Masterarbeit

ENTWICKLUNG EINES TESTSYSTEMS DURCH AUTOMATISCHE CODEGENERIERUNG

ausgeführt am

Fachhochschul-Masterstudiengang
Automatisierungstechnik-Wirtschaft

von

Ing. Christoph Assl, BSc
1810322022

betreut und begutachtet von
FH-Prof. Dipl.-Ing. Dieter Lutzmayr

Graz, im Dezember 2019

....
Unterschrift

EHRENWÖRTLICHE ERKLÄRUNG

Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die benutzten Quellen wörtlich zitiert sowie inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

....

Unterschrift

DANKSAGUNG

Ich möchte mich an dieser Stelle bei all jenen bedanken, die mich während der Entstehung dieser Masterarbeit unterstützt haben.

Weiterer Dank gilt meinen Kollegen im Integration Test Development Team, die mir beratend zur Seite gestanden sind. Dem Unternehmen LOGICDATA Electronic & Software Entwicklungs GmbH möchte ich meinen Dank aussprechen, dass mir dieses berufsbegleitende Studium ermöglicht wurde und dadurch diese Masterarbeit entstehen konnte.

Des Weiteren gilt mein Dank meiner Familie und meinen Freunden, die mir während meiner Schul- und Studienzeit Rückhalt gegeben haben. Besonders möchte ich mich bei Alexandra und Dominik für das Korrekturlesen und für die hilfreichen Tipps bedanken.

Außerdem möchte ich mich bei meinen Studienkollegen und Studienkolleginnen für die schöne Zeit an der FH Campus 02 bedanken. In den vergangenen Jahren wurde nicht nur das Wissen erweitert, sondern auch neue Freundschaften geschlossen, wofür ich sehr dankbar bin.

Mein besonderer Dank gilt Herrn FH-Prof. Dipl.-Ing. Dieter Lutzmayr, der meine Masterarbeit betreut und begutachtet hat. Für die konstruktive Kritik und für die hilfreichen Anregungen bei der Erstellung der Arbeit möchte ich mich herzlich bedanken.

KURZFASSUNG

LOGICDATA entwickelt APIs (Application Programming Interfaces) für die marktführenden Smartdevice Plattformen zur Steuerung von mechatronischen Systemen in der Möbelindustrie. Die Funktionalität dieser APIs muss durch Tests mit verschiedenen Eingabeparametern überprüft werden. Änderungen in der Implementierung der APIs während der Entwicklungs- und Testphase führen zu Wartungsaufwand des Testsystems, welcher manuell durchgeführt werden muss. Das führt zu längeren Testdurchlaufzeiten und zu Verzögerungen in der Freigabe-Phase des Projekts.

Ziel dieser Masterarbeit ist es, anfallende Wartungstätigkeiten bei API-Änderungen während der Testphase zu automatisieren, um die Entwicklungszeit des Testsystems zu reduzieren. Zu diesem Zweck soll eine Codegenerierungsmethode ausgewählt werden, mit der Code in verschiedenen Programmiersprachen und Plattformen generiert werden kann. Dafür werden verschiedene Codegenerierungsmethoden hinsichtlich ihrer Eignung für diese Aufgabenstellung miteinander verglichen und eine Wahl getroffen. Aufbauend auf diese Entscheidung wird eine Softwarearchitektur eines Testsystems mit automatischer Codegenerierung entworfen. Nach Festlegung dieser werden der Codegenerator und die Softwarekomponenten des Testsystems entwickelt und die Vorteile hinsichtlich Entwicklungs-, Wartungsaufwand und Testdurchlaufzeit aufgezeigt.

Tests mit den ersten implementierten APIs haben gezeigt, dass der Einsatz der automatischen Codegenerierung zu einer höheren Codeflexibilität und niedrigerem Entwicklungsaufwand führt. Das Testsystem kann für die Verifikation zukünftiger API-Releases eingesetzt werden.

ABSTRACT

LOGICDATA develops APIs (Application Programming Interfaces) for the market leaders of smartdevice platforms to control mechatronic systems in the furniture industry. The functionality of these APIs needs to be approved by performing tests with various input parameters. Changes in the implementation of the APIs during the development and testing phase lead to maintenance of the testing framework, which has to be done manually. This causes longer test duration and delays in the release phase of the project.

The aim of this master’s thesis is to find a code generation method to automize the maintenance tasks in case of an API change during the testing phase in order to reduce the development time of the test framework. The main requirement is that codes in different languages and for different platforms can be generated. Therefore, different code generation methods are compared and the most suitable method, regarding the requirements is determined. Resulting from this decision a new software architecture of a test system with code generation is created. After these investigations, the code generator and the software components of the test system are developed and the benefits regarding maintenance effort and test duration are demonstrated.

Tests with the first implemented APIs have demonstrated that the use of automatic code generation leads to higher code flexibility and development effort. The test framework can be utilized for the verification of future API releases.

INHALTSVERZEICHNIS

1 Einleitung ... 1
1.1 Das Unternehmen LOGICDATA Electronic & Software Entwicklungs GmbH ... 1
1.1.1 LOGIC HOME ... 2
1.1.2 LOGIC OFFICE ... 3
1.2 Ausgangssituation ... 4
1.2.1 Motion@Work App ... 4
1.2.2 SILVERmotion App ... 5
1.3 Zielsetzung ... 6
2 Codegenerierung ... 7
2.1 Codegenerator ... 7
2.2 Vorteile der Codegenerierung ... 8
3 Methoden der Codegenerierung ... 9
3.1 Unterstützende Funktionen der Entwicklungsumgebungen ... 9
3.1.1 Snippets ... 9
3.1.2 Codevervollständigung ... 10
3.2 Template-basierte Codegenerierung ... 11
3.2.1 Template-Systeme ... 11
3.2.2 Das Model-View-Controller-Muster ... 14
3.3 Modell-basierte Codegenerierung ... 17
3.3.1 Das Modell ... 17
3.3.2 Die Modelgetriebene-Software-Entwicklung ... 18
3.3.3 Die Unified Modelling Language ... 21
4 Konzept zur Codegenerierung ... 25
4.1 Anforderungen an das Konzept ... 25
4.2 Auswahl der Codegenerierungsmethode ... 26
5 Template-Engines ... 28
5.1 Vergleich von Template-Engines ... 28
5.1.1 Apache FreeMarker ... 28
5.1.2 Apache Velocity ... 30
5.2 Auswahl der Template-Engine ... 32
6 Architektur des aktuellen Testsystems ... 35
6.1 Systemkomponenten ... 35
6.1.1 API - Application Programming Interfaces ... 35
6.1.2 API-Dokumentation ... 36
6.1.3 Command Sender ... 37
6.1.4 Test-Apps ... 38
6.1.5 LD-Produkt ... 39
6.2 Entwicklungsvorgehen und Funktionsweise ... 40
6.3 Probleme der bestehenden Umsetzung ... 42

7 Architektur des Testsystems mit automatischer Codegenerierung 44
7.1 Systemkomponenten 44
7.1.1 API-Dokumentation 44
7.1.2 API-Libraries 46
7.1.3 Command Sender 48
7.1.3.1 Grafische Benutzeroberfläche 48
7.1.3.2 Schnittstelle zwischen Command Sender und Test-Apps 49
7.1.4 Test-Apps 51
7.2 Entwicklungsvorgehen und Funktionsweise 52
8 Umsetzung 54
8.1 FreeMarker Software 54
8.1.1 Java-Software als Generator 54
8.1.2 Templates 56
8.2 Protocol Buffer 58
8.3 Test-App 61
8.3.1 Grafische Benutzeroberfläche 61
8.3.2 Wrapper 62
8.4 Command Sender 64
8.4.1 API-Facade 64
8.4.1.1 Verbindungsaufbau und Initialisierung 65
8.4.1.2 Übermittlung von JSON-RPC-Requests 65
8.4.1.3 Empfangen von JSON-RPC-Responses 67
8.4.2 Grafische Benutzeroberfläche des Command Senders 69
8.4.2.1 Einlesen der JSON-Files und Initialisierung 69
8.4.2.2 Implementierung des Layouts der grafischen Benutzeroberfläche 71
8.4.2.3 Platzieren der Steuerelemente auf der Benutzeroberfläche 72
8.4.2.4 Aufrufen der entsprechenden APIs 74
8.4.2.5 Darstellung der grafischen Benutzeroberfläche 76
9 Ergebnisse und Ausblick 78
Literaturverzeichnis 80
Abbildungsverzeichnis 84
Tabellenverzeichnis 87
Abkürzungsverzeichnis 88

1 EINLEITUNG

1.1 Das Unternehmen LOGICDATA Electronic & Software Entwicklungs GmbH

Diese Masterarbeit wird für das Integrationtest Development Team der LOGICDATA Electronic & Software Entwicklungs GmbH mit Sitz in Deutschlandsberg umgesetzt.

LOGICDATA entwickelt innovative mechanische, elektronische und integrierte Software-Lösungen für verstellbare Möbel im Büro- und Heimbereich. Seit über 20 Jahren orientiert sich das Produktportfolio am neuesten Stand der Technik, besticht durch Anwenderfreundlichkeit, Funktionalität und Design. Der hohe Grad an Flexibilität und der stetige Wille, die Erwartungen der Kunden und Kundinnen zu übertreffen, machen LOGICDATA zum führenden globalen Anbieter in der Branche für elektronisch verstellbare Möbel. LOGICDATA ist in zwei Geschäftsbereiche unterteilt – LOGIC OFFICE und LOGIC HOME. In beiden Bereichen bietet LOGICDATA ihren weltweiten Partnern und Partnerinnen marktführende technologische Lösungen in Premiumqualität an. LOGICDATA ist Arbeitgeber für über 330 Mitarbeiter und Mitarbeiterinnen im Hauptsitz Deutschlandsberg und in den Niederlassungen in Grand Rapids (USA), Maribor (Slowenien), Zhuhai (China) und Zagreb (Kroatien).[1]

Abbildung 1: LOGICDATA Logo. Quelle: LOGICDATA Electronic & Software Entwicklung GmbH 1 (2019), Online-Quelle [17.08.2019].

[1] Vgl. LOGICDATA Electronic & Software Entwicklungs GmbH 2 (2019), S.44f.

1.1.1 LOGIC HOME

Der Geschäftsbereich LOGIC HOME umfasst Produkte für Heim- und Hausmöbel, die Ergonomie und einzigartige Bequemlichkeit verbinden. Diese Produktinnovationen sorgen für mehr Komfort und Individualität zuhause. Im Bereich LOGIC HOME bietet LOGICDATA Produkte folgender Produktsparten an:[2]

- Steuerungseinheiten
- Handschalter
- Konnektivität und Apps
- Zubehör
- Silver-Series (Bett-Systeme)
- Elematic Module (Mechatronik-Einheiten zur Verstellung von Betten)

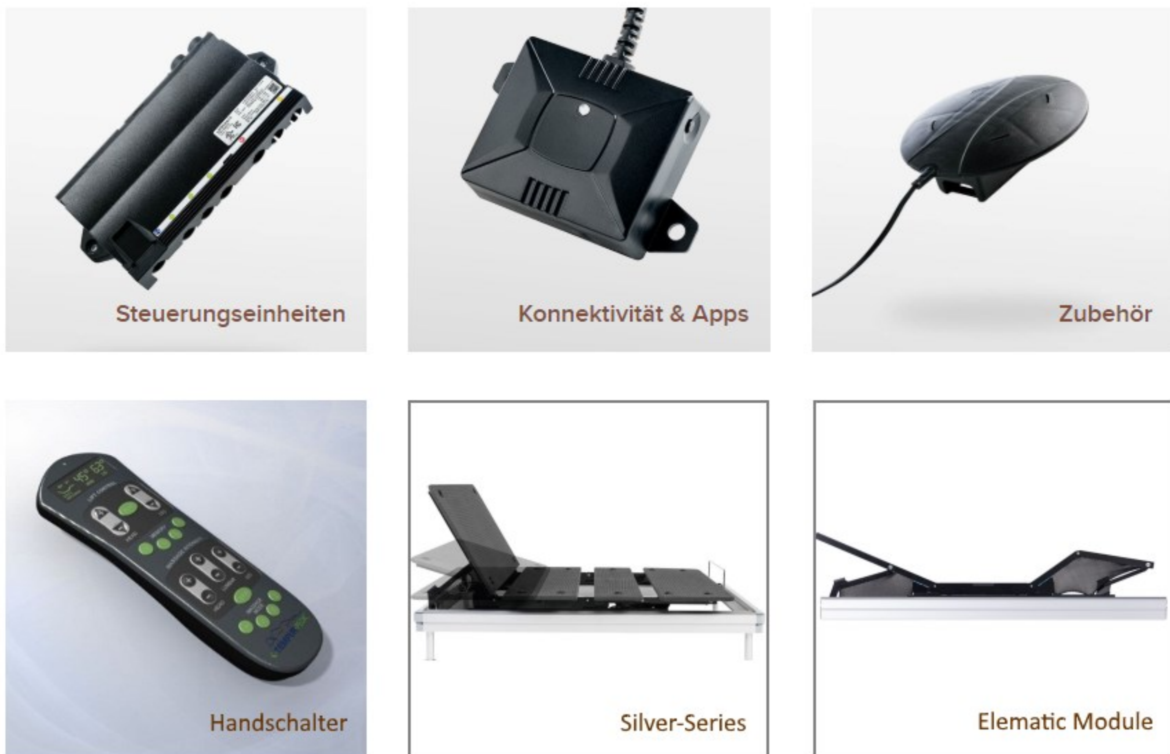

Abbildung 2: Produktübersicht LOGIC HOME, Quelle: Eigene Darstellung.

[2] Vgl. LOGICDATA Elektronic & Software Entwicklungs GmbH 3 (2019), Online-Quelle [17.08.2019].

1.1.2 LOGIC OFFICE

Im Bereich LOGIC OFFICE wird eine große Anzahl intelligenter mechatronischer Lösungen für höhenverstellbarer Büromöbel entwickelt. Der Fokus liegt dabei auf hoher Qualität und Ausführung der Produkte. In den letzten 20 Jahren hat LOGICDATA viel Erfahrung in der Entwicklung von Lösungen für ergonomische Arbeitsplätze gesammelt, um die Erwartungen des Marktes an verstellbare Büromöbel neu zu definieren. Dabei steht Benutzerfreundlichkeit immer im Mittelpunkt des LOGIC OFFICE-Konzepts. LOGICDATA arbeitet eng mit den renommiertesten Möbelherstellern der Welt zusammen, um mit marktführenden Produkten den modernen Arbeitsplatz zu optimal mit zu gestalten. Im Geschäftsbereich LOGIC OFFICE bietet LOGICDATA Produkte folgender Produktkategorien an:[3]

- Steuerungseinheiten
- Handschalter
- Antriebe
- Power-Units
- Accessoires
- Konnektivität und Apps

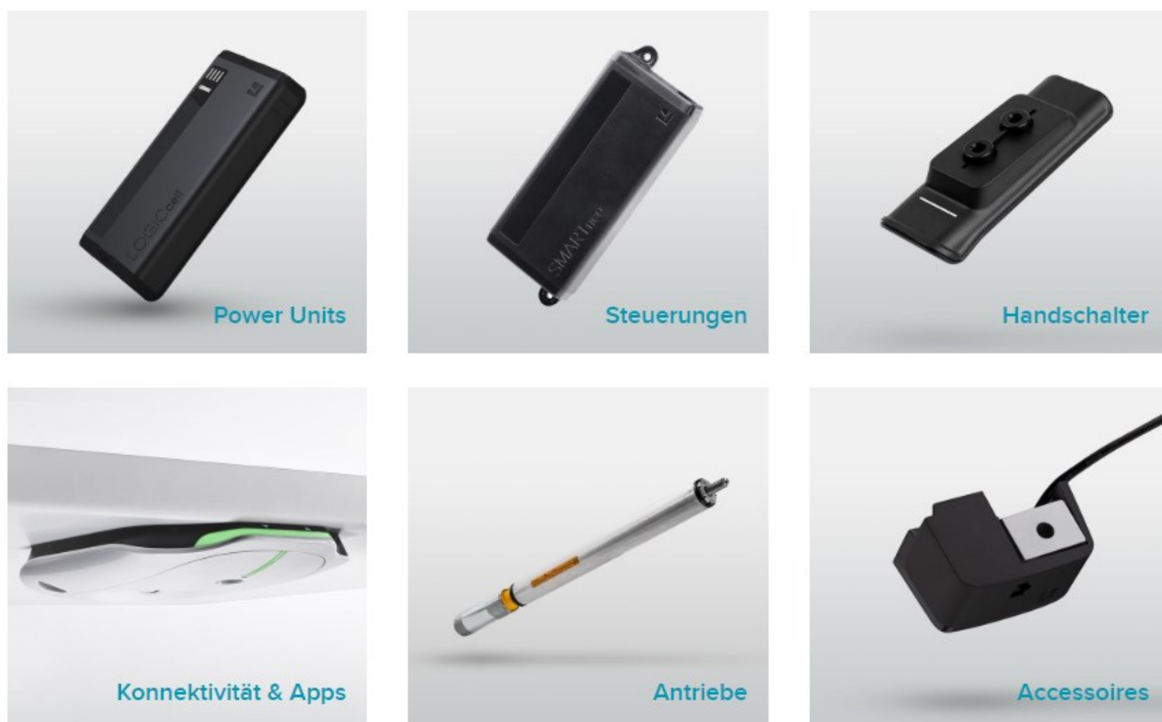

Abbildung 3: Produktübersicht LOGIC OFFICE. Quelle: LOGICDATA Elektronic & Software Entwicklungs GmbH 4 (2019), Online-Quelle [17.08.2019].

[3] Vgl. LOGICDATA Elektronic & Software Entwicklungs GmbH 4 (2019), Online-Quelle [17.08.2019].

1.2 Ausgangssituation

Die Firma LOGICDATA entwickelt APIs für die Plattformen Windows, Android und iOS. Die APIs dienen als Schnittstellen zwischen Smartdevices und den Funktionen der Produkte von LOGICDATA. Die APIs ermöglichen dem User die Bedienung dieser über das Smartdevice. Die API-Libraries können an externe Entwickler und Entwicklerinnen beziehungsweise an Kunden und Kundinnen weitergegeben werden und in Apps zur Nutzung von LOGICDATA Produkten eingesetzt werden. Um die Funktionalität der APIs zu gewährleisten, werden manuelle und automatisierte Tests durchgeführt. Zusätzlich zu den APIs wird eine Doxygen-Dokumentation, die diese genauer beschreibt, vom Entwicklungsteam zur Verfügung gestellt. Mit diesen Informationen wird eine Test-Software mit grafischer Benutzeroberfläche erstellt, die es dem Tester oder der Testerin ermöglicht, die Eingabeparameter für die zu testenden API einzugeben sowie die Antworten auszulesen. Diese Software kommuniziert via JSON-RPC-Kommandos mit Test-Apps, die auf der jeweiligen Plattform laufen und welche die APIs der jeweiligen Plattform halten.

Konnektivität und Apps

In dieser Masterarbeit liegt der Fokus auf den Bereichen Konnektivität und Apps, da durch die Codegenerierung der Test- beziehungsweise Freigabeprozess, der von der Firma LOGICDATA entwickelten Application Programming Interfaces, kurz APIs, verbessert werden soll. Die APIs ermöglichen der App beziehungsweise dem User den Zugriff auf die Funktionalitäten des jeweiligen LOGICDATA-Produkts. Momentan stehen Apps in den Geschäftsbereichen LOGIC OFFICE und LOGIC HOME zur Verfügung.

1.2.1 Motion@Work App

Die Motion@Work App (siehe Abbildung 4) von LOGICDATA erhöht die Flexibilität für den höhenverstellbaren Arbeitsbereich. Nachdem eine Verbindung zwischen dem Smartdevice und dem LOGICDATA-Produkt hergestellt wurde, kann der Schreibtisch individuell auf die Bedürfnisse des Benutzers oder der Benutzerin eingestellt werden. Neben der Anpassung der Tischhöhe können auch Lieblingspositionen gespeichert und die Maßeinheit von Zentimeter in Zoll direkt in der App verändert werden. Darüber hinaus ist es möglich, umfassende Statistiken zum Sitz-Steh-Verhalten auszugeben, persönliche Ziele zu definieren und Erinnerungen einzustellen. Die App dient daher als perfekte Unterstützung zur Verbesserung der Ergonomie am Arbeitsplatz. Sie ist für Android sowie iOS verfügbar und kann kundenspezifisch angepasst werden.[4]

[4] Vgl. LOGICDATA Electronic & Software Entwicklungs GmbH 5 (2019), Online-Quelle [19.08.2019].

Abbildung 4: Motion@Work App, Quelle: LOGICDATA Electronic & Software Entwicklungs GmbH 5 (2019), Online-Quelle [19.08.2019].

1.2.2 SILVERmotion App

Die SILVERmotion App (siehe Abbildung 5) des Geschäftsbereichs LOGIC HOME, ist eine gute Alternative zum herkömmlichen Handschalter. Der User bekommt dadurch die Möglichkeit, den Bettrahmen der Silver-Series mittels Smartdevice zu bedienen. Dazu zählen das Verstellen der einzelnen Zonen des Bettes, das Anfahren und Speichern von Speicherpositionen sowie die Aktivierung und Anpassung von Licht und Massagefunktionen. Wie auch die Motion@Work App steht die SILVERmotion App für iOS und Android zur Verfügung.[5]

[5] Vgl. LOGICDATA Electronic & Software Entwicklungs GmbH 6 (2019), Online-Quelle [19.08.2019].

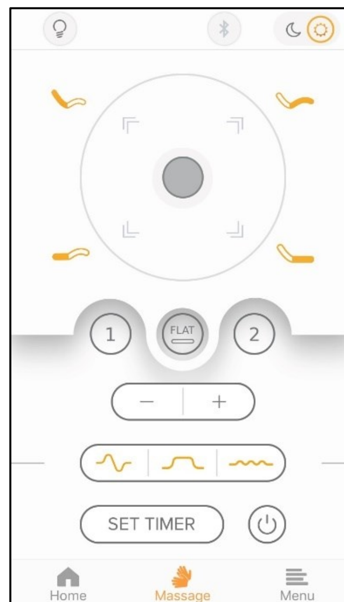

Abbildung 5: SILVERmotion App, Quelle: Eigene Darstellung.

1.3 Zielsetzung

Für die Erstellung der Test-Software, mit der die Funktionalität der APIs überprüft werden soll, müssen vorab alle API-Parameter definiert und die Dokumentation dieser vollständig sein. Zusätzlich muss bei einer Änderung der APIs die Test-Software jedes Mal angepasst werden, was vor allem im Freigabeprozess zu Verzögerungen führen kann. Aus diesem Grund soll im Zuge dieser Masterarbeit geprüft werden, inwieweit die Software inklusive grafischer Benutzeroberfläche automatisch generiert werden kann. Dafür werden vom Entwicklungsteam neben den APIs eine API-Dokumentation in einem kompakten Datenformat zur Verfügung gestellt. Mit diesen Informationen soll eine C#-Library, die von einer Test-Software genutzt wird, erstellt werden. Hierfür sollen verschiedene Methoden der Codegenerierung untersucht und auf ihre Tauglichkeit beurteilt werden. Auf der Grundlage des resultierenden Konzeptes soll dieses Softwarevorhaben in die Praxis umgesetzt werden.

Ziel dieser Arbeit ist, die Entwicklung der Test-Software mit den vom Entwicklungsteam zur Verfügung gestellten Daten zu automatisieren. Diese Test-Software soll dem Tester oder der Testerin ermöglichen, die Eingabeparameter der zu testenden API ein- und die Antworten dieser auszugeben.

Damit soll der Entwicklungs- bzw. Wartungsaufwand der Software und die zu Grunde liegende Komplexität dieser verringert werden.

2 CODEGENERIERUNG

2.1 Codegenerator

Bei einem Codegenerator handelt es sich um ein Programm, das Code basierend auf einer Eingangsspezifikation generieren kann. Diese Programme werden hauptsächlich eingesetzt, um wiederholende Tätigkeiten zu automatisieren, sowie zur Instanziierung von HTML-Webseiten im Internet. Bei der Codegenerierung findet eine Übertragung von Eingabedaten zu Ausgabedaten statt (siehe Abbildung 6). Die Eingabedaten sind Teil einer Sprache mit eigener Syntax und Semantik, welche unabhängig vom Codegenerator sind. Der Codegenerator übersetzt die Eingabedaten in eine andere Form, in den meisten Fällen eine Ausprägung mit niedrigerer Abstraktionsstufe. Die Ausgabedaten können dabei verschiedenste Gestalten annehmen; angefangen bei Maschinencode, über Compiler bis hin zu Code verschiedener Programmiersprachen bei modell-basierter Entwicklung oder rechnergestützten Softwareentwicklungen.

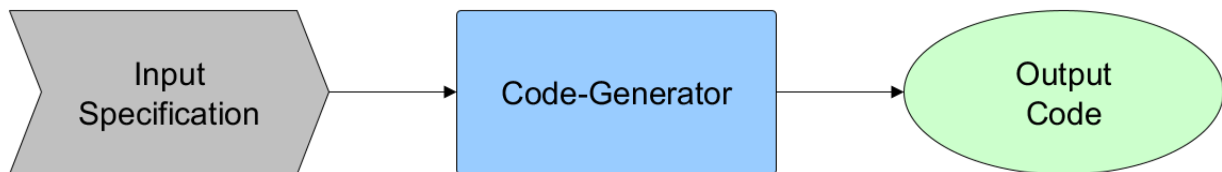

Abbildung 6: Schematische Darstellung eines Codegenerators, Quelle: Arnoldus/Brand/Serebnik/Brunekreef (2012), S.2. (leicht modifiziert).

Die Entwicklung eines Codegenerators ist eine umfangreiche Aufgabe. Es muss zuerst die Eingangssprache definiert werden, welche die nötigen Begriffe und die richtige Abstraktionsstufe für die Problemstellung besitzt. Die Begriffsdefinition erfordert eine umfassende Kenntnis des zu lösenden Problems. Eine gut definierte Sprache ist eine der wichtigsten Erfolgsfaktoren eines Codegenerators. In weiterer Folge sind das Verstehen und Implementieren des Codegenerators in vielen Fällen schwierig, da der Code aus verschiedenen Teilen besteht, die in unterschiedlichen Stufen ausgeführt werden. Der Codegenerator besteht aus Metacode, der während der Generierung ausgeführt wird, und Object-Code, dessen Teile als Building Blocks für den Ausgangscode verwendet werden. Auch das Finden von Bugs ohne zusätzlichen Tool-Support ist kompliziert, da Fehler im Metacode erst während der Kompilierung oder Generierung, und Fehler im Object-Code sogar erst danach gefunden werden können. Aus diesem Grund ist das Debuggen sehr zeitaufwändig, da bei jeder Änderung des Generators der Code neu kompiliert, generiert und getestet werden muss.[6]

[6] Vgl. Arnoldus/Brand/Serebnik/Brunekreef (2012), S.1 f.

2.2 Vorteile der Codegenerierung

Codegenerierungs-Techniken bringen in erster Linie Vorteile für die Software-Entwicklungsabteilung, jedoch profitiert auch das Management in Form von gesteigerter Produktivität und Software-Qualität davon. Im Folgenden wird auf die wichtigsten Vorteile laut Herrington eingegangen:[7]

- Qualität
  In der Regel führen größere Mengen von handgeschriebenem Code zu inkonsistenter Code-Qualität, da sich Entwickler und Entwicklerinnen ständig weiterentwickeln und neue Lösungsansätze finden. Durch den Einsatz von Templates zur Codegenerierung wird eine konsistente Codebasis generiert. Bei Änderungen des Templates und nach der Ausführung des Codegenerators werden die Fehlerbehebungen sowie Verbesserungen sofort und konsequent auf den gesamten Code angewandt.

- Code-Konsistenz
  Der vom Codegenerator generierte Code ist konsistent hinsichtlich seines Designs, der Schnittstellen und der Variablenbezeichnung. Daraus resultiert eine leicht verständliche und anwendungsfreundliche Benutzeroberfläche.

- Zentrale Wissensbasis
  Der Generator bildet eine zentrale Wissensbasis. Die Codegenerierungsarchitektur bewirkt, dass jede Änderung an der zentralen Stelle nach der neuerlichen Generierung konsequent an allen Stellen des generierten Codes wirksam wird.

- Zeitersparnis
  Projekte, bei denen Codegenerierung eingesetzt wird, unterscheiden sich maßgeblich von Projekten, bei denen ausschließlich manuell codiert wird. Oftmals fehlt es bei der manuellen Codierung an zeitlichen Ressourcen, um Analysen zur optimalen Nutzung des Systems durchzuführen. Werden falsche Annahmen getroffen, muss mit diesen Annahmen weitergearbeitet oder große Codestücke verworfen werden. Bei der automatischen Codegenerierung müssen in diesem Fall zum Beispiel nur die Templates verändert und der Code neu generiert werden. Darüber hinaus übernimmt der Generator die weiteren Tätigkeiten, die wiederum Zeitersparnisse bewirken und mehr Ressourcen für Design- und Prototypentests ermöglichen.

- Agile Entwicklung
  Ein Schlüssel-Merkmal von generiertem Code ist seine Formbarkeit. Das bedeutet, dass die Software langfristig leicht verändert und aktualisiert werden kann.

- Hohe Moral
  Lange Softwareentwicklungs-Projekte können mühsam für die Entwickler und Entwicklerinnen sein. Durch die Codegenerierung kann die Projektzeit verringert und der Fokus auf die interessanten Teile der Programmierung gelegt werden. Durch die hohe Qualität des generierten Codes steigt auch das Vertrauen und der Stolz der Beteiligten auf die Codebasis.

[7] Vgl. Herrington (2003), S. 15 ff.

3 METHODEN DER CODEGENERIERUNG

3.1 Unterstützende Funktionen der Entwicklungsumgebungen

Dieses Kapitel befasst sich mit den von den Entwicklungsumgebungen zur Verfügung gestellten Funktionen zur Generierung von Code. Bei diesen Features spricht man nicht von klassischer Codegenerierung, jedoch kann dadurch die Arbeit für den Entwickler und der Entwicklerin erleichtert und Entwicklungszeit eingespart werden. Im Folgenden wird auf die unterstützenden Funktionen der Entwicklungsumgebungen Visual Studio und Eclipse eingegangen.

3.1.1 Snippets

Unter Snippets versteht man Codeausschnitte mit wiederverwendbarem Code. Diese können über Befehle im Kontextmenü oder Tastenkombinationen eingefügt werden. Üblicherweise sind darin häufig genutzte Codeblöcke wie zum Beispiel „if-else" oder „try-finally"-Blöcke enthalten, es können jedoch auch ganze Klassen oder Methoden eingefügt werden. Visual Studio unterstützt dabei zwei verschiedene Arten von Snippets. Diese sind zum einen Erweiterungsausschnitte, die an der aktuellen Cursorposition eingefügt werden und damit eine Ausschnitts-Verknüpfung ersetzen und zum anderen umschließende Ausschnitte, die um einen markierten Codeblock herum eingefügt werden können.[8]

Beide Entwicklungsumgebungen - Visual Studio und Eclipse - verfügen über eine Vielzahl an vorinstallierten Code-Snippets. Diese Bibliotheken können auch durch eigene Code-Snippets erweitert werden, was Copy-Paste-Fehlern vorbeugen kann und auch zeitliche Vorteile mit sich bringt.

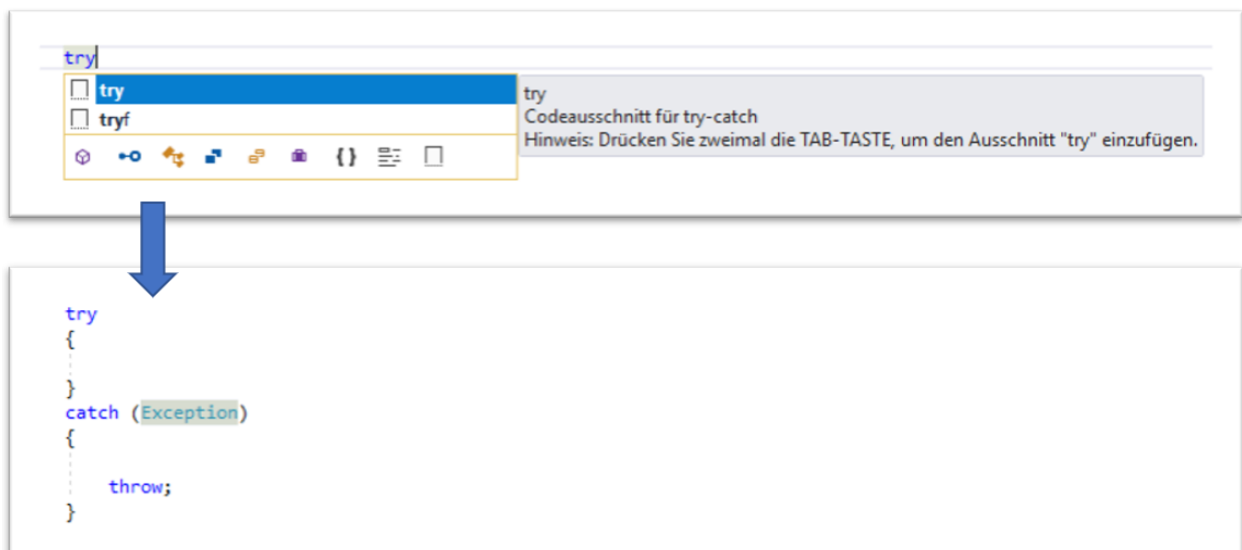

Abbildung 7: Beispiel eines Code Snippets in Visual Studio. Quelle: Eigene Darstellung.

[8] Vgl. Microsoft Corporation 1 (2016), Online-Quelle [10.08.2019].

3.1.2 Codevervollständigung

Auch dieses Feature steht in beiden Entwicklungsumgebungen mit unterschiedlichen Bezeichnungen zur Verfügung. In Visual Studio wird dieses Feature IntelliSense genannt. Es umfasst das Auflisten von Parameterinformationen, Quickinfo, Membern und die Wortvervollständigung. Durch diese Features ist es möglich, mit wenigen Tastaturanschlägen mehr über den verwendeten Code zu erfahren sowie Methoden und Eigenschaften zu Aufrufen hinzuzufügen. Die Eigenschaften von IntelliSense sind abhängig von der verwendeten Programmiersprache.[9]

In Eclipse wird dieses Feature als Content Assist bezeichnet. Es unterstützt User bei der Vervollständigung des Codes. Die Cursorposition im Code liefert dem Content Assist den notwendigen Kontext, um die möglichen Vervollständigungs-Varianten vorschlagen zu können. Der Content Assist steht für die meisten Eclipse-Text-Editioren zur Verfügung.[10]

Durch die Codevervollständigung können Tippfehler vermieden und der Programmierfluss verbessert werden.

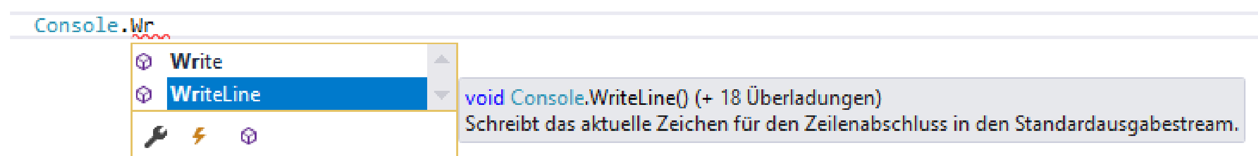

Abbildung 8: Codevervollständigung mit IntelliSense in Visual Studio. Quelle: Eigene Darstellung.

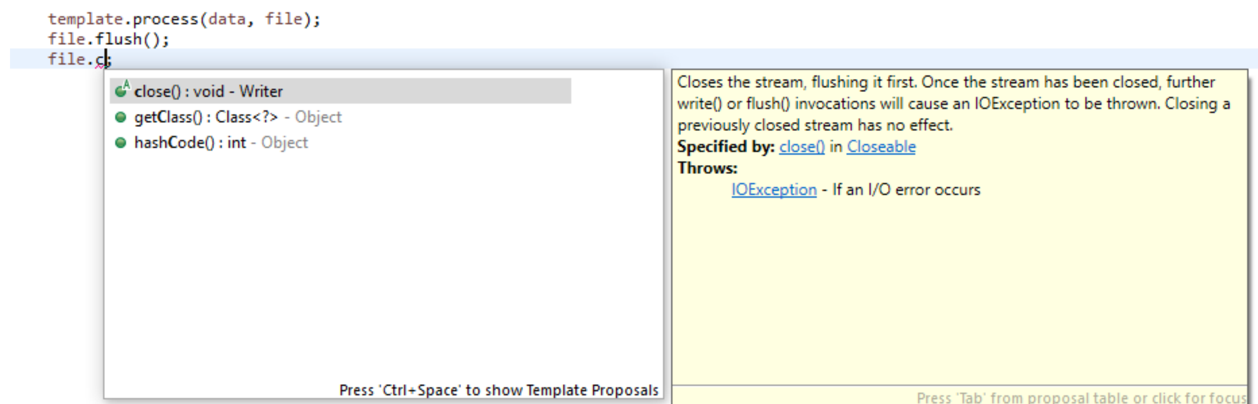

Abbildung 9: Codevervollständigung mit Content Assist in Eclipse. Quelle: Eigene Darstellung.

[9] Vgl. Microsoft Corporation 2 (2018), Online-Quelle [10.08.2019].

[10] Vgl. Eclipse Foundation 1 (2019), Online-Quelle [10.08.2019].

3.2 Template-basierte Codegenerierung

Ein Template-System ist vermutlich die meist verwendete Variante der Codegenerierung. Im Zentrum jenes steht die Template-Engine. Dabei handelt es sich im Grunde genommen um einen Compiler, der die Template-Sprache versteht. Das Template beinhaltet spezielle Notationen, die von der Template-Engine interpretiert werden können. Im einfachsten Fall muss die Template-Engine die Notationen nur durch die richtigen Daten zur Laufzeit ersetzen.[11]

3.2.1 Template-Systeme

Template-Systeme werden häufig in Web-Anwendungen eingesetzt. Aus diesem Grund gibt es auch eine große Zahl an Template-Engines am Markt, die für die Instanziierung von HTML-Seiten entwickelt wurden. Neben HTML können Templates für jegliche Art von unstrukturiertem Text, wie E-Mails oder Quellcode, eingesetzt werden. Ein Template-System (siehe Abbildung 10) setzt sich grundsätzlich aus vier Komponenten zusammen:[12]

- Eingabedaten
- Template
- Template-Engine
- Generierter Code

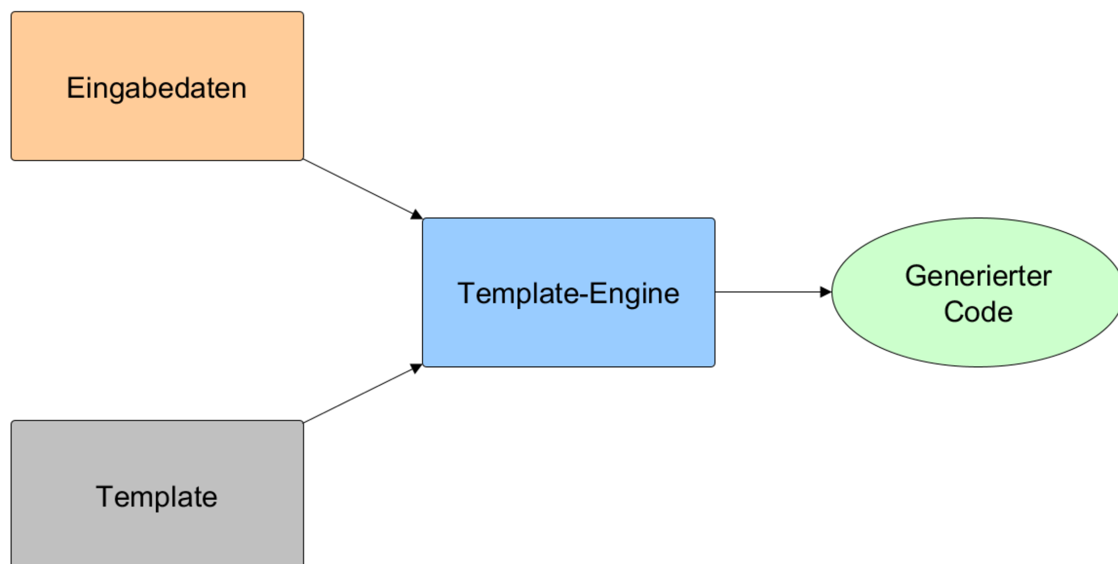

Abbildung 10: Schematische Darstellung eines Template-Systems; Quelle: Arnoldus/Brand/Serebnik/Brunekreef (2012), S.14. (leicht modifiziert).

[11] Vgl. Tomassetti (2018), Online-Quelle [20.09.2019].

[12] Vgl. Arnoldus/Brand/Serebnik/Brunekreef (2012), S. 13f.

Laut Parr unterscheidet sich ein Ausgabe-Template von einem Programm, welches eine Ausgabe erzeugt, dadurch, dass das Template nur ein Modell ist, jedoch die Ausführung des Programms zu einer Ausgabe führt. Ein Template ist ein Ausgabedokument mit integrierten Funktionen, welche die Template-Engine während der Ausführung auswertet.[13]

Bezugnehmend auf diese Definition ist ein Template ein Textdokument, welches Platzhalter beinhalten kann. Bei einem Platzhalter handelt es sich um eine Einheit, die auf ein Textstück hinweist, für das unterschiedliche Ausdrücke eingesetzt werden können. Zusätzlich beinhaltet der Platzhalter eine Funktion oder einen Ausdruck, welcher der Template-Engine erklärt wie und mit welchem Element dieser ersetzt werden soll. Der Text bildet den statischen Teil des Templates und wird direkt in das Ausgabedokument kopiert. Der Platzhalter bildet den unvollständigen Teil des Textes. Bei den Platzhaltern der Templates handelt es sich um Metacode, welche als Sätze einer Metasprache betrachtet werden können. Diese Metasprache entsteht zum Zeitpunkt, an dem die Platzhalter durch Text oder Code ersetzt werden. Das Verarbeiten der Templates wird von Template-Engines übernommen. Dabei handelt es sich um eine Anwendung, die das Template nach Platzhaltern durchsucht, die gewünschten Aktionen durchführt und die Platzhalter durch Text ersetzt. Damit wird der Ausgabecode, beziehungsweise der Ausgabetext, erstellt. Template und Template-Engine bilden in ihrer Kombination den eigentlichen Codegenerator. Die Template-Engine bildet den generischen Teil des Codegenerators, während das Template den anwendungsspezifischen Teil darstellt. Im generischen Teil werden unter anderem das Verarbeiten der Ein- und Ausgabedaten, die Dateierstellung und weitere administrative Aufgaben durchgeführt. Um die Verständlichkeit des Codegenerators zu erhöhen, sollte der Boilerplate-Code von den Ausgabe-Code-Patterns getrennt werden.[14]

Boilerplates werden in der Programmierung Codestücke genannt, die mehrfach, mit nur unwesentlich modifiziertem Inhalt, wiederverwendet werden können. Dieser Code ist oftmals Open-Source-Code, der für den Masseneinsatz geschrieben wurde und hohe Zuverlässigkeit aufweist. Bei Boilerplate-Code sind nur kleine Änderungen notwendig. So wird die Arbeit für Entwickler und der Entwicklerinnen erleichtert. Als Beispiel können Header für Webseiten genannt werden.[15]

In der folgenden Abbildung 11 ist auf der linken Seite ein Freemarker-Template abgebildet. Die durch ${ } gekennzeichneten Platzhalter werden durch definierten Text von der Template-Engine ersetzt. Das Ergebnis ist der generierte Text auf der rechten Seite.

---

[13] Vgl. Parr (2004), S. 226.

[14] Vgl. Arnoldus/Brand/Serebnik/Brunekreef (2012), S. 13f.

[15] Vgl. Techopedia Inc. (o.J.), Online-Quelle [26.08.2019].

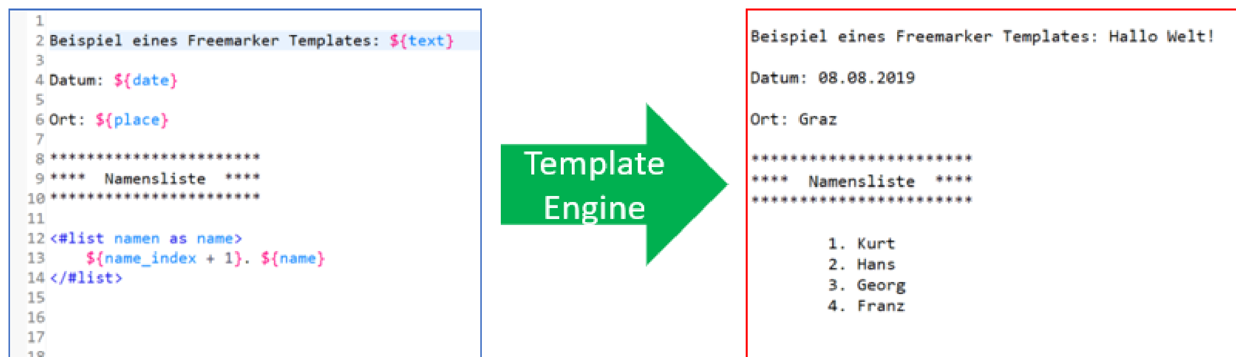

Abbildung 11: Beispiel eines Templates und des resultierenden Ausgabetexts. Quelle: Eigene Darstellung.

Syntax und Semantik

Das Schreiben von Templates und Codegeneratoren ist im Allgemeinen eine komplexe und fehleranfällige Aufgabe. Diese Komplexität ergibt sich aus dem Mischen mehrerer Sprachen in einem Template, die auf verschiedenen Ebenen ausgeführt werden und der Unvollständigkeit des Object-Codes. Die manuelle Überprüfung dieses Codes ist schwierig, da kein unvollständiger Code ausgeführt werden kann. Text-basierte Template-Evaluatoren sind nicht in der Lage, das Template vollständig auf die syntaktische Richtigkeit zu überprüfen. Es kann nur die Syntax der Metasprache kontrolliert werden, jedoch nicht die des Object-Codes, da dieser nur als String ohne Struktur betrachtet wird. Das kann zu unentdeckten Syntaxfehlern, wie zum Beispiel vergessenen Strichpunkten, führen, die die Funktion des generierten Codes beeinflussen würden. Die Aufgabe des Compilers ist es diese aufzuspüren und eine Warnung auszugeben.[16]

Zusätzlich erkennt der Compiler semantische Fehler, beispielsweise in Form von fehlenden Variablendeklarationen. Der Compiler kann jedoch keine Logikfehler im Template entdecken. Durch sich häufig ändernde Variablen in einem Template-System ergeben sich eine Vielzahl an Fällen, die bei der Implementierung bedacht werden müssen, da jede dieser Änderungen Auswirkungen auf den generierten Code mit sich zieht.

[16] Vgl. Arnoldus/Brand/Serebnik/Brunekreef (2012), S93.

3.2.2 Das Model-View-Controller-Muster

Die in dieser Masterarbeit untersuchten Template-Engines beziehen sich auf das Model-View-Controller-Muster (siehe Abbildung 12), kurz auch MVC genannt. Dabei handelt es sich um ein Architekturmuster, beziehungsweise Designmuster in der objektorientierten Programmierung, das einen flexiblen Softwareentwurf bietet, etwaige Änderungen oder Erweiterungen zu einem späteren Zeitpunkt erleichtert und die Wiederverwendbarkeit einzelner Codeteile ermöglicht. Das MVC-Muster wird vor allem bei Software mit grafischer Benutzeroberfläche, wie zum Beispiel GUI-Programmen und Webapplikationen, eingesetzt. Anwendungen, die dem MVC-Prinzip folgen, bestehen aus drei austauschbaren Komponenten.[17]

Die Aufgaben der drei Hauptkomponenten sind wie folgt definiert:[18]

- Model (Modell)
  Das Modell repräsentiert die logische Struktur von Daten in einer Anwendung und die High-Level-Klasse, die damit verbunden ist. Das Objektmodell selbst beinhaltet keine Informationen über die Benutzeroberfläche.

- View (Präsentation)
  Die Präsentation beinhaltet alle Klassen, die zur Darstellung der Benutzeroberfläche sowie der Elemente der Benutzeroberfläche benötigt werden. Die Aufgabe der Präsentation ist es, die benötigten Daten aus dem Modell darzustellen und die Benutzerinteraktionen entgegenzunehmen.

- Controller (Steuerung)
  Die Steuerung verbindet das Modell und die Präsentation und repräsentiert die Klassen. Sie verwaltet eine oder mehrere Präsentationen, nimmt die Benutzerinteraktionen entgegen, wertet diese aus und reagiert entsprechend. Die Steuerung wird auch zur Kommunikation zwischen Modell und Präsentation verwendet.

[17] Vgl. Begerow (o.J.), Online-Quelle [10.08.2019].

[18] Vgl. Rouse (2016), Online-Quelle [10.08.2019].

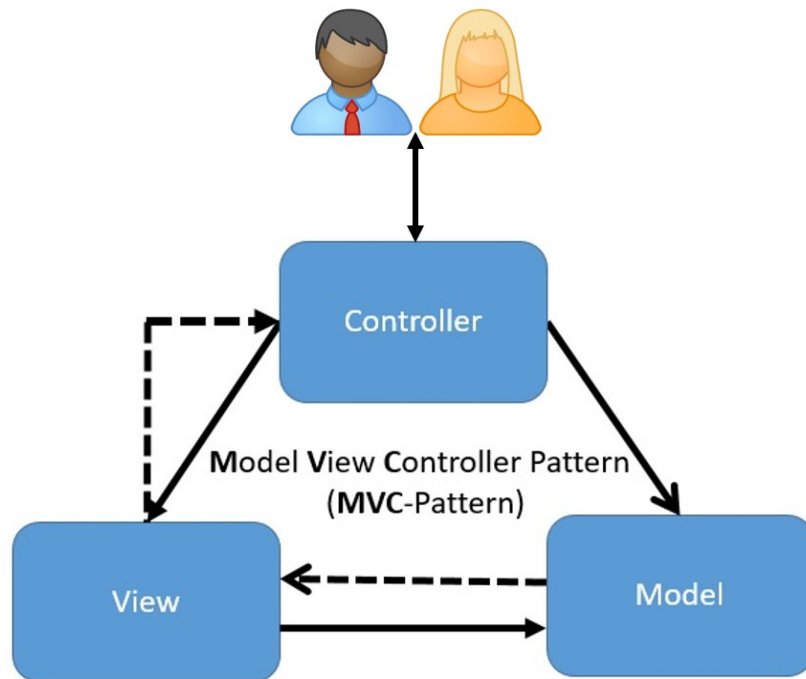

Abbildung 12: Model-View-Controller-Muster, Quelle: (leicht modifiziert) Begerow (o.J.), Online-Quelle [10.08.2019].

Das Model-View-Presenter-Muster

Als Weiterentwicklung des MVC-Musters kann in diesem Zuge das Model-View-Presenter-Muster, kurz MVP-Muster, genannt werden. Dabei wird der Controller durch einen Presenter ersetzt. Die Aufgabe dessen ist es, alle User Interface Events im Auftrag der View zu behandeln. Der Presenter empfängt User-Eingaben über die View, verarbeitet die Daten des Benutzers oder der Benutzerin und gibt sie an die View zurück. Die View und der Presenter sind im Gegensatz zu View und Controller beim MVC-Muster vollkommen voneinander getrennt und kommunizieren über ein Interface miteinander. Des weiteren ist der Presenter, anders als der Controller, nicht für die Behandlung des eingehenden Anforderungsverkehrs zuständig.[19]

[19] Vgl. Rishab Software (2016), Online-Quelle [18.09.2019].

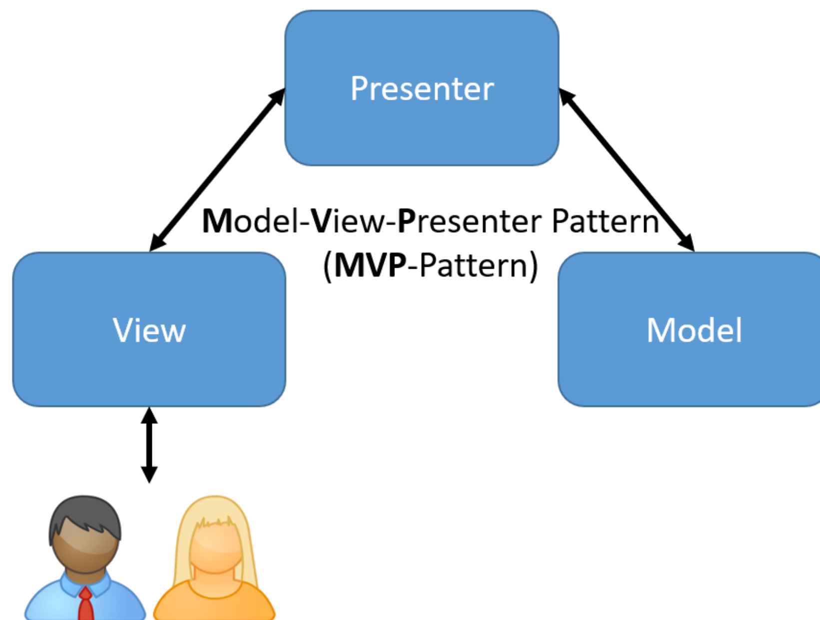

Abbildung 13: Model-View-Presenter-Muster, Quelle: In Anlehnung an Rishab Software (2016), Online-Quelle [18.09.2019].

Das Model-View-ViewModel-Muster

Eine weitere Variante des MVC-Musters ist das Model-View-ViewModel-Muster, kurz MVVM-Muster. Dieses unterstützt die bidirektionale Datenbindung zwischen View und ViewModel. Dies ermöglicht die automatische Übermittlung von Änderungen vom ViewModel zur View. Grundsätzlich übernimmt das ViewModel die Rolle des Beobachters, um Änderungen in der View dem Modell mitzuteilen. Des weiteren ist das ViewModel für die Darstellung von Methoden, Befehlen und anderen Funktionen verantwortlich, die bei der Aufrechterhaltung des View-Zustandes, bei der Manipulation des Modells als Resultat von Ergebnissen in der View und der Auslösung von Events in der View selbst unterstützen.[20]

[20] Vgl. Rishab Software (2016), Online-Quelle [18.09.2019].

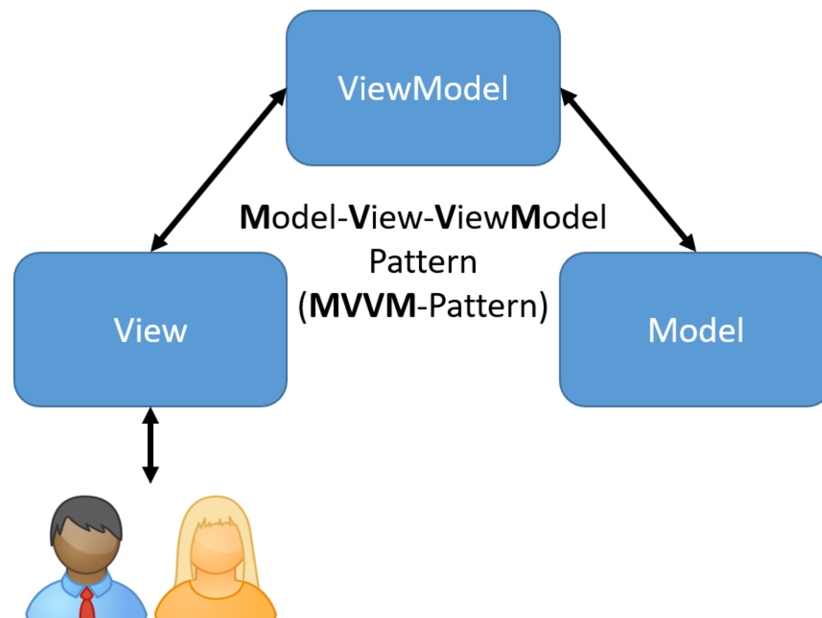

Abbildung 14: Model-View-ViewModel-Muster, Quelle: In Anlehnung an Rishab Software (2016), Online-Quelle [18.09.2019].

3.3 Modell-basierte Codegenerierung

3.3.1 Das Modell

„Ein Modell ist ein System, das als Repräsentant eines komplizierten Originals auf Grund mit diesem gemeinsamer, für eine bestimmte Aufgabe wesentlicher Eigenschaften von einem dritten System benutzt, ausgewählt oder geschaffen wird, um letzterem die Erfassung oder Beherrschung des Originals zu ermöglichen oder zu erleichtern bzw. um es zu ersetzen.“[21]

Modelle werden dazu genutzt, ein Original abzubilden und vermitteln die verschiedenen Wahrnehmungsdimensionen innerhalb des jeweiligen Abstraktionslevels. Dadurch kann ein vereinfachtes Abbild der Realität oder Teilstück der Realität wiedergegeben werden. In weiterer Folge werden Modelle zur Erklärung, Gestaltung und Beschreibung der Realität eingesetzt (siehe Abbildung 15).[22]

[21] Wüstneck (1963), S. 1522 f.

[22] Vgl. Universität Augsburg Institut für Programmierung verteilter Systeme (2009), Online-Quelle [29.07.2019].

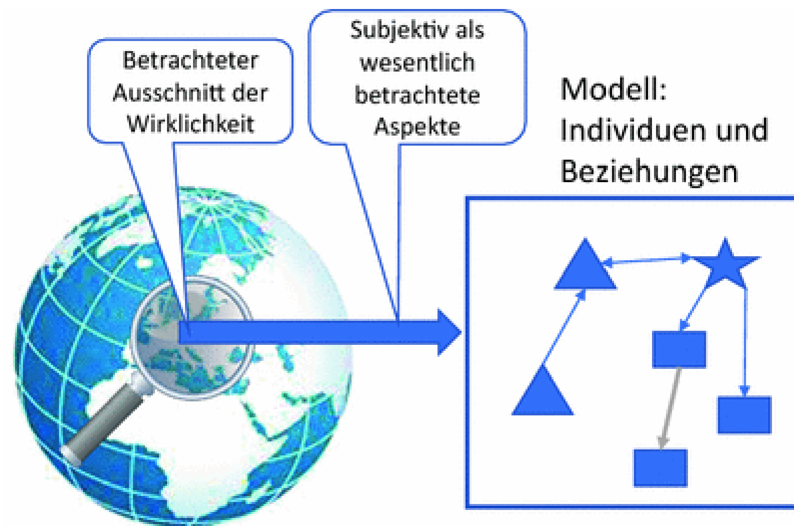

Abbildung 15: Modellbildung, Quelle: Fleischmann/Stefan/Schmidt/Stary (2018), S. 20.

3.3.2 Die Modelgetriebene-Software-Entwicklung

Modelgetriebene-Software-Entwicklung beziehungsweise Model Driven Software Development (MDSD), ist ein Überbegriff für verschiedene Techniken zur automatischen Generierung lauffähiger Software aus formalen Modellen. Die formalen Modelle liegen dafür in grafischer oder textueller Notation vor. Modelle, die lediglich der Dokumentation dienen, werden nicht zu den MDSD-Modellen gezählt. Modelle der MDSD sowie der Quellcode fließen direkt mit in die Implementierung der Software ein und stehen aus diesem Grund auf der selben Ebene wie jener. Das primäre Ziel der MDSD ist es, den Prozess der Software-Entwicklung auf Basis von Modellen zu automatisieren. Durch die Abstraktion der Anforderungen an die Software in Modell-Form soll die Transformation von Modellen zu Modellen und in weiterer Folge zu ausführbarem Code ermöglicht werden. Im Gegensatz zur traditionellen Software-Entwicklung werden Modelle hierbei nicht nur zur Dokumentation, Analyse oder Entwurf eingesetzt, sondern dienen primär der Generierung des finalen Systems. Ein wesentlicher Aspekt der MDSD ist die Definition von Modellen, die in die Software einfließen. Das bekannteste Beispiele für die Umsetzung vom MDSD ist die Model Driven Architecture (MDA) der Object Management Group (OMG).[23]

Die Phasen des Software-Entwicklungsprozesses

Um das Konzept der Modell-basierten Software-Entwicklung umsetzen zu können, müssen zuvor die grundlegenden Phasen des Software-Entwicklungsprozesses verstanden werden. Dazu kann das fünfstufige Wasserfallmodell, das sich an Royce orientiert, herangezogen werden. In Abbildung 16 werden die einzelnen Phasen dargestellt.[24]

---

[23] Vgl. DATACOM Buchverlag GmbH (2013), Online-Quelle [29.07.2019].

[24] Vgl. IONOS SE 1 (2019), Online-Quelle [31.07.2019].

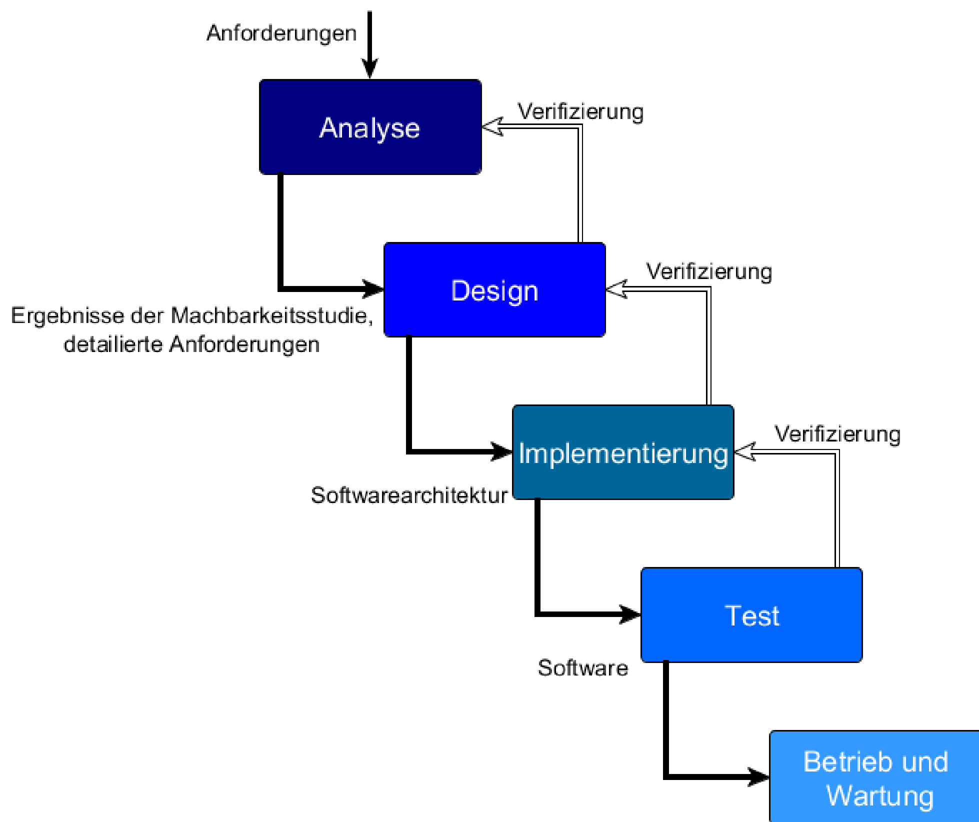

Abbildung 16: Wasserfallmodell, Quelle: IONOS SE 1 (2019), Online-Quelle [31.07.2019], (leicht modifiziert).

Die einzelnen Phasen des Wasserfallmodells reihen sich kettenförmig aneinander. Jede einzelne Phase wird mit einem Meilenstein abgeschlossen. Folgend werden die Tätigkeiten sowie die Resultate der jeweiligen Phase genauer erklärt:[25]

- Analyse
  Die Analysephase bildet den Start jedes Software-Entwicklungsprojekts. Basis für diese Phase sind Anforderungen von zum Beispiel Aufraggebern und Auftraggeberinnen oder internen Fachbereichen. Die Analyse umfasst neben der Definition von detaillierten Anforderungen auch eine Machbarkeitsstudie, in der das Projekt auf technische, zeitliche und wirtschaftliche Realisierbarkeit geprüft wird. Die Anforderungsdefinition beinhaltet eine Ist-Analyse und das Soll-Konzept. Bei der Ist-Analyse ist der Fokus auf die Problembereiche gerichtet. Das Soll-Konzept beinhaltet, welche Funktionen das Ergebnis des Software-Projektes beinhalten muss, um die Anforderungen der Auftraggeber und Auftraggeberinnen zu erfüllen. Am Ende dieser Phase findet üblicherweise noch eine Analyse der Anforderungsdefinition statt, bei der auf Basis der Anforderungen Lösungsstrategien entwickelt werden.

[25] Vgl. IONOS SE 1 (2019), Online-Quelle [31.07.2019].

- Design

  In der zweiten Phase, der Design-Phase, werden, mit den in der Analyse-Phase erarbeiteten Analysen, Strategien und Anforderungen Konzepte zur Lösung der Problemstellung zu finden. Die Software-Entwickler und Software-Entwicklerinnen erarbeiten in dieser Phase die Software-Architektur und definieren die Softwarekomponenten wie Schnittstellen, Bibliotheken und Frameworks. Zu den Ergebnissen dieser Phase zählen neben Entwurfsdokumenten inklusive Software-Architektur auch die Testpläne einzelner Komponenten.

- Implementierung

  In der Implementierungs-Phase wird die konzipierte Software-Architektur in der für das Projekt gewählten Programmiersprache umgesetzt. Dazu gehören das Programmieren und Debuggen sowie Modultests. In diesen Tests werden die entwickelten Module separat überprüft und erst danach Schritt für Schritt in das Gesamtsystem integriert. Das Resultat dieser Phase ist eine Software, die in der nächsten Phase erstmalig als Gesamtsystem getestet werden kann.

- Test

  In der Testphase wird die Software in die vorgesehene Zielumgebung integriert. Die in der Analyse-Phase entwickelten Akzeptanztests werden durchgeführt, um zu überprüfen, ob die definierten Anforderungen vom Software-Produkt erfüllt werden. Zusätzlich werden in der Regel Beta-Versionen an ausgewählte User verteilt. Die Endbenutzer und Endbenutzerinnen führen damit sogenannte Beta-Tests durch, in denen typische Verhaltensmuster getestet werden. Etwaige Fehler und Vorschläge für Verbesserungen können im Anschluss an das Entwicklungsteam weitergeleitet werden. Nach erfolgreicher Absolvierung aller Tests wird die Software freigegeben.

- Betrieb

  Auf die Freigabe der Software folgt die Auslieferung an den Kunden oder die Kundin. Diese abschließende Phase des Wasserfallmodels beinhaltet neben der Auslieferung auch die Wartung und etwaige Verbesserungen der Software.

Der Unterschied zwischen dem Prozess modell-basierter Software-Entwicklung und konventioneller objekt-orientierter Software-Entwicklung ist nur gering, jedoch ergibt sich aufgrund dieser Unterschiede der Anspruch, die einzelnen Phasen teils zu automatisieren. So sollen Fehler, die durch das Roundtrip-Engineering entstehen können, vermieden werden. Roundtrip-Engineering beschreibt die automatische Synchronisation von Model und Code.[26]

Das Problem von Roundtrip-Engineering ist, dass Änderungen und Fehlerkorrekturen auf falschen Abstraktionsniveaus liegen. Beispielsweise sind Vererbungshierarchien und Objektbeziehungen im Quelltext nicht sofort erkennbar. Modelle wie zum Beispiel Klassendiagramme hingegen erleichtern es den

[26] Vgl. Universität Augsburg Institut für Programmierung verteilter Systeme (2009), Online-Quelle [29.07.2019].

Überblick zu behalten. Das ausschließliche Arbeiten am Quelltext führt zu schlechter Wartbarkeit und Unübersichtlichkeit der Software-Lösung.[27]

Die zuvor genannten Fehler werden durch die modellgetriebene Softwareentwicklung ausgeglichen, da Teile der Phasenfolge automatisiert werden und die Implementierung zum Teil aus dem Design-Modell generiert wird (siehe Abbildung 17). Die genaue Vorgehensweise ist abhängig vom Software-Projekt, jedoch kann grundsätzlich gesagt werden, dass die Vorgehensweise bei modellgetriebener Entwicklung in zwei Bereiche unterteilt werden kann:[28]

- Generator-Entwicklung
  Festlegen der Generator-Architektur und Entwicklung des Generators
- Anwendungs-Entwicklung
  Entwicklung der Software, auf Basis der zuvor definierten Architektur und Nutzung des Generators

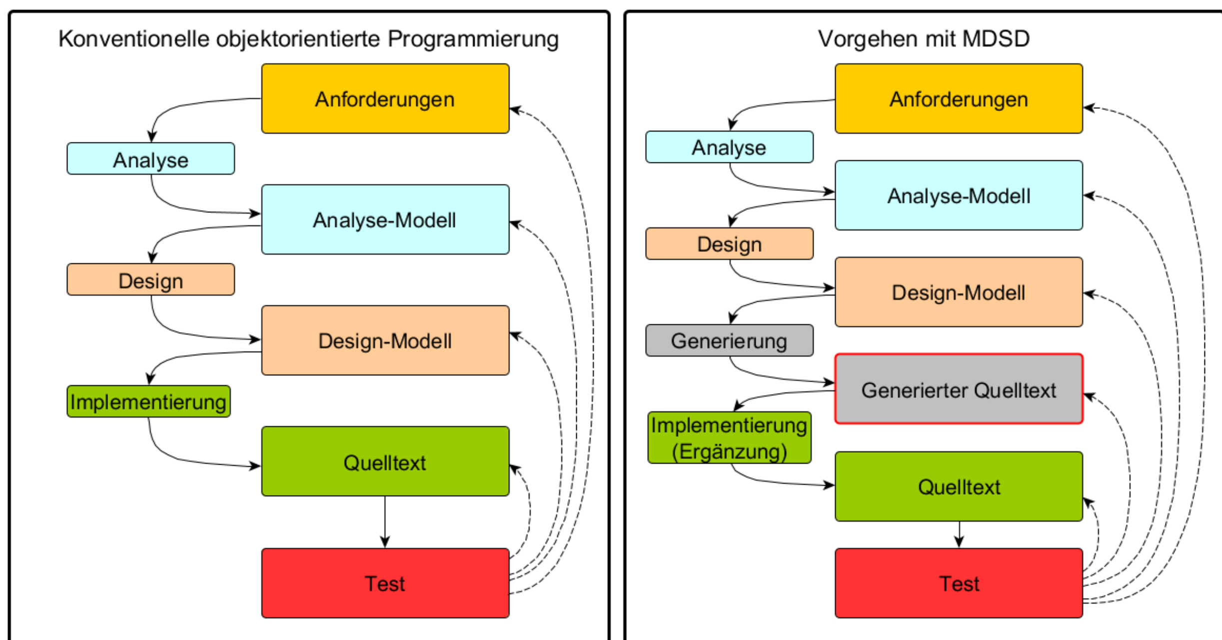

Abbildung 17: Unterschied des Softwareentwicklungsprozesses zwischen konventioneller objektorientierter Programmierung und MDSD, in Anlehnung an Pietrek (2008), Online-Quelle [30.7.2019].

3.3.3 Die Unified Modelling Language

Bei der Unified Modelling Language (UML) handelt es sich um einen Standard der zur Visualisierung von Objekten, Prozessen und Zuständen eines Systems eingesetzt und von der OMG weiterentwickelt wird. Die Modellierungssprache kann zur Planung der Architektur von Softwareprojekten herangezogen werden und Entwickler und Entwicklerinnen dabei unterstützen, die Systembeschreibung nachvollziehbarer darzustellen. Das Anwendungsgebiet von UML liegt grundsätzlich in der objektorientierten

[27] Vgl. Trompeter/Pietrek/Beltran/Holzer/Kamann/Kloss/Mork/Nieheus/Thoms (2007), S.23.

[28] Vgl. Trompeter/Pietrek/Beltran/Holzer/Kamann/Kloss/Mork/Nieheus/Thoms (2007), S.37.

Programmierung, jedoch ist UML seit der Erweiterung des Standards in der Version 2.0 auch zur Darstellung von Geschäftsprozessen geeignet. Zur Darstellung sind verschiedene Modelle bzw. Diagramme im UML Standard enthalten. In der aktuellen Version 2.5.1 sind 14 verschiedene Diagrammtypen festgelegt, welche folgende Systembestandteile darstellen:[29]

- Einzelne Objekte (grundlegende Bestandteile)
- Klassen (Zusammenfassung von Elementen mit gleichen Eigenschaften)
- Beziehungen zwischen Objekten (Darstellung der Hierarchie, Kommunikation und Verhalten zwischen Objekten)
- Aktivitäten (komplexe Kombinationen von Aktionen/Verhaltensbausteinen)
- Interaktionen zwischen Objekten und Interfaces

Grundsätzlich wird bei UML zwischen statischen und dynamischen Modellen unterschieden. Die in Abbildung 18 dargestellte Pyramide gibt einen Überblick über die von UML unterstützten statischen und dynamischen Modelle und geht auf die zuvor genannten Software-Entwicklungsphasen ein. Das statische Modell beschreibt hierbei die Systemstruktur. Es werden Datentypen und Datenstrukturen sowie die Beziehung zwischen ihnen definiert. Des Weiteren werden Kommunikationspartner ermittelt und Verantwortlichkeiten zugeordnet. Das dynamische Modell hingegen konzentriert sich auf die Beschreibung des Systemverhaltens. Mit detaillierten Szenarien für den Ablauf von Anwendungsfällen bietet das dynamische Modell einen wesentlich direkteren Zugang als das statische Modell. Darüber hinaus wird die Ausführung der im statischen Modell definierten Verantwortlichkeiten festgelegt und der zeitliche Ablauf der Prozesse definiert.[30]

[29] Vgl. IONOS SE 2 (2018), Online-Quelle [26.07.2019].

[30] Vgl. Seemann/Wolff von Gudenberg (2006), S. 7ff.

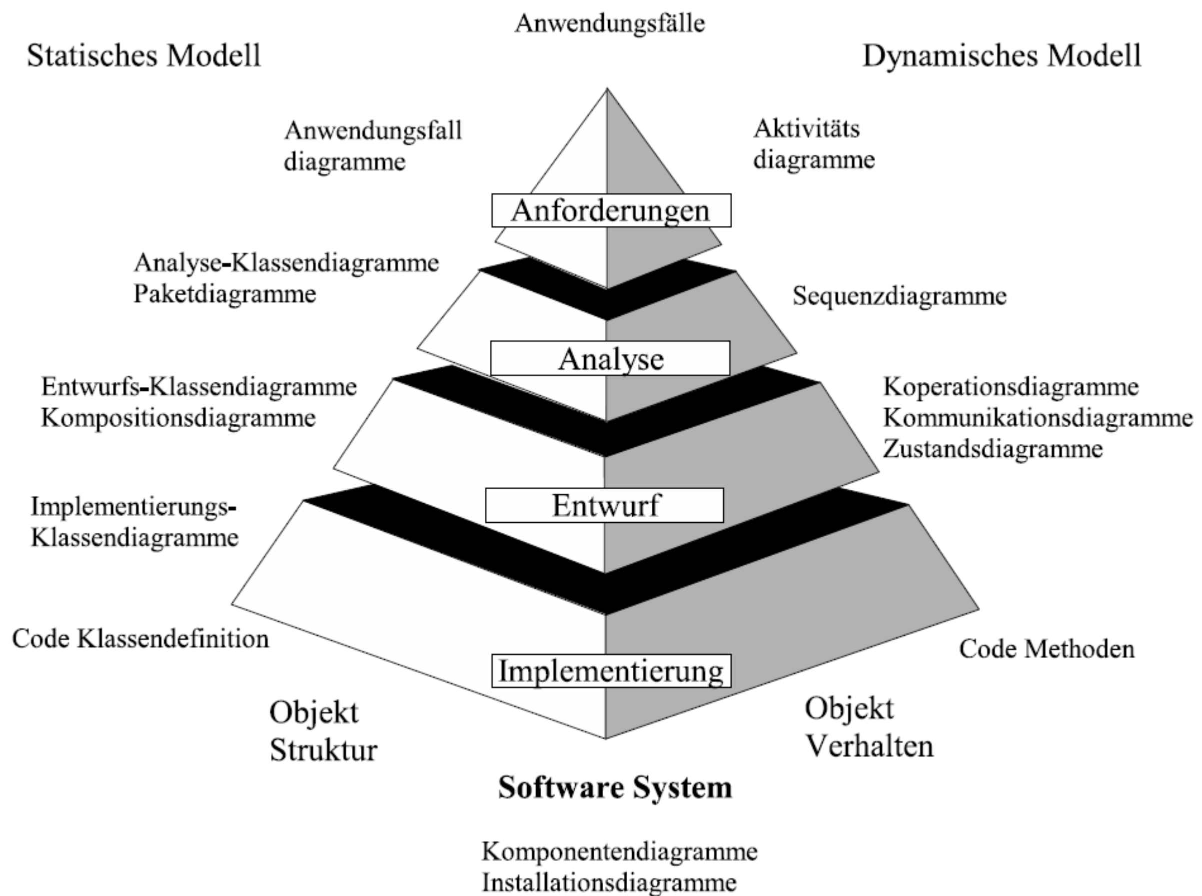

Abbildung 18: Einsatzgebiet von UML, Quelle: Seemann/Wolff von Gudenberg (2006), S.8.

Bei UML versteht man unter Codegenerierung den Prozess der Überführung von UML-Diagrammen in Quellcode. UML-Diagramme sind lediglich Ausdruck eines Modells ihrer Software und müssen deshalb erst in Quellcode übersetzt werden. Dieser Vorgang kann manuell oder automatisch vorgenommen werden, wobei der automatische Ansatz bevorzugt wird, da dadurch Arbeit und die Fehleranfälligkeit verringert werden kann. UML ist eine programmiersprachen- und plattformunabhängige Modellierungssprache und bietet die Möglichkeit, UML-Diagramme in Quellcode einer beliebigen Programmiersprache zu überführen, vorausgesetzt, es stehen den geometrischen Formen der UML-Diagramme Konzepte oder Schlüsselwörter einer Programmiersprache gegenüber. Ein wichtiger Punkt ist, dass für die Codegenerierung alles was auch später im Quellcode benötigt wird in UML-Diagrammen abgebildet werden muss. Das widerspricht zwar dem eigentlichen Sinn von UML, da nun alle Details abgebildet werden müssen, jedoch ist es für die Codegenerierung unerlässlich. Die unterstützten Programmiersprachen, beziehungsweise UML-Diagramme, sind abhängig vom gewählten Modellierungswerkzeug.[31]

[31] Vgl. Schäling (2010), Online-Quelle [22.08.2019].

Eclipse Papyrus (siehe Abbildung 19) ist ein Tool, welches zur UML-Modellierung eingesetzt wird und die Codegenerierung aus Klassendiagrammen für Java und C++ unterstützt [32].

Dadurch ist es möglich die leeren Klassen ohne eigentlichen Inhalt zu erzeugen. Wie in Abbildung 18 ersichtlich zählen Klassendiagramme zu den statischen Modellen. Das bedeutet, dass darin nur die Architektur beschrieben wird, jedoch nicht die Funktionsweise. Zur Darstellung der Funktionsweise könnten Aktivitäts- oder Sequenzdiagramme eingesetzt werden, jedoch kann mit Papyrus kein Code daraus generiert werden.

Bei Visual Studio hingegen wird der UML-Designer seit 2017 nicht mehr standardmäßig unterstützt [33]. Diese Funktionalität kann jedoch durch Installation des Class Designers, der die Generierung aus Klassendiagrammen ermöglicht, hinzugefügt werden [34].

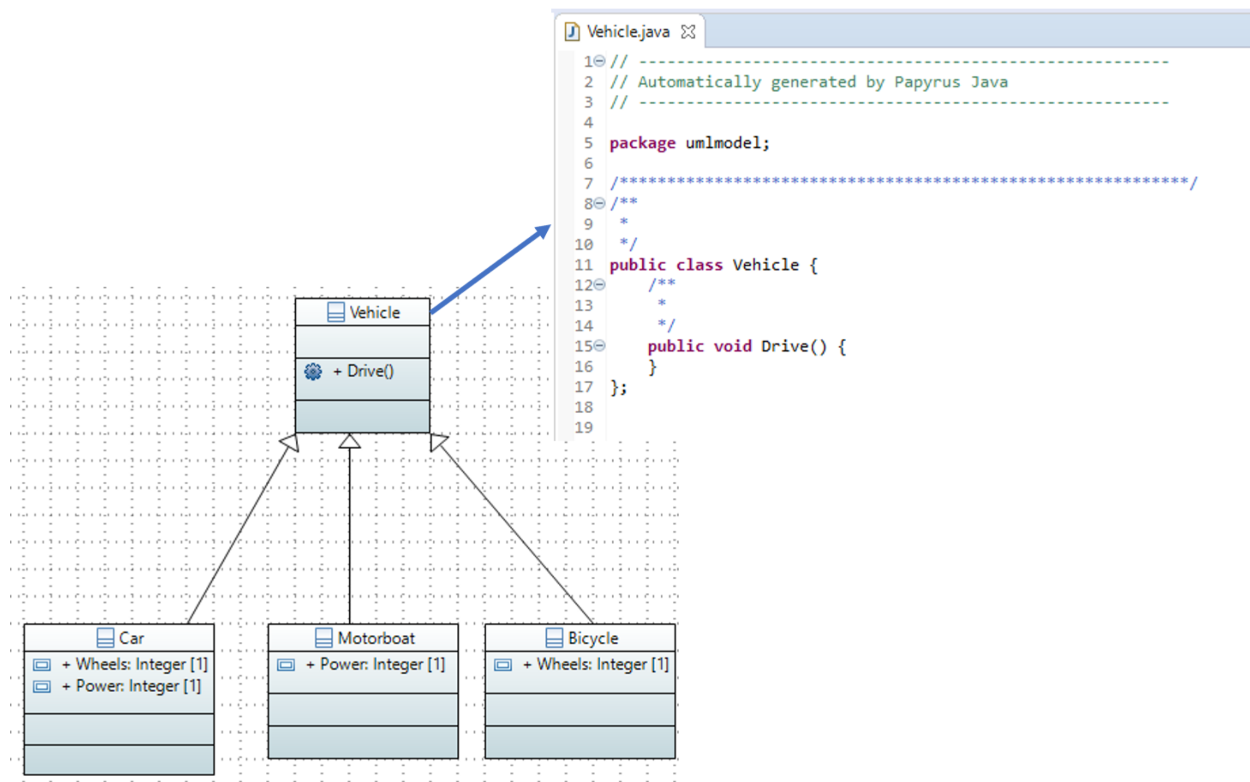

Abbildung 19: UML-Klassendiagramm und Beispiel einer generierten Klasse in Eclipse Papyrus, Quelle: Eigene Darstellung.

[32] Vgl. Eclipse Foundation 2 (2018), Online-Quelle [22.08.2019].

[33] Vgl. Microsoft Corporation 3 (2016), Online-Quelle [22.08.2019].

[34] Vgl. Microsoft Corporation 4 (2018), Online-Quelle [29.11.2019].

4 KONZEPT ZUR CODEGENERIERUNG

4.1 Anforderungen an das Konzept

Für dieses Softwarevorhaben gibt es folgende grundlegende Anforderungen:

- Import der Eingabedaten

  Das Software-Team stellt neben den Application Programming Interfaces, den APIs, auch die Beschreibung dieser in einem universellen Datenaustauschformat zur Verfügung. Diese Daten werden aus den zu testenden APIs generiert und sind deshalb mit jenen synchron. Ziel ist es, dass diese Daten direkt vom Generator eingelesen werden können. Die Daten sollen vor dem Einlesen nicht manipuliert beziehungsweise in ein anderes Format konvertiert werden müssen. Dadurch sollen zusätzliche Fehlerquellen vermieden werden.

- Flexibilität hinsichtlich der Eingabedaten

  Flexibilität stellt eine der Hauptanforderungen dieser Aufgabenstellung dar. Die Library der zu testenden APIs ändert sich während der Entwicklungsphase sehr häufig. Neben der Anzahl, Funktionalität und API-Namen ändern sich auch die Typen und die Anzahl der Übergabeparameter sowie Rückgabewerte. Neben einfachen und komplexen Datentypen sollen auch abstrakte Datentypen sowie APIs in Form von Callbacks unterstützt werden. Das erfordert eine von der Eingabedaten-Struktur unabhängige Implementierung des Codegenerators. Ziel ist es dabei, dass bei sich ändernden Eingabedaten keine Anpassung des Codegenerators notwendig wird. Bei Änderung der Eingabedaten muss der Codegenerator dynamisch auf diese reagieren können.

- Plattform-Unabhängigkeit

  Neben der Flexibilität ist es wichtig, dass mit der gewählten Codegenerierungsmethode verschiedene Plattformen und Programmiersprachen unterstützt werden können. Neben der C#-Library für den Command-Sender (siehe Kapitel 7.1.3) sollen auch die Bibliotheken für die Test-App für iOS, Android und Windows generiert werden können.

- Codegenerierungsanteil

  Ziel eines Softwareprojekts, bei dem automatisierte Codegenerierung eingesetzt wird, ist, dass der Codegenerierungsanteil so hoch wie möglich ist. Dadurch kann eine höhere Code-Qualität erzielt und die Entwicklungszeit verkürzt werden. Bei Projekten, bei denen dieser Anteil nur sehr gering ausfallen würde, muss die Frage gestellt werden, ob diese Codegenerierungsmethode für dieses Vorhaben überhaupt zweckmäßig ist.

4.2 Auswahl der Codegenerierungsmethode

Mit den Anforderungsdefinitionen aus Kapitel 4.1 können die in Kapitel 3 untersuchten Codegenerierungsmethoden auf ihre Umsetzbarkeit für dieses Softwarevorhaben überprüft werden. Vorab können bereits die in Kapitel 3.1 beschriebenen unterstützenden Funktionen der Entwicklungsumgebungen als Methode zur automatischen Codegenerierung ausgeschlossen werden. Jedoch sind Snippets und die automatische Codevervollständigung Features, welche bei der Implementierung nicht wegzudenken sind und den Programmierfluss beziehungsweise Programmierkomfort deutlich verbessern. Aus diesem Grund werden nur noch die Template-basierte Codegenerierung sowie die Modell-basierte Codegenerierung in Betracht gezogen.

Im folgenden Kapitel wird untersucht, inwieweit die Anforderungen mit der jeweiligen Codegenerierungsmethode erfüllt werden können. Dafür werden die einzelnen Anforderungen priorisiert. Höchste Priorität hat in diesem Fall definitiv die Plattform-Unabhängigkeit und die Flexibilität hinsichtlich der Eingabedaten. Mit mittlerer Priorität kann der Codegenerierungsanteil betrachtet werden, da trotzdem die Funktionalität höher als die Effizienz der Codegenerierung gewichtet werden kann. Die Anforderung der Verarbeitung der Eingabedaten wird am wenigsten gewichtet, da diese in einem bekannten Datenaustauschformat vorliegen und deshalb keine größeren Probleme zu erwarten sind.

In folgender Tabelle wird das zehnstufige Bewertungsschema dargestellt:

Beschreibung	Nicht geeignet				Mit Einschränkungen geeignet				Sehr gut geeignet	
Punkte	1	2	3	4	5	6	7	8	9	10

Tabelle 1: Bewertungsschema zur Auswahl der Codegenerierungsmethoden, Quelle: Eigene Darstellung.

- Import der Eingabedaten

  Da die Eingabedaten vom Software-Team immer im selben Datenaustauschformat zur Verfügung gestellt werden, müssen diese nicht in ein anderes Format konvertiert werden. Dadurch wird die Komplexität reduziert und die Eingabedaten können ohne Anpassung für die Template-basierte und Modell-basierte Codegenerierung eingesetzt werden. Aus diesem Grund werden für beide Ansätze zehn Punkte vergeben.

- Flexibilität

  Bei der Modell-basierten Codegenerierung wird die Logik des Codegenerators in Modellen dargestellt. Deshalb ist der Modell-basierte Ansatz kaum für die Abstraktion der Logik geeignet. Templates hingegen eignen sich jedoch sehr gut, da die Logik nur als statischer Inhalt aufgenommen wird.

- Plattform-Unabhängigkeit

  Der Template-basierte Ansatz ermöglicht es, plattformunabhängig Code zu generieren, da die Ausgabedaten lediglich Textdateien sind, die im jeweiligen Format abgespeichert werden können. UML ist prinzipiell eine plattform- und programmiersprachenunabhängige Modellierungssprache, jedoch besteht eine Einschränkung seitens der Entwicklungsumgebung Eclipse. Das Plugin Papyrus unterstützt nur die Programmiersprachen Java und C++, nicht jedoch Swift und C#.

- Codegenerierungsanteil

  Der Codegenerierungsanteil bei der Template-basierten Codegenerierung ist für dieses Projekt verhältnismäßig hoch, da sehr viele wiederholende Tätigkeiten mit den Templates generiert werden. Die meisten Plug-ins beziehungsweise Entwicklungsumgebungen unterstützen nur die Codegenerierung aus Klassendiagrammen. Daraus resultieren nur die leeren Klassen mit den Attributen ohne eigentlichen Inhalt. In diesem Fall muss die Logik erst manuell implementiert werden. Jedoch kann der Codegenerierungsanteil durch die Verwendung von Frameworks auf Basis der MDA deutlich erhöht werden.

Anforderung	Faktor	Modell-basierte Entwicklung		Template-basierte Entwicklung	
		Punkte	Zeichen	Punkte	Zeichen
Import der Eingabedaten	1	10	✓	10	✓
Flexibilität	3	2	✕	9	✓
Plattform-Unabhängigkeit	3	6	✕	9	✓
Codegenerierungsanteil	2	7	✕	8	✓
Ergebnis		48		70	

Tabelle 2: Bewertung der Codegenerierungsmethoden, Quelle: Eigene Darstellung.

Tabelle 2 zeigt, dass der Template-basierte Ansatz alle Anforderungen erfüllt und demnach die zu wählende Codegenerierungs-Variante sein sollte. Der Entwicklungsaufwand wurde in dieser Bewertung nicht berücksichtigt, jedoch scheinen auch aus diesem Aspekt die Vorteile auf Seiten der Template-basierten Methode zu liegen, da im Gegensatz zur Modell-basierten Variante keine neue Modellierungssprache erlernt werden muss.

5 TEMPLATE-ENGINES

5.1 Vergleich von Template-Engines

In diesem Kapitel werden die zwei Template-Engines Apache FreeMarker und Apache Velocity vorgestellt. Danach werden beide miteinander verglichen und eine Wahl getroffen.

5.1.1 Apache FreeMarker

Apache FreeMarker ist eine Template-Engine, die auf einer Java-Bibliothek basiert und unter Apache lizensiert ist. Sie wird eingesetzt, um Textausgaben wie zum Beispiel Quellcode, HTML-Seiten, E-Mails und Konfigurationsdateien basierend auf Templates und sich ändernden Daten zu generieren. Die Templates werden in der FTL, der FreeMarker Template Language, geschrieben. Dabei handelt es sich um eine einfache spezialisierte Programmiersprache. Zur Vorbereitung der Eingabedaten wie zum Beispiel Datenbankabfragen, oder für Kalkulationen wird eine universell einsetzbare Programmiersprache wie in diesem Fall Java verwendet. Im Anschluss stellt FreeMarker die Daten anhand von vorbereiteten FTL-Templates dar. Der Fokus liegt in den Templates darauf, wie die Daten präsentiert werden. Außerhalb der Templates konzentriert man sich darauf, welche Daten dargestellt werden. Dieser Ansatz wird auch als MVC-Muster (siehe Kapitel 3.2.2) bezeichnet. Das ermöglicht eine bessere Trennung der Programmlogik von der Daten-Darstellung. Das birgt nicht nur Vorteile bei der Entwicklung von dynamischen Webapplikationen, sondern auch bei sich ändernden nicht-webbasierten Anwendungsumgebungen.[35]

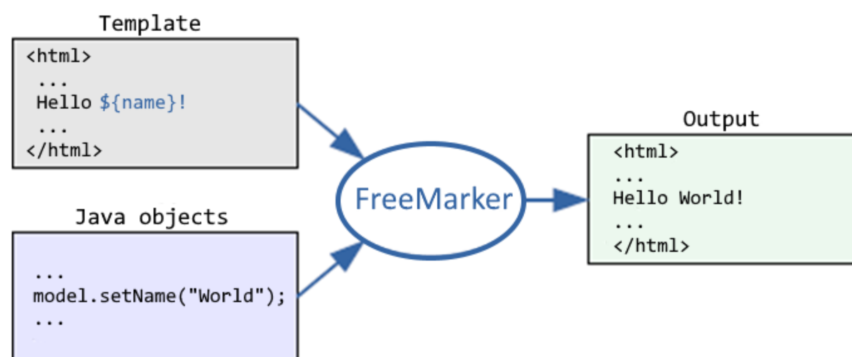

Abbildung 20: Übersicht der Funktionsweise von FreeMarker. Quelle: FreeMarker 1 (2019), Online-Quelle [11.08.2019].

Die FTL-Templates sind Textfiles, die die gewünschte Ausgabe, sowie Platzhalter, die durch ${Variablenname} gekennzeichnet sind, enthalten. Programmlogik ist in Templates nur in Form von zum Beispiel Schleifen und Bedingungen vorhanden, welche für die Darstellung der Daten notwendig sind. Die Eingabedaten der Templates werden üblicherweise in Form von Map<String, Object> oder als JavaBean vom Java-Programm zur Verfügung gestellt. Die Variablenwerte selbst können Strings, Zahlen, Maps, Listen oder beliebige Java-Objekte sein, auf deren Methode von der Vorlage aus zugegriffen wird. Die

[35] Vgl. FreeMarker 1 (2019), Online-Quelle [11.08.2019].

Ausgabe des Templates kann mit Hilfe des Writers beispielsweise in ein lokales File oder einen String geschrieben werden.[36]

FreeMarker wird von einer Vielzahl von Editoren und Entwicklungsumgebungen unterstützt, beziehungsweise können einige durch Plugins damit erweitert werden:[37]

- IntelliJ IDEA
- jEdit
- Eclipse
- NetBeans
- etc.

Eclipse, eine der bevorzugten Entwicklungsumgebungen für dieses Softwarevorhaben unterstützt FreeMarker nicht out-of-the-box, jedoch kann dieses Feature durch die Installation des Plugins „FreeMarker from JBoss Tools“ aus dem Eclipse Marketplace (siehe Abbildung 21) hinzugefügt werden. Nach Installation des Plugins werden folgende Features unterstützt:[38]

- Syntax Highlighting
- Syntax Error Marker
- Codevervollständigung von
  - Macro-Namen
  - Bean-Property-Namen

[36] Vgl. Vogel (2016), Online-Quelle [11.08.2019].

[37] Vgl. FreeMarker 2 (2019), Online-Quelle [11.08.2019].

[38] Vgl. Dekany, Daniel (2019), Online-Quelle [30.11.2019].

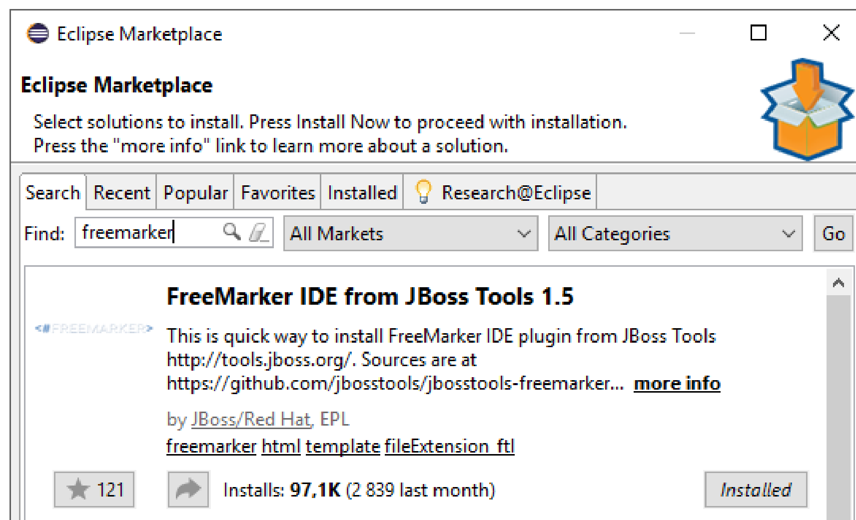

Abbildung 21: FreeMarker IDE im Eclipse Marketplace. Quelle: Eigene Darstellung.

5.1.2 Apache Velocity

Velocity ist ebenfalls wie FreeMarker eine Java-basierte Template-Engine. Velocity ist ein Open-Source-Projekt der Apache Software Foundation. Durch die Apache Software License steht Velocity den Usern kostenlos zur Verfügung. Velocity setzt den Fokus, wie die meisten Template Engines, auf Web-Anwendungen, jedoch wird es auch zur Generierung von Quellcode, E-Mails oder zur XML-Transformationen eingesetzt. Velocity folgt auch dem MVC-Entwicklungsmuster (siehe Kapitel 3.2.2), indem der Java-Code strikt vom Template-Code getrennt wird. Das führt zu besserer Wartbarkeit, besser designten Ausgabedateien und längerfristigen Entwicklungszeit-Einsparungen. Template-Designer und Template-Designerinnen haben die Möglichkeit, Markup-Statements, sogenannte Referenzen, in das Template einzubinden. Diese Referenzen werden von Context-Objects bezogen, welche Get- und Set-Methoden zur Verfügung stellen und damit das Setzen und Abrufen von Objekten ermöglichen. Diese Werte können dann direkt in das Template eingefügt werden. Velocity stellt grundlegende Steueranweisungen zum durchlaufen von mehreren Werten (foreach), oder bedingte Verzweigungen (if/else) zur Verfügung. Die Möglichkeit, Java-Methoden aufzurufen, externe Dateien einzubinden und Makros zu erstellen, die wiederholt verwendet werden können, macht Velocity zu einer leistungsstarken und dennoch einfach zu bedienenden Lösung für die Erstellung dynamischer Ausgabedateien.[39]

Die Velocity Templates werden in der Velocity Template Language, kurz VTL, geschrieben. VTL ist eine einfach zu erlernende Sprache, welche es auch Programmierern und Programmiererinnen mit wenig Erfahrung ermöglichen sollte, innerhalb kurzer Zeit Templates für Websites oder anderen Dokumenten entwickeln zu können. Velocity-Templates besitzen die Dateieiendung .vm. VTL nutzt Referenzen zur Einbindung von dynamischen Inhalten. Variablen sind eine Art von Referenzen, welche sich auf eine Definition im Java-Code oder auf ein VTL-Statement beziehen können. VTL Statements sind mit dem #-Zeichen am Anfang gekennzeichnet und beinhalten die set-Anweisung. Die set-Anweisung hält innerhalb

[39] Vgl. Apache Velocity Project 1 (2016), Online-Quelle [12.08.2019].

von Klammern eine Anweisung, die einer Variable einen Wert zuweist. Variablen selbst haben ein $-Zeichen am Beginn. Strings werden unter Anführungszeichen oder Apostroph gesetzt. Werden Apostrophe benutzt wird sichergestellt, dass der angegebene Wert der Referenz genau so zugewiesen wird, wie er da geschrieben steht. Anführungszeichen ermöglichen es, Velocity-Referenzen und Anweisungen zur Interpolation zu verwenden (siehe Abbildung 22). Wie auch bei FreeMarker werden die wichtigsten logischen Operatoren, Schleifen und Verzweigungen zur Ausgabe unterstützt.[40]

Abbildung 22: Beispiel zur Verarbeitung von Strings mit Velocity, Quelle: Eigene Darstellung.

Velocity wird von keiner Entwicklungsumgebung und keinem Editior out-off-the-box unterstützt. Für einige werden aber Plugins angeboten:[41]

- Eclipse
- IntelliJ IDEA
- jEdit
- Coda 2
- etc.

[40] Vgl. Apache Velocity Project 2 (2016), Online-Quelle [12.08.2019].

[41] Vgl. Apache Velocity Project 3 (2019), Online-Quelle [12.08.2019].

Für Eclipse stehen zwei Plugins im Eclipse Marketplace (siehe Abbildung 23) zur Verfügung. Zum einen „veloedit“ und zum anderen „veloeclipse“, welches auf veloedit aufbaut und dieses um HTML-Features erweitert[42]. Da in diesem Softwarevorhaben keine HTML-Seiten generiert werden, fällt die Entscheidung auf veloedit. Veloedit unterstützt unter anderem folgenden Features:[43]

- Syntax-Highlighting für Velocity Template Language (VTL)
- Content Assist für VTL Direktive und Referenzen
- Automatische Template-Validierung
- Integration von zusätzlichen Tastenkombinationen zur Erhöhung des Programmierflusses

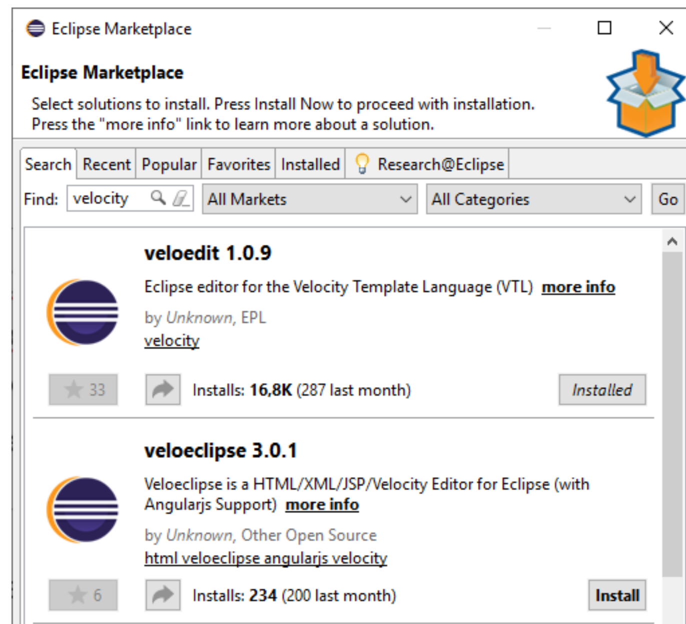

Abbildung 23: Veloedit und Veloeclipse im Eclipse Marketplace. Quelle: Eigene Darstellung

5.2 Auswahl der Template-Engine

In diesem Kapitel wird die Entscheidung getroffen, welche der im Kapitel 5.1 vorgestellten Template-Engines für diese Aufgabenstellung ausgewählt wird.

Beide Template-Engines sind in ihrer Funktionsweise ähnlich. Aus diesem Grund sind auch die Unterschiede sehr gering. Velocity sowie FreeMarker sind unter der Apache Lizenz lizensiert, und stehen deshalb den Usern kostenlos zur Verfügung. Beide Template-Engines basieren auf Java, was hinsichtlich der vorhandenen Java-Programmierkenntnisse des Teams von großem Vorteil ist. Hinsichtlich der Template-Programmiersprachen VTL und FTL bieten beide Varianten eine einfach zu erlernende Sprache, mit denen innerhalb kürzester Zeit die ersten funktionsfähigen Templates entwickelt werden können. Auf

[42] Vgl. Sarhan (o.J.), Online-Quelle [12.08.2019].

[43] Vgl. Juergeleit (2019), Online-Quelle [12.08.2019].

Grund der bisher untersuchten Punkte kann keine eindeutige Entscheidung getroffen werden. Deshalb wird die Verbreitung beziehungsweise die Community der beiden Template-Engines miteinbezogen. Eine größere Community bedeutet auch mehr Support bei etwaigen Fragen, sowie schnellere Updates bei Auftreten von Fehlern in der Engine. Dafür werden die Anzahl der Installationen der jeweiligen Plugins im Eclipse Marketplace und die Anzahl der Contributors auf GitHub verglichen.

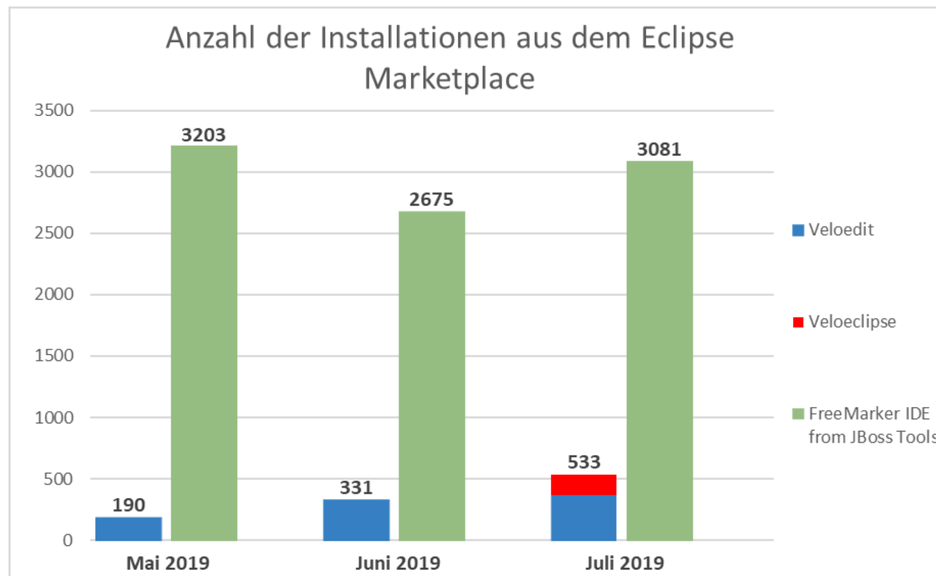

Abbildung 24: Anzahl der Installationen der jeweiligen Plugins aus dem Eclipse Marketplace. Quelle: Eigene Darstellung.

Um hinsichtlich der Anzahl der Installationen der jeweiligen Plugins vergleichbar zu sein, werden für Apache Velocity beide verfügbaren Plugins Veloeclipse und Veloedit berücksichtigt. Veloeclipse weist dabei keine Installationen vor Juli 2019 auf. Daraus kann man schließen, dass erst vor kurzer Zeit durch ein Update die Kompatibilität mit aktuellen Eclipse- beziehungsweise Velocity-Versionen ermöglicht wurde. Trotzdem übersteigen die Installationszahlen der FreeMarker IDE, die der beiden Velocity Plugins in allen drei auf einander folgenden Monate zwischen Mai und Juli 2019 (siehe Abbildung 24).

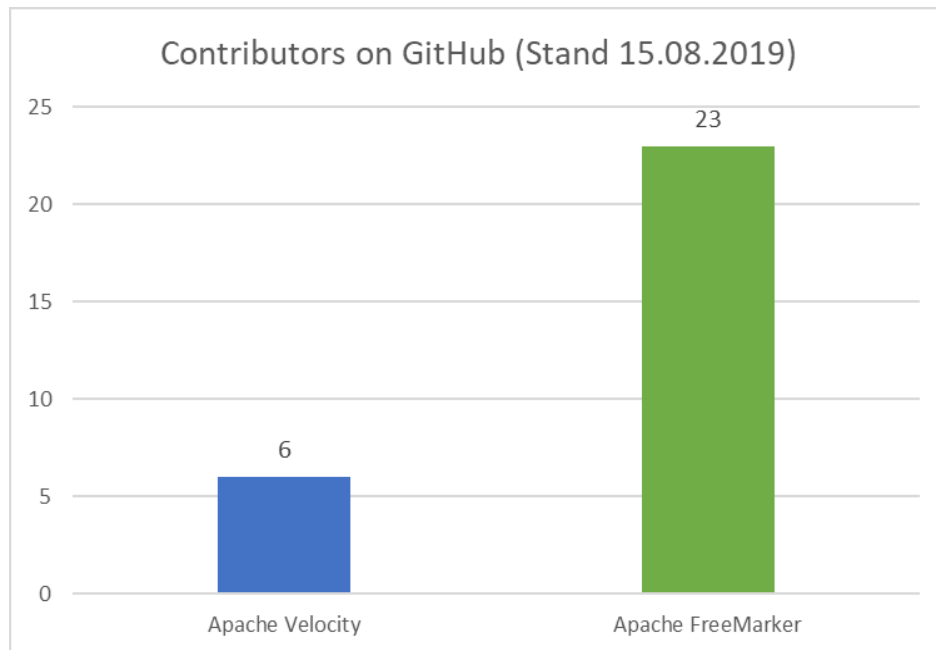

Abbildung 25: Anzahl der Contributors der jeweiligen Template-Engines auf GitHub. Quelle: Eigene Darstellung.

Auch hinsichtlich der Anzahl der mitwirkenden Personen an den jeweiligen GitHub-Projekten hat FreeMarker einen Vorteil gegenüber Velocity (siehe Abbildung 25). Das bedeutet, dass FreeMarker eine größere Community vorweist und deshalb in weiterer Folge besserer beziehungsweise schnellerer Support bei Fragen oder Implementierungsproblemen geleistet werden kann. Darüber hinaus kann eine größere Community zu höherer Qualität führen, da mehrere Entwickler und Entwicklerinnen an der Template-Engine arbeiten und dadurch verschiedene Sichtweisen bei der Lösung eines Problems betrachtet werden.

In folgender Tabelle werden die beiden Template-Engines nach dem aus Kapitel 4.2 definierten Bewertungsschema verglichen:

Vergleichsmerkmale	Apache Velocity		Apache FreeMarker	
	Punkte	Zeichen	Punkte	Zeichen
Apache Lizenz	10	✓	10	✓
Java-basierend	10	✓	10	✓
Einfache Template-Programmiersprache	8	✓	8	✓
Größe der Community	5	~	8	✓
Ergebnis	33		36	

Tabelle 3: Vergleich zwischen Apache Velocity und Apache FreeMarker. Quelle: Eigene Darstellung.

In Tabelle 3 wird ersichtlich, dass sich die beiden Template-Engines ähneln, jedoch FreeMarker auf Grund der größeren Community der Vorzug zuteilwird.

6 ARCHITEKTUR DES AKTUELLEN TESTSYSTEMS

Im folgenden Kapitel werden das aktuelle Testsystem, die Funktionsweise und die einzelnen Systemkomponenten genauer betrachtet. Am Ende dieses Kapitels wird noch genauer auf die Schwachstellen beziehungsweise auf die Elemente, die Verbesserungspotential aufweisen, eingegangen.

6.1 Systemkomponenten

6.1.1 API - Application Programming Interfaces

APIs sind Schnittstellen zwischen zwei Programmen oder zwischen einem Programm und dem Betriebssystem. APIs ermöglichen und vereinfachen die Kommunikation zwischen beiden Programmsystemen auf Quelltextebene. Durch APIs werden dem Programmierer und der Programmiererin verschiedene Möglichkeiten zur Verfügung gestellt, um mit anderen Programmen zu kommunizieren. In das Betriebssystem integrierte APIs ermöglichen zum Beispiel den Zugriff auf systeminterne Bibliotheken, das Starten von Diensten oder die Aktivierung von Hardwaregeräten. Prinzipiell können APIs in drei Arten eingeteilt werden:[44]

- Funktionsorientierte APIs
  Sie erlauben dem Programm direkten Zugriff auf die Hardware.
- Dateiorientierte APIs
  Diese erlauben die Kommunikation mit Speichermedien. Unter anderem können damit Dateien angefragt, überschrieben und ausgelesen werden.
- Protokollorientierte APIs
  Sie erlauben eine Kommunikation über definierte Kanäle zwischen zwei Programmen, unabhängig von Betriebssystem und Hardware.

Die Firma LOGICDATA entwickelt API-Bibliotheken für die Plattformen Android, iOS und Windows. Der Funktionsumfang ist für alle Plattformen bis auf wenige Betriebssystem-spezifische Unterschiede gleich. Abhängig von zu testendem Produkt stehen jedoch nicht alle Funktionen zur Verfügung. Die Libraries umfassen unter anderem den Zugriff auf das Bluetooth-, WiFi- und NFC-Modul sowie die Ansteuerung von LEDs und das Verarbeiten von User-Interaktionen.

[44] Vgl. Fuchs Media Solutions (o.J.), Online-Quelle [08.09.2019].

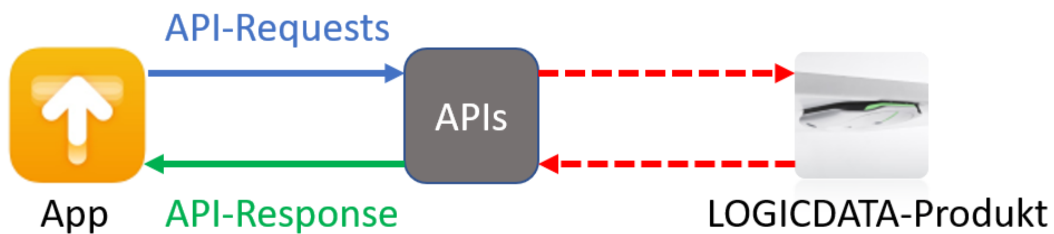

Abbildung 26: Darstellung der Funktionsweise von APIs, Quelle: Eigene Darstellung.

6.1.2 API-Dokumentation

Die Dokumentation der APIs wird vom Software-Team mit dem Tool Doxygen generiert. Die Dokumentation ist notwendig für die Entwicklung des Command Senders (siehe Kapitel 6.1.3) und für die Weitergabe an App-Entwickler und App-Entwicklerinnen. Sie beinhaltet unter anderem Informationen zu den einzelnen APIs, den Parametern und den einzelnen Modulen und Datenstrukturen. Darüber hinaus findet man zusätzliche Informationen zum Source Code und Abhängigkeitsgraphen.

Doxygen zählt zu den Standard-Tools zur Generierung von Code-Dokumentationen für C++, C, C#, Java und weiteren Programmiersprachen. Es können Online-Dokumentationen im HTML-Format oder Offline-Manuals für LATEX aus Quelldateien erzeugt werden. Zusätzlich besteht die Möglichkeit, die Dokumentation als PDF oder im RTF-Format für Microsoft Word zu erstellen. Der große Vorteil von Doxygen ist, dass die Dokumentation direkt aus dem Quellcode generiert wird. Damit ist es einfacher, den Code und die Dokumentation konsistent zu halten. Doxygen kann so konfiguriert werden, dass die Code-Struktur aus undokumentierten Quelldateien extrahiert wird. Darüber hinaus können unter anderem Vererbungsdiagramme und Abhängigkeitsgraphen automatisch generiert werden, was die Übersicht vor allem bei großen Software-Projekten deutlich verbessert.[45]

[45] Vgl. van Heesch (2019), Online-Quelle [10.09.2019].

LOGICDATA MOTION FOR YOUR LIFE LOGIClink API ver1.0

Main Page | Related Pages | Modules | Data Structures | Files | Search

Data Structures | Data Structure Index | Data Fields

Data Fields

ll_scan_result Struct Reference

`#include <LOGIClinkTypes.h>`

Data Fields

ll_intf_t	intf
uint8_t	ID [6]

Detailed Description

Definition at line 243 of file LOGIClinkTypes.h.

Field Documentation

uint8_t ll_scan_result::ID[6]

Definition at line 246 of file LOGIClinkTypes.h.

ll_intf_t ll_scan_result::intf

Definition at line 245 of file LOGIClinkTypes.h.

The documentation for this struct was generated from the following file:

- LOGIClinkTypes.h

Generated by doxygen 1.8.9.1

Abbildung 27: Dokumentation einer Datenstruktur mit Doxygen, Quelle: Eigene Darstellung.

6.1.3 Command Sender

Beim Command Sender (siehe Abbildung 28) handelt es sich um ein Testtool, das als Client in der Testumgebung fungiert. Der Command Sender ermöglicht dem Tester und der Testerin den Aufruf aller APIs und die Kontrolle der Resultate. Das Tool verfügt über eine grafische Benutzeroberfläche, durch die der User die Möglichkeit bekommt, eine Verbindung mit dem Testobjekt herzustellen. Hierfür müssen Port und IP-Adresse des Servers, der in der jeweiligen Test-App gestartet werden kann, bekannt sein. Nachdem die Kommunikation zwischen Client und Server hergestellt wurde, kann auch die Verbindung zwischen Test-App und Testobjekt aufgebaut werden. Ab diesem Zeitpunkt können die APIs der Library aufgerufen werden. Abhängig von der aufgerufenen API müssen Übergabeparameter definiert und eingegeben werden. Die Wahl der Werte liegt dabei in der Verantwortung des Testers beziehungsweise der Testerin. Danach werden die Daten im JSON-RPC Format an den Server gesendet. Die Antwort wird dann wiederum an den Command Sender zurückgesendet und von der testenden Person evaluiert.

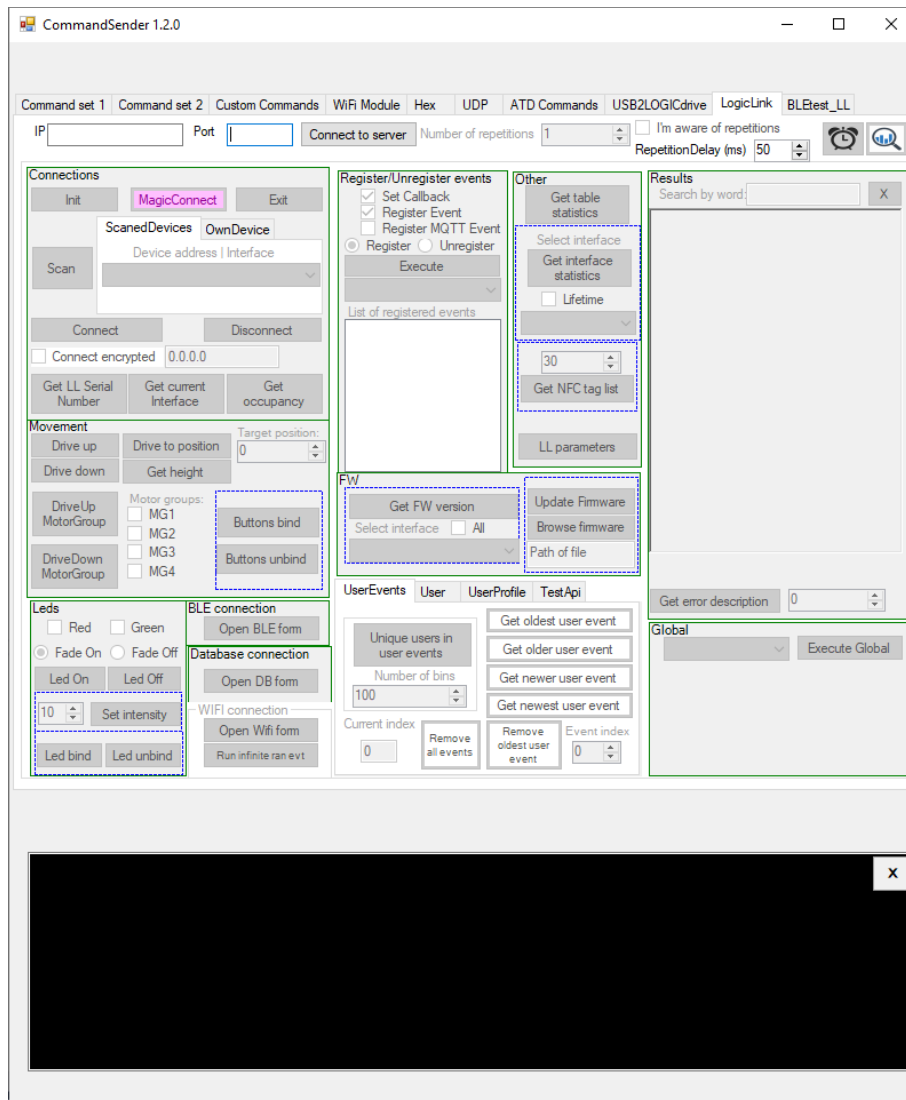

Abbildung 28: Command Sender für den Aufruf von APIs, Quelle: Eigene Darstellung.

6.1.4 Test-Apps

Bei den Test-Apps handelt es sich um Applikationen, die als Server in der Client-Server-Architektur fungieren und gleichzeitig alle APIs halten. Diese können vom Client in Form des Command Senders (siehe Kapitel 6.1.3) aufgerufen werden. Für jede Plattform wird eine eigene Test-App in der jeweiligen Programmiersprache und mit der entsprechenden API-Library entwickelt. Die grafische Benutzeroberfläche ist dabei sehr einfach gehalten. Es wird lediglich die IP-Adresse sowie der Port des Servers angezeigt, der beim Aufruf der Applikation gestartet wird. Nachdem eine Verbindung zwischen Command Sender und Test-App aufgebaut wurde, können die JSON-RPC Requests vom Server empfangen werden. Die übermittelten Daten beinhalten die Methodennamen sowie deren Übergabeparameter, die für den Aufruf der APIs notwendig sind. Diese Informationen werden für die Ausführung der API benötigt. Die Rückgabedaten werden im Anschluss als JSON-RPC Response an den Command Sender übermittelt.

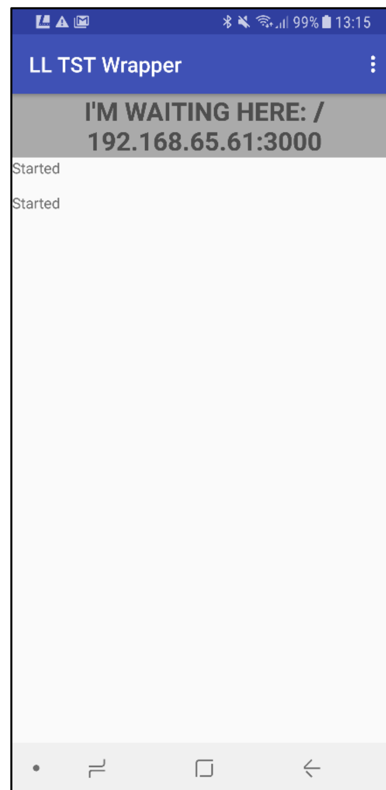

Abbildung 29: Android-Version der Test-App, Quelle: Eigene Darstellung.

6.1.5 LD-Produkt

Als Beispiel für ein LOGICDATA-Produkt wird hier der LOGIClink Corporate gewählt. Das Entwicklungsvorgehen beziehungsweise die Funktionsweise des Test-Systems ist jedoch auch für andere LD-Produkte mit App-Anbindung ähnlich.

Der LOGIClink ist ein moderner Kommunikationshub, der die Kontrolle über den Arbeitsplatz verbessert und dem Nutzer beziehungsweise der Nutzerin eine individuell angepasste Arbeitsumgebung ermöglicht. Er bildet eine Schnittstelle zwischen Arbeitsplatz und Endbenutzer sowie Endbenutzerin. Neben den ergonomischen Vorteilen einer Sitz-Steh-Arbeitsumgebung bietet LOGIClink die Möglichkeit, den Arbeitsplatz mit umgebenden Geräten zu verbinden. Durch die Anbindung an die mechatronischen Produkte von LOGICDATA wird ein dynamischer sowie aktivitätsbasierter Arbeitsplatz geschaffen, der individuell auf die Bedürfnisse der Nutzer und Nutzerinnen abgestimmt werden kann.

Der LOGIClink Corporate verfügt neben einer Auf- und einer Ab-Taste, die die Verstellung des Tisches ohne Smartdevice ermöglichen soll, über einen Belegungssensor und LEDs. Dieser Sensor wird dazu eingesetzt, um den Belegungsstatus des Arbeitsplatzes zu ermitteln. Der User kann diesen sofort an der Farbe der LEDs erkennen. Die Verbindung zwischen Smartdevice und LOGIClink kann über Bluetooth, NFC und WLAN erfolgen. Die Einbindung von LOGIClinks in ein WiFi-Netzwerk ermöglicht eine Datenerfassung und Analyse der Tischbelegung im Unternehmen. Dadurch kann eine effizientere Nutzung von Arbeitstischen erreicht werden. Nach Aufbau der Verbindung zwischen Smartdevice und LOGIClink

kann der Arbeitsplatz über die App personalisiert, Nutzungsdaten analysiert sowie Erinnerungen aktiviert werden.[46]

Abbildung 30: LOGIClink, Quelle: LOGICDATA Elektronic & Software Entwicklungs GmbH 8 (2018), Online-Quelle [21.09.2019].

6.2 Entwicklungsvorgehen und Funktionsweise

Ausgangspunkt der Entwicklung des Testsystems ist die Bereitstellung der APIs der jeweiligen Plattformen sowie die dazugehörige API-Dokumentation des Software-Teams. Wie in Abbildung 31 ersichtlich, wird die API-Dokumentation mit dem Tool Doxygen (siehe Kapitel 6.1.2) aus dem Quellcode der APIs generiert. Damit kann davon ausgegangen werden, dass die Dokumentation der APIs mit den APIs konsistent ist. Mit der im HTML- und RTF-Format vorliegenden Dokumentation kann mit der Implementierung des User Interfaces und der Logik für den Command Senders (siehe Kapitel 6.1.3) begonnen werden. Vorbereitend kann zuvor bereits die Kommunikation zwischen Client und Server entwickelt werden, da diese unabhängig von den APIs ist und üblicherweise im Laufe der Entwicklung nicht verändert werden muss. Als Protokoll wird hierfür JSON-RPC in der Version 2.0 eingesetzt, dass das Client-Server-Modell unterstützt. Auch auf Seiten der Test-App (siehe Kapitel 6.1.4) kann bereits vor Erhalt der APIs das Kommunikationsmodul fertiggestellt werden. Die Test-App beinhaltet alle APIs der jeweiligen Plattform, die vom LOGICDATA-Produkt (siehe Kapitel 6.1.5) unterstützt wird. Die APIs bilden die Schnittstellen zwischen der App und der Firmware des LOGICDATA-Produkts.

Um die APIs zu testen, muss der Tester beziehungsweise die Testerin zuerst eine Verbindung zwischen Command-Sender und der Test-App herstellen. Nach der Initialisierung müssen alle Übergabeparameter

[46] Vgl. LOGICDATA Elektronic & Software Entwicklungs GmbH 7 (2019), Online-Quelle [21.09.2019].

der zu testenden API eingegeben und im Anschluss abgesendet werden. Damit wird ein JSON-RPC-Request mit den notwendigen Daten an die Test-App gesendet. Diese übergibt die Parameter und ruft die jeweilige API auf. Daraufhin wird die angeforderte Aktion ausgeführt und die Test-App nimmt die Rückgabewerte entgegen und sendet sie wiederrum im JSON-Format an den Command-Sender zurück. Hier werden die empfangenen Daten strukturiert dargestellt. Die Aufgabe des Testers und der Testerin ist es, die Werte mit den zu erwarteten Ergebnissen zu vergleichen. Dieser Test wird mit verschiedenen Übergabeparametern und in verschiedenen Systemzuständen wiederholt. Erst nach erfolgreicher Absolvierung aller Tests kann die Überprüfung der jeweiligen API als bestanden betrachtet werden. Diese Tests müssen für alle APIs auf allen unterstützten Plattformen sowie Plattform-Versionen durchgeführt werden.

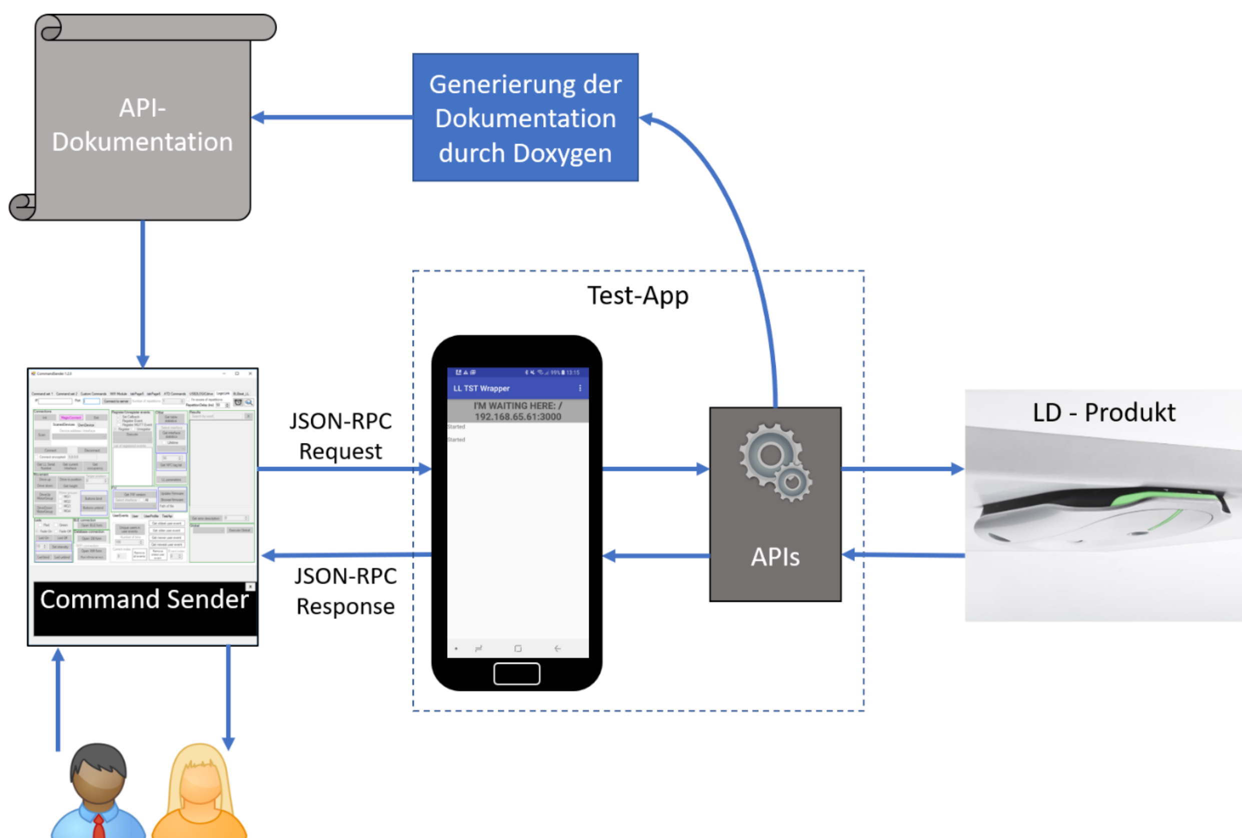

Abbildung 31: Schematische Darstellung des aktuellen Testsystems, Quelle: Eigene Darstellung.

6.3 Probleme der bestehenden Umsetzung

Wie in Kapitel 6.2 beschrieben, kann mit der eigentlichen Entwicklung des Command Senders und der Test-App erst nach der Bereitstellung der API-Dokumentation und der APIs durch das Software-Team begonnen werden. Da die Funktionalität der einzelnen APIs ohne passendes Testsystem kaum überprüfbar ist, können vor Fertigstellung jenes auch keine Tests durchgeführt werden. Das führt in weiterer Folge auch zu einer Verzögerung im Freigabeprozess und erhöhtem Zeitdruck für die Entwickler und Entwicklerinnen des Testsystems. Nachdem der Command Sender und die Test-Apps fertig entwickelt wurden, müssen auch diese getestet werden, um die Wahrscheinlichkeit für durch das Testsystem hervorgerufene Fehlverhalten der APIs zu verringern. Diese können jedoch auch nach Tests nicht zur Gänze ausgeschlossen werden.

Üblicherweise befinden sich im ersten Release Candidate noch Bugs und Logikfehler, mit denen die API-Library noch nicht freigegeben werden kann. Dafür müssen je nach Schwere des Fehlers angemessene Aktionen im Software-Team gesetzt werden. Diese reichen von einfachen Bugfixes bis hin zu großen Änderungen ganzer Softwaremodule. Diese Änderungen wirken sich in weiterer Folge auch auf den Command Sender und die Test-Apps aus, welche manuell umgesetzt werden müssen. Damit entsteht vor der zweiten Testschleife eine weitere Zeitverzögerung, die der Anpassung und dem Test des Testsystems geschuldet ist. Dieser Prozess wiederholt sich bis zur Freigabe der APIs durch das Integration Test Team. Vor allem bei sehr umfangreichen API-Libraries mit sehr vielen Testschleifen entsteht durch diesen Ansatz ein großer Zeitverlust sowie ein erhöhter Entwickler- und Entwicklerinnenbedarf für die Anpassung des Testsystems. Das kann zu Personalengpässen beziehungsweise zu weiteren Verzögerungen in anderen Projekten führen.

Im folgenden Ablaufdiagramm wird der Prozess von der Entwicklung bis zur Freigabe der API-Library vereinfacht dargestellt.

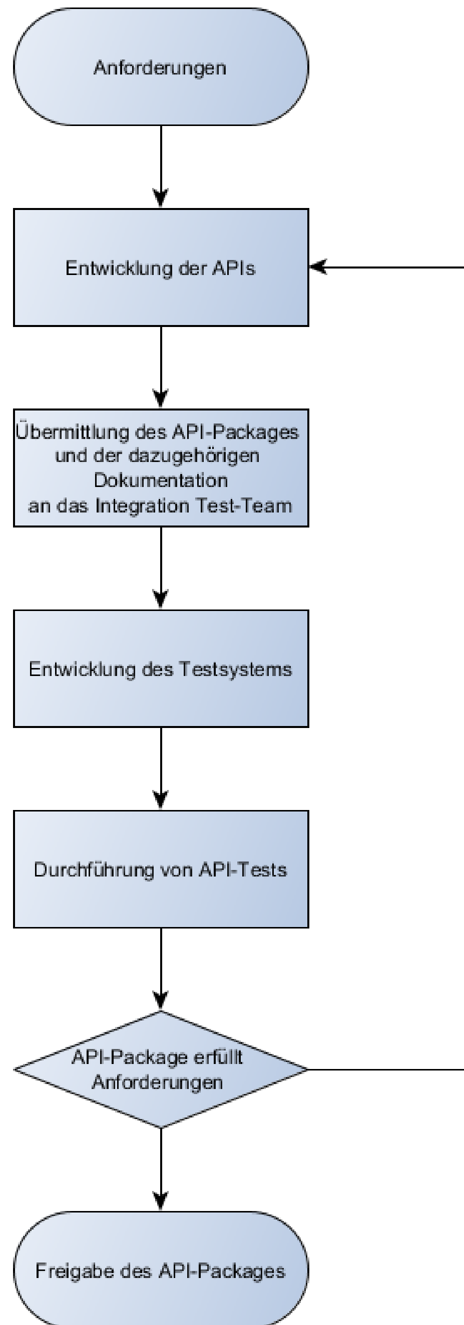

Abbildung 32: Ablaufdiagramm des aktuellen Entwicklungsvorgehens, Quelle: Eigene Darstellung.

7 ARCHITEKTUR DES TESTSYSTEMS MIT AUTOMATISCHER CODEGENERIERUNG

Das Ziel des neuen Testsystems ist es, den Freigabeprozess zu verkürzen, den Entwicklungsaufwand sowie Wartungsarbeiten zu verringern und Fehler, die auf Grund von Bugs im Testsystem aufgetreten sind, zu vermeiden. Diese Punkte sollen durch den Einsatz von Codegenerierung realisierbar sein. Das Problem des alten Testsystems ist, dass bis zur Bereitstellung der APIs und der dazugehörigen Dokumentation nur sehr wenige Elemente entwickelt werden können und deshalb der Großteil des Entwicklungsaufwands an das Ende geschoben wird. Durch den Einsatz von Template-Engines und den dazugehörigen Templates soll es möglich sein die Entwicklung bereits vor Erhalt der APIs voranzutreiben. Dies erfordert zusätzlich eine Anpassung der Architektur des Testsystems, auf welche in diesem Kapitel näher eingegangen wird.

7.1 Systemkomponenten

7.1.1 API-Dokumentation

Die API-Dokumentation dient als Basis für die Entwicklung der einzelnen Systemkomponenten, da sie alle notwendigen Informationen der APIs beinhaltet. Der bestehende Ansatz mit von Doxygen generierten Dokumentationsdateien in Form von beispielsweise HTML-Seiten und Word-Dateien erfüllt zwar den Zweck der Beschreibung der APIs, ist jedoch als Datenaustauschformat weniger geeignet. Aus diesem Grund muss eine Alternative gefunden werden, mit der die Beschreibung als auch der Austausch der Informationen der APIs möglich ist. In Frage kommen dafür bekannte Datenaustauschformate wie XML und JSON. Beide Formate ermöglichen eine kompakte Beschreibung der APIs und eignen sich durch die Plattformunabhängigkeit und einfachen Verarbeitungsmöglichkeiten ideal für dieses Softwarevorhaben. Die Wahl fällt jedoch auf JSON, da die Syntax im Vergleich einfacher aufgebaut ist und dadurch die enthaltenen Informationen für den Menschen lesbarer sind.

JSON API-Dokumentation

Die API-Dokumentation ist eine aus mehreren JSON-Files bestehende Beschreibung der einzelnen APIs. Zur Verbesserung der Übersicht gibt es für jedes Produkt drei Files, die die APIs in essenzielle, gemeinsame und produktspezifische Methoden aufteilt. Bei den essenziellen APIs spricht man von Methoden, die beispielsweise für die Initialisierung, Verbindungsherstellung und Verbindungstrennung notwendig sind und von allen Produkten gemeinsam genutzt werden. Im JSON-File für die gemeinsamen APIs werden alle Methoden geführt, die von mehr als einem Produkt genutzt werden, aber nicht essenziell für die grundlegende Funktion sind. Das dritte File beschreibt die produktspezifischen APIs, die nur vom jeweiligen Produkt genutzt werden. Durch diese Separierung wird die Dokumentation übersichtlicher.

In den jeweiligen JSON-Files entspricht ein JSON-Objekt einer API. In folgender Tabelle werden die definierten Eigenschaften eines Objekts sowie deren Zweck genauer erläutert. Der Aufbau der Objekte im JSON-File wird in Abbildung 33 dargestellt.

Name der Eigenschaft	Beschreibung	Zweck der Eigenschaft
function_name	API-Name	Gibt den Namen der API an.
function_description	API-Beschreibung	Beschreibt die Funktion der API und dient in weiterer Folge als Kommentar zur Verbesserung der Übersicht.
function_name_API	API-Name im API-Package	Gibt den Namen der jeweiligen Methode der API im Package an.
function_name_lib	API-Name in der Bibliothek	Gibt den Namen der jeweiligen Methode der API in der Library an.
function_type	API-Typ	Unterscheidet, ob es sich bei der API um einen Callback oder einer einfachen API handelt.
function_role	Rolle der API	Unterscheidet, ob es sich bei der API um eine essenzielle, eine gemeinsam genutzte oder produktspezifische API handelt.
input_params	Eingabeparameter	Beinhaltet alle Eingabeparameter der API.
output_params	Rückgabeparameter	Beinhaltet alle Rückgabeparameter der API.

Tabelle 4: Beschreibung der Eigenschaften eines JSON-Objekts der Dokumentation, Quelle: Eigene Darstellung.

Der Wert für die Eingabeparameter kann auch leer sein, da nicht jede API über Eingabeparameter verfügt. Zusätzlich kann noch eine Eigenschaft, die die von der API unterstützten Produkte beinhaltet, Bestandteil des Objekts sein. Diese Daten werden in weiterer Folge als Eingabeparameter für die FreeMarker Template-Engine (siehe Kapitel 8.1) und zur Implementierung der grafischen Benutzeroberfläche des Command-Senders (siehe Kapitel 7.1.3.1) genutzt.

```
}, {
  "function_name": "Disconnect",
  "function_description": "Disconnects from currently connected LD device.",
  "function_name_API": "LD_Disconnect",
  "function_name_lib": "LDlib_Disconnect",
  "function_type": "regular",
  "function_role": "essential",

  "input_params": [
  ],

  "output_params": [
    ["LDStatus", "status", "Output status."]
  ]
}, {
  "function_name": "RegisterEvent",
  "function_description": "Makes a registration of a specific event on LD device.",
  "function_name_API": "LD_RegisterEvent",
  "function_name_lib": "LDlib_RegisterEvent",
  "function_type": "regular",
  "function_role": "essential",

  "input_params": [
    ["LDEvent", "event", "Object that defines an event that should be registered on LD device."]
  ],

  "output_params": [
    ["LDStatus", "status", "Output status."]
  ]
}, {
```

Abbildung 33: Aufbau der JSON-Objekte in der API-Dokumentation, Quelle: Eigene Darstellung

7.1.2 API-Libraries

Die zu testenden APIs werden in Form von Libraries vom Software-Team zur Verfügung gestellt. Dabei ist das Format abhängig von der jeweiligen Plattform. Android-Libraries liegen dabei im .aar, iOS-Libraries im .dylib und Windows-Libraries im .dll Format vor. Um die Funktion der APIs testen zu können, werden sie in eine Test-App integriert.

Um auf die APIs zugreifen zu können, werden definierte Datenstrukturen benötigt. Diese müssen in einem passenden Datenformat festgelegt werden. Zur Auswahl steht dabei neben XML und JSON auch Google Protocol Buffer, kurz Protobuf. Alle drei Varianten werden dabei von den für dieses Softwarevorhaben relevanten Programmiersprachen unterstützt. Protobuf ist jedoch wesentlich kompakter hinsichtlich des Speicherplatzbedarfes und ist darüber hinaus schneller in der Verarbeitung der Daten. Zusätzlich wird die CPU bei der Serialisierung und Deserialisierung weniger belastet, was in weiterer Folge eine Ressourcenoptimierung mit sich bringt. Aus diesem Grund fällt die Entscheidung auf Protobuf als Format zur Definition der Datenstrukturen.

Google Protocol Buffers

Bei Google Protocol Buffer handelt es sich um einen effizienten, flexiblen und automatisierten Mechanismus zur Serialisierung von strukturierten Daten. Dabei muss die Struktur der Daten nur einmal festgelegt werden, um mit speziell generiertem Source Code strukturierte Daten von verschiedenen Quellen und verschiedenen Programmiersprachen lesen und schreiben zu können. Die Datenstruktur kann

sogar nachträglich verändert werden, ohne bereits mit dem „alten“ Format kompilierte Programme beeinträchtigen zu müssen. Die Struktur der Protobuf Messages wird in .proto-Dateien definiert. Jede Protobuf Message beinhaltet eine Reihe von Name-Wert-Paaren. Das Format der Nachrichten ist sehr einfach gehalten – jede Message besitzt ein oder mehrere nummerierte Felder mit jeweils einem Namen und einem Wertetyp. Die Wertetypen können zum Beispiel Integer, Gleitkommazahlen, Bool, Strings oder sogar andere Protobuf Messagetypen sein. Nach Definition der Nachrichten kann der Compiler für die jeweilige Programmiersprache für die .proto-Datei ausgeführt werden, um Klassen zu generieren. Diese bieten Zugriff auf die einzelnen Felder sowie Methoden zum Serialisieren und Verarbeiten der Strukturen.[47]

Vom Software Team wird pro Produkt eine .proto-Datei zur Verfügung gestellt. Beim Kompilieren dieser Dateien mit dem protoc-Compiler werden aus den Messages (siehe Abbildung 34) Dateien generiert. Abhängig von der gewünschten Programmiersprache entstehen dadurch Dateien in unterschiedlichen Formaten. Im Fall von Java entsteht dabei zum Beispiel eine .java-Datei, im Fall von C# hingegen eine .cs-Datei. Diese Dateien enthalten die Datenstrukturen, die notwendig sind, um die Schnittstellen beschreiben zu können. Ein Vorteil von Protobuf ist, dass aus einer einzigen .proto-Datei die Datenstrukturen für alle Plattformen in der jeweiligen Programmiersprache generiert werden können. Sie werden in weiterer Folge in die API-Facade (siehe Kapitel 7.1.3.2) für die Nutzung der Datentypen sowie in die Test-Apps (siehe Kapitel 7.1.4) eingebunden, um auf die APIs zugreifen zu können.

```
syntax = "proto3";

option java_package = "at.logicdata.uldapilib";

/* LD API SYSTEM VARS */

message LDByteArray {
    repeated uint32 value = 1;
}

/* LD API TYPES */
message LDStatus {
    enum ld_status_t {
        ld_status_success = 0;
        ld_status_busy = 200; ///<  Busy.
        ld_status_no_response = 201; ///<  No response.
        ld_status_communication_lost = 202; ///<  Communication lost.
        ld_status_memory_allocation_error = 203; ///<  Memory allocation error.
        ld_status_not_connected = 204; ///<  Not connected.
        ld_status_not_initialized = 205; ///<  Not initialized.
        ld_status_already_initialized = 206; ///<  Already initialized.
        ld_status_already_connected = 207; ///<  Already connected.
        ld_status_invalid_parameter = 208; ///<  Invalid parameter.
        ld_status_execution_of_function_not_allowed = 213; ///< Execution of function is not allowed
    }

    ld_status_t status = 1;
}
```

Abbildung 34: Beispiel einer Protobuf Message, Quelle: Eigene Darstellung.

[47] Vgl. Google LLC. (o.J), Online-Quelle [25.09.2019].

7.1.3 Command Sender

Der Command Sender erfüllt im Testsystem mit Codegenerierung die gleiche Funktion wie im Testsystem ohne Codegenerierung. Jedoch besteht der Unterschied im Entwicklungsvorgehen desselben.

7.1.3.1 Grafische Benutzeroberfläche

Die grafische Benutzeroberfläche, kurz GUI, ist die direkte Schnittstelle zwischen Tester oder Testerin und Testsystem. Der User bekommt dadurch die Möglichkeit, die Verbindung zum Server einzuleiten, APIs mit Werten zu versehen, diese aufzurufen und die Ergebnisse zu überprüfen.

Ziel ist es, die grafische Benutzeroberfläche zu großen Teilen generativ zu erstellen. Als Eingabedaten dienen dazu die vom Software-Team zur Verfügung gestellten API-Dokumentationen, die alle notwendigen Informationen beinhalten. Eine Möglichkeit ist, FreeMarker Templates zu entwickeln und die Steuerelemente und den dazugehörigen Code aus diesen zu generieren. Ein weiterer Ansatz ist, die grafische Benutzeroberfläche direkt aus den Eingabedaten zur Laufzeit zu erstellen. Diese Herangehensweise bringt den Vorteil, dass keine zusätzlichen Templates entworfen werden müssen. Bei Änderungen in der Dokumentation muss diese lediglich ausgetauscht und bei Programmstart eingelesen werden. Aus diesem Grund ist dieser Ansatz zu wählen, da damit die Flexibilität gesteigert wird und die Wartungstätigkeiten zwischen den Testschleifen weiter verringert werden können.

Da die Dokumentation im JSON-Format vorliegt, können die Daten einfach verarbeitet werden. Im JSON-File entspricht ein Objekt einer API mit allen Eingabeparametern und Informationen. Diese Daten werden eingelesen und abhängig von den Parametern werden Kontrollelemente für die jeweilige API generiert. Somit kann gewährleistet werden, dass bei Vorliegen der richtigen Dokumentationsdateien die Steuerelemente in der GUI mit diesen übereinstimmen.

Das Design des Command Senders soll sehr übersichtlich und intuitiv gehalten werden. Im Kopf der GUI befinden sich alle für den Verbindungsaufbau notwendigen Steuerelemente. Im Konkreten können die IP-Adresse sowie der Port des Servers mit Textfeldern konfiguriert und mit dem Connect Button die Verbindung hergestellt werden. Dieses Design-Konzept ist sehr stark an dem des bereits bestehenden Command Sender angelehnt. Dadurch soll dem User der Umstieg auf die neue Version so einfach wie möglich gemacht werden. Darunter befindet sich das Konsolenfenster, das die übermittelten Daten grafisch darstellt. Damit bekommen der Tester und die Testerin die Möglichkeit, die rückgegebenen Antworten mit den zu erwartenden Ergebnissen vergleichen zu können. Auch dieses Element hat sich im bestehenden Command Sender bewährt und wird deshalb wieder eingesetzt.

Da die Dokumentation APIs mehrerer Produkte beinhalten kann, ist es sinnvoll, diese zu separieren. Dafür bietet es sich an, für jedes Produkt eine eigene Registerkarte anzulegen, die die Trennung der APIs ermöglicht. Um diese besser gliedern zu können, wird das Fenster des jeweiligen Produkts nochmals in drei Teile unterteilt. Im ersten Teil werden die APIs gelistet, die für die Nutzung unbedingt notwendig und für alle Produkte gleich sind. Der zweite Teil beinhaltet die APIs, die von mehr als einem Produkt genutzt werden, aber nicht essenziell sind. Im letzten Teil können die produktspezifischen APIs ausgewählt und die Parameter dieser gesetzt werden. Das Verändern der Eingabeparameter erfolgt über Steuerelemente, die für den jeweiligen Datentyp angemessen sind. Der Aufruf der APIs erfolgt über einen Button, der mit dem API-Namen beschriftet ist. Zur besseren Abgrenzung werden alle Steuerelemente einer API von einer

GroupBox umschlossen. In der folgenden Abbildung ist der von der JSON API-Dokumentation abgeleitete Command Sender dargestellt.

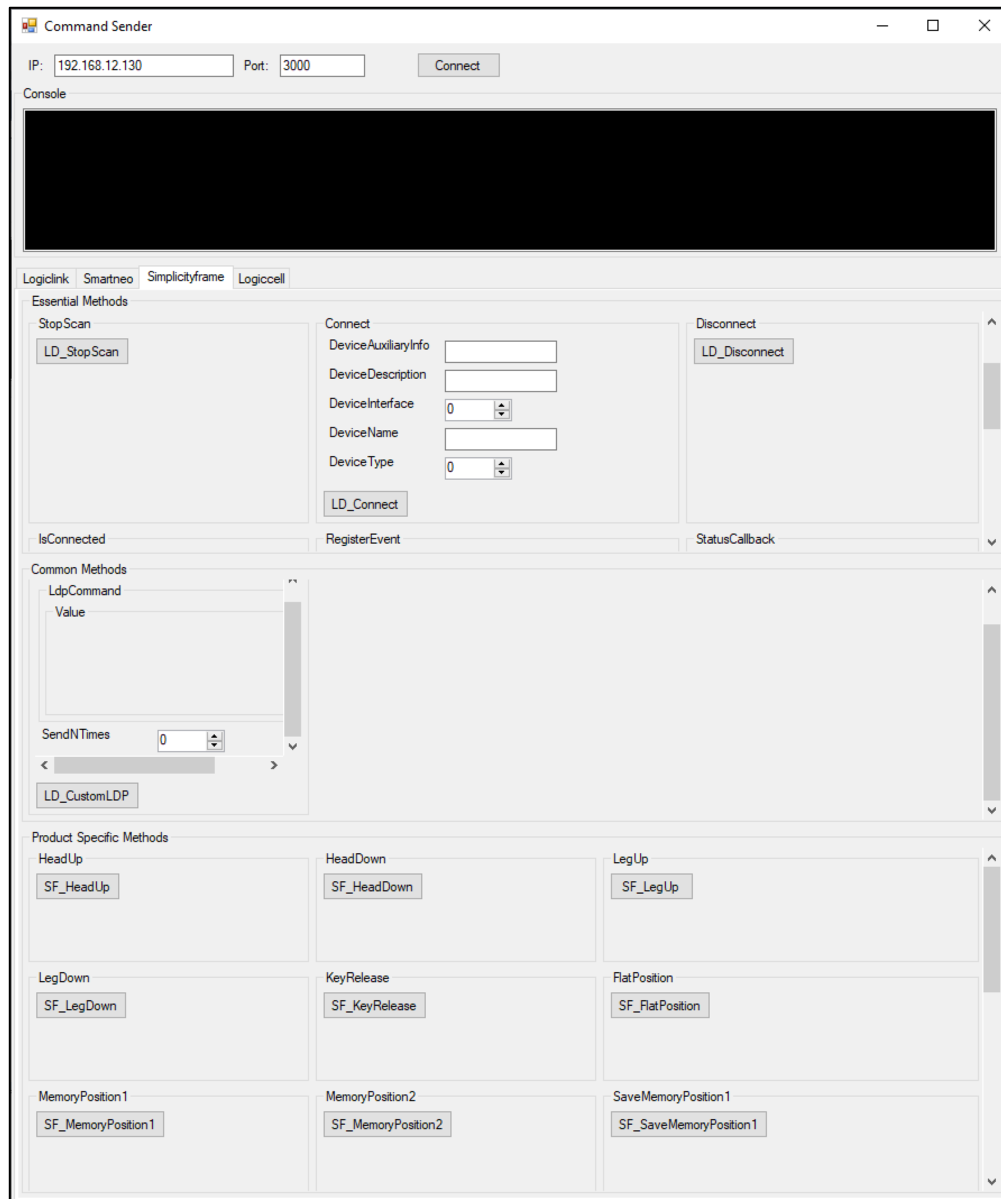

Abbildung 35: Grafische Oberfläche des Command Senders, Quelle: Eigene Darstellung.

7.1.3.2 Schnittstelle zwischen Command Sender und Test-Apps

Für die Kommunikation zwischen Command Sender und Test-Apps muss eine Schnittstelle definiert werden. Als Protokoll zwischen Server und Client wird JSON-RPC 2.0 gewählt, weil es sich im bestehenden System bewährt hat und dadurch große Codeteile wiederverwendet werden können. Für die Verbindungsherstellung müssen zuvor einige Objekte im Code initialisiert und Abhängigkeiten berücksichtigt werden. Diese Tatsache macht den Code unübersichtlich und schwierig zu warten. Außerdem soll diese Schnittstelle zukünftig auch für weitere Testsoftware, zum Beispiel für automatisierte Tests, eingesetzt werden. Aus diesem Grund wird der Einsatz eines Entwurfsmuster, der zum einen die Komplexität der Schnittstelle verringert und zum anderen die Schnittstelle vom restlichen Code entkoppelt,

angedacht. Da diese Themen hauptsächlich die Beziehungen zwischen Klassen und Objekten betreffen, kann die Wahl des Entwurfsmusters auf Strukturmuster eingegrenzt werden. Für diese Anwendung kommt jedoch nur das Facade Entwurfsmuster in Frage, sie übernimmt die Initialisierung der Objekte des Subsystems und vereinfacht gleichzeitig den Zugriff auf dieses für den User. Des weiteren kann sie einfach gewartet werden und bietet die Möglichkeit, sie auch in zukünftiger Testsoftware einzusetzen.

API-Facade

Das Facade Entwurfsmuster ist eine einfache Schnittstelle zur Nutzung eines Systems oder Objekten. Man betrachtet dabei Clients, die auf ein Subsystem mit vielen Elementen und Abhängigkeiten zwischen ihnen zugreifen möchten. Dafür müssen sich die Clients mit verschiedenen Schnittstellen der Elemente befassen und die Funktionsweise dieser verstehen. Aus diesem Grund bauen sich Abhängigkeiten zu verschiedenen Objekten auf und koppeln sich mit den Elementen des Subsystems.

Die Facade steht dabei zwischen den Clients und dem Subsystem (siehe Abbildung 36). Sie kapselt das Subsystem, enthält die Logik zum Arbeiten mit dem Subsystem und ermöglicht dem Client eine vereinfachte Schnittstelle nach außen. Die Facade verteilt die Aufrufe der Clients an das Subsystem. Dadurch können die Clients auf das System zugreifen, ohne die Elemente, deren Abhängigkeiten und Beziehungen zu kennen.[48]

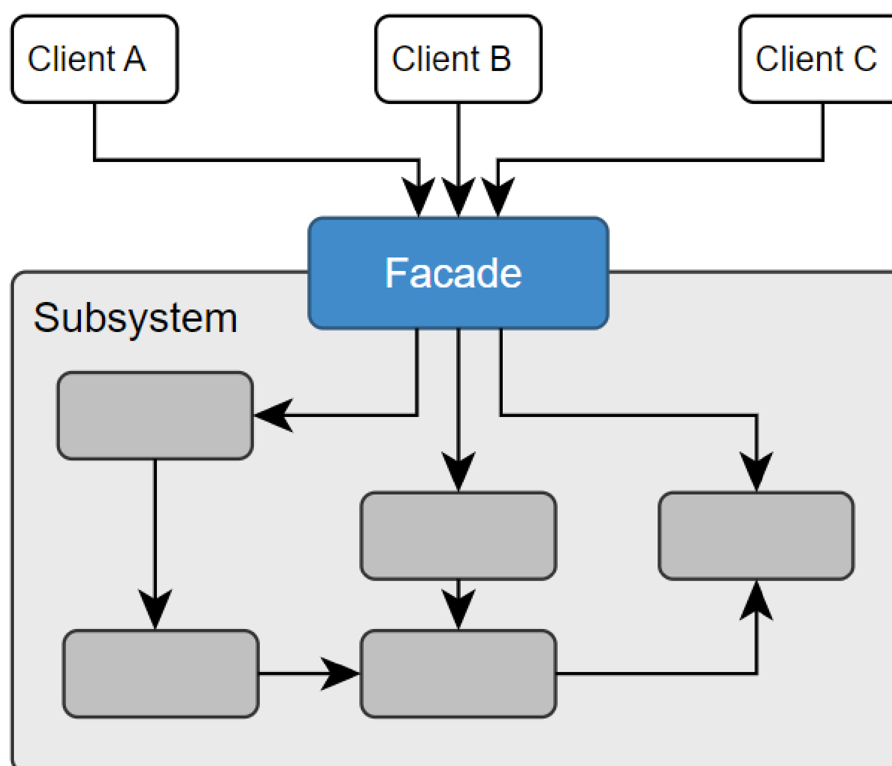

Abbildung 36: Schematische Darstellung des Facade Design Patterns, Quelle: Hauer (2010), Online-Quelle [08.10.2019].

[48] Vgl. Hauer (2010), Online-Quelle [08.10.2019].

Die Rolle des Clients wird im Falle dieses Testsystems vom Command Sender übernommen. Das Subsystem wird von der Implementierung des Websockets und den Test-Apps der verschiedenen Plattformen repräsentiert. Die Aufgabe der API-Facade ist es in erster Linie, die Verbindung mit dem Server, der Test-App, einzuleiten. Des Weiteren müssen die vom Tester oder der Testerin getätigten Eingaben entgegengenommen und in das JSON-RPC-Format für die Übermittlung umgewandelt werden. Der Aufbau der Requests der einzelnen APIs wird mit Hilfe von FreeMarker und dazugehörigen Templates generiert. Als Eingabedaten dienen dazu die JSON-API-Dokumentationen. Die vom Server ankommenden JSON-Responses werden hingegen wieder in das kompaktere Protobuf-Format konvertiert. Dazu werden wieder, wie in Kapitel 7.1.2 beschrieben, die im Protobuf-File definierten Datenstrukturen herangezogen.

7.1.4 Test-Apps

Auch die Test-App unterscheidet sich hinsichtlich der Funktionsweise kaum von den Test-Apps (siehe Kapitel 6.1.4) des Testsystems ohne Codegenerierung. Auch diese Test-Apps fungieren als Server des Server-Client Modells und empfangen die JSON-RPC-Requests mit den notwendigen Parametern vom Command Sender. Die jeweilige API wird mit den jeweiligen Übergabeparametern aufgerufen. Die Antwort jener wird am Display des Smartdevices ausgegeben und in Form einer JSON-RPC Response an den Command Sender zurückgesendet.

Der große Unterschied liegt jedoch beim Entwicklungsvorgehen. Im Gegensatz zu den Test-Apps des bisherigen Testsystems wird der Code nicht ausschließlich manuell implementiert, sondern auch zum Teil automatisch generiert. Statische Elemente der Apps, wie zum Beispiel die grafische Benutzeroberfläche oder Code zur Kommunikation zwischen Command Sender und Test-App, werden manuell implementiert. Für diese Teile des Codes ist es jedoch auch nicht erforderlich, eine Codegenerierungsmethode einzusetzen, da dieser Code üblicherweise nach einmalig erfolgreicher Entwicklung nicht weiter verändert werden muss. Der dynamische Teil der Apps besteht im Grunde genommen aus zwei Teilen. Der erste Teil setzt sich aus den API-Libraries und den dazugehörigen Datenzugriffsklassen, die aus den .proto-Files generiert werden, zusammen. Diese müssen nach Erhalt in die Test-App eingebunden werden. Beim zweiten Teil handelt es sich um automatisch generierte Wrapper, die für den Zugriff auf die APIs notwendig sind. Wrapper ist Code, der andere Codeteile kapselt und verfügbar macht. Die Daten werden vom Command Sender im JSON-RPC Format an die Test-App gesendet. Diese Informationen müssen, bevor sie an die jeweilige API übergeben werden können, noch weiterverarbeitet werden. Die empfangenen Daten werden dafür vom Wrapper in das Protobuf-Format umgewandelt, um die angeforderte API aufzurufen (siehe Abbildung 37). Gleichermaßen werden die Rückgabewerte der API wieder entgegengenommen und für das Rücksenden an den Client vorbereitet. Dafür muss die von der API übermittelte Antwort vor dem Senden vom Protobuf-Format wieder in das JSON-RPC-Format konvertiert werden. Jede API wird dafür von Wrapper-Code umschlossen. Der grundsätzliche Aufbau dieses Codes ist für alle APIs gleich. Sie unterscheiden sich nur in der Anzahl und der Typen der Übergabe- und Rückgabeparameter der APIs. Diese wiederkehrenden Code-Muster sind für Template-basierte Codegenerierung besonders geeignet. Aus diesem Grund können diese Wrapper mit FreeMarker und passenden Templates automatisch generiert werden. Als Eingabedaten für die Template Engine wird die

JSON API-Dokumentation herangezogen, die alle notwendigen Informationen, wie zum Beispiel API-Namen sowie Übergabe- und Rückgabewerte, beinhaltet.

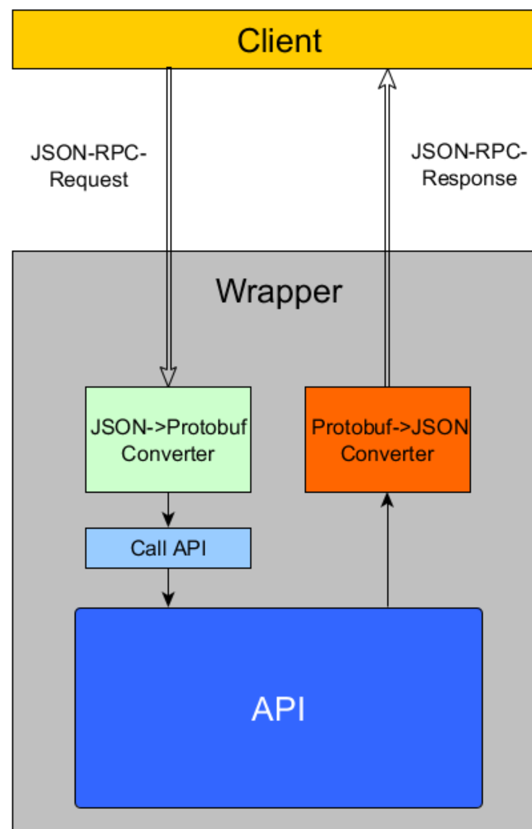

Abbildung 37: Schematische Darstellung der Funktionsweise von Wrapper, Quelle: Eigene Darstellung.

7.2 Entwicklungsvorgehen und Funktionsweise

Die in diesem Kapitel entwickelte Architektur begünstigt es, große Teile des Testsystems bereits vor Erhalt des ersten Relase Candidates zu entwickeln. FreeMarker ermöglicht die Generierung von großen Teilen der Wrapper für die APIs sowie für die API-Facade, die vom Command Sender zur Herstellung der Kommunikation und zur Verarbeitung der eingehenden Datenströme genutzt wird. Da es sich bei der API-Facade und den Wrappern um Elemente handelt, die einem wiederkehrendem Muster folgen, kann bereits nachdem die grundlegende Struktur der APIs im Software-Team festgelegt wurde, auch auf Seiten des Integration Test Team mit der Entwicklung der Templates und der dazugehörigen Java Software begonnen werden. Wie auch beim bestehenden Testsystem kann bereits vorab die Kommunikation zwischen Client und Server entwickelt werden, wobei dabei keine automatische Codegenerierung zum Einsatz kommt. Das Graphical User Interface des Command Senders basiert auf der JSON API-Dokumentation. Jedoch können die Klassen, die zur Generierung der Steuerelemente benötigt werden, bereits im Vorfeld implementiert werden. Nach Erhalt der Dokumentation müssen lediglich die JSON-Files in dem in der Software definierten Pfad abgelegt werden.

Nach Fertigentwicklung der APIs im Software-Team werden die Libraries sowie das .proto-File an das Integration Test Development Team übergeben. Die zur Kommunikation mit den APIs notwendigen

Datenstrukturen werden aus den Protobuf Objekten für die jeweiligen Programmiersprachen generiert. Diese sowie die Library Files müssen in die jeweilige Test-App eingebunden und neu kompiliert werden.

Mit jeder neuen API-Version müssen lediglich die Libraries der jeweiligen Plattformen, die JSON-API-Dokumentation sowie die aus den Protocol Buffer File generierten Dateien ausgetauscht und neu kompiliert werden. Dadurch wird der eigentliche Entwicklungsaufwand während der Freigabephase auf ein Minimum reduziert und gleichzeitig die Fehlerwahrscheinlichkeit im Testsystem verringert.

In folgender Abbildung wird die Architektur des Testsystems mit automatischer Codegenerierung schematisch dargestellt:

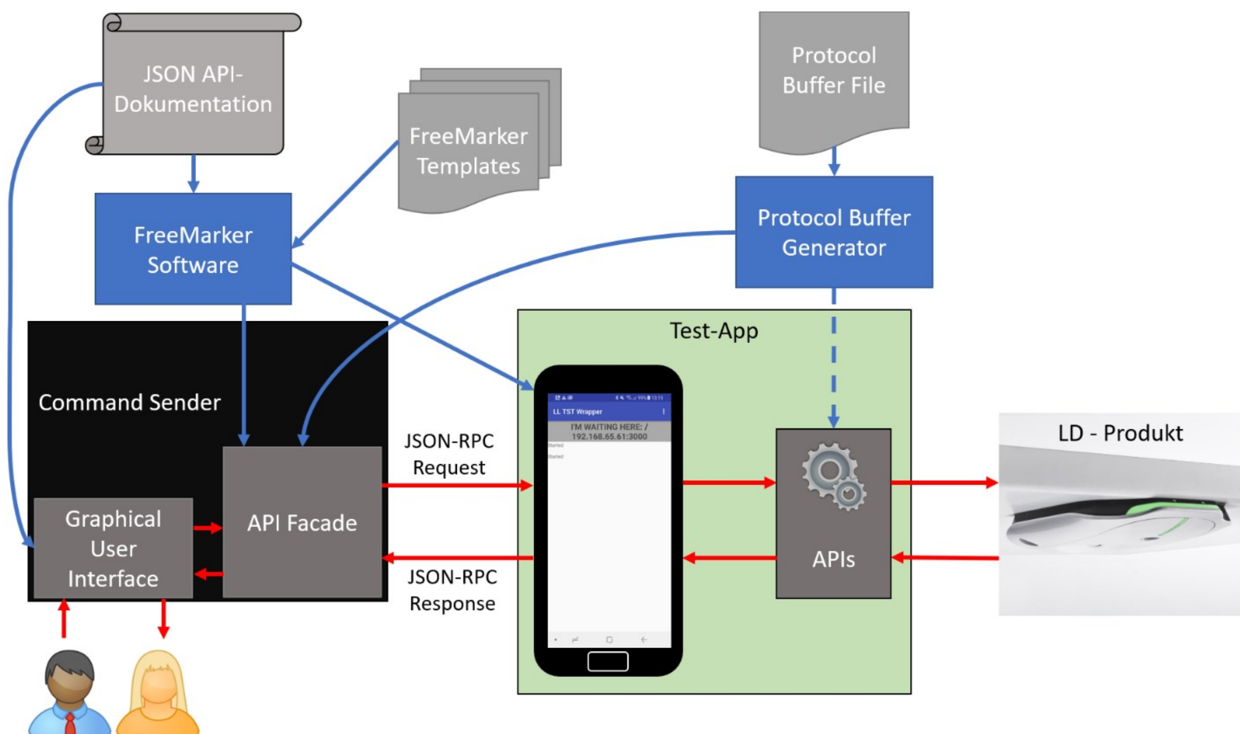

Abbildung 38: Schematische Darstellung des Testsystems mit automatischer Codegenerierung, Quelle: Eigene Darstellung.

8 UMSETZUNG

In diesem Kapitel wird auf die Umsetzung der in Kapitel 7 beschriebenen Softwarekomponenten eingegangen. Dazu wird die Funktionsweise dieser anhand von Codeabschnitten beziehungsweise Beispielen genauer erklärt.

8.1 FreeMarker Software

Die in Kapitel 5.2 für dieses Softwarevorhaben ausgewählte Template-Engine FreeMarker wird von der Eclipse IDE nach Installation des Plug-Ins FreeMarker IDE from JBoss Tools aus dem Eclipse Marketplace unterstützt. Dem User stehen damit unter anderem Syntax Highlighting und Codevervollständigung für die Entwicklung von Templates zur Verfügung. Darüber hinaus muss die FreeMarker.jar Library zum angelegten Java-Projekt hinzugefügt werden.

Die FreeMarker Software wird zur Generierung der API-Wrapper der jeweiligen Test-Apps für die verschiedenen Plattformen sowie für Teile der API-Facade eingesetzt. Als Eingabedaten dienen dabei immer die in Kapitel 7.1.1 beschriebenen API-Dokumentationen. Die Generierung muss für jedes Ausgabefile separat geschehen, was bei einer größer werdenden Anzahl an Produkten beziehungsweise Versionen sehr aufwändig werden kann. Aus diesem Grund ist es ein Bestreben, die Generierung aller Files in einer zentralen Software zu ermöglichen. Da FreeMarker auf Java basiert ist es naheliegend, eine Java-Software um die Template-Engine zu entwickeln. Mit dieser Software soll es möglich sein, verschiedene Eingabedaten und Templates einlesen zu können und damit die jeweiligen Ausgabedaten zu generieren.

8.1.1 Java-Software als Generator

Um die zentrale Generierung der einzelnen Files für die verschiedenen Plattformen zu ermöglichen, wird eine Software mit dem Namen AutoWrapper entwickelt. Als User Interface wird eine Konsole gewählt, da der Software nur wenige Argumente übergeben werden müssen und es keiner grafischen Ausgabe bedarf. Der AutoWrapper beinhaltet alle Generatoren für alle unterstützten Programmiersprachen. Der User muss lediglich die Eingabedaten, die gewünschte Programmiersprache, das passende Template und den Pfad und Namen der Ausgabedatei definieren. Des Weiteren können mit der Angabe von weiteren Argumenten zusätzliche Generierungs-Einstellungen, die sich auf programmiersprachenabhängige Eigenschaften und auf interne Funktionen beziehen, getätigt werden. Der User wird durch einen beim Start angezeigten Text bei der Eingabe des Kommandos zur Generierung unterstützt (siehe Abbildung 39).

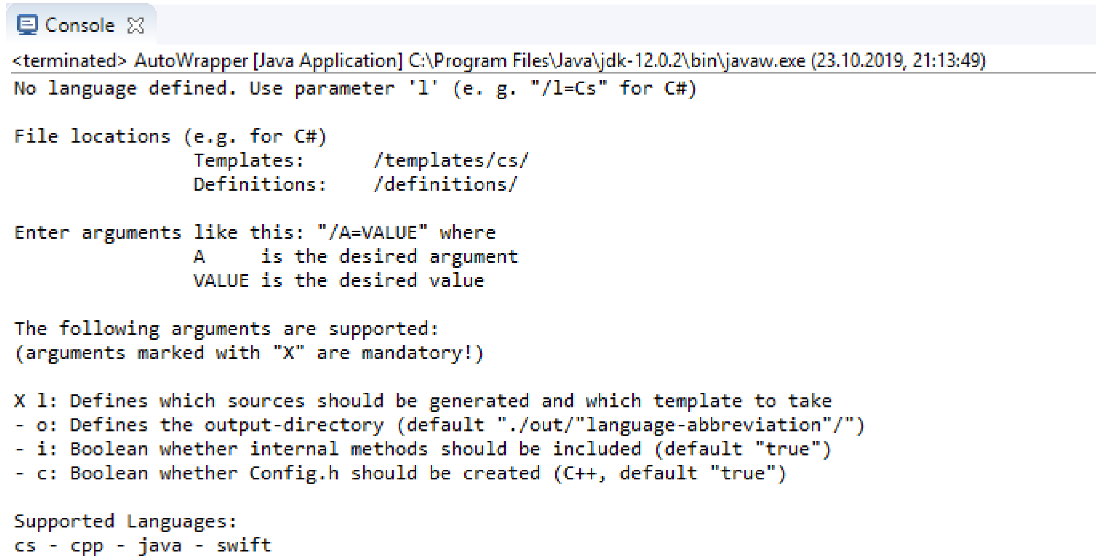

Abbildung 39: Darstellung der Konsolenanwendung des Autowrappers, Quelle: Eigene Darstellung.

Wie in Abbildung 39 ersichtlich, werden die Programmiersprachen C#, C++, Java und Swift vom Autowrapper unterstützt. In C# werden die Klassen, die von der API-Facade zur Erstellung der JSON-RPC-Requests genutzt werden, generiert. In den Sprachen C++, Java und Swift werden die API-Wrapper für die jeweiligen Plattformen Windows, Android und iOS erstellt.

Für die Generierung der oben genannten Codeteile wird je eine Java-Klasse pro Programmiersprache entwickelt. Zusätzlich wird eine Java-Superklasse „LanguageDefinitionBase“ definiert, die die gemeinsamen Methoden und Attribute der spezifischen Klassen hält. Diese Elternklasse beinhaltet beispielsweise die notwendige FreeMarker-Konfiguration, wie Versionsnummer und Encoding-Format der Templates. Darüber hinaus werden die Eingabedaten, in Form von JSON-Files, in eine Map eingelesen. Dadurch kann auf die JSON-Objekte, die jeweils eine API repräsentieren, sowie auf die einzelnen Schlüssel-Wert-Paare zugegriffen werden. In den abgeleiteten Java-Klassen für die Generierung in die verschiedenen Programmiersprachen wird die Basisklasse um plattformspezifische Methoden und Attribute erweitert, die für das jeweilige Generierungsvorhaben benötigt werden. Des Weiteren werden die jeweiligen Templates definiert und die Ausgabedirectory festgelegt. In der execute-Methode (siehe Abbildung 40) wird die Methode processTemplate aufgerufen, die alle für die Generierung notwendigen Parameter übergeben bekommt und die Generierung des Codes ausführt.

```
@Override
public void execute() throws Exception {
    Configuration cfg = getConfiguration();
    Map<String, Map<String, Object>> definitionMap = processDefinitions();

    // create folder
    new File(outputDirectory).mkdirs();

    // process LDInterface.java
    processTemplate(cfg, definitionMap.get(ALL_FUNCTIONS), templatesLocation + templateNameLDInterface,
            outputDirectory + fileNameLDInterface);

    // process RPCHandler.java
    processTemplate(cfg, definitionMap.get(ALL_FUNCTIONS), templatesLocation + templateNameRPCHandler,
            outputDirectory + fileNameRPCHandler);

    // process LDApi.java
    processTemplate(cfg, definitionMap.get(ALL_FUNCTIONS), templatesLocation + templateNameLDApi,
            outputDirectory + fileNameLDApi);

    System.out.println("\n" + "Source-files have been put into \"" + outputDirectory + "\"");
}
```

Abbildung 40: Execute-Methode der Java-Subklasse, Quelle: Eigene Darstellung.

8.1.2 Templates

Die Templates als Kernelement der Template-basierten Codegenerierung sind ein essenzieller Bestandteil der Template-Engine FreeMarker. Für jedes generierte Ausgabe-File muss ein Template geschrieben werden. Da mit dieser Methode ausschließlich Text generiert werden kann, muss der Entwickler oder die Entwicklerin neben der Kenntnis der Template-Sprache auch über Erfahrung in der Programmiersprache, in der der Code generiert werden soll, verfügen. Der im Template-File geschriebene Code wird zwar von der Entwicklungsumgebung hinsichtlich der Richtigkeit der Syntax der Templatesprache überprüft, jedoch nicht auf Fehler im Code der jeweiligen Programmiersprache, da dieser von der Template-Engine nur als Text interpretiert wird. Diese Fehler würden selbst zum Zeitpunkt der Generierung des Ausgabefiles nicht entdeckt werden, jedoch bei Integration des Codes in die Softwarekomponente zu Kompilierungsfehlern oder Fehlverhalten führen. Aus diesem Grund werden bei der Entwicklung von Templates intensive Code-Reviews durchgeführt.

Die folgende Abbildung zeigt das Template-File, das für die Generierung der C++ Header Files eingesetzt wird. Diese Header Files beinhalten die Funktionsdeklarationen der API-Wrapper für die Windows Plattform. Das Ziel dabei ist es, mit einem einzigen Template die Header-Files aller produktspezifischen sowie allgemeinen API-Wrapper zu generieren. Möglich wird das zum einen, weil die Dokumentation der APIs bereits produktspezifisch auf mehreren Files aufgeteilt vorliegt, und zum anderen, weil durch den Einsatz von if-Anweisungen in den Templates bedingte Anweisungen beziehungsweise Verzweigungen gemacht werden können. Des Weiteren ermöglicht das wiederkehrende Codemuster in den aus der Generierung resultierenden Header-Files ein kompaktes Design des Templates.

```
#ifndef CMAKE_GENERATED
#include "Config.h"
#endif
<#-- device_specific header files have one ifdef at the top -->
<#if !definitionName?matches("Common") && !definitionName?matches("Essential")>
#ifdef ${definitionName?lower_case}
</#if>

#include <string>
#include "LDTypes.pb.h"
<#list functions as f>

    <#-- each common or internal method has its own ifdef with all affected devices -->
    <#if f.function_role?matches("common") || f.function_role?matches("internal")>
        <#list f.affected_devices>
        #ifdef <#t>
            <#items as device>
                <#t>${device} <#sep> || </#sep>
            </#items>
        </#list>

    </#if>
        // ${f.function_description} <#lt>
    <#if f.input_params?has_content && !f.function_type?matches("callback")>
        extern std::string ${f.function_name_API}_Wrap(std::string inputString); <#lt>
    <#elseif !f.input_params?has_content && !f.function_type?matches("callback")>
        extern std::string ${f.function_name_API}_Wrap(void); <#lt>
    <#elseif f.function_type?matches("callback")>
        extern std::string ${f.function_name_API}_Wrap(std::string input); <#lt>
        extern void ${f.function_name?lower_case}(${f.output_params[0][0]}& input); <#lt>
    </#if>
        <#-- each common or internal method has its own endif, also the last device_specific method has an endif-->
    <#if f.function_role?matches("common") || f.function_role?matches("internal")>
        #endif <#lt>

    </#if>
</#list>
<#-- device_specific header files have one endif at the botton -->
<#if !definitionName?matches("Common") && !definitionName?matches("Essential")>

#endif
</#if>
```

Abbildung 41: Template-File zur Generierung von C++ Header-Files, Quelle: Eigene Darstellung.

Das Ergebnis der Generierung wird in folgender Abbildung anhand eines Codeausschnitts des generierten Header-Files Common.h, das die Funktionsdeklarationen der produktübergreifenden APIs beinhaltet, dargestellt. Die aus der API-Dokumentation entnommene Beschreibung der API wird dabei zur Verbesserung der Übersicht als Kommentar hinzugefügt. Die Funktionsdeklarationen werden abhängig von Art der API und den Übergabeparametern aus dem Template generiert.

```
#ifndef CMAKE_GENERATED
#include "Config.h"
#endif

#include <string>
#include "LDTypes.pb.h"

#ifdef logiclink  || smartneo
// Drives up using a LD device.
extern std::string LD_DriveUp_Wrap(void);
#endif


#ifdef logiclink  || smartneo
// Drives down using a LD device.
extern std::string LD_DriveDown_Wrap(void);
#endif


#ifdef logiclink  || smartneo
// Move table to wanted position.
extern std::string LD_MoveToPosition_Wrap(std::string inputString);
#endif


#ifdef logiclink  || smartneo
// Gets height of table.
extern std::string LD_GetHeight_Wrap(void);
#endif


#ifdef smartneo  || simplicityframe
// Sends a defined custom LDP command to a connected LD device.
extern std::string LD_CustomLDP_Wrap(std::string inputString);
#endif


#ifdef logiclink  || smartneo
// Returns a height.
extern std::string LD_SetHeightCb_Wrap(std::string input);
extern void heightcallback(LDHeightData& input);
#endif
```

Abbildung 42: Codeausschnitt des generierten C++ Headers, Quelle: Eigene Darstellung.

8.2 Protocol Buffer

Das Protocol Buffer File, das die Datenstrukturen zur Beschreibung der APIs beinhaltet, liegt im .proto-Format vor. Die Datenzugriffsklassen werden dabei in Protocol Buffer Messages definiert. Um diese nutzen zu können, muss die .proto Datei zuvor von einem Compiler in die richtige Sprache übersetzt werden. Diese werden für viele verschiedene Programmiersprachen und Plattformen zur Verfügung gestellt. Darunter finden sich auch die für diese Masterarbeiten relevanten Programmiersprachen C#, C++ und Java. Zur Kompilierung in Swift muss ein zusätzliches Add-On zum von Google entwickelten proto-Compiler hinzugefügt werden. Der protoc kann kostenlos von GitHub gedownloadet werden.

Um die .proto-Datei kompilieren zu können, muss der Proto-Compiler über die Command-Line aufgerufen werden und folgende Informationen übergeben bekommen:

- Verzeichnis der .proto-Quelldatei
- Programmiersprache nach der kompiliert werden soll
- Verzeichnis der generierten Ausgabedatei
- Name der .proto-Quelldatei

In folgender Abbildung wird der Aufruf des Compilers für die Programmiersprachen C#, Java und C++ dargestellt.

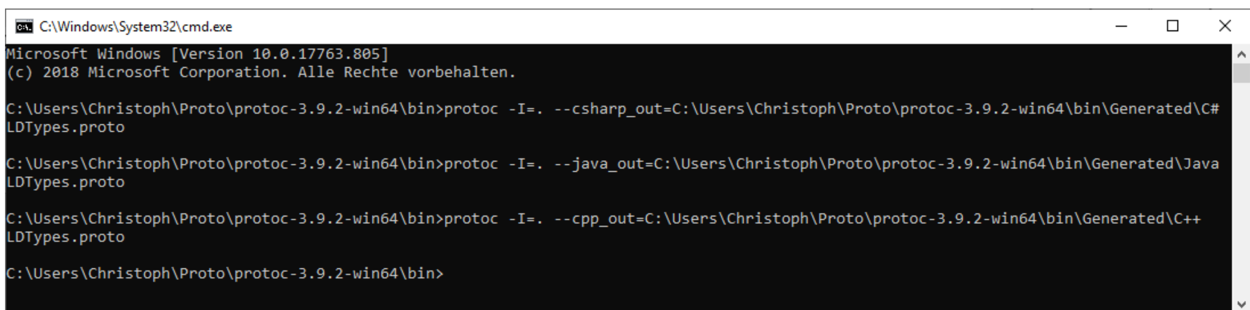

Abbildung 43: Aufruf des Proto-Compilers über die CMD-line, Quelle: Eigene Darstellung.

Um die Datenzugriffsklassen für Swift generieren zu können, muss der proto-Compiler auch für das MacOS Betriebssystem gedownloadet und um das Swift Protobuf Add-On erweitert werden. Das kann zum einen manuell erfolgen, zum anderen über den Paketmanager Homebrew für macOS. Nach Installation desselben muss lediglich ein Installationskommando für das Swift Protobuf Add-On im Terminal aufgerufen werden. Danach kann der Compiler ähnlich wie unter Windows genutzt werden.

In folgender Abbildung wird der Aufruf des Swift Compilers unter macOS dargestellt:

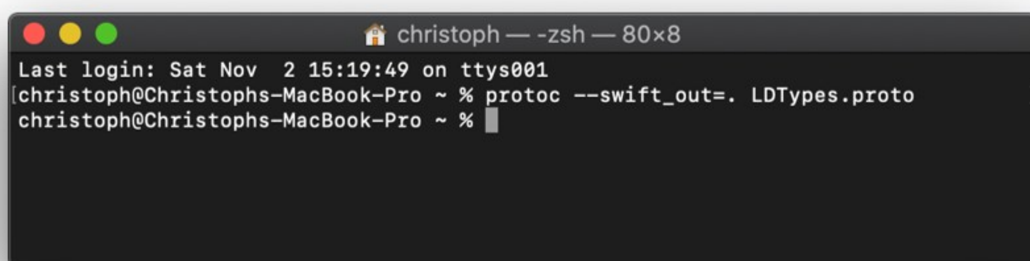

Abbildung 44: Aufruf des Proto-Compilers für Swift über den macOS Terminal, Quelle: Eigene Darstellung.

Die kompilierten Dateien werden in dem definierten Ausgabeordner abgelegt. Diese liegen je nach Programmiersprache in verschiedenen Formaten vor – für C# im .cs, für Java im .java, für C++ im .cc und für Swift im .swift Format. Zusätzlich wird für C++ ein dazugehöriges Header-File mit den Funktionsdeklarationen generiert. In den generierten Files sollten keine Änderungen durchgeführt werden, da ansonsten der Konsens zwischen Eingabedaten und Ausgabedaten verloren geht. Um vom proto-

Compiler generierte Files einfacher von anderen unterscheiden zu können, werden diese mit einem .pb vor der Dateiendung gekennzeichnet.

In der folgenden Abbildung wird der aus einer Protobuf Message generierte Code für die Programmiersprache C# dargestellt. Die entsprechende Protobuf Message findet sich in Abbildung 34. Dabei handelt es sich um eine Deklaration des ld_status_t Enumerator, der den Status des LOGICDATA-Produkts beschreibt. Zur Verbesserung der Lesbarkeit werden die Deklarationen dieser Enumeratoren in partielle Klassen aufgeteilt.

```
#region Nested types
/// <summary>Container for nested types declared in the LDStatus message type.</summary>
[global::System.Diagnostics.DebuggerNonUserCodeAttribute]
public static partial class Types {
  public enum ld_status_t {
    [pbr::OriginalName("ld_status_success")] LdStatusSuccess = 0,
    /// <summary>
    ////&lt;  Busy.
    /// </summary>
    [pbr::OriginalName("ld_status_busy")] LdStatusBusy = 200,
    /// <summary>
    ////&lt;  No response.
    /// </summary>
    [pbr::OriginalName("ld_status_no_response")] LdStatusNoResponse = 201,
    /// <summary>
    ////&lt;  Communication lost.
    /// </summary>
    [pbr::OriginalName("ld_status_communication_lost")] LdStatusCommunicationLost = 202,
    /// <summary>
    ////&lt;  Memory allocation error.
    /// </summary>
    [pbr::OriginalName("ld_status_memory_allocation_error")] LdStatusMemoryAllocationError = 203,
    /// <summary>
    ////&lt;  Not connected.
    /// </summary>
    [pbr::OriginalName("ld_status_not_connected")] LdStatusNotConnected = 204,
    /// <summary>
    ////&lt;  Not initialized.
    /// </summary>
    [pbr::OriginalName("ld_status_not_initialized")] LdStatusNotInitialized = 205,
    /// <summary>
    ////&lt;  Already initialized.
    /// </summary>
    [pbr::OriginalName("ld_status_already_initialized")] LdStatusAlreadyInitialized = 206,
    /// <summary>
    ////&lt;  Already connected.
    /// </summary>
    [pbr::OriginalName("ld_status_already_connected")] LdStatusAlreadyConnected = 207,
    /// <summary>
    ////&lt;  Invalid parameter.
    /// </summary>
    [pbr::OriginalName("ld_status_invalid_parameter")] LdStatusInvalidParameter = 208,
    /// <summary>
    ////&lt; Execution of function is not allowed
    /// </summary>
    [pbr::OriginalName("ld_status_execution_of_function_not_allowed")] LdStatusExecutionOfFunctionNotAllowed = 213,
  }

}
#endregion
```

Abbildung 45: Generierter Enumerator in C#, Quelle: Eigene Darstellung.

8.3 Test-App

Die Test-Apps der jeweiligen Plattformen dienen zum einen als Server der Server-Client-Architektur des Testsystems und zum anderen halten sie alle APIs, die für das jeweilige Produkt beziehungsweise für den Release relevant sind. In diesem Kapitel wird die Implementierung der Android Test-App betrachtet. Die iOS-Version der Test-App unterscheidet sich funktionell nur unwesentlich von der Android-Version. Optisch differenzieren sich die beiden Versionen nur in der plattformabhängigen unterschiedlichen Darstellung von Steuerelementen. Bei der Version für das Windows Betriebssystem handelt es sich um ein Konsolenprogramm in C++, dass sich auf die Funktion als Server und mit der Ausgabe der empfangenen und gesendeten Nachrichten beschränkt.

8.3.1 Grafische Benutzeroberfläche

Die grafische Benutzeroberfläche der Test-Apps ist sehr einfach gehalten. Beim Öffnen der Applikation wird gleichzeitig der Server am jeweilig definierten Port gestartet. Der User kann diesen sowie die IP-Adresse des Devices am Bildschirm des Smartdevices ablesen (siehe Abbildung 46, links). Darunter befindet sich ein Textfeld, das sich über den restlichen Screen erstreckt. Dieses Textfeld wird zur Darstellung von eingehenden JSON-RPC-Requests und ausgehenden JSON-RPC-Responses genutzt. Dadurch können etwaige Fehler in der Kommunikation zwischen Command Sender und Test-App einfacher detektiert werden. Zur besseren Übersicht verfügt das Textfeld über eine ScrollView und zusätzlich werden die Daten JSON-formatiert dargestellt. Bevor jedoch die Tests durchgeführt werden können, muss eine Verbindung zwischen App und LOGICDATA-Produkt hergestellt werden. Abhängig vom Produkt kann diese Verbindung via WiFi oder Bluetooth erfolgen. Diese Suche kann im Overflow-Menü der Android-App getriggert werden (siehe Abbildung 46, Mitte). Des Weiteren können im Menü der Port des Web Socket Servers geändert, sowie ein Reset des Servers ausgelöst werden. Als zusätzliches Test-Feature verfügt die App über das sogenannte GUI Window (siehe Abbildung 46, rechts), das dem User ermöglicht, Basic-APIs aufzurufen. Damit können zum Beispiel Fahrkommandos ausgesendet, oder LEDs ein- und ausgeschaltet werden. Die Ergebnisse dieser APIs sind sehr einfach zu kontrollieren und bieten eine einfache Möglichkeit, die Basisfunktionalitäten zu überprüfen. Da sich mehrere LOGICDATA-Produkte im selben Netzwerk befinden können, kann jedes separat per IP-Adresse angesprochen werden.

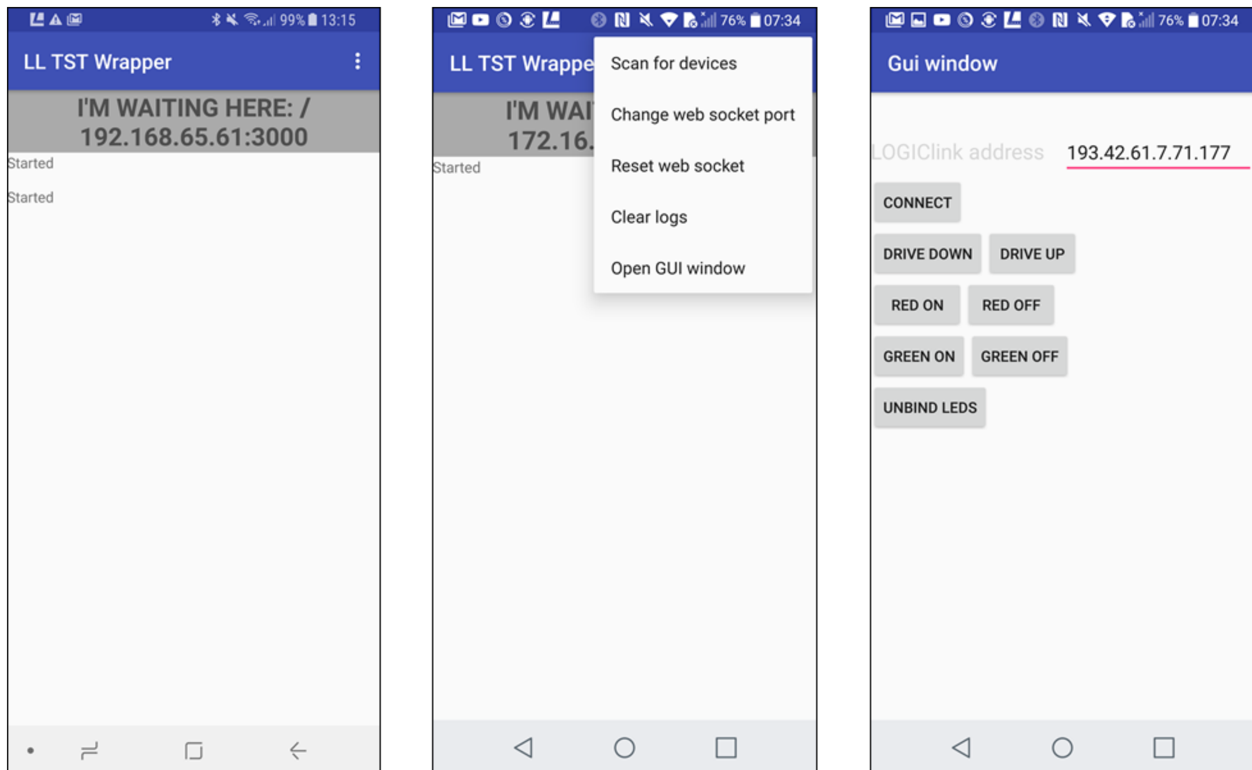

Abbildung 46: Darstellung der Android Test-App, Quelle: Eigene Darstellung.

8.3.2 Wrapper

Wrapper sind wie bereits in Kapitel 7.1.4 erwähnt, notwendig, um die vom Command Sender empfangenen JSON-RPC-Requests entgegenzunehmen und die darin enthaltenen Daten in das Protobuf-Format umzuwandeln. Danach wird die entsprechende API aufgerufen und die Rückgabewerte vom Wrapper in das JSON-RPC-Format konvertiert.

Zur Entwicklung der Wrapper wird jeweils ein Template pro Plattform entwickelt. Aus diesen Templates soll pro API eine Methode generiert werden, die diese kapselt und verfügbar macht. In folgender Abbildung wird ein Teil des Templates, das zur Generierung der Java-Wrapper für Android eingesetzt wird, dargestellt. Prinzipiell muss bei den APIs zwischen einfachen APIs und Callbacks unterschieden werden. Ein Callback ist eine Funktion, die als Argument an eine andere Funktion übergeben wird und deren Ausführung nach Eintritt eines Ereignisses ausgeführt wird[49]. Callbacks werden mit einem Cb am Ende des Namens gekennzeichnet, um sie leichter von den einfachen APIs unterscheiden zu können. Des Weiteren wird die Information, ob es sich bei der jeweiligen API um einen Callback handelt, in der JSON-API-Dokumentation mit der Eigenschaft function_type übermittelt. Im Template wird diese Eigenschaft zur Unterscheidung herangezogen, um die Wrapper-Methode zu generieren.

[49] Vgl. Jindal (o.J.), Online-Quelle [27.10.2019].

```
    // ${f.function_description}
    @Override
<#if !f.function_type?matches("callback")>
    public String ${f.function_name_API}(<#if f.input_params?has_content>String inputString</#if>) {
        try {
            if (ldApi == null)
                ldApi = at.logicdata.uldapilib.LDApi.create(MainActivity.appContext);
            <#if f.input_params?has_content>
            ${f.input_params[0][0]} input = (${f.input_params[0][0]}) ProtobufToJson.msgFromJson(inputString, ${f.input_params[0][0]}.class);
            ${f.output_params[0][0]} output = ldApi.${f.function_name_API}(input);
            <#else>
            ${f.output_params[0][0]} output = ldApi.${f.function_name_API}(<#if f.function_name_API?matches("LD_Init")>MainActivity.appContext</#if>);
            </#if>
            return ProtobufToJson.jsonFromMsg(output);
        }
        catch (Exception e)
        {
            return "{\"error\": 2000}";
        }
    }
<#else>
    public String ${f.function_name_API}(String input) {
        if (input.compareTo("true") == 0) {
            ${f.function_name} callback = new ${f.function_name}() {
                @Override
                public void ${f.function_name?uncap_first}(${f.output_params[0][0]} input) {
                    try {
                        String callbackResponse = "{\"isCallback\":1," +
                                "\"Method\":\" ${f.function_name} \"," +
                                "\"result\":" + ProtobufToJson.jsonFromMsg(input) + "}";
                        MainActivity.sendResponse(callbackResponse);
                    } catch (Exception e)
                    {
                        MainActivity.sendResponse("{\"error\": 2000}");
                    }
                }
            };
            ldApi.${f.function_name_API}Listener(callback);
        }
        else
        {
            ldApi.${f.function_name_API}Listener(null);
        }

        return "{\"status\": 0}";
    }
```

Abbildung 47: Codeausschnitt des Templates zur Generierung des Java-Wrappers, Quelle: Eigene Darstellung.

In folgender Abbildung wird der generierte Code der Wrapper für eine einfache API sowie eines Callbacks dargestellt. Im oben dargestellten Callback-Wrapper wird ein Objekt der Klasse HeightCallback erzeugt. Dieses Objekt wird dem Callback-Listener übergeben. In der darin aufgerufenen Methode werden die Eingabedaten in das JSON-Format konvertiert und in der MainActivity als Callback Response zurückgesendet.

In der unteren Wrapper-Methode der einfachen API wird diese aufgerufen und der Rückgabewert im Protobuf-Format wird mit der Methode jsonFromMsg in das JSON-Format umgewandelt.

Die Rückgabewerte der Wrapper-Methoden werden von einer weiteren Methode entgegengenommen und an den Command Sender zurückgesendet.

```
// Returns a height.
@Override
public String LD_SetHeightCb(String input) {
    if (input.compareTo("true") == 0) {
        HeightCallback callback = new HeightCallback() {
            @Override
            public void heightCallback(LDHeightData input) {
                try {
                    String callbackResponse = "{\"isCallback\":1," +
                            "\"Method\":\" HeightCallback \"," +
                            "\"result\":" + ProtobufToJson.jsonFromMsg(input) + "}";
                    MainActivity.sendResponse(callbackResponse);
                } catch (Exception e)
                {
                    MainActivity.sendResponse("{\"error\": 2000}");
                }
            }
        };
        ldApi.LD_SetHeightCbListener(callback);
    }
    else
    {
        ldApi.LD_SetHeightCbListener(null);
    }

    return "{\"status\": 0}";
}

// Function for connection to LD device.
@Override
public String LD_Init() {
    try {
        if (ldApi == null)
            ldApi = at.logicdata.uldapilib.LDApi.create(MainActivity.appContext);
        LDStatus output = ldApi.LD_Init(MainActivity.appContext);
        String returnString = ProtobufToJson.jsonFromMsg(output);
        return returnString;
    }
    catch (Exception e)
    {
        return "{\"error\": 2000}";
    }
}
```

Abbildung 48: Codeausschnitt eines generierten Java-Wrappers für Android, Quelle: Eigene Darstellung.

8.4 Command Sender

8.4.1 API-Facade

Die API-Facade als Teil des Command Senders dient in erster Linie zum Aufbau der Kommunikation mit der Test-App, die als Server fungiert, und wird als .DLL in das Command Sender Projekt eingebunden. Nachdem der Server gestartet wurde, kann der Command Sender über die API-Facade eine Verbindung herstellen. Dafür muss neben der IP-Adresse des Servers auch der Port bekannt sein. Folgend kann der Tester oder die Testerin über die grafische Benutzeroberfläche etwaige Übergabeparameter der APIs bestimmen und diese durch Drücken des jeweiligen Buttons aufrufen. Dabei werden die Daten an die API-Facade weitergegeben. Eine weitere Aufgabe der API-Facade ist es, diese übergebenen Daten in das JSON-RPC 2.0 Format zu konvertieren. Der Grund dafür liegt darin, dass die JSON-RPC-Server-Client Architektur bereits vollständig in der ursprünglichen Version implementiert wurde und deshalb

wiederverwendet werden kann. Diese JSON-Requests werden von der API-Facade an den Server gesendet. Die Struktur der Requests für die einzelnen APIs wird über FreeMarker Templates, die die JSON API-Dokumentation als Eingabedaten erhalten, generiert. Die vom Server zurückgesendeten JSON-Responses werden von der API-Facade entgegengenommen und in weiterer Folge in der grafischen Benutzeroberfläche des Command Senders ausgegeben.

8.4.1.1 Verbindungsaufbau und Initialisierung

Zu Beginn muss über die API-Facade eine Verbindung mit dem Server hergestellt werden. Wie bereits im Abschnitt zuvor erwähnt, können große Teile des Codes für die Client-Server-Kommunikation vom bereits bestehenden Teil übernommen werden. In Abbildung 49 wird die Verbindungsherstellung über die bereits bestehende Websocket Server Klasse sowie die Instanziierung der Klasse LL_Actions, zur Vorbereitung der JSON-RPC-Requests dargestellt. Dafür wird eine neue Instanz der Klasse WS_ConnectionActions erschaffen. Durch Aufruf der Methode WS_Init, die die IP-Adresse und Port des Servers übergeben bekommt, wird die Initialisierung durchgeführt und die Verbindung zum Websocket Server hergestellt. Danach wird eine Instanz der Template generierten Klasse LL_Actions erzeugt, die die Struktur der JSON-RPC-Requests der einzelnen APIs beinhaltet.

```
public static void CreateApiInstance(string executionMachineIP, int port, bool runLLServer)
{
    facade = new LL_ApiFacade();

    facade.ws = WS_ConnectionActions.GetInstance();
    facade.ws.WS_Init(executionMachineIP, Convert.ToInt16(port)); //Init + connect to webSocket

    facade.ll_Action = new LL_Actions(facade.ws);

}
```

Abbildung 49: Verbindungsherstellung und Instanziierung der Klasse LL_Actions, Quelle: Eigene Darstellung.

8.4.1.2 Übermittlung von JSON-RPC-Requests

Für die Klasse LL_Actions wird ein Template geschrieben, das eine Methode pro API generiert. Diese Methoden haben die Aufgabe, die JSON-RPC Requests für die Übermittlung zum Server vorzubereiten und diese durchzuführen. Der Aufbau der einzelnen Methoden ist dabei sehr ähnlich und nur abhängig von den Übergabe-, Rückgabeparametern und des API-Typs. Aus diesem Grund kann mit einem kompakten Template eine große Menge an Code generiert werden. Die Ausgabe der API-Beschreibung als Kommentar erhöht zusätzlich die Lesbarkeit des Codes. In folgender Abbildung ist das Template für das LOGICDATA Produkt LOGIClink dargestellt. Für jedes weitere Produkt muss lediglich ein neues, leicht modifiziertes Template erstellt werden.

```
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

namespace LLApiFacade
{
    public class LL_Actions : LD_Actions
    {
        internal LL_Actions(WS_ConnectionActions ws) : base(ws) { }

<#list functions as f>
<#if (f.function_role?matches("common") && !f.function_type?matches("callback")) ||
(f.function_role?matches("device_specific") && f.affected_devices?seq_contains("logiclink"))>
        // ${f.function_description}
        public ${f.output_params[0][0]} ${f.function_name_API}(<@fill_declaration_params f/>)
        {
            var jsonRequest = new JsonRpcRequest()
            {
                Id = 69,
                Method = "${f.function_name_API}",
            };

            <#if f.input_params?has_content>
            jsonRequest.Params.Add(JsonConvert.SerializeObject(${f.input_params[0][1]}, new JsonSerializerSettings
            { ContractResolver = contractResolver, Formatting = Formatting.None }));
            </#if>

            ws.SendRequest(jsonRequest);
            return ws.WaitForResponseResult${"<"}${f.output_params[0][0]}${">"}(jsonRequest.Id);
        }

</#if>
</#list>
    }
}
```

Abbildung 50: Template zur Generierung der Methoden zur Vorbereitung der JSON-RPC-Requests, Quelle: Eigene Darstellung.

Die aus dem Template generierte Klasse LL_Actions ist von der ebenfalls generierten Klasse LD_Actions, die alle Methoden für die essenziellen APIs beinhaltet, abgeleitet. Dadurch erbt sie deren Verhalten und Eigenschaften. In folgender Abbildung werden die Methoden für zwei APIs dargestellt. Die Methode LD_Drive_Up besitzt keine Übergabeparameter, deshalb muss im Objekt der Klasse JsonRpcRequest, die für die Konvertierung der Daten in das JSON-RPC-Format zuständig ist, nur die ID sowie der API-Name gesetzt werden. Die Methode LD_MoveToPosition besitzt ein Objekt als Übergabeparameter des Typs LDHeightData. Dieses Objekt muss bevor es für die weitere Verarbeitung weitergegeben werden kann, in einen String konvertiert werden. Dazu wird die Methode SerializeObjekt der Klasse JsonConvert, die im Newtonsoft.Json Framework inkludiert ist, verwendet. Der String wird dem Objekt jsonRequest als Parameter übergeben. Dieses Objekt wird als Request an den Server gesendet.

```
internal LL_Actions(WS_ConnectionActions ws) : base(ws) { }

// Drives up using a LD device.
public LDStatus LD_DriveUp()
{
    var jsonRequest = new JsonRpcRequest()
    {
        Id = 69,
        Method = "LD_DriveUp",
    };


    ws.SendRequest(jsonRequest);
    return ws.WaitForResponseResult<LDStatus>(jsonRequest.Id);
}

// Move table to wanted position.
public LDStatus LD_MoveToPosition(LDHeightData height_data)
{
    var jsonRequest = new JsonRpcRequest()
    {
        Id = 69,
        Method = "LD_MoveToPosition",
    };

    jsonRequest.Params.Add(JsonConvert.SerializeObject(height_data, new JsonSerializerSettings
    { ContractResolver = contractResolver, Formatting = Formatting.None }));

    ws.SendRequest(jsonRequest);
    return ws.WaitForResponseResult<LDStatus>(jsonRequest.Id);
}
```

Abbildung 51: Generierte Methoden zur Erstellung von API abhängigen JSON-RPC-Requests. Quelle: Eigene Darstellung.

8.4.1.3 Empfangen von JSON-RPC-Responses

Das Empfangen der JSON-RPC-Requests wird in der Methode WS_MessageReceived der Klasse WS_ConnectionActions (siehe Abbildung 52) implementiert, die vom bestehenden Code übernommen werden kann. Dazu wird die empfangene JSON-RPC-Response in ein JSON Objekt konvertiert, um einfach auf die Eigenschaften jenes zugreifen zu können. Zuerst wird überprüft, ob das Resultat der Antwort leer ist beziehungsweise ein Fehler in dieser anliegt. In diesem Falle wird eine Fehlermeldung ausgegeben und diese in einem Dictionary gespeichert. Ist die Antwort des Servers valide, wird unterschieden, ob es sich bei der Antwort um eine Response eines Callbacks oder um eine einfache Response handelt. In beiden Fällen werden die Resultate ausgegeben und dem Log hinzugefügt. Bei Callbacks wird zusätzlich ein Eintrag im Callback Log angelegt.

```
private void WS_MessageReceived(object sender, MessageEventArgs e)
{
    //Check if received data is from callBack function or normal one
    var response = JsonConvert.DeserializeObject<JsonRpcResponse>(e.Data);

    //If in response is error, that we log error
    if (response.Result == null && response.Error != null)
    {
        LL_Log logError = new LL_Log(DateTime.Now, "ResponseError: " + Environment.NewLine +
        response.Error.Message + "(Code: " + response.Error.code + ") -" +
        response.Error.message + Environment.NewLine);
        Logs.Enqueue(logError);

        //Save error in Error dictionary
        Error[response.Id] = response.Error.Message +
        "(Code: " + response.Error.code + ") -" +
        response.Error.message;
    }
    else//no error
    {
        if (response.IsCallback) //callback results
        {
            Console.WriteLine("Response callback: ", e.Data);

            CallbackResults.Enqueue(new CallbackResponseProtobuf(response.Result.ToString(),
            response.Method.ToString()));
        }
        else //json rpc function
        {
            Console.WriteLine("Response: ", e.Data);
            Result[response.Id] = response.Result;
        }

        //Write to log queue
        LL_Log log = new LL_Log(DateTime.Now, "Response: " + Environment.NewLine +
        e.Data.ToString() + Environment.NewLine);
        Logs.Enqueue(log);
    }

    WaitForResponse[response.Id] = false;
}
```

Abbildung 52: Verarbeitung der empfangenen JSON-RPC-Responses, Quelle: Eigene Darstellung.

8.4.2 Grafische Benutzeroberfläche des Command Senders

Die Hauptaufgabe des Command Senders ist es, dem Tester und der Testerin die Eingabe von Parametern und den Aufruf der zu testenden API zu ermöglichen. Durch sich ändernde Eingabeparameter und Eigenschaften der APIs während der Entwicklungsphase und damit einhergehenden notwendigen Änderungen im Command Sender ist der Einsatz von Codegenerierung besonders sinnvoll. Jedoch wird für die Entwicklung der grafischen Benutzeroberfläche des Command Senders, wie in Kapitel 7.1.3.1 angemerkt, nicht auf die Template-Engine FreeMarker zurückgegriffen. Die Steuerelemente der Benutzeroberfläche werden stattdessen direkt aus der eingelesenen JSON API-Dokumentation generiert. Das bringt den Vorteil, dass keine zusätzlichen Templates geschrieben werden müssen und dadurch die Benutzeroberfläche zur Laufzeit dynamisch generiert werden kann.

8.4.2.1 Einlesen der JSON-Files und Initialisierung

In der Methode Initalize (siehe Abbildung 53) wird zunächst der Pfad zum Ordner, wo die JSON-Files der Dokumentation abgelegt sind, definiert. Dieser ist hartkodiert, was in weiterer Folge dazu führt, dass neue JSON-Files immer am selben Ort abgelegt werden müssen. Dadurch muss nicht bei jedem Start der Pfad neu eingegeben werden und dadurch können Fehler, die durch die Wahl falscher Pfade entstehen, vermieden werden. Sofern der angegebene Pfad existiert, werden die sich darin befindenden JSON-Files eingelesen. Jede API in der Dokumentation entspricht einem JObject in einem JArray. Zunächst werden alle von den enthaltenen APIs unterstützten Produkte einer Liste hinzugefügt, die in weiterer Folge für die Generierung der Benutzeroberfläche notwendig ist. Danach werden alle APIs hinsichtlich ihrer Rolle aufgeteilt. Prinzipiell werden dabei die Rollen „Essentiell“, „Gemeinsam genutzt“ und „Produktspezifisch“ unterschieden. Dazu wird die in der Klasse ULDAPI_Methods definierte Enumeration FunctionRoleEnum eingesetzt und mit der jeweiligen Rolle der API verglichen. Abhängig davon werden die APIs den jeweiligen Listen des Typs ULDAPI_Methods, die die JSON-Eigenschaften der APIs beinhaltet, hinzugefügt. Diese sind zur Separierung in der Benutzeroberfläche notwendig. Sollte eine andere Rolle als die definierten in der Dokumentation enthalten sein oder sollte diese Information fehlen, wird eine Exception weitergeleitet.

```
//INIT
private void Initialize()
{
    string json = string.Empty;

    //PATH TO JSON folders
    string jsonFolderPath = String.Format("{0}\\{1}", Directory.GetParent(Directory.GetCurrentDirectory()).Parent.Parent.FullName,
        C_JSON_FOLDER_NAME);

    if (Directory.Exists(jsonFolderPath))
    {
        string[] files = Directory.GetFiles(jsonFolderPath);

        string functionRole = string.Empty;

        foreach (string jsonFile in files)
        {
            json = File.ReadAllText(jsonFile.ToString());

            JArray jArray = JArray.Parse(json);

            if (jArray.Count != 0)
            {
                foreach (JObject jsonObject in jArray)
                {
                    //ADDS affected devices in to list
                    if (jsonObject.ContainsKey(C_AFFECTED_DEVICES))
                    {
                        var affectedDevice = jsonObject[C_AFFECTED_DEVICES].Values<string>();
                        AddToProductList(affectedDevice);
                    }
                }
                //We only look at the first to see what his role is and than we deserialize the whole json.
                JObject jObject = jArray.Children<JObject>().First();

                if (jObject.ContainsKey(C_FUNCTION_ROLE))
                {
                    ULDAPI_Methods.FunctionRoleEnum role;
                    //We get the role from the object
                    if (!Enum.TryParse<ULDAPI_Methods.FunctionRoleEnum>((string)jObject[C_FUNCTION_ROLE], out role))
                        throw new Exception(string.Format("Cannot find device type: '{0}' in file '{1}'. Contact software admin.",
                            (string)jObject[C_FUNCTION_ROLE], jsonFile));

                    //ESSENTIAL
                    if (role == ULDAPI_Methods.FunctionRoleEnum.essential)
                    {
                        _essentialMethods = JsonConvert.DeserializeObject<List<ULDAPI_Methods>>(json);
                    }

                    //COMMON + INTERNAL
                    else if (role == ULDAPI_Methods.FunctionRoleEnum.common || role == ULDAPI_Methods.FunctionRoleEnum.@internal)
                    {
                        _commonMethods.AddRange(JsonConvert.DeserializeObject<List<ULDAPI_Methods>>(json));
                    }

                    //PRODUCT SPECIFIC
                    else if (role == ULDAPI_Methods.FunctionRoleEnum.device_specific)
                    {
                        ULDAPI_Methods.FunctionDevicesEnum device;

                        //We need this to get the device in device specific json file
                        var specificAffectedDevice = jObject[C_AFFECTED_DEVICES].Values<string>();

                        //effectedDevice.First() because there is only one obj. in the IEnumebrable
                        if (!Enum.TryParse<ULDAPI_Methods.FunctionDevicesEnum>(specificAffectedDevice.First(), out device))
                            throw new Exception(string.Format("Cannot find device type: '{0}' in file '{1}'. Contact software admin.",
                                specificAffectedDevice.First(), jsonFile));
                        _productSpecificMethods.Add(device, JsonConvert.DeserializeObject<List<ULDAPI_Methods>>(json));
                    }
                    else...
                }
                else...
            }
            else...
        }
    }
    else
        throw new Exception(String.Format("Could not find path: {0}", C_JSON_FOLDER_NAME));
}
```

Abbildung 53: Einlesen der JSON-Files und Initialisierung, Quelle: Eigene Darstellung.

8.4.2.2 Implementierung des Layouts der grafischen Benutzeroberfläche

Beim Ausführen des Command Senders wird zu Beginn nur eine fast leere Benutzeroberfläche dargestellt, die nur die Labels und die Textfelder für die IP-Adresse und den Port einen Button zum Verbinden mit dem Server, sowie das Konsolenfenster beinhaltet. Der Grund dafür liegt darin, dass zu diesem Zeitpunkt die APIs, wie in Kapitel 8.4.2.1 beschrieben, erst eingelesen werden müssen. Als nächsten Schritt müssen diese eingelesenen Daten genutzt werden, um die Steuerelemente zu generieren. Bevor diese jedoch hinzugefügt werden können, muss das Layout festgelegt werden. Zunächst wird für jedes beim Einlesen identifizierte Produkt eine eigene TabPage erstellt, dadurch entsteht eine strikte Separierung, die das Aufrufen von nicht produktrelevanten APIs verhindern soll. Über jede Seite wird ein TableLayoutPanel gelegt, das erlaubt, den Inhalt in einem Raster anzuordnen. Der Hauptbereich der GUI wird in drei Unterbereiche geteilt. Die bereits beim Einlesen durchgeführte Unterscheidung hinsichtlich der API-Rolle wird hier herangezogen. Jeder Bereich wird hierbei von einer GroupBox mit dem jeweiligen Namen umschlossen. Innerhalb dieser sollen die Steuerelemente für die APIs platziert werden. Dafür wird jeweils ein Objekt der Klassen ucEssentialMethods, ucCommonMethods und ucProductSpecificMethods angelegt, die dafür verantwortlich sind, dass die richtigen Steuerelemente für die APIs generiert werden.

In folgender Abbildung ist beispielhaft die Implementierung des Layouts der TabPages und der darin enthaltene Aufruf zur Generierung der Steuerelemente der essenziellen APIs dargestellt.

```
if (products.Count > 0)
{
    foreach (var product in products)
    {
        //TAB PAGE
        tabPage = new TabPage();
        tabPage.Text = Capitalize(product);
        tabPage.Location = new Point(4, 22);
        tabPage.Size = new Size(800, 700);
        //tabPage.Dock = DockStyle.Fill;

        TableLayoutPanel tlp = new TableLayoutPanel();
        tlp.AutoScroll = true;
        tlp.Margin = new Padding(10);
        tlp.ColumnCount = 1;
        tlp.AutoSize = true;
        tlp.Dock = DockStyle.Fill;
```

```
        //ESSENTIAL METHODS

        //ESSENTIALMETHODS GROUPBOX
        GroupBox gbEssentials = new GroupBox();
        gbEssentials.Text = "Essential Methods";
        gbEssentials.Dock = DockStyle.Fill;
        gbEssentials.Location = new Point(0, 0);
        gbEssentials.Size = new Size(tlp.Width, (tabPage.Height / 3));

        essentialMethods = new ucEssentialMethods();
        essentialMethods.Initialize();
        essentialMethods.Dock = DockStyle.Fill;

        gbEssentials.Controls.Add(essentialMethods);
        tlp.Controls.Add(gbEssentials);

        //ADD TableLayoutPannel on TabPage
        tabPage.Controls.Add(tlp);

        //ADD TAB TO TABCONTROL
        tabControl.Controls.Add(tabPage);
        tabControl.Size = new Size(this.ClientSize.Width, (this.ClientSize.Height - 40));

    }
}
```

Abbildung 54: Beispielhafte Darstellung der Implementierung des Layouts der TabPages, Quelle: Eigene Darstellung.

8.4.2.3 Platzieren der Steuerelemente auf der Benutzeroberfläche

Die Klassen ucEssentialMethods, ucCommonMethods und ucProductSpecificMethods sind von der Klasse ucULDAPI_Main abgeleitet und erweitern diese um die Methode Initialize. In dieser werden für alle APIs mit der jeweiligen Rolle ein TableLayoutPanel und ein Button, mit dem der User die Möglichkeit bekommt, diese aufzurufen, angelegt. Sollte die API Übergabeparameter besitzen, müssen zusätzliche Steuerelemente, abhängig vom Parametertyp, generiert werden. Zur Definition, welche Steuerelemente für welche Parametertypen beziehungsweise API angelegt werden sollen, wird die Methode GetClassProperties aufgerufen (siehe Abbildung 55).

```
if (method.InputParams != null && method.InputParams.Count > 0)
{
    foreach (var inputParam in method.InputParams)
    {
        string inputName = inputParam.First(); //parameter type
        var classType = list.SingleOrDefault(x => x.Name == inputName);
        if (classType != null)
        {
            GetClassProperties(classType, flpMethod, null);
        }
    }
}
```

Abbildung 55: Aufruf der Methode GetClassProperties bei mindestens einem Übergabeparameter der API, Quelle: Eigene Darstellung.

Die Methode GetClassProperties durchläuft alle Parameter, die der jeweiligen API übergeben werden. Für jeden Parameter soll dabei das passende Steuerelement generiert werden. Die Wahl welches gewählt wird, ist von dem Datentyp des Übergabeparameters abhängig. Dieses Attribut kann durch den Einsatz von Klassen des namespaces Systems.Reflection ermittelt werden. Mit diesen Informationen können die Steuerelemente, die für die Eingabe der Parameter notwendig sind, innerhalb der GroupBox der jeweiligen API platziert werden.

In folgender Tabelle werden alle unterstützten Datentypen inklusive der dazugehörigen Steuerelemente dargestellt:

Datentyp	Steuerelement
String	Textbox
Int32	NumericUpDown
UInt32	NumericUpDown
Bool	Checkbox
Enum	Combobox

Tabelle 5: Auflistung der unterstützten Datentypen und dazugehörigen Steuerelemente. Quelle: Eigene Darstellung.

In der folgenden Abbildung wird ein Codeausschnitt, der zur typabhängigen Steuerelement Generierung eingesetzt wird, dargestellt. Die Informationen über die Datentypen der Parameter werden über die Methode PropertyType ermittelt. In Form einer if-elseif-else Verzweigungsstruktur werden alle Datentypen der Übergabeparameter mit den in Tabelle 5 aufgelisteten verglichen. Stimmen beide überein, wird das jeweilige Steuerelement sowie ein Label mit dem Namen des Übergabeparameters dem TableLayoutPanel hinzugefügt. Sollte es sich beim übergebenen Parameter um einen Aufzählungstyp handeln muss zusätzlich noch die Datenquelle der auszuwählenden Elemente definiert werden. Wird ein anderer Datentyp als die oben dargestellten ermittelt, wird eine Exception weitergeleitet.

```
//We get properties of the classType that we passed
var properties = classType.GetProperties(BindingFlags.Instance | BindingFlags.Public | BindingFlags.DeclaredOnly);

//We go through every prop
foreach (var prop in properties.OrderBy(x => x.Name))
{
    if (prop.PropertyType == typeof(string))
    {
        ULDAPI_TextBox tb = new ULDAPI_TextBox(prop.Name, classType);
        Label tblbl = new Label { Text = prop.Name };
        tlpParameters.Controls.Add(tblbl);
        tlpParameters.Controls.Add(tb);
    }
    else if (prop.PropertyType == typeof(Int32) || prop.PropertyType == typeof(UInt32))
    {
        ULDAPI_NumericUpDown nud = new ULDAPI_NumericUpDown(prop.Name, classType) { Width = 60 };
        Label nudlbl = new Label { Text = prop.Name };
        tlpParameters.Controls.Add(nudlbl);
        tlpParameters.Controls.Add(nud);
    }
    else if (prop.PropertyType == typeof(bool))
    {
        ULDAPI_CheckBox chb = new ULDAPI_CheckBox(prop.Name, classType);
        Label chblbl = new Label { Text = prop.Name };
        tlpParameters.Controls.Add(chblbl);
        tlpParameters.Controls.Add(chb);
    }
    else if (prop.PropertyType.BaseType == typeof(Enum))
    {
        ULDAPI_ComboBox cob = new ULDAPI_ComboBox(prop.Name, classType);
        Label coblbl = new Label { Text = prop.Name };
        cob.MinimumSize = new Size(200, 200);
        cob.DataSource = Enum.GetValues(prop.PropertyType);

        tlpParameters.Controls.Add(coblbl);
        tlpParameters.Controls.Add(cob);
    }
    else
    {
        throw new Exception(string.Format("Not recognised property type: {0}. Contact software admin.",
        prop.PropertyType.Name));
    }

}
```

Abbildung 56: Codeausschnitt zur typabhängigen Steuerelement Generierung. Quelle: Eigene Darstellung.

8.4.2.4 Aufrufen der entsprechenden APIs

Nachdem die Verbindung mit dem Server über die API-Facade (siehe Kapitel 8.4.1) hergestellt, die JSON API-Dokumentation eingelesen und alle Steuerelemente erstellt wurden, kann der User mit dem Aufruf der APIs beginnen. Dazu müssen nur die entsprechenden Parameter eingegeben und anschließend der zur jeweiligen API zugeordnete Button geklickt werden. Durch Klicken dieser Buttons wird die Methode ButtonClickMethodHandler (siehe Abbildung 57) aufgerufen. Da diese Methode von allen generierten Buttons aufgerufen wird, müssen innerhalb dieser verschiedene Fälle berücksichtigt werden. Im Konkreten muss zwischen APIs mit Übergabeparametern und APIs ohne Übergabeparametern unterschieden werden. Das kann durch die Überprüfung der Anzahl der Elemente in dem Container, der den Button und alle zugehörigen Steuerelemente enthält, geschehen. Eine Anzahl von eins bedeutet, dass es sich um eine API ohne Übergabeparameter handelt und die Methode SetULDAPIMethod aufgerufen werden kann. Ist die Anzahl größer eins, handelt es sich um eine API mit Übergabeparametern, das das Auslesen der Werte

aus den Steuerelementen durch die Methode SetPropertyValues erfordert. Das von dieser Methode rückgegebene Objekt kann anschließend der Methode SetULDAPIMethod übergeben werden.

```
public void ButtonClickMethodHandler(object sender, EventArgs e)
{
    if (sender.GetType() != typeof(Button))
        return;
    Button btn = (Button)sender;

    //if it has no childs return
    if (btn.Parent.Controls.Count < 1)
        throw new Exception(string.Format("{0} - The parent control - {1} of button {2} has 0 children. Contact software admin.",
            MethodBase.GetCurrentMethod().Name, btn.Parent.Parent.Name, btn.Name));

    //button and further children
    else if (btn.Parent.Controls.Count > 1) {
        ULDAPI_ResponseBase rBase = SetPropertyValues(btn.Parent.Controls);
        SetULDAPIMethod(btn, rBase);
    }
    // == 1 only 1 child = button
    else if (btn.Parent.Controls.Count == 1)
    {
        SetULDAPIMethod(btn);
    }
}
```

Abbildung 57: Implementierung des ButtonClickMethodHandler. Quelle: Eigene Darstellung.

Die Methode SetULDAPIMethod (siehe Abbildung 58) überprüft zunächst, zu welchem Produkt die gewählte API zählt. Kommt es zu einer Übereinstimmung, werden dem Objekt method die Metadaten der zur jeweiligen API gehörigen Methode der Klasse LD_ApiFacade zugewiesen. Wie in Kapitel 8.4.1.2 beschrieben, haben diese Methoden die Aufgabe, die JSON-RPC Requests für die Übermittlung zum Server vorzubereiten. Sofern eine zur API gehörige Methode in der LD_ApiFacade existiert, wird die Methode InvokeULDAPIMethod aufgerufen.

```
private void SetULDAPIMethod(Button btn, params ULDAPI_ResponseBase[] parameters)
{
    string methodName = btn.Text;
    Form form = btn.FindForm();

    MethodInfo method = null;

    if (LD_ApiFacade.Instance.GetCurrentProduct() == LD_Enums.ld_product.ULDAPI_SN)
        method = LD_ApiFacade.Instance.ULDApiSNMethods.GetType().GetMethod(methodName);
    else if (LD_ApiFacade.Instance.GetCurrentProduct() == LD_Enums.ld_product.ULDAPI_SF)
        method = LD_ApiFacade.Instance.ULDApiSFMethods.GetType().GetMethod(methodName);
    else if (LD_ApiFacade.Instance.GetCurrentProduct() == LD_Enums.ld_product.ULDAPI_LL)
        method = LD_ApiFacade.Instance.ULDApiLLMethods.GetType().GetMethod(methodName);
    else if (LD_ApiFacade.Instance.GetCurrentProduct() == LD_Enums.ld_product.ULDAPI_LC)
        method = LD_ApiFacade.Instance.ULDApiLCMethods.GetType().GetMethod(methodName);

    //Invoke method
    if (method != null)
        InvokeULDAPIMethod(method, parameters);
    else
        throw new Exception(string.Format("Method could not be found. Contact software admin."));

}
```

Abbildung 58: Implementierung der Methode SetULDAPIMethod. Quelle: Eigene Darstellung.

Die aufgerufene Methode InvokeULDAPIMethod (siehe Abbildung 59) hat die Aufgabe, die in SetULDAPIMethod ermittelte, zur API gehörende Methode aufzurufen. Dabei muss wiederrum zwischen APIs mit und ohne Übergabeparameter unterschieden werden. In beiden Fällen werden die aktuelle Zeit sowie der aufgerufene Methodenname an die Konsole gesendet und ausgegeben. Der Aufruf der Methode erfolgt über die Methode Invoke des Namenraums System.Reflection. Dabei wird der Methodenname reflektiert und jene mit demselben Namen aufgerufen. Innerhalb dieser Methode wird auf die JSON-RPC-Response gewartet und anschließend als Objekt an die Methode InvokeULDAPIMethod zurückgegeben und für die weitere Verarbeitung in eine ULDAPI_ResponseBase konvertiert. Dieses Ergebnis, der Methodenname der API und ein Zeitstempel werden abschließend an die Konsole weitergeleitet und ausgegeben.

```
private void InvokeULDAPIMethod(MethodInfo method, params ULDAPI_ResponseBase[] parameters)
{
    string timeString;
    ULDAPI_ResponseBase result = null;

    if (parameters == null || parameters.Length == 0)
    {
        timeString = DateTime.Now.ToString("HH:mm:ss");
        //Sends the time and methodname to the console
        OnMessageSent(String.Format("SEND: {0} - {1}", timeString, method.Name));
        result = method.Invoke(LD_ApiFacade.Instance.ULDApiEssentialMethods, null) as ULDAPI_ResponseBase;

    }
    else
        timeString = DateTime.Now.ToString("HH:mm:ss");
        //Sends the time and methodname to the console
        OnMessageSent(String.Format("SEND: {0} - {1}", timeString, method.Name));
        result = method.Invoke(LD_ApiFacade.Instance.ULDApiEssentialMethods, parameters) as ULDAPI_ResponseBase;

    //Waits for the response and sends it to the console
    if (OnResponseReceived != null)
    {
        timeString = DateTime.Now.ToString("HH:mm:ss");
        OnResponseReceived(this, result, "RECEIVED: t" + method.Name, timeString);
    }
}
```

Abbildung 59: Aufruf der Methoden zur Vorbereitung der JSON-RPC Requests und Ausgabe in der Konsole. Quelle: Eigene Darstellung.

8.4.2.5 Darstellung der grafischen Benutzeroberfläche

Die grafische Benutzeroberfläche ist die Schnittstelle zwischen Tester beziehungsweise Testerin und dem Testsystem. Erst nach dem Verbindungsaufbau, dem Einlesen der JSON-Files (siehe Kapitel 8.4.2.1) und dem Platzieren der Steuerelemente (siehe Kapitel 8.4.2.3) ist die komplette grafische Benutzeroberfläche sichtbar und alle Funktionen nutzbar. In folgender Abbildung ist die fertig generierte Benutzeroberfläche dargestellt. Im Kopf der Applikation sind die Steuerelemente zum Verbindungsaufbau sowie das Konsolenfenster zur Ausgabe der übermittelten Daten zu sehen. Das gewünschte Produkt kann durch Wahl der jeweiligen Registerkarte ausgewählt werden. Darunter finden sich die generierten Steuerelemente der APIs unterteilt in essenziellen, gemeinsam genutzte und produktspezifische. Im Falle des unten dargestellten LOGIClinks, wird der produktspezifische Bereich nicht dargestellt, da keine der APIs diese Rolle für dieses Produkt einnimmt.

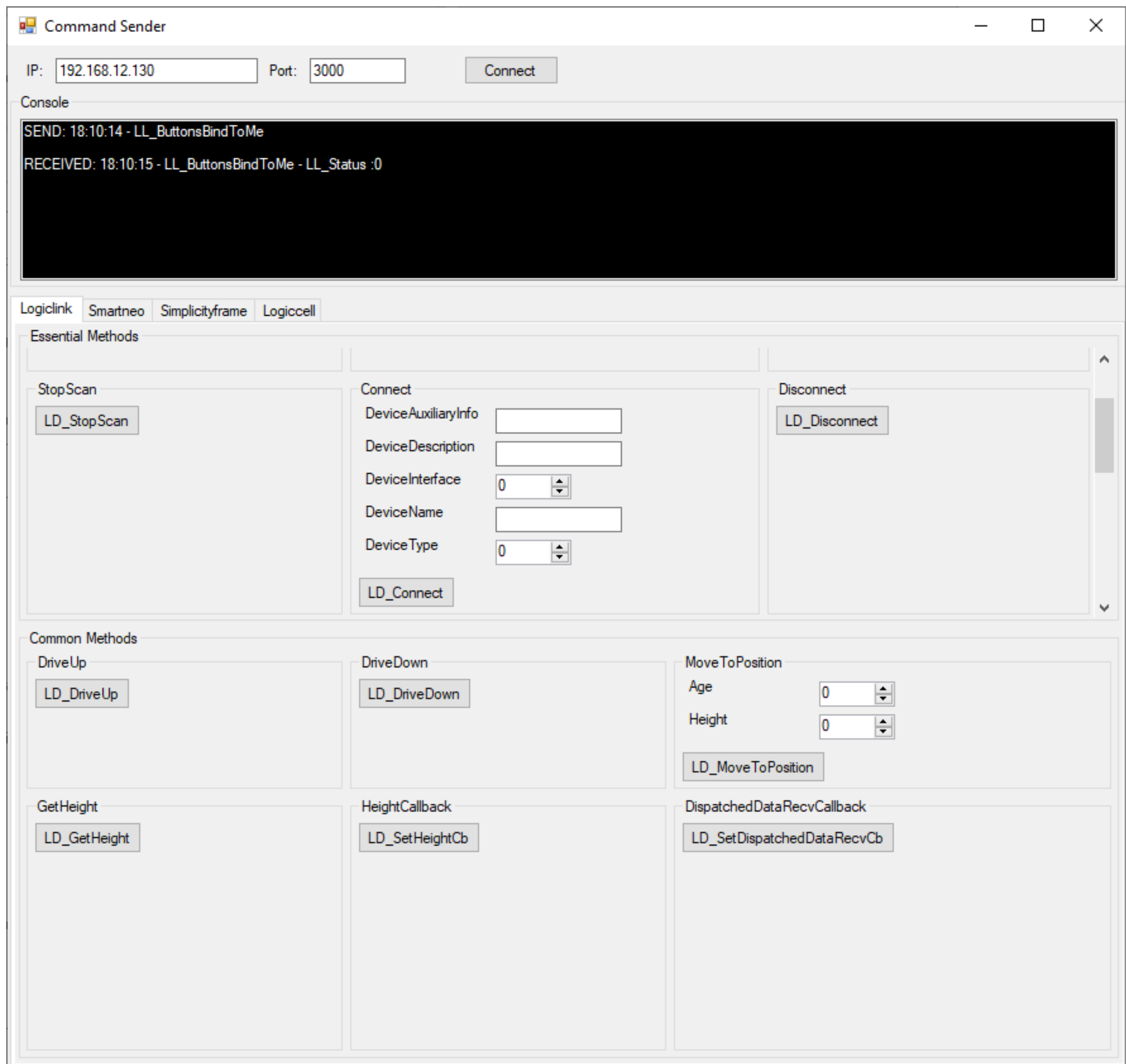

Abbildung 60: Darstellung des generierten Command-Senders, Quelle: Eigene Darstellung.

9 ERGEBNISSE UND AUSBLICK

Diese Masterarbeit zeigt wie der Einsatz von automatischer Codegenerierung bei der Implementierung bei sich ändernden Eingabedaten unterstützen und in weiterer Folge den Entwicklungs- und Wartungsaufwand verringern kann. Der Einsatz von Template-Engines, im konkreten FreeMarker, erweist sich für diese Aufgabenstellung als richtige Wahl. Die generierten Files enthalten große Teile sich wiederholender Codesequenzen, die durch die Entwicklung von kompakten Templates einfach und schnell generiert werden können.

Die API-Dokumentation in Form von JSON-Files bietet eine einfache Lösung, um die Eigenschaften aller APIs zu beschreiben. Die Aufteilung der APIs hinsichtlich ihrer Rollen auf mehreren Files verbessert zusätzlich die Übersicht. JSON als Datenaustauschformat zeichnet sich auch dadurch aus, dass es von fast allen Programmiersprachen unterstützt wird – darunter auch die für dieses Softwarevorhaben relevanten Programmiersprachen C#, Java, C++ und Swift. Das ist auch ein Grund warum auch an der bestehenden JSON-RPC-Server-Client Architektur zwischen Test-App und Command Sender festgehalten wird.

Der Command Sender, bei dem auch auf die API-Dokumentation als Eingabedaten zurückgegriffen wird, nutzt Reflection zur Generierung der Steuerelemente. Dieses geschieht während der Laufzeit und der Wartungsaufwand zwischen den Testschleifen beschränkt sich dafür auf den Austausch der JSON-API-Dokumentationsfiles im entsprechenden Ordner. Auch bezüglich des initialen Entwicklungsaufwands ergeben sich deutliche Zeitersparnisse, da nicht alle Steuerungselemente manuell eingefügt und auskodiert werden müssen, was in weiterer Folge auch die Fehlerwahrscheinlichkeit verringert.

Für die API-Facade als Teil des Command Senders wird FreeMarker zur Generierung der Klasse zur Vorbereitung der JSON-RPC-Requests genutzt. Des Weiteren können bewährte Codeteile des bestehenden Systems wiederverwendet werden. Auch in diesem Teil des Systems können dadurch die Fehlerwahrscheinlichkeit und personelle Ressourcenaufwand verringert werden.

Die Test-Apps auf den verschiedenen Plattformen unterscheiden sich optisch nur geringfügig von den Test-Apps, die im Testsystem ohne Codegenerierung genutzt wurden. Die Entwicklungszeit wird auch in diesem Teil des Testsystems durch den Einsatz von Templates verringert. Das ist jedoch sehr stark davon abhängig wie viele APIs die Library des jeweiligen Releases beinhaltet. Je höher die Anzahl desto höher auch das Einsparungspotential. Der Wartungsaufwand reduziert sich auf die Generierung der Wrapper-Methoden, den Austausch dieser, der API-Libraries, sowie der aus dem Protocol Buffer File generierten Datenzugriffsklassen.

Hinsichtlich der Usability gibt es nur sehr geringe Unterschiede zum Testsystem ohne Codegenerierung. Das bringt den Vorteil, dass sich die Testprozedur für den Tester und der Testerin nicht ändert und der Umstieg vom bestehenden auf das neue Testsystem einfach vollzogen werden kann. Die einzigen für den Endbenutzer und Endbenutzerin merkbaren Änderungen beziehen sich dabei auf die veränderte Benutzeroberfläche des Command Senders.

Durch den Einsatz von automatischer Codegenerierung und Optimierung der Architektur des Testsystems konnte die initiale Entwicklungszeit im Vergleich mit, der des Testsystems ohne Codegenerierung um ca. 50% reduziert werden. Dabei muss aber erwähnt werden, dass einige bewährte Teile aus der bestehenden Codebasis übernommen werden konnten. Die größte Einsparung resultiert jedoch im Bereich des Wartungsaufwandes. Dadurch das nach Erhalt des neuen API-Release Candidates lediglich die Files ausgetauscht, Klassen neu generiert und die einzelnen Software-Projekte neu kompiliert werden müssen, reduziert sich der Wartungsaufwand auf wenige Minuten pro Testschleife. Im Vergleich zur vorherigen manuellen Anpassung des Testsystems bringt der neu entwickelte Ansatz eine erhebliche Verbesserung hinsichtlich des personellen Ressourcenaufwands. Die Zeitersparnis hängt dabei stark von der Anzahl und Größe der Änderungen des Testobjekts zwischen den Testschleifen ab. Damit einhergehend reduziert sich auch die Durchlaufzeit der Testschleifen und ermöglicht bestenfalls eine frühere Freigabe der API-Libraries. Ein weiterer Vorteil, der sich aus dem Einsatz von automatischer Codegenerierung ergibt, ist eine geringere Anzahl an Fehlern im Code des Testsystems, da durch Templates Tipp- und Flüchtigkeitsfehler vermieden werden.

Automatische Codegenerierung ermöglicht durch seine Flexibilität ein breites Anwendungsspektrum. Des weiteren zeigen die Ergebnisse dieser Masterarbeit das große Potential zur Ressourceneinsparung auf. Aus diesem Grund sind bereits weitere Entwicklungsprojekte im Integration Test Team geplant, die auf Codegenerierungsmethoden zurückgreifen. Ein Bestreben ist es natürlich den Codegenerierungsanteil sowie den Automatisierungsgrad noch weiter zu erhöhen und dadurch die Effizienz zu steigern. Als nächsten Schritt soll der Dateienaustausch, die anschließende Generierung des Codes sowie Kompilierung der einzelnen Projekte automatisch erfolgen, um den Wartungsaufwand noch weiter zu minimieren.

Ein weiteres Ziel ist es die gewonnen Erkenntnisse mit weiteren Teams zu teilen und dadurch den Einsatz von Codegenerierung weiter zu forcieren.

Literaturverzeichnis

Gedruckte Werke (5)

Arnoldus, Jeroen; Brand, Mark; Serebnik, A.; Brunekreef, J.J. (2012): *Code Generation with Templates*, 1. Auflage, Atlantis Press, Eindhoven

Fleischmann, Albert; Stefan, Oppl; Schmidt, Werner; Stary, Christian (2018): *Ganzheitliche Digitalisierung von Prozessen*, 1. Auflage, Springer Vieweg, Wiesbaden

Herrington, Jack (2003): *Code Generation in Action*, Manning Publications Co., Greenwich

Seemann, Jochen; Wolff von Gudenberg, Jürgen (2006): *Software-Entwurf mit UML 2*, 2. Auflage Auflage, Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg

Trompeter, Jens; Pietrek, Georg; Beltran, Juan; Holzer, Boris; Kamann, Thorsten; Kloss, Michael; Mork, Stefen; Nieheus, Benedikt; Thoms, Karsten (2007): *Modellgetriebene Softwareentwicklung MDA und MDSD in der Praxis*, 1. Auflage, Entwickler-Press, Frankfurt am Main

Wissenschaftliche Artikel (2)

LOGICDATA Electronic & Software Entwicklungs GmbH 2 (2019): *Fastfacts*, in: Dynamik life, Ausgabe 1/2019, LOGICDATA Electronic & Software Entwicklungs GmbH, S. 44-45

Wüstneck, Klaus (1963): *Zur philosophischen Verallgemeinerung und Bestimmung des Modellbegriffs*, in: Deutsche Zeitschrift für Philosophie, 11/1963, Akademie Verlag, S. 1504-1523

Konferenzbeiträge (1)

Parr, Terrence (2004): *Enforcing Strict Model-View Separation in Tempate Engines*, in: *Proceedings of the 13th international conference on World Wide Web*, San Francisco, S. 224-233

Online-Quellen (38)

Apache Velocity Project 1 (2016): *The Apache Velocity Project* https://velocity.apache.org/engine/devel/overview.html [Stand: 12.08.2019]

Apache Velocity Project 2 (2016): *Velocity User Guide* https://velocity.apache.org/engine/1.7/user-guide.html#velocity-template-language-vtl-an-introduction [Stand: 12.08.2019]

Apache Velocity Project 3 (2019): *Velocity Editiors* https://cwiki.apache.org/confluence/display/velocity/VelocityEditors [Stand: 12.08.2019]

Begerow, Markus (o.J.): *Datenbanke verstehen* http://www.datenbanken-verstehen.de/lexikon/model-view-controller-pattern/ [Stand: 10.08.2019]

DATACOM Buchverlag GmbH (2013): *ITWissen.info* https://www.itwissen.info/MDSD-model-driven-software-development-Modellgetriebene-Software-Entwicklung.html [Stand: 29.07.2019]

Dekany, Daniel (2019): *Eclipse Marketplace*
https://marketplace.eclipse.org/content/freemarker-ide [Stand: 30.11.2019]

Eclipse Foundation 1 (2019): *Eclipse documentation*
https://help.eclipse.org/2019-06/index.jsp?topic=%2Forg.eclipse.wst.sse.doc.user%2Ftopics%2Fcsrcedt006.html&resultof=%22%43%6f%6e%74%65%6e%74%22%20%22%63%6f%6e%74%65%6e%74%22%20%22%61%73%73%69%73%74%22%20 [Stand: 10.08.2019]

Eclipse Foundation 2 (2018): *Papyrus User Guide*
https://wiki.eclipse.org/Papyrus_User_Guide#Code_Generation_Support [Stand: 22.08.2019]

FreeMarker 1 (2019): *Freemarker*
https://freemarker.apache.org [Stand: 11.08.2019]

FreeMarker 2 (2019): *FreeMarker*
https://freemarker.apache.org/editors.html [Stand: 11.08.2019]

Fuchs Media Solutions (o.J.): *SEO-Analyse*
https://www.seo-analyse.com/seo-lexikon/a/api/ [Stand: 08.09.2019]

Google LLC. (o.J): *Protocol Buffers*
https://developers.google.com/protocol-buffers/docs/overview [Stand: 25.09.2019]

Hauer, Philipp (2010): *Das Facade Design Pattern*
https://www.philipphauer.de/study/se/design-pattern/facade.php [Stand: 08.10.2019]

IONOS SE 1 (2019): *Ionos SE Wasserfallmodell*
https://www.ionos.de/digitalguide/websites/web-entwicklung/wasserfallmodell/ [Stand: 31.07.2019]

IONOS SE 2 (2018): *Digital Guide UML*
https://www.ionos.at/digitalguide/websites/web-entwicklung/uml-modellierungssprache-fuer-objektorientierte-programmierung/ [Stand: 26.07.2019]

Jindal, Rishabh (o.J.): *Geeksforgeeks*
https://www.geeksforgeeks.org/asynchronous-synchronous-callbacks-java/ [Stand: 27.10.2019]

Juergeleit, Torsten (2019): *Github-Veloedit*
https://github.com/vaulttec/veloedit [Stand: 12.08.2019]

LOGICDATA Electronic & Software Entwicklung GmbH 1 (2019): *LOGICDATA*
https://www.logicdata.net/de/ [Stand: 17.08.2019]

LOGICDATA Electronic & Software Entwicklungs GmbH 5 (2019): *Motion@Work*
https://www.logicdata.net/de/blog/product/motionwork-2/ [Stand: 19.08.2019]

LOGICDATA Electronic & Software Entwicklungs GmbH 6 (2019): *SILVERmotion App*
https://www.logicdata.net/silvermotionapp/ [Stand: 19.08.2019]

LOGICDATA Elektronic & Software Entwicklungs GmbH 3 (2019): *LOGIC HOME*
https://www.logicdata.net/logic-home/ [Stand: 17.08.2019]

LOGICDATA Elektronic & Software Entwicklungs GmbH 4 (2019): *LOGIC OFFICE*
https://www.logicdata.net/de/logic-office/ [Stand: 17.08.2019]

LOGICDATA Elektronic & Software Entwicklungs GmbH 7 (2019): *LOGICDATA LOGIClink*
https://www.logicdata.net/de/blog/product/logiclink/ [Stand: 21.09.2019]

LOGICDATA Elektronic & Software Entwicklungs GmbH 8 (2018): *Handbuch LOGIClink*
https://www.logicdata.net/de/wp-content/uploads/sites/2/2018/06/Manual_LOGIClink_de_Doc1.pdf
[Stand: 21.09.2019]

Microsoft Corporation 1 (2016): *Microsoft Visual Docs*
https://docs.microsoft.com/de-at/visualstudio/ide/code-snippets?view=vs-2019 [Stand: 10.08.2019]

Microsoft Corporation 2 (2018): *Visual Studio Docs*
https://docs.microsoft.com/de-at/visualstudio/ide/using-intellisense?view=vs-2019 [Stand: 10.08.2019]

Microsoft Corporation 3 (2016): *Neuerungen beim Entwurf in Visual Studio 2017*
https://docs.microsoft.com/de-de/visualstudio/modeling/what-s-new-for-design-in-visual-studio?view=vs-2017&viewFallbackFrom=vs-2019 [Stand: 22.08.2019]

Microsoft Corporation 4 (2018): *Visual Studio Docs*
https://docs.microsoft.com/en-us/visualstudio/ide/class-designer/how-to-add-class-diagrams-to-projects?view=vs-2019 [Stand: 29.11.2019]

Pietrek, Georg (2008): *DocPlayer*
https://docplayer.org/4498444-Modellgetriebene-softwareentwicklung.html [Stand: 30.7.2019]

Rishab Software (2016): *Linkedin*
https://www.linkedin.com/pulse/understanding-difference-between-mvc-mvp-mvvm-design-rishabh-software [Stand: 18.09.2019]

Rouse, Margaret (2016): *ComputerWeekly*
https://www.computerweekly.com/de/definition/Model-View-Controller-MVC [Stand: 10.08.2019]

Sarhan, Akmal (o.J.): *Sourceforge-Veloeclipse*
http://propsorter.sourceforge.net/veloeclipse [Stand: 12.08.2019]

Schäling, Boris (2010): *Highscore*
http://www.highscore.de/uml/codegenerierung.html [Stand: 22.08.2019]

Techopedia Inc. (o.J.): *Techopedia-Boilerplate*
https://www.techopedia.com/definition/1259/boilerplate [Stand: 26.08.2019]

Tomassetti, Gabriele (2018): *Tomassetti*
https://tomassetti.me/code-generation/ [Stand: 20.09.2019]

Universität Augsburg Institut für Programmierung verteilter Systeme (2009): *Einführung in die modellbasierte Softwareentwicklung*
https://www.informatik.uni-augsburg.de/lehrstuehle/swt/vs/lehre/archiv/WS_08_09/seminar/downloads/Seminarband.pdf [Stand: 29.07.2019]

van Heesch, Dimitri (2019): *Doxygen*

http://www.doxygen.nl/index.html [Stand: 10.09.2019]

Vogel, Lars (2016): *Vogella*

https://www.vogella.com/tutorials/FreeMarker/article.html [Stand: 11.08.2019]

ABBILDUNGSVERZEICHNIS

Abbildung 1: LOGICDATA Logo. Quelle: LOGICDATA Electronic & Software Entwicklung GmbH 1 (2019), Online-Quelle [17.08.2019]. 1

Abbildung 2: Produktübersicht LOGIC HOME, Quelle: Eigene Darstellung. 2

Abbildung 3: Produktübersicht LOGIC OFFICE. Quelle: LOGICDATA Elektronic & Software Entwicklungs GmbH 4 (2019), Online-Quelle [17.08.2019]........ 3

Abbildung 4: Motion@Work App, Quelle: LOGICDATA Electronic & Software Entwicklungs GmbH 5 (2019), Online-Quelle [19.08.2019]. 5

Abbildung 5: SILVERmotion App, Quelle: Eigene Darstellung. 6

Abbildung 6: Schematische Darstellung eines Codegenerators, Quelle: Arnoldus/Brand/Serebnik/Brunekreef (2012), S.2. (leicht modifiziert). 7

Abbildung 7: Beispiel eines Code Snippets in Visual Studio. Quelle: Eigene Darstellung. 9

Abbildung 8: Codevervollständigung mit IntelliSense in Visual Studio. Quelle: Eigene Darstellung. 10

Abbildung 9: Codevervollständigung mit Content Assist in Eclipse. Quelle: Eigene Darstellung. 10

Abbildung 10: Schematische Darstellung eines Template-Systems; Quelle: Arnoldus/Brand/Serebnik/Brunekreef (2012), S.14. (leicht modifiziert). 11

Abbildung 11: Beispiel eines Templates und des resultierenden Ausgabetexts. Quelle: Eigene Darstellung. 13

Abbildung 12: Model-View-Controller-Muster, Quelle: (leicht modifiziert) Begerow (o.J.), Online-Quelle [10.08.2019]........ 15

Abbildung 13: Model-View-Presenter-Muster, Quelle: In Anlehnung an Rishab Software (2016), Online-Quelle [18.09.2019]. 16

Abbildung 14: Model-View-ViewModel-Muster, Quelle: In Anlehnung an Rishab Software (2016), Online-Quelle [18.09.2019]. 17

Abbildung 15: Modellbildung, Quelle: Fleischmann/Stefan/Schmidt/Stary (2018), S. 20. 18

Abbildung 16: Wasserfallmodell, Quelle: IONOS SE 1 (2019), Online-Quelle [31.07.2019], (leicht modifiziert). 19

Abbildung 17: Unterschied des Softwareentwicklungsprozesses zwischen konventioneller objektorientierter Programmierung und MDSD, in Anlehnung an Pietrek (2008), Online-Quelle [30.7.2019]........ 21

Abbildung 18: Einsatzgebiet von UML, Quelle: Seemann/Wolff von Gudenberg (2006), S.8. 23

Abbildung 19: UML-Klassendiagramm und Beispiel einer generierten Klasse in Eclipse Papyrus, Quelle: Eigene Darstellung. 24

Abbildung 20: Übersicht der Funktionsweise von FreeMarker. Quelle: FreeMarker 1 (2019), Online-Quelle [11.08.2019]. 28

Abbildung 21: FreeMarker IDE im Eclipse Marketplace. Quelle: Eigene Darstellung. 30

Abbildung 22: Beispiel zur Verarbeitung von Strings mit Velocity, Quelle: Eigene Darstellung. 31

Abbildung 23: Veloedit und Veloeclipse im Eclipse Marketplace. Quelle: Eigene Darstellung 32

Abbildung 24: Anzahl der Installationen der jeweiligen Plugins aus dem Eclipse Marketplace. Quelle: Eigene Darstellung. 33

Abbildung 25: Anzahl der Contributors der jeweiligen Template-Engines auf GitHub. Quelle: Eigene Darstellung. 34

Abbildung 26: Darstellung der Funktionsweise von APIs, Quelle: Eigene Darstellung. 36

Abbildung 27: Dokumentation einer Datenstruktur mit Doxygen, Quelle: Eigene Darstellung. 37

Abbildung 28: Command Sender für den Aufruf von APIs, Quelle: Eigene Darstellung. 38

Abbildung 29: Android-Version der Test-App, Quelle: Eigene Darstellung. 39

Abbildung 30: LOGIClink, Quelle: LOGICDATA Elektronic & Software Entwicklungs GmbH 8 (2018), Online-Quelle [21.09.2019] 40

Abbildung 31: Schematische Darstellung des aktuellen Testsystems, Quelle: Eigene Darstellung. 41

Abbildung 32: Ablaufdiagramm des aktuellen Entwicklungsvorgehens, Quelle: Eigene Darstellung. 43

Abbildung 33: Aufbau der JSON-Objekte in der API-Dokumentation, Quelle: Eigene Darstellung 46

Abbildung 34: Beispiel einer Protobuf Message, Quelle: Eigene Darstellung. 47

Abbildung 35: Grafische Oberfläche des Command Senders, Quelle: Eigene Darstellung. 49

Abbildung 36: Schematische Darstellung des Facade Design Patterns, Quelle: Hauer (2010), Online-Quelle [08.10.2019]. 50

Abbildung 37: Schematische Darstellung der Funktionsweise von Wrapper, Quelle: Eigene Darstellung. 52

Abbildung 38: Schematische Darstellung des Testsystems mit automatischer Codegenerierung, Quelle: Eigene Darstellung. 53

Abbildung 39: Darstellung der Konsolenanwendung des Autowrappers, Quelle: Eigene Darstellung. 55

Abbildung 40: Execute-Methode der Java-Subklasse, Quelle: Eigene Darstellung. 56

Abbildung 41: Template-File zur Generierung von C++ Header-Files, Quelle: Eigene Darstellung. 57

Abbildung 42: Codeausschnitt des generierten C++ Headers, Quelle: Eigene Darstellung. 58

Abbildung 43: Aufruf des Proto-Compilers über die CMD-line, Quelle: Eigene Darstellung. 59

Abbildung 44: Aufruf des Proto-Compilers für Swift über den macOS Terminal, Quelle: Eigene Darstellung. 59

Abbildung 45: Generierter Enumerator in C#, Quelle: Eigene Darstellung. 60

Abbildung 46: Darstellung der Android Test-App, Quelle: Eigene Darstellung. 62

Abbildung 47: Codeausschnitt des Templates zur Generierung des Java-Wrappers, Quelle: Eigene Darstellung. 63

Abbildung 48: Codeausschnitt eines generierten Java-Wrappers für Android, Quelle: Eigene Darstellung. 64

Abbildung 49: Verbindungsherstellung und Instanziierung der Klasse LL_Actions, Quelle: Eigene Darstellung. 65

Abbildung 50: Template zur Generierung der Methoden zur Vorbereitung der JSON-RPC-Requests, Quelle: Eigene Darstellung. 66

Abbildung 51: Generierte Methoden zur Erstellung von API abhängigen JSON-RPC-Requests. Quelle: Eigene Darstellung. 67

Abbildung 52: Verarbeitung der empfangenen JSON-RPC-Responses, Quelle: Eigene Darstellung....... 68

Abbildung 53: Einlesen der JSON-Files und Initialisierung, Quelle: Eigene Darstellung. 70

Abbildung 54: Beispielhafte Darstellung der Implementierung des Layouts der TabPages, Quelle: Eigene Darstellung. 72

Abbildung 55: Aufruf der Methode GetClassProperties bei mindestens einem Übergabeparameter der API, Quelle: Eigene Darstellung. 72

Abbildung 56: Codeausschnitt zur typabhängigen Steuerelement Generierung. Quelle: Eigene Darstellung. 74

Abbildung 57: Implementierung des ButtonClickMethodHandler. Quelle: Eigene Darstellung................. 75

Abbildung 58: Implementierung der Methode SetULDAPIMethod. Quelle: Eigene Darstellung................ 75

Abbildung 59: Aufruf der Methoden zur Vorbereitung der JSON-RPC Requests und Ausgabe in der Konsole. Quelle: Eigene Darstellung. 76

Abbildung 60: Darstellung des generierten Command-Senders, Quelle: Eigene Darstellung.................. 77

TABELLENVERZEICHNIS

Tabelle 1: Bewertungsschema zur Auswahl der Codegenerierungsmethoden, Quelle: Eigene Darstellung. 26

Tabelle 2: Bewertung der Codegenerierungsmethoden, Quelle: Eigene Darstellung. 27

Tabelle 3: Vergleich zwischen Apache Velocity und Apache FreeMarker. Quelle: Eigene Darstellung. ... 34

Tabelle 4: Beschreibung der Eigenschaften eines JSON-Objekts der Dokumentation, Quelle: Eigene Darstellung. 45

Tabelle 5: Auflistung der unterstützten Datentypen und dazugehörigen Steuerelemente. Quelle: Eigene Darstellung. 73

ABKÜRZUNGSVERZEICHNIS

API .... Application Programming Interface

FTL .... FreeMarker Template Language

IDE .... Integrated Development Environment

JSON-RPC .... JavaScript Object Notation Remote Procedure Call

MDA .... Model-Driven Architecture

MDSD .... Model Driven Software Development

MVC .... Model-View-Controller

MVP .... Model-View-Presenter

MVVM .... Model-View-ViewModel

OMG .... Object Management Group

UML .... Unified Modelling Language

VTL .... Velocity Template Language

XML .... Extensible Markup Language