen werden, auch wenn diese “private” ist.

Nach Abwiegen der positiven und negativen Argumente ist die Wahl hauptsächlich aufgrund der Funktionalität auf Java Reflection gefallen. Mit Java Reflections ist es nämlich ohne Probleme möglich die Methodennamen, deren Attribute und den Rückgabetype als String abzuspeichern. Die negativen Punkte werden hier abgeschwächt, indem die Funktionalität vom Server abgekoppelt wird. D.h. der Code wird vom Administrator händisch ausgeführt und liegt nicht im Source Code der beim Starten des Servers geladen wird. Die Ergebnisse werden über eine JSON Datei zum User gegeben, deswegen gibt es keine funktionalen Verbindungen vom Server zu diesem Code mit Java Reflections. Nachdem das Projekt zudem im Gesamten auf dem Server liegt und nur die Personen darauf zugreifen, die sowieso in dem Projekt programmieren, ist das Problem mit dem fehlenden “Information Hiding” und damit den uneingeschränkten Zugang zu allen Methoden und Attributen auch nicht schlimm. Zudem werden die Methoden, welche dem User über die Datei übergeben werden, in der Klasse gekennzeichnet. Somit bekommt der User nicht kompletten Zugang zu allen möglichen Methoden, sondern nur bestimmten, welche davor ausgewählt wurden.

2.3 Datenformate

2.3.1 JavaScript Object Notation (JSON)

Bei einer JSON Datei handelt es sich um eine Datei die als schlankes Datenaustauschformat genutzt wird [24]. Vorteil der JSON ist, dass Menschen diese Struktur lesen können und Computer diese einfach Parsen und Generieren können. Obwohl der Name JavaScript enthält, ist dieses Dateiformat komplett Programmiersprachen unabhängig. Bei den meisten Programmiersprachen ist die Implementierung in Packages/Modulen (so auch bei Java) vorhanden, wohingegen JavaScript nativ mit JSON arbeiten kann. Aufgrund der Projekt größe und Komplexität wird das JSON Package jedoch nicht eingebunden, sondern die JSON als String erstellt. Ein JSON bildet ähnlich wie Dictonaries (Python) oder HashMaps (Java) Key/Value Paare. Anhand folgender Grafik erkennt man die Struktur.

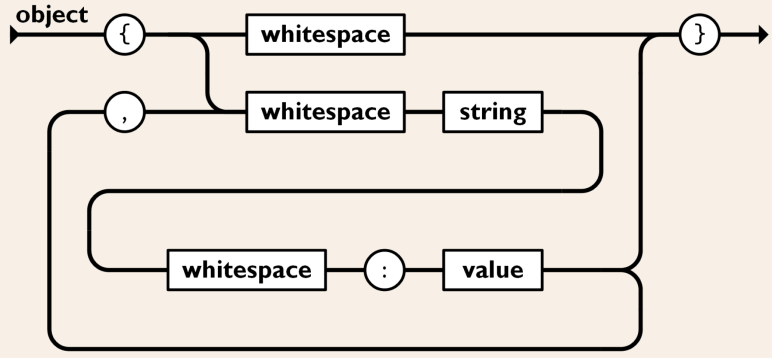

Abbildung 2.9: JSON Object Darstellung.

Anfang und Ende eines Objects (JSON) werden mit “{” bzw. “}” markiert. Dazwischen befindet sich eine ungeordnete Menge von Key/Value Paare welche jeweils mit einem “ , ” voneinander getrennt sind (Abbildung 2.9). Ein Key ist immer eine Zeichenkette (string), wohingegen ein Value eines der folgenden Datenformate haben muss: string, number, boolean, null, object (Abbildung 2.10) Es ist auch möglich als Value ein Object zu definieren, welches natürlich wieder aus Key/Value Paaren besteht. Zusätzlich ist es möglich über eine geordnete Liste von Werten (Array) wie in Abbildung 2.11 mehrere Values einem Key zuzuweisen.
Die Wahl ist hier auf ein JSON gefallen, weil dieses im späteren Verlauf einfach der Webseite übergeben werden kann. Dies ist von Vorteil, weil die Oberfläche mit der Validierungsfunktionalität in JavaScript programmiert wird.

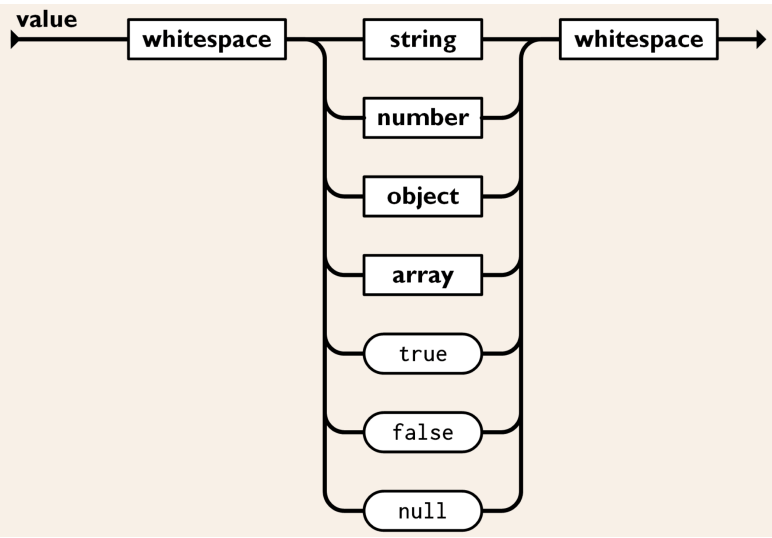

Abbildung 2.10: JSON Value-Typen

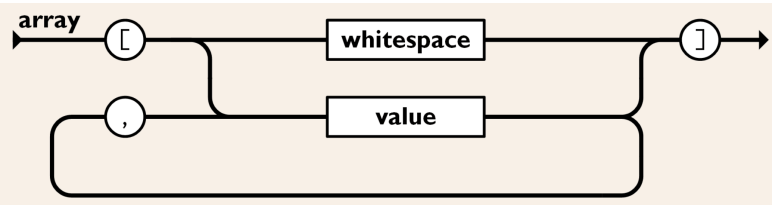

Abbildung 2.11: JSON Array Darstellung.

2.3.2 Extensible Markup Language (XML)

Bei XML handelt es sich um eine weit verbreitete Auszeichnungssprache zur Darstellung und Strukturierung von Daten. Hierbei handelt es sich um eine einheitliche Organisation und Formatierung von Daten in einer hierarchischen Struktur, welche die Daten plattformunabhängig und in maschinenlesbarere Weise darstellt. [3]

2.3.3 Vergleich

Im Folgenden werden die beiden Datenformate JSON und XML, im Bezug auf dieses Projekt, verglichen. Für die Umsetzung ist einerseits die Lesbarkeit, andererseits die Möglichkeit, die Datei einfach, also ohne vorgeschriebene Funktionen, zu bearbeiten und zu ergänzen, wichtig. XML ist komplexer und basiert auf umfangreichen Tags, Verschachtelungen und eigenen Funktionen. JSON hingegen bietet durch eine klare und kompakte Syntax, die leicht verständlich ist, eine lesbare Struktur. Darüber hinaus ermöglicht JSON eine direkte Integration mit JavaScript. Nachdem JavaScript für die Umsetzung im Frontend genutzt wird, ist JSON das Format, welches im Weiteren genutzt wird. Dadurch ist das Einbinden der Daten einfacher und effizienter.

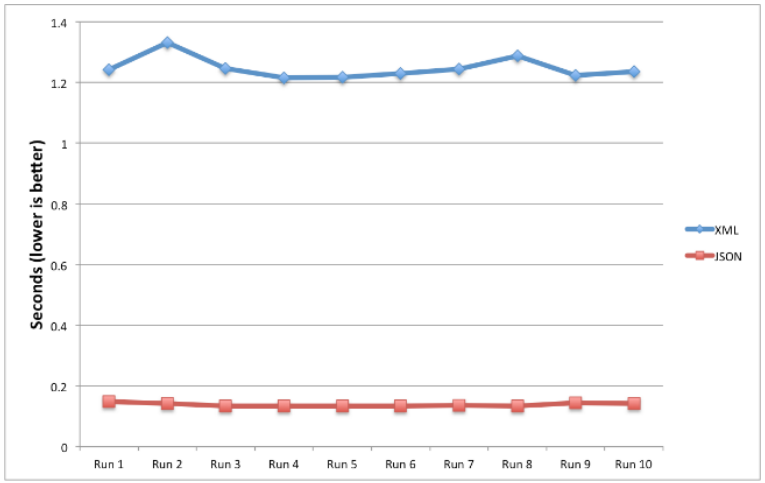

Abbildung 2.12: JSON APIs compared to XML APIs [8]

Wie in Abbildung 2.12 aufgezeigt wird, ist die Datenübertragung mit JSON effizienter und kompakter. Daraus resultiert eine besseren Datenübertragungszeit und ein geringerer Speicherbedarf als bei XML. Zusätzlich ist JSON in modernen Webtechnologien und APIs nicht mehr weg zu denken. Für Webanwendungen ist JSON das bevorzugte Format für den Datenaustausch zwischen Client und Server [3, 8, 24]. Aus Sichtweise der Cybersicherheit ist JSON ebenfalls zu bevorzugen, hierbei sind die gleichen Punkte wie oben zu nennen. Aufgrund all dieser Punkte wird in diesem Projekt auch JSON genutzt.

2.4 Python

Abbildung 2.13: Python Logo

Nachdem hier verschiedene JSON Layouts erstellt wurden, sollte getestet werden, welches im späteren Verlauf am schnellsten ist. Dieser Test wurde in Python programmiert, weil Python [29] nicht für seine schnelle Ausführungszeit bekannt ist und mit relativ wenig “einfachem” Code die gewünschte Funktionalität schnell einsetzbar ist. Weil Python auf Einfachheit und Übersichtlichkeit ausgelegt wurde, unterscheidet sich die Syntax stark von anderen Programmiersprachen. Anstelle von Klammern und Semikolons wird mit Einrückungen des Codes gearbeitet. Bei grundsätzlichen Funktionalitäten wird gerne auch auf ein Wort gesetzt wie z.b. “x in list” (Foreach) oder “a and b” (&&). Python ist nicht typsicher, die angelegten Variablen sind Objekte, welche mit Strings, Boolean, int, usw. gefüllt werden. Aufgrund der Einfachheit eignet sich Python bei dem Testen der JSON perfekt.

Version: 3.9.13

2.5 Portierung auf MacOS

Mit der neuen hauseigenen “apple silicon” Chip-Generation sind MacBooks in punkto Leistung und Effizienz klar an den gängigen anderen Laptops vorbeigezogen. Durch die sehr lange Akkulaufzeit kombiniert mit einem gestochen scharfen Retina Display und dem auf Unix aufbauenden Betriebsystem MacOS sind die MacBooks für dieses Studium sehr gut geeignet [4, 5]. Deswegen werden sie auch von der Fakultät an Studierende verliehen. “ARM” - Chips wurden bis dahin hauptsächlich in Smartphones verbaut, haben aber inzwischen auch in einem der schnellsten Supercomputer (zur Zeit Platz 2) der Welt Einzug gefunden [18, 20].

Leider ergibt sich aus dem neuen Design auch eine weitere Herausforderung. Das Chipdesign wurde von “x86” auf “ARM” umgestellt. Die hauptsächliche Unterscheidung ist hier, dass die Assembler Befehle bei “ARM” immer gleich lang sind. Bei “x86” sind die Befehle unterschiedlich lang. Deswegen sind die beiden Designs nicht ohne Übersetzung kompatibel miteinander. Bei der Software ziehen viele Hersteller mit und optimieren ihre Programme für “ARM” Chips. Bei den verbliebenen Programmen schafft MacOS mit dem integrierten Übersetzungstool “Rosetta2” Abhilfe. Leider funktioniert dies bei ganzen Betriebssystemen, also Virtuellen Maschinen, nicht wirklich gut. Die virtuelle Maschine, auf welcher Windows 10 läuft und der gesamte labAlive Sourcecode zum entwickeln eingebunden ist, funktioniert deswegen auf dem neuen MacBook nicht mehr. Also musste, bevor diese Arbeit richtig beginnen konnte, das ganze Projekt auf MacOS lauffähig gemacht werden. Diese Dokumentation würde diese Arbeit etwas sprengen, deswegen verweise ich hier auf die von mir erstellte Dokumentation [13].

2.6 labAlive

Im labAlive [25] Projekt werden Simulationsumgebungen für kommunikationstechnische Schaltungen erstellt. Im folgenden wird genauer auf die Logik bzw. Funktionalität eingegangen. Dies Unterteilt sich in die reine Logik und die programmatische Umsetzung der Logik in Java.

2.6.1 Systeme - Logik

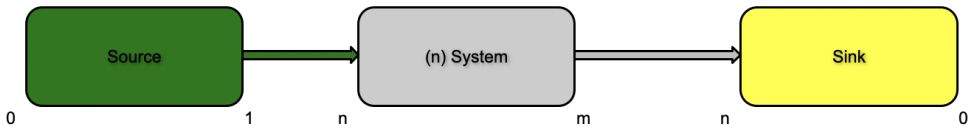

Abbildung 2.14: labAlive Code-Aufbau Systeme

Die Umsetzung besteht aus Systemen, welche mit Wirings (dargestellt durch "-") verbunden werden können. Anhand von Abbildung 2.14 wird gezeigt wie eine Schaltung aufgebaut ist. Am Anfang muss ein System eingesetzt werden, welches keine Eingänge besitzt (z.b. Source) aber ein oder mehrere Ausgänge. In der Mitte müssen Systeme sein mit Ein- und Ausgängen. Am Ende ist entweder ein System ohne Ausgang, oder es wird automatisch eine "Sink" angefügt. Im ersten Schritt muss syntaktisch überprüft werden, ob die angegebenen Systeme überhaupt vorhanden sind. Im zweiten Schritt wird die Semantik überprüft. Ein System besteht aus einer Klasse und dessen Konstruktoren bzw. Methoden, mehr dazu im nächsten Punkt.

2.6.2 LabAlive - Code

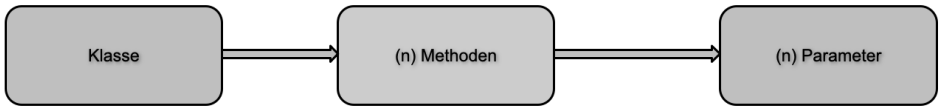

Abbildung 2.15: labAlive Code-Aufbau Klassen

Wie in Abbildung 2.15 zu sehen ist, besteht ein System aus einer Klasse und dessen Konstruktoren bzw. Methoden aufrufen, welche gefolgt sind von den jeweiligen Parametern. In der Umsetzung sieht dies dann so aus:

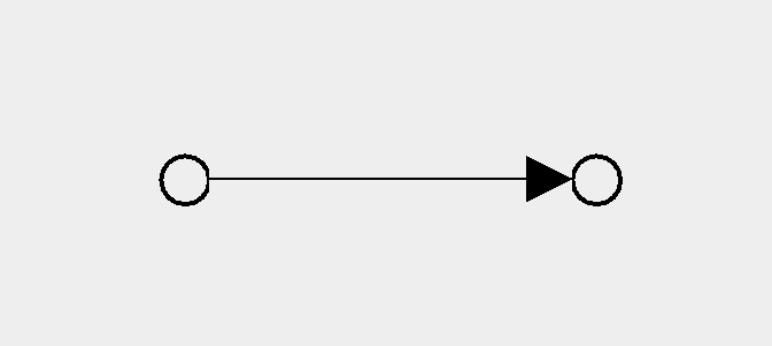

Abbildung 2.16: Schaltung mit einem SineGenerator

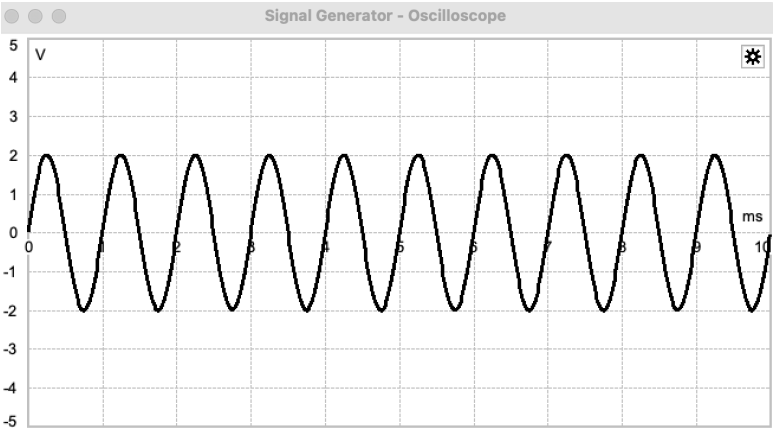

Abbildung 2.17: Oszilloskop am SineGenerator Ausgang

```
1        SineGenerator amplitude(2.0) frequenz(1k)
```

Diese Zeile beschreibt einen Signal Generator, welcher, wie in Abbildung 2.16 und Abbildung 2.17 gezeigt, ein Sinus Signal mit der Amplitude von 2 V und einer Frequenz von 1kHz erstellt. Am Anfang steht die Klasse, gefolgt von Methoden, welchen Werte übergeben werden. Zum Verbinden von zwei Systemen wird ein “Wiring” genutzt. Als einfaches Beispiel dient folgender Code:

```
1        SineGenerator amplitude(2.0) frequenz(1k) - sink
```

Der Code macht nichts anderes wie der vorherige (Abbildung 2.16 und Abbildung 2.17), da bei diesem ein System ohne Ausgang am Ende fehlt wird automatisch eine Sink hinzugefügt. Auch komplexere Schaltungen können ohne Probleme dargestellt werden. Im nächsten Beispiel sieht man anhand des “Adders”, dass Systeme eindeutig identifiziert sind.

```
1        SineGenerator - adder - adder2 - sink
2        CosineGenerator - adder
3        SineGenerator - adder2
```

In der ersten Zeile wird ein “Sinus Generator” mit dem “Adder” verbunden, welcher dann auf eine “Sink” läuft. In der zweiten Zeile läuft auf den selben “Adder” noch ein “Lowpass”. In der dritten Zeile wird zu dem zweiten “Adder” (adder2) nochmal der erste “Sinus Generator” zugeschaltet. Dies wird in folgender Abbildung 2.18 dargestellt.

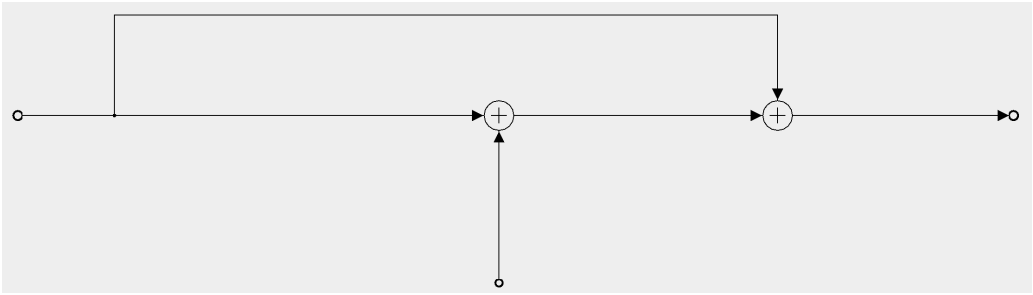

Abbildung 2.18: Schaltungsbeispiel mit zwei Adder

2.6.3 Apps

Das Erstellen einer App ist derzeit etwas kompliziert gestaltet und wird hier anhand eines Beispiels dargestellt.

```
1 SineGenerator amplitude(2.0) frequenz(1k)
```

Angenommen der “SineGenerator” hat einen Konstruktor dem ein Argument übergeben wird und einen dem kein Argument übergeben wird. Dann wird zu erst getestet, ob der Konstruktor mit einem Argument funktioniert, hier wird als Argument das nächste Wort übergeben. Ist dem nicht so, dann wird der nächste Konstruktor mit weniger Argumenten aufgerufen, dies passiert solange bis es funktioniert, oder keine Argumente übergeben werden. Als nächstes wird das darauffolgenden Wort getestet. Hierbei wird in der Klasse überprüft ob es eine Methode mit diesem Namen gibt. Nachdem es meistens mehrere überladene Methoden gibt, wird hier auch wieder die mit den meisten Argumenten als erstes getestet und “abgearbeitet”, wie bei dem Konstruktor. Nach diesem Muster wird die gesamte App erstellt, dies führt in einigen Fällen auch dazu, dass sich eine App beim Starten sechs mal öffnet und wieder schließt, bis diese beim 7. mal endgültig geöffnet bleibt.

2.7 Compiler

2.7.1 Syntax und Semantik

Die Syntax macht keine Aussagen bzgl. der Semantik einer Schnittstelle/Klasse/Methode. Die Kenntnis einer Schnittstelle z.B. reicht zwar aus um syntaktisch korrekt auf die Methoden einer Klasse zugreifen zu können, legt aber die Wirkung der Methoden und damit die Semantik der Klasse nicht fest. Also wird mit der Syntax definiert wie auf eine Methode zugegriffen wird (Methodenname, Parametertypen, ...) und mit der Semantik definiert was die Methode macht. Beispiel Ersetzungskompatibel:
Eine Klasse A heißt zu einer Klasse B funktional ersetzbar (ersetzungskompatibel), wenn A die syntaktische Schnittstelle von B umfasst und in beliebigen Programmen die Klasse B durch die Klasse A ersetzt werden kann, ohne dass sich die Ergebnisse der Programme ändern (d.h. die Methoden der Schnittstellen müssen auch semantisch gleich sein).

2.7.2 Interpreter

Ein Compiler erzeugt, einen Zieltext, dies macht der Interpreter nicht. Bei beiden wird der Quelltext aufbereitet. Der Interpreter setzt Anweisungen aus dem Quelltext direkt um und stellt auch in der Regel eine Laufzeitumgebung zur Verfügung. In Abbildung 2.19 werden die verschiedenen Phasen der Zielcode-Erzeugung in der Interpretation dargestellt.

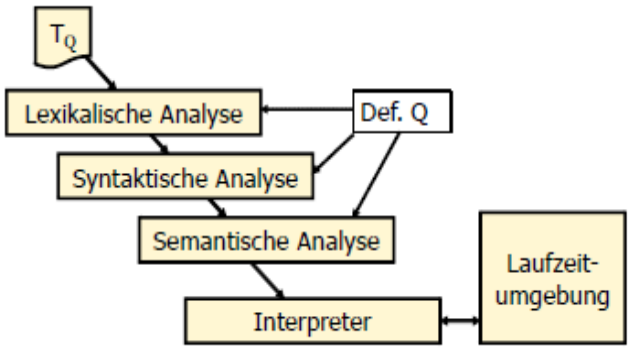

Abbildung 2.19: Die Phasen eines Interpreters [15, S.6]

"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."[15, S. 6].
Der Interpreter kann den Code schrittweise ausführen, dafür nutzt er den abstrakten Syntaxbaum, zudem führt der Interpreter den Code Zeile für Zeile in Echtzeit aus. Interpretierte Anweisungen werden in Maschinenbefehle umgewandelt und dann vom Computer ausgeführt [9, 16].

2.7.3 Lexikalische Analyse

Das Zerlegen des Quellcodes in einzelne Token oder lexikalische Einheiten während des Compiler- und Interpreterprozesses wird lexikalische Analyse genannt. Dieser Schritt ist der Grundstein für die weitere Analyse und Verarbeitung des Quellcodes. In der Regel umfasst der lexikalische Analyseprozess das Lesen und Eingruppieren von Zeichen in Token gemäß festgelegter Regeln. Die Regeln werden als reguläre Ausdrücke oder endgültige Automaten dargestellt. Die Analyse wird in einem lexikalischen Scanner (“lexer”), der für die lexikalische Analyse verantwortlich ist, durchgeführt. Der Scanner sortiert die Token im Quelltext nach ihrer Art und überspringt unnötige Leerzeichen, Kommentare usw. Die Token können verschieden wichtige Codebausteine (Schlüsselwörter, Identifikationen, Operatoren, Zahlen, usw.) darstellen. Das Alles ermöglicht eine frühzeitige Fehler- und Problemerkennung, sowie eine effektive Verarbeitung großer Codebasen [9, 16, 17].

2.7.4 Automat

Automaten werden verwendet, um die Funktionalität bzw. das Verhalten eines Systems zu beschreiben. Automaten sind ein abstraktes Modell, mit welchen Zustände und Übergänge dargestellt werden.
Endliche Automaten sind Systeme, welche die Zustände durchlaufen und auf Eingaben reagieren. Sie bestehen aus einem Startzustand, einer Menge von Endzuständen, einer Menge von Zuständen und einer Übergangsfunktion, die den Übergang zwischen Zuständen auf der Grundlage der Eingabezeichen steuert. Dabei können diese in zwei Kategorien unterteilt werden, deterministisch und nicht deterministisch. Bei Ersterem handelt es sich um einen Automaten, der einen genauen Übergang für jeden Zustand und jedes Eingabezeichen besitzt. Beim Zweiten gibt es mehrere mögliche Übergänge, was zu einer gewissen Unsicherheit führt.
Innerhalb der Endautomaten befindet sich der abstrakte Automat, welcher die Modellierung und Analyse komplexer Systeme und Prozesse ermöglicht. Abstrakte Automaten können zusätzliche Funktionen und Eigenschaften, wie Speicher oder erweiterte Ausdrucksfähigkeit, die über die Merkmale eines

endlichen Automaten hinausgehen, enthalten. Diese werden in vielen Bereichen der Informatik und Mathematik, darunter auch Sprachverarbeitung und Compilerbau genutzt [9, 16, 17].

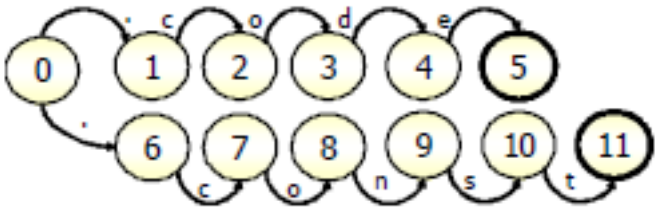

Abbildung 2.20: Beispiel eines endlichen Automaten [15, S.36]

2.7.5 Syntaktische Analyse

Unter syntaktischer Analyse, auch Parsing genannt, versteht man den Prozess der Analyse einer Zeichenkette oder eines Textes gemäß bestimmter Grammatik oder Syntaxregeln. Die Eingabe, welche eine Abfolge von Zeichen oder Tokens ist, muss in kleinere Einheiten aufgeteilt werden, um die Struktur und Bedeutung dieser zu verstehen. Dabei wird diese in eine Datenstruktur, die als “Parsebaum” oder “Syntaxbaum” bekannt ist, umgewandelt. Beim Finden von Syntaxfehler im Quellcode wird eine Fehlermeldung, mit den Informationen und der Position des Fehlers, erstellt. Normalerweise ist ein Interpreter mit einem Parser verbunden. Dabei untersucht der Parser den Quellcode und erzeugt eine interne Darstellung, welche vom Interpreter genutzt wird um Anweisungen auszuführen. Beim Parsing unterscheidet man verschiedene Methoden, wie z.B. “Top-Down” und “Bottom-Up” [9, 16, 17].

2.7.6 Semantische Analyse

Aufgabe der semantischen Analyse ist die Untersuchung der Bedeutung und logischen Konsistenz des Codes. Dabei wird sichergestellt, dass der Code die semantischen Regeln und Bedingungen der Programmiersprache erfüllt. Dabei wird überprüft, ob die Behandlung von Deklarationen, Ausdrücke und Anweisungen, sowie Typen korrekt sind. Während der semantischen Analyse wird der abstrakte Syntaxbaum, welcher in der vorherigen syntaktischen Analyse erstellt wird, verwendet. Im Ast des Syntaxbaums sind die Struktur und der Kontext des Quellendes enthalten. Der semantische Analyseprozess verwendet diese Informationen um sicherzustellen, dass Variablen, Funktionen und Elemente des Codes korrekt verwendet werden. Dabei ist ein wesentlicher Bestandteil die Typeanalyse. Mit der semantischen Analyse wird garantiert, dass Ausdrücke und Operationen kompatiblen Typen zugeordnet werden und dass keine inkonsistenten oder undefinierten Verwendungen von Variablen oder Funktionen vorliegen. Es können auch Fehler, wie unbelegte Variablen, nicht erreichbarer Code oder unzulässige Typumwandlungen, erkannt werden. Um sicherzustellen, dass der Code logischen Regeln und Bedeutungen folgt, ist die semantische Analyse ein wichtiger Schritt. Dabei hilft diese Fehler und Unstimmigkeiten zu erkennen und eine Grundlage für die effiziente Ausführung des Codes zu schaffen [9, 16, 17].

2.7.7 Parser

Ein wesentlicher Bestandteil des Interpreters ist ein Parser. Dieser untersucht den Quellcode und erzeugt eine interne Repräsentation, welche vom Interpreter verstanden werden kann. Dies geschieht, indem die Syntax des Codes untersucht wird und daraus eine Struktur (abstrakter Syntaxbaum) erschaffen wird.

Mithilfe dieses Syntaxbaums kann der Interpreter den Quellcode in einzelne Bestandteile aufschlüsseln sowie die entsprechenden Aktionen darin schrittweise ausführen. In Echtzeit werden die Anweisungen interpretiert und in Maschinenbefehle, welche vom Computer ausgeführt werden können, umgewandelt.

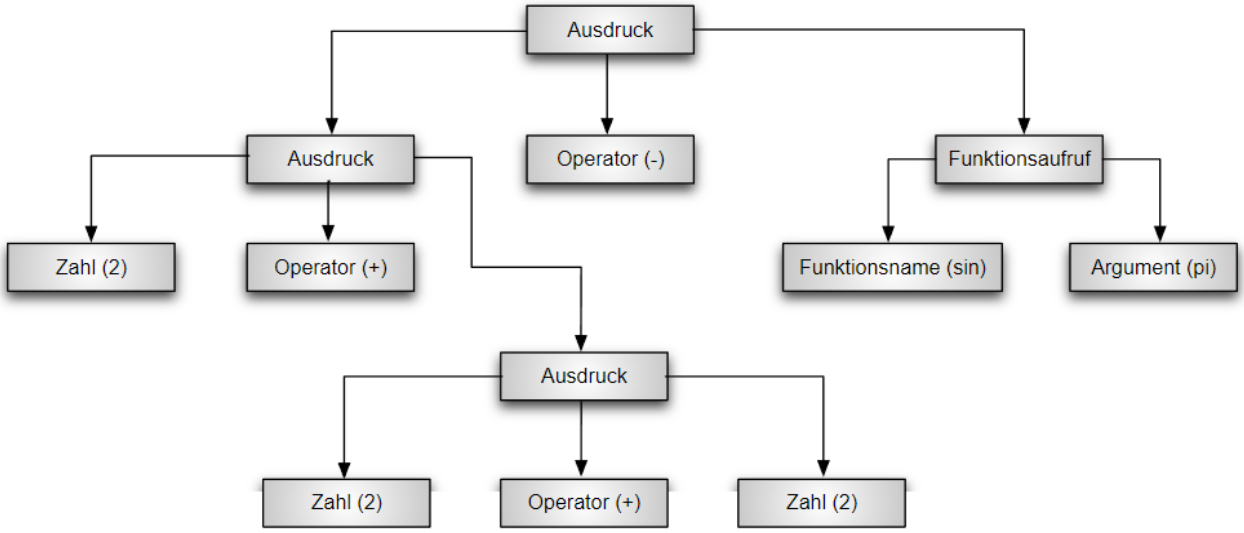

Abbildung 2.21: Parser Organigram [31]

In Abbildung 2.21 wird Schritt für Schritt gezeigt, wie ein Parser aussehen kann. Die Eingabedaten, welche als einfache Zeichenkette erscheinen, werden mit dem “Lexer” in der Mitte des ersten Schritts in Token zerlegt. Hierbei wird in der Regel ein endlicher Automat verwendet. Als nächstes findet die Syntaxanalyse statt, welche durch einen abstrakten Automaten durchgeführt wird. Dabei wird ein Ableitungsbaum (Parserbaum) erstellt. In der vorletzten Ebene wird die semantische Analyse durch den eben erstellten Baum unterstützt. Dieser bestimmt die Bedeutung der Eingabe. Als letzter Schritt folgt die Ausführung, welche entweder zur Erstellung von Code im Compiler oder zur Ausführung durch einen Interpreter führt [9, 16, 17].

2.7.8 Top Down Parser

Es gibt verschieden Arten von Parsern. Der Top-Down-Parser wird verwendet um eine Eingabesequenz von Tokens oder Symbolen gemäß einer bestimmten Grammatik zu analysieren. Es wird mit dem Startsymbol der Grammatik begonnen. Danach werden rekursive Grammatikregeln, um die Eingabe in der vorgegebenen Reihenfolge zu untersuchen, verwendet. Dabei nutzt der Top-Down-Parser die Vorwärtsanalyse, um die Eingabe durch Ableitungen, welche von Startsymbolen (Oben) nach Unten (Eingabe) verlaufen, zu erzeugen. Auf Grundlage des aktuellen Eingabesymbols wird eine geeignete Regel ausgewählt. Es wird versucht die Regelproduktion zu verwenden. Entdeckte Nichtterminalsymbole werden solange vom Parser erweitert, bis dieser ausschließlich Terminalsymbole, welche der Eingabe entsprechen, entdeckt.

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

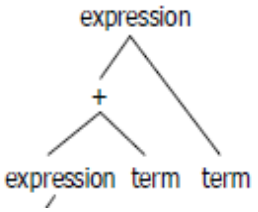

Abbildung 2.22: LL(k) und Syntaxbaum eines Top-Down-Parsers [16, S.54]

Der Top-Down-Parser funktioniert, wie in Abbildung 2.22 dargestellt, auch mit LL(k)-Grammatik (Links-Nach-Rechts / Linksableitung mit k-Symbolen). Als Hauptvorteil ist hier zu nennen, dass dieser der Struktur der Grammatik natürlich folgt und somit einfacher zu lesen ist als andere Parser. Der Top-Down-Parser kann Schwierigkeiten mit mehrdeutiger oder linksrekursiver Grammatik haben [9, 16, 17].

2.7.9 Unterschied zwischen TopDownParser und BottomUpParser

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 [17].

2.8 Cybersecurity

Generell unterscheidet man zwischen “Safety” und “Security”. Wobei “Safety” sinnvoll mit Betriebssicherheit übersetzt wird. Hierbei wird die Ausfallsicherheit im Bezug auf z.B. Hardwaredefekten beachtet. Nachdem der labAlive - Server auf einer militärischen Liegenschaft befindet und dort einerseits die Regeln strenger sind und andererseits auch z.b. eine Werksfeuerwehr zur Verfügung steht, wird diese Gefahr in dieser Arbeit eher vernachlässigt.
Wenn man über “Security” spricht ist also die Cybersicherheit gemeint, in der es in den nächsten Punkten hauptsächlich geht. Zusätzlich werden in jedem Punkt auch mögliche Gegenmaßnahmen beschrieben.

2.8.1 Hacking

Im Allgemeinen wird unter Hacking das Aufspüren und Ausnutzen von Schwachstellen zur Erlangung eines unbefugten Zugangs und erweiterten Rechten auf den angegriffenen Systemen gemeint. Dies hängt oft vom Können der Hacker ab. Ein klarer Vorteil ist hier die Liegenschaft, nachdem labAlive nicht zu einem privaten Unternehmen gehört und somit kein “eigenes” Geld hat, fallen die finanziell motivierten Hacker als Bedrohung weg. Andererseits ist der gerade benannte Vorteil auf einer Liegenschaft der Bundeswehr zu liegen leider auch ein Nachteil. Es gibt “ideologische” Hacker, welche nicht einverstanden mit der Bundesregierung bzw. der Bundeswehr sind, zusätzlich gibt es professionelle Gruppierungen anderer Staaten, welche oft viel Zeit haben ihre Fähigkeiten zu verbessern, professionell

strukturiert und finanziert sind und vor Entdeckungen nicht abgeschreckt werden [10]. Grundsätzlich sollte man die verwendeten Systeme “up to date”, also immer auf dem aktuellen Softwarestand halten. Sobald gefährliche/bedrohliche Schwachstellen bekannt sind, sollte das System überprüft werden. Als Beispiel wird an dieser Stelle “Log4J” genannt. Nach bekanntwerden der Schwachstelle, sollte die betroffene Funktionalität bis zu einem Sicherheitsupdate, welches die Schwachstelle schließt, abgeschaltet werden und bleiben.
Es gibt gewisse Mechanismen und Methodiken, welche in der Entwicklungsphase angewendet werden können um die Fehlerwahrscheinlichkeit zu reduzieren [14]. Genauso sollte auch ein “cleaner code Style” eingeführt werden. Um es den Angreifern schwerer zu machen, sollte man auf zertifizierte oder offene Software zurückgreifen. Der Vorteil ist, dass diese Software meistens auf Schwachstellen überprüft wurde. Als letzten Punkt, könnte das labAlive Projekt geöffnet, als “open source”, werden. Dadurch werden sich einige Nutzer den Code anschauen und ggf. übersehene Schwachstellen erkenne und melden.

2.8.2 Information Hiding

“Information Hiding” in der objektorientierung Programmierung bedeutet, dass es eine klare Trennung von Nutzersicht und Implementierungssicht gibt. Konkret heißt das, dass versucht wird so wenig Informationen wie möglich über die interne Implementierung eines Programms/Funktion/Klasse etc. nach außen zu geben. Die Sichtbarkeit von Attributen einer Klasse wird auf private gesetzt, der Nutzer bekommt nur die Schnittstellen, die er benötigt [7].

2.8.3 Malware

Malware setzt sich aus “Malicious Software” zusammen und bezeichnet Software, welche unerwünschte und schädliche Funktionen auf dem angegriffenen System ausführt. Diese Schadsoftware ist speziell so entwickelt, dass sie grundsätzlich auf allen Betriebsystemen umgesetzt werden kann. Somit macht Malware auch nicht vor Servern oder mobilen Geräten halt. Verteilt wird Malware oft über Mail-Anhänge, manipulierte Websites (Downloadlinks) oder Datenträger. Ransomware ist eine spezielle Art von Malware und wurde entwickelt, um ausgewählte Daten auf dem angegriffenen System zu verschlüsseln. Zum Entschlüsseln wird meistens eine finanzielle Leistung über Kryptowährungen gefordert. Spionagesoftware wurde im Gegensatz zur Ransomware dafür ausgelegt, Informationen aus einem System unerkannt auszuleiten [6].

2.8.4 Phishing

Unter Phishing (zusammengesetzt aus Passwort und Fishing) versteht man, dass Angreifer über z.B. Links in Mails versucht Passwörter zu fischen und im besten Fall wird über den Link Malware o.ä. Runtergeladen und der Angreifer hat kompletten Zugriff auf das angegriffene System. Das Rechenzentrum der Universität der Bundeswehr hat hier mit der Markierung “[Ext]” vor dem Betreff einer Mail die von einer externen Mailadresse kommt schon einen sehr guten Schritt gemacht. Leider werden auch Mails so markiert, welche von Bundeswehr Mailadressen verschickt werden. Zusätzlich scannt das Rechenzentrum gewisse Massenmails und kann so bekannte Links die zu Schadsoftware führen herausfiltern. Dies ist ein sehr guter Schutzmechanismus, außer wenn das Rechenzentrum diese Mails dann trotzdem an alle weiterschickt. Das ist in letzter Zeit leider schon vorgekommen. Als einfache Gegenmaßnahme ist das Sensibilisieren bzw. Schulen aller betroffenen Personen zu nennen. Oft ist die Schwachstelle der Mensch, durch Aufklärung kann dieser aber besser erkennen, welche Gefahr sich hinter so einem oft unscheinbaren Link befinden kann.

2.8.5 Social Engineering

Unter Social Engineering versteht man, das Ausnutzen des Menschen um Zugang zu dem System zu bekommen. Hierin liegt die größte Gefahr, denn der Mensch ist meistens das schwächste Glied in der Kette [11]. Streng genommen ist Phishing auch ein Teil des Social Engineering, deswegen ist die effektivste Gegenmaßnahme hier auch wieder das Schulen und Sensibilisieren der Personen. Selbstverständlich könnte man auch die Zugangsberechtigungen von allen Personen entziehen, aber solange ChatGPT noch nicht eigenständig in der Lage ist, komplexe Systeme zu erstellen und selbstständig ohne menschliche Hilfe zu verwalten, ist es nicht möglich bzw. sinnlos allen Personen den Zugang zu entziehen.

3 Implementierung

3.1 Logik

3.1.1 Datenmodell

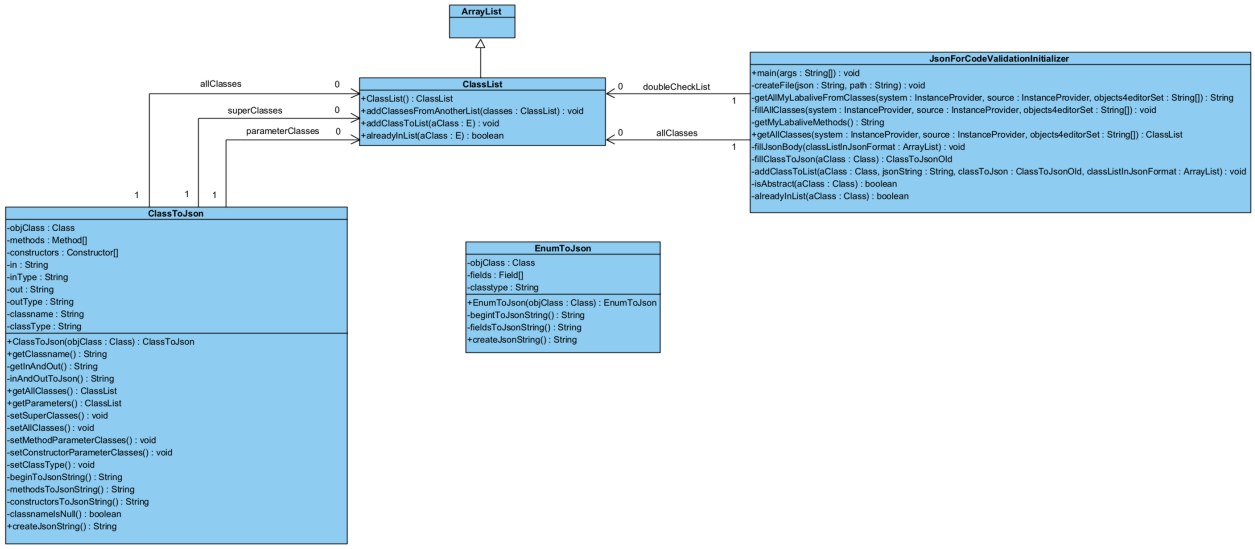

Abbildung 3.1: Fachliches Datenmodell

Im fachlichen Datenmodell (Abbildung 3.1) wird der Zusammenhang zwischen den erstellten Klassen dargestellt. Hier werden nur die Verbindungen aufgezeigt welche als Attribute vorhanden sind. Ein Objekt der Klasse “EnumToJson” wird z.B. in einer Methode der Klasse “ClassToJson” instanziiert. Danach wird aber nur ein String von dem instanziierten Objekt zurückgegeben und das Objekt nicht in einem Attribut gespeichert. Deswegen ist keine Verbindung zwischen den Klassen “EnumToJson” und “ClassToJson”.

3.1.2 Anwendungsfälle

Das Aufzählen aller Anwendungsfälle würde den Rahmen dieser Arbeit sprengen. Außerdem sind diese bereits in meiner Projektarbeit dargestellt. Deswegen wird hier auf meine Projektarbeit verwiesen [12].

3.2 Annotations

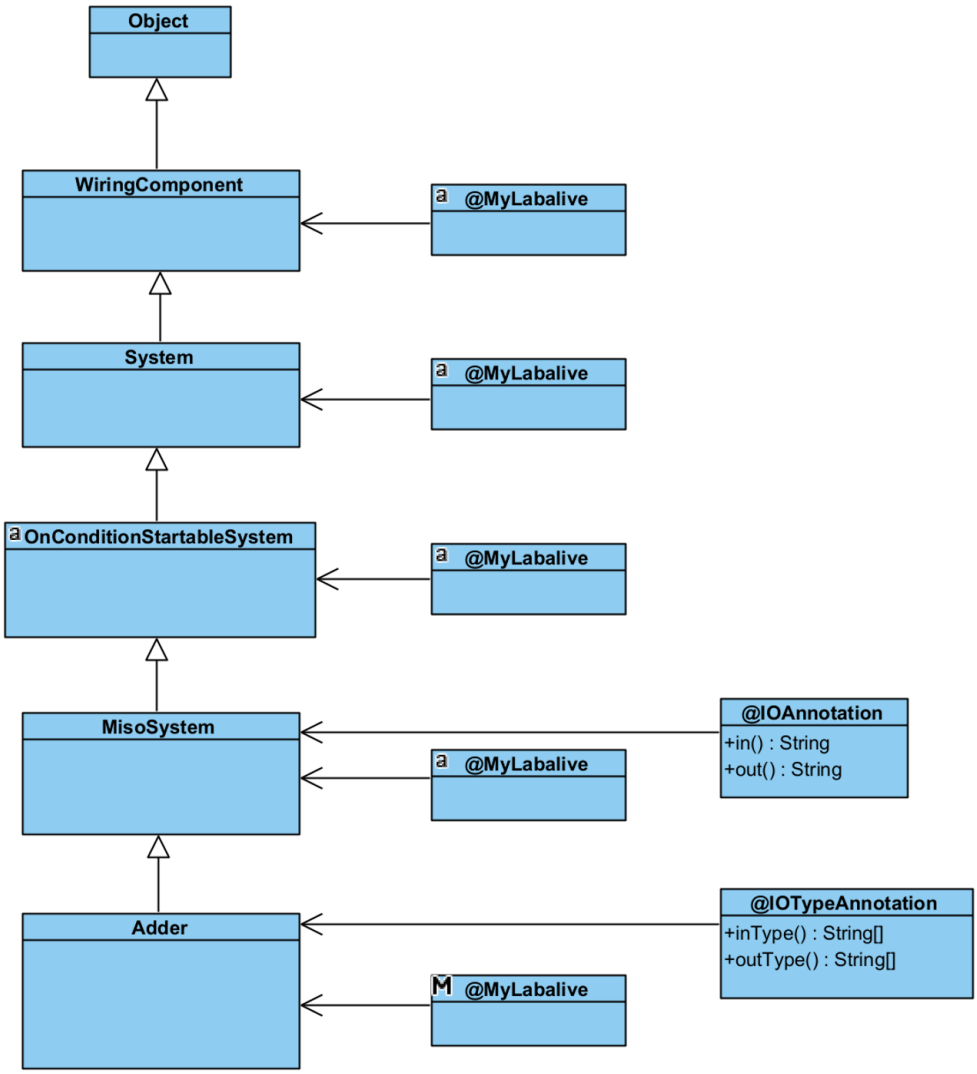

Abbildung 3.2: Adder Annotation Beispiel

Am Beispiel der Adder Klasse (Abbildung 3.2) werden die Annotations dargestellt. Die "@MyLabalive" Annotation kann sowohl im Adder als auch in dessen Oberklassen auf die Methoden der jeweiligen Klasse gesetzt werden. Die "@IOAnnotation" Annotation kann nur auf Klassen gesetzt werden. Sollte diese in einer Oberklasse und in der eigentlichen Klasse, hier Adder, vorhanden sein, dann wird die vom Adder bevorzugt. Die letzte Annotation ist die "@IOTypeAnnotation" Annotation. Diese kann genauso, wie die "@IOAnnotation" auf Klassen gesetzt werden. Es wird auch wieder der Annotation in der eigentlichen Klasse, gegenüber denen der Elternklasse, Vorzug gewährt.

3.2.1 @MyLabalive

```
1 @Retention(RetentionPolicy.RUNTIME)
2 // Target: TYPE = Class/Enum, METHOD = Method, CONSTRUCTOR = ↩
    Constructor, FIELD = Enum-field
```

```
3 @Target({ElementType.TYPE, ElementType.METHOD, ↩
      ElementType.CONSTRUCTOR, ElementType.FIELD})
4 public @interface MyLabalive {
5     // no values in this Annotation
6 }
```

Listing 3.1: Code der MyLabalive Annotation

Zu beginn einer Annotation muss die “Retention” gesetzt werden. Nachdem die Annotations zur Laufzeit überprüft werden, wird hier “RetentionPolicy.RUNTIME” ausgewählt. Dies kann man in Listing 3.1 erkennen.
Mit der “MyLabalive” Annotation werden vorrangig Methoden und Konstruktoren in einer Klasse annotiert. Dies wird unter “Target” wie im Kommentar (Listing 3.1) beschrieben gesetzt. Um in Zukunft noch einzelne Klasse gesondert annotieren zu können, wurde hier das Target “TYPE” noch mit angegeben. Genauso ist es mit dem Eintrag “Field” möglich einzelne Felder in einem Enum zu annotieren.
In dieser Annotation sind keine Attribute gespeichert. “MyLabalive” wird in diesem Projekt, zum jetzigen Zeitpunkt, nur zum annotieren von Methoden genutzt, welche später an die Code-Validierung weitergegeben werden. Diese können vom User gesehen und im LabAlive-Code eingesetzt werden. Im Listing 3.2 ist eine beispielhafte Annotierung der Methode “label” aus der Klasse “System” dargestellt.

```
1     @MyLabalive
2     public System label(String name) {
3         getOutWire().name(name);
4         return this;
5     }
```

Listing 3.2: Anwendungsbeispiel MyLabalive Annotation in der Klasse System

3.2.2 @IOAnnotation

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target({ElementType.TYPE})
3 public @interface IOAnnotation {
4     String in(); // count of inputs
5     String out(); // count of outputs
6 }
```

Listing 3.3: Code der IOAnnotation

Die “IOAnnotation”, dessen Code in Listing 3.3 gezeigt wird, soll nur auf Klassen angewendet werden können, deswegen wird als “Target” nur “TYPE” angegeben. “In” und “out” beschreiben hierbei die Anzahl, der möglichen Ein- und Ausgänge. Nachdem es keine, einen oder viele Ein- und Ausgänge geben kann, werden die Attribute hier als String gesetzt. Im nächsten Listing 3.4 wird gezeigt, wie die Annotation auf die Klasse “MisoSystem” (MultipleInputSingleOutput) gesetzt wird. Das “n” steht hierbei für beliebig viele, also mehr als einen möglichen Eingang.

```
1 @IOAnnotation(in = "n", out = "1")
2 public abstract class MisoSystem extends ↩
      OnConditionStartableSystem {
```

Listing 3.4: Anwendungsbeispiel IOAnnotation

3.2.3 @IOTypeAnnotation

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface IOTypeAnnotation {
    String[] inType();
    String[] outType();
}
```

Listing 3.5: Code der IOTypeAnnotation

Die “IOTypeAnnotation” bestimmt welchen Typ der Eingang und Ausgang hat. Der Code ist in Listing 3.5 zu erknennen. Bei einem “Adder” kann sowohl ein digitales als auch ein analoges Signal ein- und ausgegeben werden. Hier steht in den Attributen dann jeweils ein “*” (Listing 3.6). Die beiden “IO-Annotations” mussten getrennt werden, da es auch Systeme gibt, die von “MisoSystem” erben aber z.B. nur digitale Signale an Ein- und Ausgang nutzen können.

```
@IOTypeAnnotation(inType = "*", outType = "*")
public class Adder extends MisoSystem {
```

Listing 3.6: Anwendungsbeispiel IOTypeAnnotation

3.3 ClassList

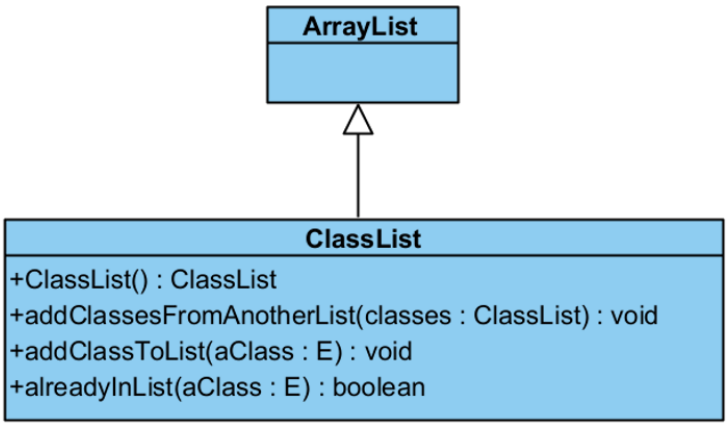

Abbildung 3.3: ClassList

3.3.1 Aufgabe

Wird als Transportobjekt für alle Klassen genutzt.

3.3.2 Funktionsweise und Aufbau

In dieser Klasse werden einige Funktionalitäten umgesetzt, welche in einer “ArrayList” standardmäßig nicht hinterlegt sind. Hierbei handelt es sich v.a. darum, beim Hinzufügen einer Klasse, in die Liste, zu überprüfen, ob diese bereits in der Liste vorhanden ist. Ist dies der Fall, dann wird die Klasse nicht

hinzugefügt. Es wird aber kein Fehler oder ähnliches zurückgegeben. Mit der Methode “addClassesFromAnotherList()”, wie in Abbildung 3.3 dargestellt, wurde die Funktionalität umgesetzt, dass aus einer “ClassList” alle Klassen in die aufrufende “ClassList” übergeben werden. In Zukunft kann diese Klasse auch dazu dienen, eine Liste der möglichen Klassen weiterzugeben, um z.b. vor dem erstellen der Apps dessen Code zu validieren.

3.4 ClassToJson

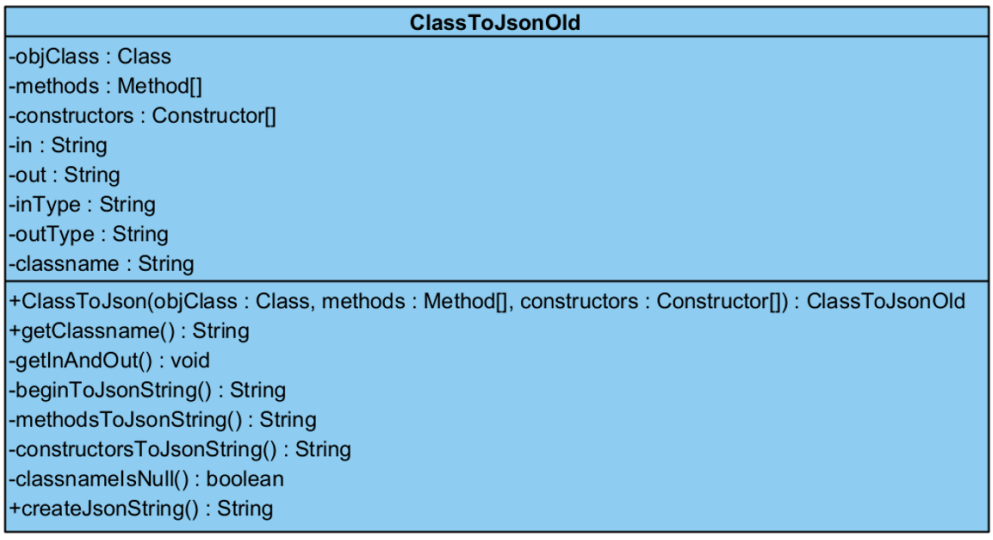

Abbildung 3.4: ClassToJson alter Aufbau

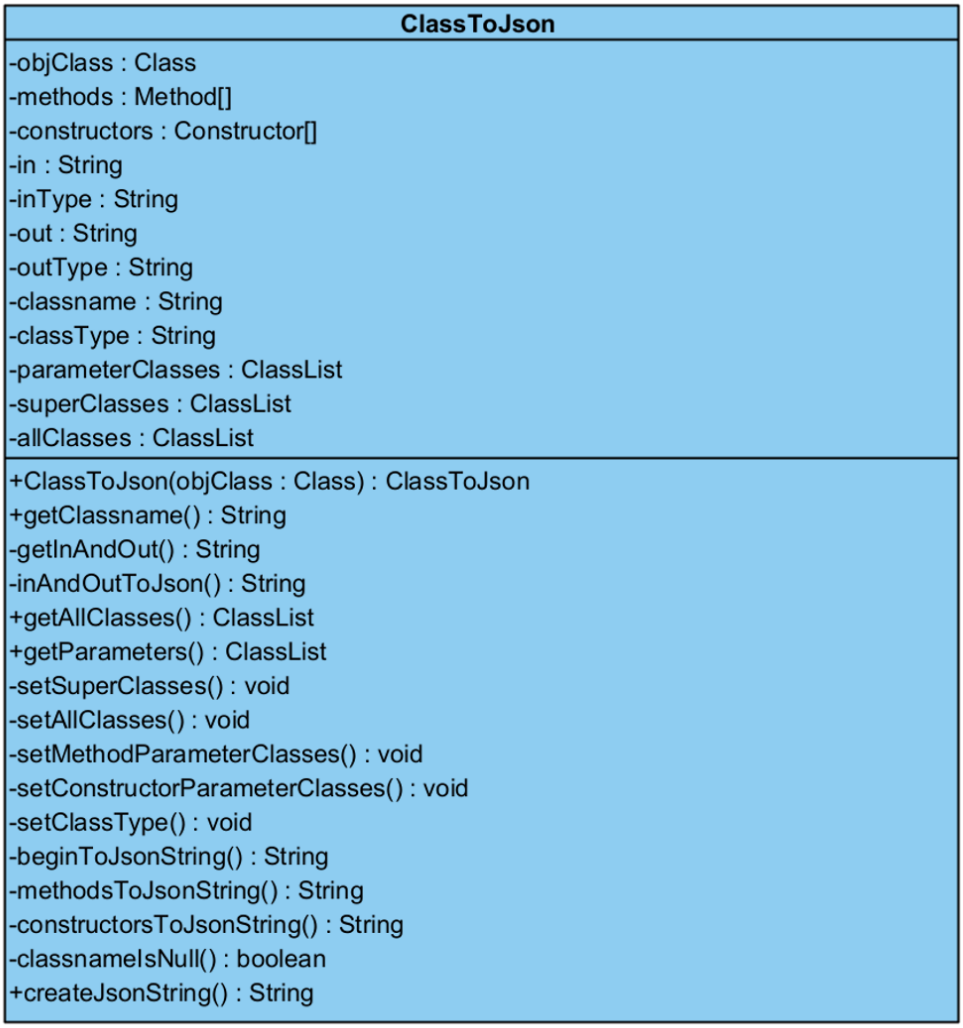

Abbildung 3.5: ClassToJson neuer Aufbau

Die Abbildung 3.4 zeigt wie die Klasse vor der Anpassung ausgesehen hat. Diese war auch noch auf das alte Datenformat ausgelegt, deswegen sind in der neuen Umsetzung (Abbildung 3.5) von “ClassToJson” mehr Methoden. Mit den neuen Methoden sind die Funktionalitäten noch erweitert wurden und die Grundlagen für zukünftige Projekte gelegt.

3.4.1 Aufgabe

Erstellen eines JSON-Strings für die jeweilige Klasse.

3.4.2 Funktionsweise und Aufbau

Beim Erstellen eines Objekts dieser Klasse muss nur die eigentliche Klasse (hier objClass genannt) dem Konstruktoraufruf übergeben werden. In “ClassToJson” sind alle Funktionalitäten umgesetzt, um für eine Klasse ein JSON zu erstellen. Dies funktioniert schrittweise.

```
1     public ClassToJsonV2(Class objClass){
2         this.objClass = objClass;
```

```
3           this.methods = objClass.getDeclaredMethods();
4           this.constructors = objClass.getDeclaredConstructors();
5           this.classname = objClass.getSimpleName();
6           setClassType();
7           setSuperClasses(objClass);
8           setMethodParameterClasses();
9           setConstructorParameterClasses();
10          setAllClasses();
11      }
```

Listing 3.7: Codeausschnitt ClassToJson Konstruktor

Im Konstruktor (Listing 3.7) werden erstmal die Methoden und Konstruktoren der objClass geholt. Hierfür werden die “getDeclared” Methoden benutzt. Die Besonderheit dieser Methoden ist, dass alle Methoden (auch private) der Klasse objClass, nicht jedoch Methoden welche die Klasse erbt, geholt werden. Es wird der Klassenname mit “getSimpleName()” gesetzt. Hierbei wurde aus Performancegründen auf den langen Namen verzichtet, da im Frontend sonst der String erst gesplittet werden müsste, was weitere Arbeit bedeuten würde. Mit der folgenden Set-Methode wird der Klassentyp (System, Source, Object) herausgefunden und gesetzt. Die restlichen Methodenaufrufe sind dafür da, um alle weiteren Klassen, die mit dieser Klasse in Verbindung stehen, in einer “ClassList” zu speichern. Diese “ClassList” beinhaltet dann, alle Klassen die als Parametertypen in Methoden und Konstruktoren übergeben werden können, sowie dessen Oberklassen bis zur Klasse “Object” und die Oberklassen der objClass bis zur Klasse “Object”. Diese Funktionalität wird später in der Klasse “JsonForCodeValidationInitializer” genutzt.

Durch das Aufrufen von “createJsonString()” auf ein erstelltes Objekt von “ClassToJson” wird das komplette JSON mithilfe eines “Stringbuilders” von Anfang an erstellt. Es wird unterschieden ob es sich bei objClass um eine normale Klasse oder ein Enum handelt. Sollte objClass ein Enum sein, dann wird die Klasse “EnumToJson” genutzt.

Um eine Klasse als JSON zu erstellen, wird als erstes die objClass, sowie dessen Oberklassen auf die “IOAnnotation” und “IOTypeAnnotation” überprüft. Die Ergebnisse werden in “in”, “out”, “inType” und “outType” gespeichert. In den nächsten Schritten, wird das JSON als String erstellt. Hierzu werden die Attribute der Klasse genutzt. Am Ende wird der Komplette JSON String für eine Klasse zurückgegeben.

3.5 EnumToJson

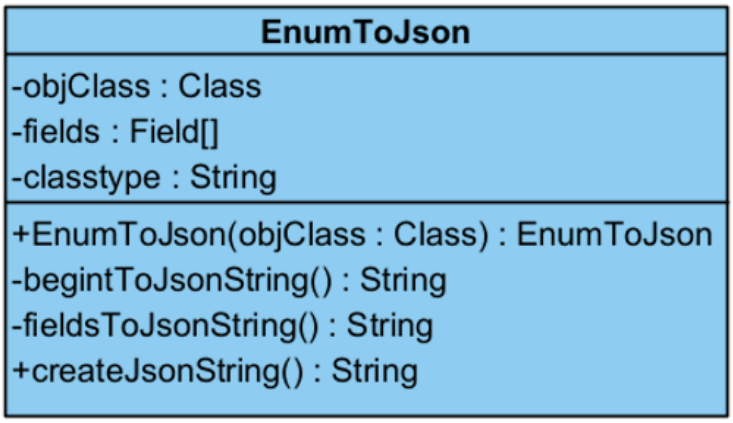

Abbildung 3.6: EnumToJson

3.5.1 Aufgabe

Erstellen eines JSON-Strings für die jeweilige Klasse.

3.5.2 Funktionsweise und Aufbau

Bei einem Enum sind nur die Enum-Felder von Interesse für den Enduser, deswegen gibt es hierfür eine eigene Klasse, wie sie in Abbildung 3.6 abgebildet wird. Diese hat auch eine Methode “createJsonString()”, welche einen kompletten JSON String mithilfe eines “Stringbuilders” zurückgibt. Diese Methode wird auf ein Objekt der Klasse “EnumToJson” in der Klasse “ClassToJson” angewendet, wenn es sich bei der übergebenen Klasse um ein Enum handelt. Um die Entwicklung im Frontend zu vereinfachen wird hier auch ein “classtype” gesetzt, dieser wird aber auf “enum” gesetzt.

3.6 JsonCreaterforInitialize

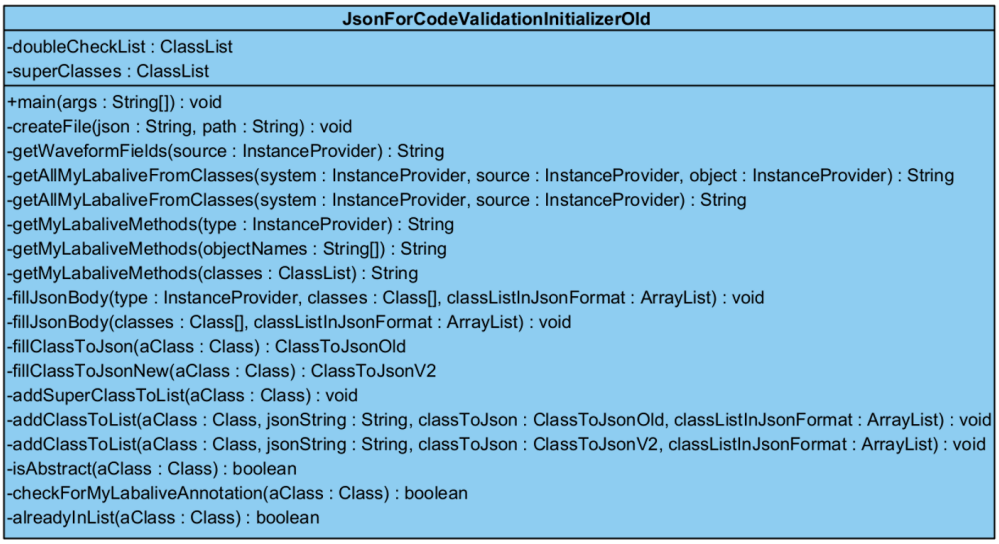

Abbildung 3.7: JsonForCodeValidationInitializer alter Aufbau

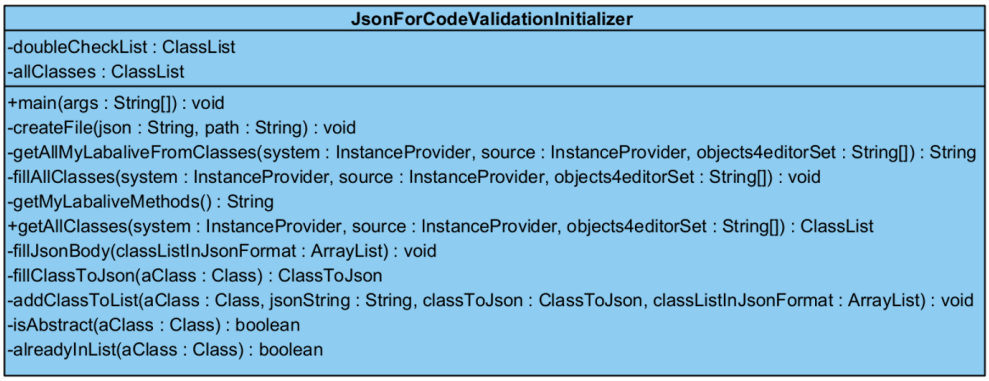

Abbildung 3.8: JsonForCodeValidationInitializer neuer Aufbau

Die Abbildung 3.7 zeigt die “JsonForCodeValidationIinitializer” Klasse im alten Aufbau. Im Gegensatz zur Abbildung 3.8, welche die Klasse im neuen Aufbau zeigt, sind weniger überladene Methoden vorhanden. Dies dient einerseits der neuen Umsetzung, andererseits der einfachen Lesbarkeit. Aus sicht der Cybersecurity ist ein einfacher Aufbau besser als ein komplizierter, da im einfachen Aufbau schneller Schwachstellen und Fehler erkannt werden können. Zudem ist die neue Funktionalität mehr auf einen sicheren Ablauf ausgelegt.

3.6.1 Aufgabe

Erstellen und speichern aller Klassen in einer JSON Datei.

3.6.2 Funktionsweise und Aufbau

Diese Klasse ist im Endeffekt die “Main” Klasse. Durch das Aufrufen der Methode “main()” wird für alle ausgewählten Klassen aus dem labAlive Projekt jeweils ein JSON String erstellt, welcher dann in ein komplettes JSON geschrieben werden. Am Ende wird noch eine neue Datei erstellte, in welcher das komplette JSON gespeichert wird. Die Datei wird später genutzt um das JSON in der Webseite also dem Frontend für die Code-Validierung einzubinden. Aufgrund der Komplexität wurden anfangs viel mehr Methoden benötigt, welche auch überladen wurden, weil die Funktionalität später unabhängig weiter genutzt werden sollte. Im Zuge der Einführung der Klasse “ClassList” wurde einiges an Funktionalität verändert. Gegen Ende dieser Arbeit wurde diese Klasse noch einmal verändert um diese einfacher und performanter zu gestallten. Als positiver Nebeneffekt ist noch eine Art Grundstein entstanden, jetzt kann die von dieser Klasse erstellte “ClassList” für zukünftige Zwecke weitergegeben und -verwendet werden.
Mit der Methode “getallMyLabaliveFromClasses()” in “main()” startet die Magie. Es müssen die “InstanceProvider” “System” und “Source”, sowie das “Arrayset” “Objects4Editor.Set” übergeben werden. Alle Klassen aus den Pfaden “System” und “Source” werden in die “ClassList” im Attribut “allClasses” hinzugefügt. Zusätzlich werden alle Klassen die etwas mit der Klasse zu tun haben (siehe “ClassToJson.setAllClasses()”) in die “allClasses” hinzugefügt. Bei “Object4Editor.Set” handelt es sich um einzelnen Strings, welche als erstes mit dem Packagenamen zu einem Klassenname zusammengesetzt werden müssen. Danach kann mit der Methode “Class.forName(Klassenname)” diese als Klasse identifiziert werden und der Liste übergeben werden. Es werden auch wieder alle Klassen die etwas mit der übergebenen Klasse zu tun haben in die “allClasses” hinzugefügt. Mit diesem ersten wichtigen Schritt liegt nun eine komplette Liste aller Klassen vor. Diese wird jetzt der Reihe nach durchgearbeitet. Für jede Klasse in der Liste wird ein Objekt der Klasse “ClassToJson” erstellt und mit “ClassToJson.createJsonString()” das jeweilige JSON Key-Value Paar für diese Klasse als String erstellt. Die Key-Value-Strings werden in einer “ArrayList” zwischengespeichert. Wenn alle Klassen fertig sind, dann wird die “ArrayList” in einen String mithilfe der Methode “String.join()” umgewandelt. Dabei wird nach jedem Key-Value Paar ein Komma gesetzt, außer es handelt sich um das Letzte oder die Liste besteht nur aus einem Paar. Der neue erstellte String beinhaltet dann das komplette JSON und wird in der “main()” an die Methode “createFile()” übergeben, so dass eine neue JSON Datei entsteht.

3.7 Datenformat

Die Wahl des Datenformats fiel leichter aus. Da es sich im Frontend um JavaScript handelt wurde als passendes Datenformat natürlich JSON gewählt. Bei der genauen Gestaltung der einzelnen Key-Value Paare wurden zwei verschiedene Formate getestet. Das wichtigste Kriterium ist die Geschwindigkeit, denn der User soll später nicht auf die Code-Validierung warten müssen. Weitere Kriterien die berücksichtigt wurden, waren der einfache Zugriff auf die Variablen und die Erweiterbarkeit um weitere Key-Value Paare. So können für jede Klasse beliebig viele Methoden und Konstruktoren hinzugefügt werden. Bei manchen Keys ist die Value ein eigenes JSON mit innenliegenden Key-Value Paaren.

3.7.1 Erstes Klassen Format

```
1 "classes": [
2         {
```

```
            "classname": "String",
            "in": "int",
            "out": "int",
            "superclass": "String",
            "methods": [
                {
                    "name": "String",
                    "parameters": ["String", "String"],
                    "returntype": "String"
                }
            ],
            "constructors": [
                        {
                            "name": "String",
                            "parameters": ["String", "String"],
                            "returntype": "String"
                        }
            ]
        }
    ]
```

Listing 3.8: JSON erstes Klassen Format

Der große Unterschied war im ersten Format (Listing 3.8), dass die JSON Datei aus einem einzigen Key-Value Paar bestand. Der Key war “classes” und als Value wurde ein Array aus einzelnen JSON genutzt. Jedes JSON hat eine eigene Klasse dargestellt. Der Klassenanme wurde in diesem JSON hinter dem Key “classname” als Value gesetzt. Im Listing (Listing 3.8) wird das erste Format dargestellt. Die Key-Value Paare werden im zweiten Format (Listing 3.9) genauer beschrieben.

3.7.2 Zweites Klassen Format

```
"classnameAsString":
    {
            "in":
                {
                    "count": "string",
                    "types": ["String","class"]
                },
            "out":
                {
                    "count": "string",
                    "types": ["String","class"]
                },
            "superclass": "String",
            "classtype": "String",
            "methods":
            [
                {
                    "name": "String",
                    "parameters": ["String", "String"],
                    "returntype": "String"
```

```
21                  }
22              ],
23              "constructors":
24              [
25                  {
26                      "name": "String",
27                      "parameters": ["String", "String"],
28                      "returntype": "String"
29                  }
30              ]
31      }
```

Listing 3.9: JSON zweites Klassen Format

Im zweiten Format (Listing 3.9) besteht das komplette JSON aus vielen (ca. 400) Key-Value Paaren. Nachdem die Klassennamen eindeutig sind, konnten diese jeweils als Key verwendet werden. Als Value wurde wieder ein eigenes JSON genutzt. In diesem ersten verschachtelten JSON wurden die Key-Value Paare wie in folgender Tabelle (Tabelle 3.1) gesetzt. Die JSON welche in der ersten Tabelle als Value gesetzt werden, werden in den folgenden beiden Tabellen, für die Keys “in” & “out” (Tabelle 3.2) und “methods” & “constructors” (Tabelle 3.3) genauer beschrieben.

Key	Bedeutung	Valuetype
in	Eingang des Bauteils	JSON
out	Ausgang des Bauteils	JSON
superclass	Elternklasse der Klasse	String
classtype	Klassentyp (System, Source, Object, Enum)	String
methods	Annotierte Methoden der Klasse	Array aus JSON
constructors	Annotierte Konstruktoren der Klasse	Array aus JSON

Tabelle 3.1: Erklärung Klassen JSON

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

Tabelle 3.2: Erklärung in&out JSON

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

Tabelle 3.3: Erklärung methods&constructors JSON

3.7.3 Enum Format

```
"classname":
      {
        "classtype": "String",
        "fields":
              [
                    "String",
                    "String"
              ]
      }
```

Listing 3.10: JSON Enum Format

Das Enum-Format (Listing 3.10) ist genauso wie ein Klassenformat aufgebaut. Der Klassenname ist der Key und in dem dazugehörigen JSON Value sind die Key-Value Paare “classtype”, welches den Klassentype hier “enum” beschreibt und “fields” welches als Value ein Array aus Strings hat und die Felder des Enums abspeichert.

3.7.4 Python Format Test

In dem Python Test Script (Kapitel 6.1.11 “Python Test Script”) werden als erstes verschiedene leere Listen angelegt, diese sind zum Testen da, ob alle Klassen, die als Value bei “superclass” stehen auch als Klassen im JSON vorhanden sind, sowie zu überprüfen ob eine Klasse mehrfach im JSON vorhanden ist. In der letzten Liste werden vier Klassen aufgenommen (z.B. SignalGenerator, Generator, Adder, String) und in dem JSON gesucht. Dabei wird am Ende das JSON zu jeder dieser Klassen ausgegeben, sowie die Ergebnisse aus den anderen Listen. Zusätzlich wird die Anzahl der Klassen und Enums im JSON ausgegeben. Dieser Test dient einerseits zum Überprüfen, ob das Format fehlerhaft ist andererseits zum Testen welches Format bei Zugriffen schneller ist.

3.7.5 Fazit

Nach verschiedenen Performance Versuchen mit einem einfachen Python-Script hat sich der gewünschte Geschwindigkeitsvorteil des zweiten Formates in Grenzen gehalten. Allerdings ist aufgefallen, dass das erste Format beim ersten Laden, also wenn der Cache leer ist immer schneller war als das erste Format. Bezüglich der weiteren Kriterien hat das zweite Format minimale Vorteile gegenüber dem Ersten. Die Entscheidung, welches Format zum Einsatz kommt, wurde relativ früh in der Entwicklung getroffen, deswegen unterscheiden sich die Datenformate auch in den Ein- und Ausgängen, außerdem hatte das erste Format zum damaligen Zeitpunkt noch kein Key-Value Paar für den Klassentyp.
Zum jetzigen Projektstand unterscheiden sich die beiden Formate um 0.001 Sekunden. Wobei abwechselnd mal das Eine, dann wieder das Andere schneller ist. Dabei muss nochmal erwähnt werden, dass beim ersten Laden, also wenn der Cache noch leer ist das zweite Format um 0.05 Sekunden schneller ist als das Erste. Allerdings ist die Art und Weise wie hier die Performance getestet wird nicht ganz fair.
Im ersten Klassen Format befinden sich 175 Klassen, weil dort z.b. die Klassen welche als Parameter von Methoden vorhanden sein können noch nicht berücksichtigt wurden, darunter fallen auch alle Enums. Zudem fehlen noch der Klassentyp und das eigene JSON von den Ein-/Ausgängen.
Im zweiten Format sind 407 Klassen vorhanden und es fehlen keine JSON bzw. Key-Value Paare. Zudem wird hier die Ausgabe erst noch bearbeitet (hinzufügen des Klassennamens), da in der Ausgabe nur zu dem Key die jeweilige Value, nicht aber der Key, ausgegeben wird.

Abschließend fällt bei dem Vergleich also auf, dass das zweite Format mehr arbeiten muss (Füllen der Enumliste, Aufbauen Ausgabestring) über doppelt so viele Einträge hat und trotzdem noch genauso schnell ist wie das erste Format. Beim ersten Laden war das zweite Format sogar immer schneller. Deswegen bestätigt sich hier die Anfangsannahme, dass das zweite Format eine bessere Performance hat als das erste und die Wahl fiel auf das zweite Format.

4 Anwendung

4.1 Erweiterung der Klassenliste

Nachdem die gesamte Anwendung erweiterbar gestaltet ist wird hier erklärt, wie weitere Klassen in das JSON bzw. die Klassenliste eingetragen werden können.
Die erste Möglichkeit ist, im Source oder System Pfad neue Klassen anzulegen. Nachdem die beiden Pfade automatisch komplett übernommen werden, muss hier nichts weiteres getan werden, außer die Klasse zu erstellen. Bei der zweiten Möglichkeit muss in der Klasse “Objects4Editor” das “Objects4Editor.SET” erweitert werden. Hierbei muss beachtet werden, dass der Anfang “de.labAlive” nicht mit in die Liste mit aufgenommen wird, da dieser im Code automatisch hinzugefügt wird.

```
1  public static final String[] SET = {
2              // Params for measures
3          "measure.ConstellationDiagram",
4          "measure.Power",
5          "measure.Scope"
6
7          // further objects
8
9          "java.lang.String"
10 }
```

Wie in Listing 4.1 am Beispiel des “Scope” zu erkennen ist, wird der Anfang “de.labAlive” weggelassen. Außerdem können auch andere Klassen wie “java.lang.String” in das JSON bzw. die Klassenliste aufgenommen werden.

4.2 Ausführung

Um das JSON zu erstellen muss in der Klasse “JsonForCodeValidationInitializer” die Methode “main()” aufgerufen werden. Im Listing 4.2 ist nochmal die “main()” Methode aus der Klasse “JsonForCodeValidationInitializer” aufgezeigt.

```
1      public static void main(String[] args) throws ↩
           ClassNotFoundException {
2          String path = "LabAliveClasses.json";
3          String json = getAllMyLabaliveFromClasses(SYSTEM, ↩
               SOURCE, Objects4Editor.SET);
4          System.out.println(json);
5
6          // uncomment to create jsonfile -> change path first
7          createFile(json, path);
8
9      }
```

In weiteren Projekten wird ggf. die JSON Datei nicht benötigt. Um nur eine Klassenliste im “ClassList” Format zu bekommen, muss die Methode “getAllClasses()” (Listing 4.2) aufgerufen werden.

```
1    public static ClassList<Class> ↩
         getAllClasses(InstanceProvider system, InstanceProvider ↩
         source, String[] object4editorSet) throws ↩
         ClassNotFoundException {
2         fillAllClasses(system, source, object4editorSet);
3         return allClasses;
4    }
```

5 Fazit

5.1 Ergebnisse

Die Website hat jetzt für den User eine einfache Entwicklungsoberfläche mit Code Validierung in welcher der User seinen Code auf Syntax und Semantik überprüfen kann. Die Code Validierung vor dem Starten der Apps konnte in der Kürze der Zeit leider nicht verwirklicht werden. Die erforderlichen Informationen für die Code Validierung werden im Backend gesammelt und als JSON Datei an das Frontend übergeben. Die Datei ist nur 77kB groß umfasst 407 Klassen und der Zugriff erfolgt in Python in unter 0.01 Sekunden. Dies ist schnell genug, dass sogar eine automatische Code Ergänzung möglich ist. Die Funktionalität ist so aufgebaut, dass über das Objects4Editor.Set weitere Klassen angegeben werden können. Bei diesen weiteren Klassen werden dann auch die Parameterklassen und Elternklassen übernommen. Das JSON kann beliebig um Key-Value Paare erweitert werden. Vererbte Methoden werden nicht doppelt übernommen, somit werden keine unnötigen Daten gespeichert. In der JsonForCodeValidationInitializer Klasse sind Methoden erstellt, welche noch nicht genutzt werden. Diese geben eine komplette ClassList zurück, in welcher alle Klassen, die auch im JSON vorhanden sind, gespeichert sind. Mit dieser können spätere Funktionen verwirklicht werden.
Zusätzlich ist als Nebenprodukt eine Anleitung entstanden, mit der das labAlive Projekt ohne Virtuelle Maschine auf MacOS bzw. auch auf Windows mit IntelliJ und/oder Eclipse weiter entwickelt werden kann.

5.2 Herausforderungen

Grundsätzlich ist das Anfangsproblem immer das gleiche bei so einem riesigen Projekt. Bei ca. 1700 Klassen muss man sich erst einmal einen Überblick verschaffen. Eine weitere Herausforderung war das gleichzeitige Arbeiten von vielen Personen an dem Projekt. Alles musste sehr gut abgesprochen werden und nicht jeder findet den Code so leserlich wie der der diesen erstellt hat. Ein Beispiel ist das JSON Format, bei einer Änderung mussten einige Zugriffsfunktionen auf der Weboberfläche umgeschrieben werden, was natürlich einiges an Zeit kostet.

5.3 Ausblick

Mit den Funktionalitäten im neuen Java Reflections Package kann können einige Methoden aus dieser Arbeit vereinfacht werden. Dazu müsste aber das gesamte Projekt auf die neueste Java Version hochgezogen werden.
Nachdem eine projektweite Umsetzung der Validierung innerhalb dieser drei Monate zeitlich nicht möglich war, bietet diese Arbeit die Grundlagen für weitere studentische Arbeiten. Diese Grundlagen können in weiteren Projekten genutzt werden, um einen einheitlichen Validierungsservice zu erstellen. Somit kann der Code für jede App erst einmal überprüft werden und dann die App direkt mit dem richtigen Systemen gestartet werden. Eine weiter Vorstellung wäre natürlich, den labAlive Code in eine einfachen Programmiersprache mit Algorithmen zu entwickeln, sowie dafür einen eigenen Compiler

zu erstellen. Weiterhin kann z.B. mit Python ein “neuronales Netzwerk” erstellt werden, welches die meist genutzten Klassen im Code automatisch als erstes vorschlägt. In Abbildung 5.1 wird ein möglicher zeitlicher Verlauf für zukünftige studentische Arbeiten nach dieser Arbeit aufgezeigt.
5.1

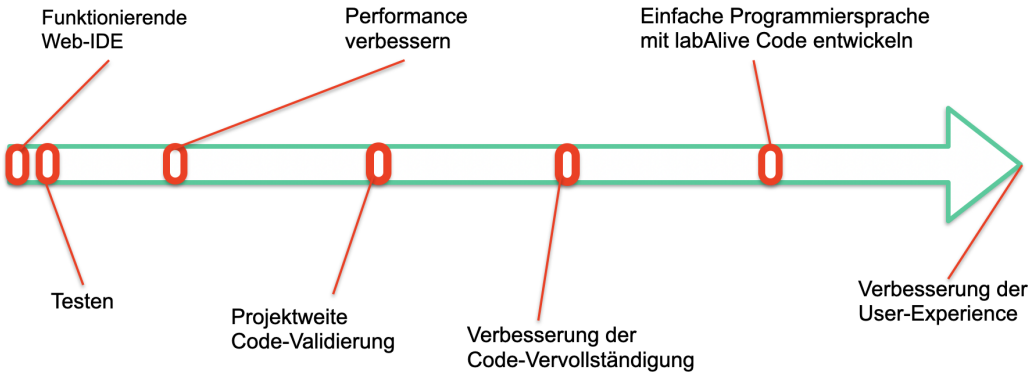

Abbildung 5.1: Ausblick

6 Anhang

6.1 QuellCode

Hier wird der Code, der einzelnen Klassen, sowie des Python Tests eingefügt. Zusätzlich werden die JSON Formate für die Klassen und Enums nochmal aufgezeigt.

6.1.1 @MyLabAlive

```
1  package de.labAlive;
2
3  import java.lang.annotation.ElementType;
4  import java.lang.annotation.Retention;
5  import java.lang.annotation.RetentionPolicy;
6  import java.lang.annotation.Target;
7
8  @Retention(RetentionPolicy.RUNTIME)
9  // Target: TYPE = Class/Enum, METHOD = Method, CONSTRUCTOR = ↩
       Constructor, FIELD = Enum-field
10 @Target({ElementType.TYPE, ElementType.METHOD, ↩
       ElementType.CONSTRUCTOR, ElementType.FIELD})
11 public @interface MyLabalive {
12     // no values in this Annotation
13 }
```

Listing 6.1: MyLabalive Annotation

6.1.2 @IOAnnotation

```
1  package de.labAlive;
2
3  import java.lang.annotation.ElementType;
4  import java.lang.annotation.Retention;
5  import java.lang.annotation.RetentionPolicy;
6  import java.lang.annotation.Target;
7
8
9  @Retention(RetentionPolicy.RUNTIME)
10 @Target({ElementType.TYPE})
11 // Target: TYPE = Class/Enum, METHOD = Method, CONSTRUCTOR = ↩
       Constructor, FIELD = Enum-field
12 public @interface IOAnnotation {
13     String in(); // count of inputs
14     String out(); // count of outputs
15 }
```

Listing 6.2: IOAnnotation Annotation

6.1.3 @IOTypeAnnotation

```java
package de.labAlive;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
// Target: TYPE = Class/Enum, METHOD = Method, CONSTRUCTOR = Constructor, FIELD = Enum-field
public @interface IOTypeAnnotation {
    String[] inType();
    String[] outType();
}
```

Listing 6.3: IOTypeAnnotation Annotation

6.1.4 ClassList

```java
package de.labAlive.wiring.editor.parse.creation;

import java.util.ArrayList;
import java.lang.Class;

public class ClassList<E extends Class> extends ArrayList<E> {

    public ClassList () {

    }

    public void addClassesFromAnotherList(ClassList<E> classes){
        for(E aClass : classes){
            this.addClassToList(aClass);
        }
    }

    public void addClassToList(E aClass){
        //String[] packageNameSplit = aClass.getPackageName().split("\\.");
        if(aClass != null && !alreadyInList(aClass) && !aClass.equals(Object.class)){ //&& !aClass.isInterface() && packageNameSplit[1].equals("labAlive")){
            this.add(aClass);
        }
    }

    private boolean alreadyInList(E aClass){
        if(this.contains(aClass)){
```

```
27              return true;
28          }
29          for(E classInList : this){
30              if(aClass.getSimpleName().equals(classInList.getSimpleName())){
31                  return true;
32              }
33          }
34          return false;
35      }
36
37  }
```

Listing 6.4: ClassList

6.1.5 ClassToJson

```
1  package de.labAlive.wiring.editor.parse.creation;
2
3  import de.labAlive.IOAnnotation;
4  import de.labAlive.IOTypeAnnotation;
5  import de.labAlive.MyLabalive;
6
7  import java.lang.annotation.Annotation;
8  import java.lang.reflect.Constructor;
9  import java.lang.reflect.Method;
10
11 import static ↩
       de.labAlive.wiring.editor.parse.creation.InstanceProvider.*;
12
13 public class ClassToJson {
14      private final Class objClass;
15      private final Method[] methods;
16      private final Constructor[] constructors;
17      private String in;
18      private String[] inType;
19      private String out;
20      private String[] outType;
21      private final String classname;
22      private final ClassList<Class> parameterClasses = new ↩
            ClassList<>();
23      private final ClassList<Class> superClasses = new ↩
            ClassList<>();
24      private String classType;
25      private final ClassList<Class> allClasses = new ↩
            ClassList<>();
26
27      public ClassToJson(Class objClass){
28          this.objClass = objClass;
29          // getDeclared gibt auch private methoden zurück, aber ↩
               nicht die der oberklasse
```

```
30          this.methods = objClass.getDeclaredMethods();
31          this.constructors = objClass.getDeclaredConstructors();
32          this.classname = objClass.getSimpleName();
33          setClassType();
34          setSuperClasses(objClass);
35          setMethodParameterClasses();
36          setConstructorParameterClasses();
37          setAllClasses();
38      }
39
40      public ClassList<Class> getAllClasses(){
41          return allClasses;
42      }
43
44      public ClassList<Class> getParameters(){
45          return parameterClasses;
46      }
47
48      public String getClassname() {
49          return classname;
50      }
51
52      private void setAllClasses(){
53          allClasses.addClassesFromAnotherList(superClasses);
54          allClasses.addClassesFromAnotherList(parameterClasses);
55      }
56
57      private void setSuperClasses(Class aClass){
58          Class superclass = aClass;
59          while (superclass != null && ↩
                !superclass.equals(Object.class)){
60              superClasses.addClassToList(superclass);
61              superclass = superclass.getSuperclass();
62          }
63      }
64
65      private void setMethodParameterClasses(){
66          for (Method method : methods){
67              for(Annotation annotationtype : ↩
                    method.getAnnotations()) {
68                  if (annotationtype instanceof MyLabalive) {
69                      Class [] parameters = ↩
                            method.getParameterTypes();
70                      for(Class aParameterClass : parameters){
71                          parameterClasses.addClassToList(aParameterClass);
72                          setSuperClasses(aParameterClass);
73                          /*
74                          Class superclass = aParameterClass;
75                          while (superclass != null && ↩
                                !superclass.equals(Object.class)){
```

```
76                              superClasses.addClassToList(superclass);
77                              superclass = ↩
                                    superclass.getSuperclass();
78                          }
79                          */
80                      }
81                      break;
82                  }
83              }
84          }
85      }
86
87      private void setConstructorParameterClasses(){
88          for (Constructor constructor : constructors){
89              for(Annotation annotationtype : ↩
                    constructor.getAnnotations()) {
90                  if (annotationtype instanceof MyLabalive) {
91                      Class [] parameters = ↩
                            constructor.getParameterTypes();
92                      for(Class aParameterClass : parameters){
93                          parameterClasses.addClassToList(aParameterClass);
94                          setSuperClasses(aParameterClass);
95                          /*
96                          Class superclass = aParameterClass;
97                          while (superclass != null && ↩
                                !superclass.equals(Object.class)){
98                              superClasses.addClassToList(superclass);
99                              superclass = ↩
                                    superclass.getSuperclass();
100                         }
101                         */
102                     }
103                     break;
104                 }
105             }
106         }
107     }
108
109     private void getInAndOut(){
110         // * = all types
111         // n = any count
112         Class superclass = objClass;
113         while (superclass != null && ↩
                !superclass.equals(Object.class)) {
114             for(Annotation annotationIO : ↩
                    superclass.getAnnotations()){
115                 if (annotationIO instanceof IOAnnotation){
116                     in = ((IOAnnotation) annotationIO).in();
117                     out = ((IOAnnotation) annotationIO).out();
118                 }
```

```
119                 else if(annotationIO instanceof ↩
                        IOTypeAnnotation){
120                     inType = ((IOTypeAnnotation) ↩
                            annotationIO).inType();
121                     outType = ((IOTypeAnnotation) ↩
                            annotationIO).outType();
122                 }
123             }
124             superclass = superclass.getSuperclass();
125         }
126         for(Annotation annotationType : ↩
                objClass.getAnnotations()){
127             if (annotationType instanceof IOTypeAnnotation) {
128                 inType = ((IOTypeAnnotation) ↩
                        annotationType).inType();
129                 outType = ((IOTypeAnnotation) ↩
                        annotationType).outType();
130             }
131             else if (annotationType instanceof IOAnnotation){
132                 in = ((IOAnnotation) annotationType).in();
133                 out = ((IOAnnotation) annotationType).out();
134             }
135         }
136         if(classType.equals("object")){
137             in = "0";
138             out = "0";
139         }
140         if(inType == null){
141             inType = new String[]{"?"};
142         }
143         if(outType == null){
144             outType = new String[]{"?"};
145         }
146         if(in == null){
147             in = "?";
148         }
149         if(out == null){
150             out = "?";
151         }
152     }
153
154     private void setClassType(){
155         if(SOURCE.CLASS.isAssignableFrom(objClass)){
156             classType = "source";
157         }
158         else if(SYSTEM.CLASS.isAssignableFrom(objClass)){
159             classType = "system";
160         }
161         else if(OBJECT.CLASS.isAssignableFrom(objClass)){
162             // TODO: classtype = object if class is from ↩
```

```
                Objects4Editor.SET
163             classType = "object";
164         }
165         else if(OBJECT.CLASS.isAssignableFrom(objClass)){
166             // TODO: OBJECT is the hole project ???
167             classType = "object";
168         }
169         else {
170             classType = "null";
171         }
172     }
173 
174     private String beginToJsonString(){
175         String inAndOut = inAndOutToJson();
176         String superclass = "";
177         if(objClass.getSuperclass() != null){
178             superclass = ↩
                    objClass.getSuperclass().getSimpleName();
179         }
180         String jsonString = "\"" + classname + "\": {" +
181                             inAndOut +
182                             "\"superclass\": \"" + superclass ↩
                                + "\"," +
183                             "\"classtype\": \"" + classType + ↩
                                "\",";
184         return jsonString;
185 
186         /* Format:
187 
188             "classnameasString": {
189                 "in": {
190                     "count": "string",
191                     "types": ["String","String"]
192                 },
193                 "out": {
194                     "count": "string",
195                     "types": ["String","String"]
196                 },
197                 "superclass": "String",
198                 "classtype": "String",
199 
200          */
201     }
202 
203     private String inAndOutToJson(){
204         getInAndOut();
205         StringBuilder inTypeJsonLine = new StringBuilder();
206         inTypeJsonLine.append(" ");
207         for(String aType : inType ){
208             inTypeJsonLine.append("\"" + aType + "\",");
```

```
209         }
210         String inLine =  inTypeJsonLine.substring(0, ↩
               inTypeJsonLine.length()  -1);
211         StringBuilder outTypeJsonLine = new StringBuilder();
212         outTypeJsonLine.append(" ");
213         for(String aType : outType ){
214             outTypeJsonLine.append("\"" + aType + "\",");
215         }
216         String outLine =  outTypeJsonLine.substring(0, ↩
               outTypeJsonLine.length()  -1);
217         String inAndOutJsonLine = "\"in\": { \"count\": \"" + ↩
               in + "\"," +
218                                             "\"types\": [" + ↩
                                                inLine + "]}," +
219                                     "\"out\": { \"count\": \"" + ↩
                                        out + "\"," +
220                                             "\"types\": [" + ↩
                                                outLine + "]}," +
221                                     "";
222         return inAndOutJsonLine;
223     }
224
225     private String methodsToJsonString(){
226         StringBuilder sb = new StringBuilder();
227         sb.append("\"methods\": [  ");
228         for (Method method : methods){
229             Annotation[] annotations = method.getAnnotations();
230             for (Annotation annotation : annotations){
231                 if (annotation instanceof MyLabalive){
232                     String name = method.getName();
233                     sb.append("{\"name\": \"" + name + "\", ");
234                     sb.append("\"parameters\": [");
235                     Class[] parameters = ↩
                           method.getParameterTypes();
236                     if(parameters.length == 0){
237                         sb.append("],");
238                     }
239                     for(int i = 0; i < parameters.length; i++){
240                         String parametername = ↩
                               parameters[i].getName();
241                         if (i < parameters.length - 1){
242                             sb.append("\"" + parametername + ↩
                                   "\", ");
243                         }
244                         else{
245                             sb.append("\"" + parametername + ↩
                                   "\"], ");
246                         }
247                     }
248                     String returntype = ↩
```

```
                    method.getReturnType().toString();
249                 sb.append("\"returntype\": " + "\"" + returntype + "\"");
250                 sb.append("}, ");
251             }
252         }
253     }
254     String tmp = sb.toString();
255     String jsonString = tmp.substring(0, tmp.length() -2) + "]";
256     return jsonString;
257
258     /* Format:
259     "methods": [
260             {
261                 "name": "String",
262                 "parameters": ["String", "String"],
263                 "returntype": "String"
264             },
265             {
266                 "name": "String",
267                 "parameters": ["String", "String"],
268                 "returntype": "String"
269             }
270         ],
271      */
272 }
273
274 private String constructorsToJsonString(){
275     StringBuilder sb = new StringBuilder();
276     sb.append("\"constructors\": [  ");
277     for (Constructor constructor : constructors) {
278         Annotation[] annotations = constructor.getAnnotations();
279         for (Annotation annotation : annotations) {
280             if (annotation instanceof MyLabalive) {
281                 String name = constructor.getName();
282                 sb.append("{\"name\": \"" + name + "\", ");
283                 sb.append("\"parameters\": [");
284                 Class[] parameters = constructor.getParameterTypes();
285                 if (parameters.length == 0) {
286                     sb.append("]");
287                 }
288                 for (int i = 0; i < parameters.length; i++) {
289                     String parametername = parameters[i].getName();
290                     if (i < parameters.length - 1) {
291                         sb.append("\"" + parametername + "
```

```
                                "\", ");
292                         } else {
293                             sb.append("\"" + parametername + ↩
                                    "\"]");
294                         }
295                     }
296                     sb.append("}, ");
297                 }
298             }
299         }
300         String tmp = sb.toString();
301         String jsonString = tmp.substring(0, tmp.length() -2) ↩
                + "]";
302         return jsonString;
303 
304         /* Format:
305 
306         "constructors": [
307                         {
308                             "name": "String",
309                             "parameters": ["String", "String"]
310                         },
311                         {
312                             "name": "String",
313                             "parameters": ["String", "String"]
314                         }
315                     ]
316          */
317     }
318 
319 
320     public String createJsonString(){
321         if(classnameIsNull()){
322             return null;
323         } else if (objClass.isEnum()) {
324             classType = "enum";
325             EnumToJson enumToJson = new EnumToJson(objClass);
326             String json = enumToJson.createJsonString();
327             return json;
328         }
329         String json = beginToJsonString() + ↩
                methodsToJsonString() + "," + ↩
                constructorsToJsonString() + "}";
330         return json;
331     }
332 
333     private boolean classnameIsNull(){
334         return objClass.getSimpleName().equals("");
335     }
336 
```

```
337
338     /* New Format without "classes" at the beginning:
339     {
340             "classnameasString": {
341                 "in": {
342                     "count": "string",
343                     "types": ["String","class"]
344                 },
345                 "out": {
346                     "count": "string",
347                     "types": ["String","class"]
348                 },
349                 "superclass": "String",
350                 "classtype": "String",
351                 "methods": [
352                 {
353                     "name": "String",
354                     "parameters": ["String", "String"],
355                     "returntype": "String"
356                 },
357                 {
358                     "name": "String",
359                     "parameters": ["String", "String"],
360                     "returntype": "String"
361                 }
362             ],
363             "constructors": [
364                         {
365                             "name": "String",
366                             "parameters": ["String", "String"],
367                             "returntype": "String"
368                         },
369                         {
370                             "name": "String",
371                             "parameters": ["String", "String"],
372                             "returntype": "String"
373                         }
374                     ]
375             }
376         }
377      */
378
379 }
```

Listing 6.5: ClassToJson

6.1.6 EnumToJson

```
1 package de.labAlive.wiring.editor.parse.creation;
2
```

```
3  import java.lang.reflect.Field;
4
5  public class EnumToJson {
6      private final Class<?> objClass;
7      private final Field[] fields;
8      private final String classtype;
9
10     public EnumToJson(Class<?> objClass){
11         this.objClass = objClass;
12         this.fields = objClass.getFields();
13         classtype = "enum";
14     }
15
16     private String beginToJsonString(){
17         String sb = "\"" + objClass.getSimpleName() + "\": {" +
18                 "\"classtype\": \"" + classtype + "\", " +
19                 "\"fields\": " +
20                 "[";
21         return sb;
22     }
23
24     private String fieldsToJsonString(){
25         StringBuilder sb = new StringBuilder();
26         for (Field field : fields) {
27             String fieldName = field.getName();
28             sb.append("\"" + fieldName + "\",");
29         }
30         String tmp = sb.toString();
31         if(tmp.length()<1){
32             return "";
33         }
34         String jsonString = tmp.substring(0, tmp.length() -1) ↩
               + "]}";
35         return jsonString;
36     }
37
38     public String createJsonString(){
39         StringBuilder sb = new StringBuilder();
40         String jsonfields = fieldsToJsonString();
41         if(jsonfields.equals("") || ↩
               objClass.getSimpleName().equals("")){
42             return null;
43         }
44         sb.append(beginToJsonString());
45         sb.append(jsonfields);
46         return sb.toString();
47     }
48
49     /*
50     FORMAT:
```

```
51
52      {
53        "classname":
54        {
55          "classtype": "String",
56          "fields":
57              [
58                  "String",
59                  "String"
60              ]
61        }
62      }
63
64      */
65
66  }
```

Listing 6.6: EnumToJson

6.1.7 JsonCreaterForInitialize

```
1  package de.labAlive.wiring.editor.parse.creation;
2
3  import static ↩
       de.labAlive.wiring.editor.parse.creation.InstanceProvider.SOURCE;
4  import static ↩
       de.labAlive.wiring.editor.parse.creation.InstanceProvider.SYSTEM;
5
6  import java.io.File;
7  import java.io.FileWriter;
8  import java.lang.reflect.Constructor;
9  import java.lang.reflect.Method;
10 import java.lang.reflect.Modifier;
11 import java.util.ArrayList;
12 import de.labAlive.wiring.editor.objects.Objects4Editor;
13 import de.labAlive.wiring.editor.parse.util.Classfinder;
14
15 public class JsonForCodeValidationInitializer {
16     private static final ClassList<Class> allClasses = new ↩
           ClassList<>();
17     private static final ClassList<Class> doubleCheckList = ↩
           new ClassList<>();
18
19
20     public static void main(String[] args) throws ↩
           ClassNotFoundException {
21         String path = "LabAliveClassesFinal.json";
22         String json = getAllMyLabaliveFromClasses(SYSTEM, ↩
               SOURCE, Objects4Editor.SET);
23         System.out.println(json);
```

```
24
25          // uncomment to create jsonfile -> change path first
26          createFile(json, path);
27
28          // Object öffnet Java Anwendung? deswegen mit exit ↩
               beenden
29          // System.exit(0);
30      }
31
32      private static void createFile(String json, String path){
33          try {
34              File myFile = new File(path);
35              if (myFile.createNewFile()) {
36                  FileWriter myWriter = new FileWriter(path);
37                  myWriter.write(json);
38                  myWriter.close();
39                  System.out.println("File created: " + ↩
                       myFile.getName());
40              } else {
41                  System.out.println("File already exists.");
42              }
43          } catch (Exception e) {
44              e.printStackTrace();
45          }
46      }
47
48      private static String ↩
           getAllMyLabaliveFromClasses(InstanceProvider system, ↩
           InstanceProvider source, String[] object4editorSet) ↩
           throws ClassNotFoundException {
49          fillAllClasses(system, source, object4editorSet);
50          String completeJSON;
51          String allClasses = getMyLabaliveMethods();
52
53          completeJSON = "{" +
54                  allClasses +
55                  "}";
56          return completeJSON;
57      }
58
59      private static void fillAllClasses(InstanceProvider ↩
           system, InstanceProvider source, String[] ↩
           object4editorSet) throws ClassNotFoundException {
60          for(Class aClass : Classfinder.getClasses(system.PATH)){
61              ClassToJson classToJson = new ClassToJson(aClass);
62              allClasses.addClassesFromAnotherList(classToJson.getAllClasses());
63              allClasses.addClassToList(aClass);
64          }
65          for(Class aClass : Classfinder.getClasses(source.PATH)){
66              ClassToJson classToJson = new ClassToJson(aClass);
```

```
67              allClasses.addClassesFromAnotherList(classToJson.getAllClasses());
68              allClasses.addClassToList(aClass);
69          }
70          for (String objectName : object4editorSet){
71              String[] tmp = objectName.split("\\.");
72              if (tmp[0].equals("java")){
73                  Class<?> aClass = Class.forName(objectName);
74                  ClassToJson classToJson = new ↩
                        ClassToJson(aClass);
75                  allClasses.addClassesFromAnotherList(classToJson.getAllClasses());
76                  allClasses.addClassToList(aClass);
77              }
78              else{
79                  String classname = "de.labAlive." + objectName;
80                  Class<?> aClass = Class.forName(classname);
81                  ClassToJson classToJson = new ↩
                        ClassToJson(aClass);
82                  allClasses.addClassesFromAnotherList(classToJson.getAllClasses());
83                  allClasses.addClassToList(aClass);
84              }
85          }
86
87      }
88
89      public static ClassList<Class> ↩
           getAllClasses(InstanceProvider system, InstanceProvider ↩
           source, String[] object4editorSet) throws ↩
           ClassNotFoundException {
90          fillAllClasses(system, source, object4editorSet);
91          return allClasses;
92      }
93
94      private static String getMyLabaliveMethods(){
95          ArrayList<String> classListInJsonFormat = new ↩
               ArrayList<>();
96          String classStringInJsonFormat;
97          fillJsonBody(classListInJsonFormat);
98          // more than one class -> seperate json Format strings ↩
               with ,
99          if (classListInJsonFormat.size() > 1) {
100             classStringInJsonFormat = String.join(", ", ↩
                    classListInJsonFormat);
101         }
102         // only one class -> no seperation
103         else{
104             classStringInJsonFormat = String.join("", ↩
                    classListInJsonFormat);
105         }
106         // build the complete JsonString
107         String jsonString = classStringInJsonFormat;
```

```
108         return jsonString;
109     }
110 
111     private static void fillJsonBody(ArrayList<String> ↩
        classListInJsonFormat){
112         for(Class aClass : allClasses){
113             if(!alreadyInList(aClass)) {
114                 ClassToJson classToJson = ↩
                    fillClassToJson(aClass);
115                 String jsonString = ↩
                    classToJson.createJsonString();
116                 addClassToList(aClass, jsonString, ↩
                    classToJson, classListInJsonFormat);
117             }
118         }
119     }
120 
121     private static ClassToJson fillClassToJson(Class aClass){
122         Constructor[] constructors = aClass.getConstructors();
123         // getDeclared gibt auch private methoden zurück, aber ↩
            nicht die der oberklasse
124         Method[] methods = aClass.getDeclaredMethods();
125         ClassToJson classToJson = new ClassToJson(aClass);
126         return classToJson;
127     }
128 
129     private static void addClassToList(Class aClass, String ↩
        jsonString, ClassToJson classToJson, ArrayList<String> ↩
        classListInJsonFormat){
130         if (jsonString != null){
131             classListInJsonFormat.add(classToJson.createJsonString());
132             doubleCheckList.addClassToList(aClass);
133         }
134     }
135 
136     private static boolean isAbstract(Class<?> aClass) {
137         // not in use
138         int mod = aClass.getModifiers();
139         return Modifier.isAbstract(mod);
140     }
141 
142     private static boolean alreadyInList(Class aClass){
143         if(doubleCheckList.contains(aClass)){
144             return true;
145         }
146         for(Class classInList : doubleCheckList){
147             if(aClass.getSimpleName().equals(classInList.getSimpleName())){
148                 return true;
149             }
150         }
```

```
151             return false;
152         }
153
154 }
```

Listing 6.7: JsonCreaterForInitialize

6.1.8 Erstes Klassen Datenformat

```
1  "classes": [
2          {
3              "classname": "String",
4              "in": "int",
5              "out": "int",
6              "superclass": "String",
7              "methods": [
8                  {
9                      "name": "String",
10                     "parameters": ["String", "String"],
11                     "returntype": "String"
12                 }
13             ],
14             "constructors": [
15                         {
16                             "name": "String",
17                             "parameters": ["String", "String"],
18                             "returntype": "String"
19                         }
20             ]
21         }
22     ]
```

Listing 6.8: Erstes Klassen Datenformat

6.1.9 Zweites Klassen Datenformat

```
1  "classnameAsString":
2      {
3              "in":
4                  {
5                      "count": "string",
6                      "types": ["String","class"]
7                  },
8              "out":
9                  {
10                     "count": "string",
11                     "types": ["String","class"]
12                 },
13             "superclass": "String",
14             "classtype": "String",
```

```
            "methods":
            [
                {
                    "name": "String",
                    "parameters": ["String", "String"],
                    "returntype": "String"
                }
            ],
            "constructors":
            [
                {
                    "name": "String",
                    "parameters": ["String", "String"],
                    "returntype": "String"
                }
            ]
    }
```

Listing 6.9: Zweites Klassen Datenformat

6.1.10 Enum Datenformat

```
"classname":
      {
        "classtype": "String",
        "fields":
              [
                    "String",
                    "String"
              ]
      }
```

Listing 6.10: Enum Datenformat

6.1.11 Python Test Script

```
import time
import json

def oldVersion():
    start = time.time()
    print("Altes Format: ")
    list = []
    superclasses = []
    notinlist = []
    searchlist = ["SignalGenerator", "Generator", "Adder", ↩
       "String"]
    with open("labaliveoldformat.json") as file:
        jsonfile = json.loads(file.read())
    for aClass in jsonfile.get("classes"):
```

```
14          list.append(aClass["classname"])
15          superclasses.append(aClass["superclass"])
16          if aClass["classname"] in searchlist:
17              print(aClass)
18          else:
19              for tmp in aClass["methods"]:
20                  if tmp["name"] in searchlist:
21                      print(aClass)
22
23      for x in superclasses:
24          if x not in list:
25              if x not in notinlist:
26                  if x != "Object":
27                      notinlist.append(x)
28
29      visited = set()
30      doubles = [x for x in list if x in visited or ↩
            (visited.add(x) or False)]
31      if len(doubles) < 1:
32          print("Doppelte: " + str(len(doubles)))
33      else:
34          print("Anzahl Doppelte: " + str(len(doubles)) + ↩
              "\nDoppelte Klassen: " + str(doubles))
35      print("Anzahl Klassen im JSON: " + str(len(list)))
36      print("Nicht im JSON: " + str(len(notinlist)))
37      print(notinlist)
38      print("Alle Klassen: " + str(list))
39
40      end = time.time()
41      print('{:5.5f}s'.format(end - start))
42      print("\n")
43      return end - start
44
45
46  def newVersion():
47      start = time.time()
48      print("Neues Format: ")
49      enums = []
50      list = []
51      superclasses = []
52      notinlist = []
53      searchlist = ["SignalGenerator", "Generator", "Adder", ↩
            "String"]
54      with open("LabAliveClassesFinal.json") as file:
55          jsonfile = json.loads(file.read())
56      newVersionCheck(jsonfile, list, superclasses, notinlist, ↩
            searchlist, enums)
57      end = time.time()
58      print('{:5.5f}s'.format(end - start))
59      print("\n")
```

```
60      return end - start
61
62
63  def newVersionCheck(jsonfile, list, superclasses, notinlist, ↩
        searchlist, enums):
64      for search in searchlist:
65          if search in jsonfile:
66              print(str(search) + " : " + str(jsonfile[search]))
67      for aClass in jsonfile:
68          list.append(aClass)
69          try:
70              superclasses.append(jsonfile[aClass]['superclass'])
71          except:
72              enums.append(jsonfile[aClass])
73      for x in superclasses:
74          if x not in list:
75              if x not in notinlist:
76                  if x != "Object":
77                      notinlist.append(x)
78
79      visited = set()
80      doubles = [x for x in list if x in visited or ↩
            (visited.add(x) or False)]
81      if len(doubles) < 1:
82          print("Doppelte: " + str(len(doubles)))
83      else:
84          print("Anzahl Doppelte: " + str(len(doubles)) + ↩
              "\nDoppelte Klassen: " + str(doubles))
85      print("Anzahl Klassen im JSON: " + str(len(list)))
86      print("Nicht im JSON: " + str(len(notinlist)))
87      print(notinlist)
88      print("Alle Klassen: " + str(list))
89      print("Anzahl von Enums: " +str(len(enums)))
90
91
92  def main():
93      oldTime = oldVersion()
94      newTime = newVersion()
95      if oldTime > newTime:
96          print("Neues Format ist um " + ↩
              '{:5.5f}s'.format(oldTime - newTime) + " schneller")
97      elif oldTime < newTime:
98          print("Altes Format ist um " + ↩
              '{:5.5f}s'.format(newTime - oldTime) + " schneller")
99      else:
100         print("Gleich schnell")
101
102
103 main()
```

Listing 6.11: Python Test Scriptt

```
Altes Format:

#Ausgabe gekürzt und mit schönem Format#

{
  'classname': 'Adder',
  'in': 'n', 'out': '1',
  'superclass': 'MisoSystem',
  'methods': [],
  'constructors': []}

Doppelte: 0
Anzahl Klassen im JSON: 176
Nicht im JSON: 0
[]

Alle Klassen: [ "..." ] #Ausgabe gekürzt

0.00667s


Neues Format:

#Ausgabe gekürzt und mit schönem Format#

Adder :
{
    'in':
    {
      'count': 'n',
      'types': ['*']
    },
    'out':
    {
      'count': '1',
      'types': ['*']
    },
    'superclass': 'MisoSystem',
    'classtype': 'system',
    'methods': [],
    'constructors': []
}

Doppelte: 0
Anzahl Klassen im JSON: 407
Nicht im JSON: 1
[""]
Alle Klassen: [ "..." ]   #Ausgabe gekürzt#

Anzahl von Enums: 12
```

```
51
52 0.00157s
53
54
55 Neues Format ist um 0.00510s schneller
```

Listing 6.12: Ausgabe Python Test Script

6.2 Zeitaufwand

Vor der Bachelorarbeit wurde in der Projektarbeit ein Ausblick gegeben, dieser wird in Abbildung 6.1 noch einmal dargestellt. Diese Zeitplanung konnte leider nicht eingehalten werden. Anhand des Zeitstrahls auf Abbildung 6.2 wird die genaue zeitliche Abfolge dargestellt.

Mit der Implementierung in Abbildung 6.3 ist nicht nur die Implementierungsphase (Abbildung 6.2) gemeint, sondern auch ein Teil des annotierens und das Verheiraten. Ebenso wurden auch die “Try and Error” Programmierung in diese Zeit gerechnet.

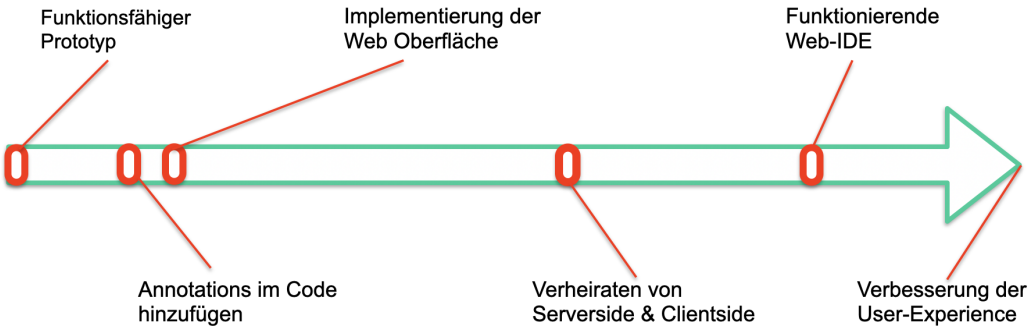

Abbildung 6.1: Ausblick Projektarbeit geplant

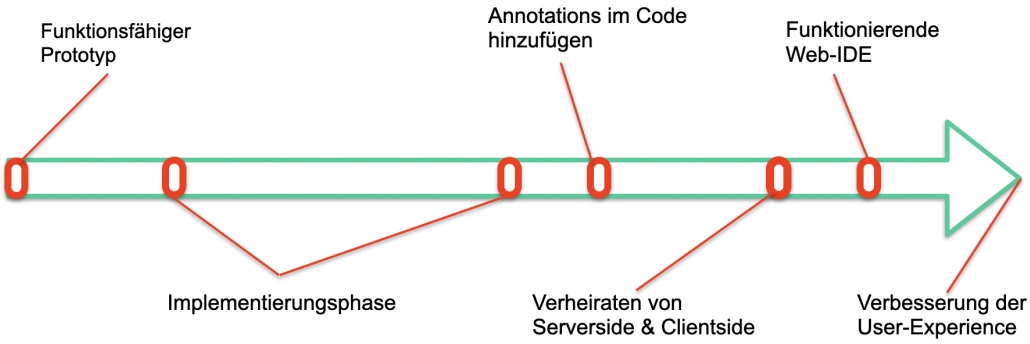

Abbildung 6.2: Ausblick Projektarbeit wirklich

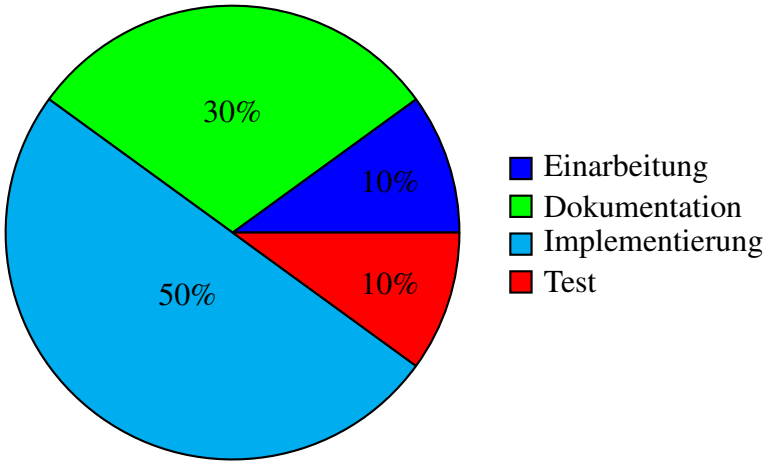

Abbildung 6.3: Themenbezogene Stundenaufschlüsselung für die Abschlussarbeit.

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[23a] Juni 2023. URL: https://jsoneditoronline.org (siehe Seite 15).

[23b] Juni 2023. URL: https://www.visual-paradigm.com (siehe Seite 17).

[23c] Juni 2023. URL: https://wiki.selfhtml.org/wiki/XML (siehe Seite 20).

[23d] Juni 2023. URL: https://www.apple.com/de/macbook-pro/ (siehe Seite 21).

[23e] Juni 2023. URL: https://github.com/apple-open-source/macos (siehe Seite 21).

[23f] Juni 2023. URL: https://www.bsi.bund.de/DE/Themen/Unternehmen-und-Organisationen/Informationen-und-Empfehlungen/Empfehlungen-nach-Gefaehrdungen/Malware/malware_node.html (siehe Seite 29).

[Bau22] Prof. Dr. Andrea Baumann. *Höhere Programmierung*. Apr. 2022 (siehe Seite 29).

[Bet23] Torrey Betts. *Mobile Performance Testing - JSON vs XML Infragistics Blog*. Juni 2023. URL: https://www.infragistics.com/community/blogs/b/torrey-betts/posts/mobile-performance-testing-json-vs-xml (siehe Seite 20).

[Dr 94] Hagen Langer Dr. Sven Neumann. *Eine Einführung in die maschinelle Analyse natürlicher Sprache*. Uni Trier, 1994 (siehe Seiten 25–28).

[Gra21] Prof. Dr.-Ing. Klaus-Peter Graf. *FolienKapitel4(Angreifer, Bedrohungen)*. Jan. 2021 (siehe Seite 29).

[Had11] Christopher Hadnagy. *Die Kunst des Human Hacking Social Engineering - Deutsche Ausgabe*. 2. Auflage. mitp, 2011 (siehe Seite 30).

[Hir23a] Daniel Hirlimann. "Anforderungen und Architektur für die Validierung und Fehlerbehandlung von labAlive Text2App Code". Apr. 2023 (siehe Seite 31).

[Hir23b] Daniel Hirlimann. *LabAlive Entwicklung auf einem MacBook (Eclipse und IntelliJ)*. April 23. Apr. 2023 (siehe Seiten 17, 21).

[Kry18] Veikko Krypczyk. *Handbuch für Softwareentwickler*. 1. Rheinwerk Computing, 2018 (siehe Seite 29).

[Paw] 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 25, 26).

[Paw23] Prof. Dr.-Ing. Dieter R. Pawelczak. *Programmerzeugungssysteme*. Universität der Bundeswehr München - Institut für System Engineering. München, Juni 23 (siehe Seiten 25–28).

[Sim23] Lt. Laurence Arthur Simon. "Entwicklung von Komponenten einer IDE für die labAlive-App myApps Webanwendung". Juni 2023 (siehe Seiten 13, 25–28).

[23g] *Website 500 fastest PCs*. Juni 2023. URL: https://www.top500.org/lists/top500/2023/06/ (siehe Seite 21).

[23h] *Website Annotations*. Juni 2023. URL: https://docs.oracle.com/javase/tutorial/java/annotations/index.html (siehe Seite 17).

[23i] *Website Fugaku Specifications*. Juni 2023. URL: https://www.fujitsu.com/global/about/innovation/fugaku/specifications/ (siehe Seite 21).

[23j] *Website Git*. Juni 2023. URL: https://git-scm.com/ (siehe Seite 15).

[23k] *Website IntelliJ*. Juni 2023. URL: https://www.jetbrains.com/de-de/idea/ (siehe Seite 15).

[23l] *Website Java*. Juni 2023. URL: https://www.java.com/de/ (siehe Seite 17).

[23m] *Website Json*. Juni 2023. URL: https://www.json.org/json-de.html (siehe Seiten 18, 20).

[23n] *Website labAlive*. Juni 2023. URL: https://www.etti.unibw.de/labalive/ (siehe Seite 22).

[23o] *Website MacTEX*. Juni 2023. URL: https://www.tug.org/mactex/ (siehe Seite 16).

[23p] *Website OpenJDK*. Juni 2023. URL: https://openjdk.org/ (siehe Seite 16).

[23q] *Website PyCharm*. Juni 2023. URL: https://www.jetbrains.com/de-de/pycharm/ (siehe Seite 16).

[23r] *Website Python*. Juni 2023. URL: https://www.python.org/ (siehe Seite 21).

[23s] *Website Reflection*. Juni 2023. URL: https://www.oracle.com/technical-resources/articles/java/javareflection.html (siehe Seite 18).

[Zin23] Rafael Zink. *Parser organigram*. Dez. 2023 (siehe Seite 27).

Index

M

O

P

Q

R

S

T

V

X

Z