FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Modularer Treiber zur Sprachsteuerung eines Terminals für ältere Anwender und Anwenderinnen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medizinische Informatik

eingereicht von

Clemens Igler, BSc
Matrikelnummer 0925411

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao.Univ.Prof.i.R. Dipl.-Ing. Dr. techn. Wolfgang Zagler
Mitwirkung: Projektass. Dipl.-Ing. Peter Mayer

Wien, 08.04.2018

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Clemens Igler, BSc
Van der Nüll Gasse 79-81/28, 1100 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

1100 Wien, 08.04.2018
Ort, Datum

Unterschrift

Danksagung

Hiermit möchte ich einen Dank an all jene Personen aussprechen, die mich im Zuge des Diplomarbeitsprozesses begleitet und unterstützt haben.

Für die geduldige, freundliche und kompetente Unterstützung bei der Aufbereitung, Entwicklung und Fertigstellung meiner Arbeit möchte ich meinem Betreuer Prof. Dr. Wolfgang Zagler herzlich danken. Ebenso gilt mein Dank Dipl.-Ing. Peter Mayer und Dipl.-Ing. Christian Beck, deren wegweisende Ideen und Rückmeldungen einen wesentlichen Teil zur Erstellung dieser Arbeit beigetragen haben.

Den größten Dank möchte ich jedoch an meine Familie und Freunde aussprechen, die mir meinen universitären Werdegang nicht nur ermöglicht haben, sondern auch während den letzten Monaten eine große Stütze gewesen sind. Hier möchte ich mich insbesondere bei meiner Schwester Claudia Igler für die langwierigen Korrekturlesearbeiten und die aufbauenden Worte bedanken, die mir immer wieder die nötige Kraft und Motivation für die herausfordernden Momente des Studiums gaben. Meinen Eltern, Renate und Gerhard Igler, ist es zu verdanken, dass ich meinen persönlichen wie universitären Weg stets nach meinen Interessen gestalten konnte. Für ihre Unterstützung während des Studiums, sowie Zeit meines Lebens, möchte ich ihnen hiermit großen Dank aussprechen.

Besonders erwähnen möchte ich an dieser Stelle auch meine Lebenspartnerin Anna-Helene Giefing, die mir in allen Lebenslagen liebevoll zur Seite steht und stets ihr offenes Ohr, Geduld und Zuversicht für die schönen und herausfordernden Momente mit mir teilt. Dafür möchte ich ihr mit aller Liebe danken.

Kurzfassung

Durch die großen Fortschritte im Bereich der Spracherkennung befindet sich diese Technologie als Eingabemethode auf dem Vormarsch und findet in einer zunehmenden Anzahl an Geräten und Bereichen Einsatz. Speziell im Bereich Smart Home und Active and Assisted Living (AAL; ehemals Ambient Assisted Living) bietet sich eine Steuerung mittels Spracheingabe an und öffnet unter anderem die Möglichkeit einer barrierefreien Bedienung. Solch ein AAL-System ist an der Technischen Universität Wien bereits in Arbeit und besitzt als zentrale Steuereinheit ein Local User Interface (LUI), das bislang primär über Touchscreen bedient wird.
In dieser Arbeit wird die Möglichkeit einer Implementierung der Bedienung des LUI mittels eines Treibers zur Spracherkennung untersucht und mit einem Prototyp verwirklicht. Der Prototyp besteht aus einer Kernkomponente und einer Komponente zur Spracherkennung. Die Kernkomponente wurde mittels des Node.js-Frameworks umgesetzt und übernimmt die Kommunikation mit dem LUI und der Komponente zur Spracherkennung, sowie die Verarbeitung und Verteilung der Daten. Diese Komponente bietet außerdem ein klar definiertes und dokumentiertes Interface, welches es ermöglicht, die unterschiedlichsten Anwendungen zur Spracherkennung anzubinden und für die Steuerung des LUI zu nutzen. Für die endgültige Umsetzung des Treibers wurde die Microsoft Speech Platform eingesetzt, für die Evaluierung der Spracherkennung wurde aber auch die neuere Microsoft Technologie – Windows Speech API – herangezogen.

Die Steuerung des LUI mittels Spracherkennung erwies sich als zufriedenstellend, wobei sich die Windows Speech API als alltagstauglicher herausstellte. Im Weiteren wurden durch die Evaluierung wichtige Ansatzpunkte für die Weiterentwicklung der Steuerung des LUI mittels Spracherkennung sowohl seitens der Spracherkennungstechnologie, als auch des LUI identifiziert.

Schlüsselwörter: Active and Assisted Living, Ambient Assisted Living, Local User Interface, LUI, automatische Spracherkennung, Windows Speech API, Microsoft Speech API

Abstract

Owing to the big advances in the field of speech recognition, this technology is on the rise as a new input method and is used in an increasing number of fields and devices. Particularly, the field of Smart Home and Active and Assisted Living (AAL, former Ambient Assisted Living) lends itself to control by means of speech recognition and provides, among other possibilities, a barrier-free operation. Such a system is already under construction at the University of Technoloy in Vienna and currently uses as a central control unit a Local User Interface (LUI) that is mainly controlled through a touchscreen.
In this study the possibility of implementing a control via a speech recognition driver for the LUI is evaluated and realized in a prototype. The prototype consists of a core component and a component for speech recognition. The core component was implemented using the node.js-framework and carries out the communication with the LUI and with the component for speech recognition, as well as it processes and distributes the data. Moreover, this component offers a clearly defined and well-documented interface, which provides the possibility to integrate different speech recognition technologies to control the LUI. The Microsoft Speech Platform was chosen for the final driver implementation but for the evaluation of the speech recognition control the newer Microsoft technology called Windows Speech API was employed as well.

Using speech recognition to control the LUI proved to be satisfactory with the Windows Speech API showing a higher suitability for daily use. Furthermore, the evaluation identified important starting points for further development of a speech recognition control for the LUI regarding the speech recognition technology as well as the LUI itself.

Keywords: Active and Assisted Living, Ambient Assisted Living, Local User Interface, LUI, automatic speech recognition, Windows Speech API, Microsoft Speech API

INHALTSVERZEICHNIS

1. Einleitung

1.1. Problemstellung

Durch den fortwährenden demographischen Wandel und den zunehmenden Altersdurchschnitt unserer Gesellschaft, haben sich auch die sozialen Bedürfnisse vieler Menschen gewandelt. Für Österreich wird beispielweise prognostiziert, dass bis zum Jahr 2020 der Anteil der Bevölkerung mit einem Alter von 65 Jahren oder mehr bei 19,1% liegen wird [1]. Die wachsenden Bedürfnisse dieser Bevölkerungsgruppe stellen die Entwicklerinnen und Entwickler neuer Technologien zweifelsohne vor neue Herausforderungen: Der stetig steigende Altersdurchschnitt verlangt eine Anpassung neuer Technologien, um die Partizipation aller Menschen gleichermaßen gewährleisten zu können, sowie die Entwicklung von Technologien, die speziell an diese Altersgruppe angepasst sind. Insbesondere ältere Menschen haben einen wachsenden Bedarf, moderne Kommunikations- und Telekommunikationsmittel zu nutzen, unter anderem, um dem Bedürfnis nachzukommen, den eigenen Wohnraum solange wie möglich selbstständig nutzen zu können. Die Weiterentwicklung und Anpassung einer barrierefreien Nutzung verschiedener Kommunikationstechnologien erweist sich daher als unabdingbar.
Alle Konzepte und Methoden, die entwickelt werden, um diese Anforderungen zu erfüllen, werden unter dem Begriff altersgerechte Assistenzsysteme zusammengefasst. International werden solche Assistenzsysteme vor allem unter dem Begriff Active and Assisted Living (AAL; ehemals Ambient Assisted Living) zusammengefasst. Am Zentrum für Angewandte Assistierende Technologien (AAT) der Technischen Universität Wien wird bereits an einem solchen AAL System gearbeitet (Projekt CongeniAAL). In diesem System wird ein auf Windows basierendes Touch Terminal, namens Local User Interface (LUI), als zentrale Steuereinheit eingebunden und bildet dort das Herzstück des Systems. Das LUI ist ein speziell entwickeltes User Interface, das auf die Bedürfnisse von älteren Menschen ausgerichtet ist und sich momentan primär über einen Touchscreen oder mit herkömmlichen Eingabegeräten, wie Maus und Tastatur, bedienen lässt. Daraus ergibt sich die Notwendigkeit, die vorhandenen Eingabemodalitäten zu erweitern, um eine barrierefreie Nutzung des LUI zu gewährleisten.

1.2. Zielsetzung

Ziel dieser Diplomarbeit ist es, die vom LUI angebotenen Schnittstellen, welche der Erweiterung der Funktionalität der Software dienen, für die Entwicklung und Anbindung einer weiteren Eingabemodalität zu nutzen. Konkret soll eine Steuerung implementiert werden, welche ausschließlich mittels lokaler Spracherkennung bedient werden kann, sodass keine der bereits vorhandenen Eingabemethoden zusätzlich benötigt wird. Ausgehend von dieser Zielsetzung wurde folgende Forschungsfrage für diese Arbeit aufgestellt:
Wie sieht eine geeignete Lösung aus, um ein User-Interface, am Beispiel des Local User Interface (LUI), um einen Treiber zur Bedienung mittels einer lokalen (embedded) Spracherkennung von Microsoft, zu erweitern? Wie kann diese Erweiterung implementiert werden und in wie weit werden die identifizierten Anforderungen erfüllt?

1.3. Motivation

Die stetigen Fortschritte und Weiterentwicklungen im Bereich der Spracherkennung bieten die Möglichkeit, diese Technologien nicht mehr nur als zusätzliches Feature oder ergänzende Bedienmöglichkeit, sondern in erster Linie als primäre Eingabemethode zu nutzen. Die Möglichkeit einer vollständigen Bedienung mittels Spracherkennung bedeutet einen weiteren Schritt in Richtung einer barrierefreien Nutzung von Kommunikationstechnologien für alle Menschen. Ausgangspunkt für die Beschäftigung mit dieser Thematik war für den Autor also nicht nur das technische Interesse an modernen Spracherkennungssystemen, sondern auch die Möglichkeit, bei der Umsetzung einer barrierefreien Kommunikationstechnologie mitzuwirken.

1.4. Methodisches Vorgehen

- Literaturrecherche und Erhebung des State of the Art
  Zu Beginn der Diplomarbeit wird durch umfassende Recherchen der bestehenden Literatur im Bereich der Spracherkennung eine fundierte Basis für ein strukturiertes, weiteres Vorgehen geschaffen. Im Rahmen dessen wird ein Überblick über die gesamte Thematik der Automatic Speech Recognition (ASR) gegeben. Die Darstellung des State of the Art dient hierbei der Erhebung der für die Aufgabenstellung relevanten Technologien. Diese Vorgehensweise war notwendig, um ein technisches Grundverständnis für die Thematik zu erlangen, welches in weiterer Folge die Basis für die Entwicklung der Software darstellt.

- Auswahl der technischen Möglichkeiten
  Zunächst erfolgt eine Evaluierung der technischen Alternativen, welche für die Umsetzung der Anforderungen dieser Arbeit in Frage kommen. Das inkludiert ein Abwägen der Vor- und Nachteile der unterschiedlichen Technologien. Dies ist nicht nur für diese Arbeit, sondern auch für mögliche zukünftige Weiterentwicklungen der zu implementierenden Treibersoftware relevant. Dieser Auswahlprozess beinhaltet eine Analyse der angebotenen Software Development Kits (SDK) im Bereich der Spracherkennung, sowie der Beispielimplementierungen dieser Technologien. Diese Ergebnisse haben infolge dessen direkten Einfluss auf die im nächsten Schritt folgende Entwicklung des Treibers.

- Entwickeln eines Prototyps des Treibers
  Zu Beginn dieses Arbeitsschritts werden die Zielsetzung und Forschungsfrage dieser Arbeit mittels Requirements Engineering analysiert und definiert. Dazu werden zunächst die allgemeinen Anforderungen festgehalten, welche anschließend aus einer detaillierteren technischen Perspektive in verschiedene Teilaufgaben aufgeschlüsselt werden. Im Rahmen dieses Vorgangs erfolgt auch das Identifizieren von Use Cases mittels einer Anwendungsfallanalyse. Anschließend werden für alle Schnittstellen der zu entwickelnden Software Interface-Definitions erstellt. Anhand dieser Interface-Definitions, sowie der bisher im Auswahlprozess gewonnenen Erkenntnisse, soll in dieser Phase ein Prototyp eines Treibers zur Spracherkennung für das LUI entwickelt

werden. Die Entwicklung wird, soweit eine Anwendung der einzelnen Ebenen auf dieses Projekt möglich ist, auf einer agilen Vorgehensweise basieren und unter Einhaltung von Best Practices durchgeführt. Somit kann durch das zyklische (iterative) Durchlaufen der einzelnen Entwicklungsphasen auf entstehende Probleme oder geänderte Anforderungen flexibel eingegangen werden. Anschließend wird der Treiber ausführlich getestet und in das LUI Environment integriert. Das erfolgt sowohl durch individuelle Unit-Tests, als auch Integrationstests.

- Evaluierung mit Anwenderinnen und Anwendern
  Nachdem die Funktion der Software, beziehungsweise des Systems, überprüft wurde, werden des Weiteren Tests mit Spracheingaben von Probandinnen und Probanden durchgeführt. Bei diesen Tests soll ausschließlich die Erkennungsleistung der Spracherkennung unter realitätsnahen Bedingungen evaluiert werden. Im letzten Teil dieses Abschnitts erfolgt die Evaluierung des gesamten, im Rahmen dieser Arbeit entwickelten, Treibers. Anhand der Ergebnisse dieser Evaluierung kann festgestellt werden, ob die anfangs definierten Anforderungen erfüllt wurden. Außerdem werden etwaige Probleme identifiziert und transparent gemacht, um nicht zuletzt einen Ausblick auf Weiterentwicklungsmöglichkeiten geben zu können.

1.5. Aufbau der Arbeit

Die vorliegende Arbeit gliedert sich in sieben Teile. In einem einleitenden Kapitel wird auf die Fragestellung, Zielsetzung, Motivation sowie das methodische Vorgehen näher eingegangen. Im zweiten Teil wird der aktuelle Forschungsstand („State of the Art“) von Spracherkennungssystemen dargelegt und anschließend werden im Kapitel *Grundlagen* jene Methoden und Konzepte offen gelegt, die für die Forschungsarbeit eingesetzt wurden. Der vierte Teil beinhaltet die Anforderungs- und Use Case-Analyse, sowie eine Ausarbeitung einer detaillierten Zielsetzung. Die Realisierung und Testung des entwickelten Prototypen des Treibers wird im fünften Teil vorgestellt. An dieser Stelle werden auch jene Technologien beschrieben, die im Zuge der Implementierung des Treibers und seiner Komponenten zum Einsatz kamen. Im vorletzten Abschnitt der Arbeit erfolgt mit Hilfe von Anwenderinnen und Anwendern schließlich die Evaluierung der Spracherkennung und des Treibers, sowie die Präsentation der erhobenen Forschungsdaten. Den Abschluss der Arbeit bildet das Resümee, in welchem die wichtigsten Ergebnisse auf den Punkt gebracht und Perspektiven für weiterführende Forschungsarbeiten in diesem Bereich aufgezeigt werden sollen.

2. State of the Art

Im folgenden Kapitel dieser Arbeit wird der Stand der Technik (State of the Art) im Bereich der Spracherkennung dargelegt. Die Aufbereitung des Forschungsstandes trägt wesentlich zum Verständnis, sowie der Konzeptualisierung des weiteren Forschungsverlaufs bei und wird daher wie folgt dargestellt: Im ersten Abschnitt dieses Kapitels werden ausschließlich der technische Hintergrund und die Funktionsweise von Systemen zur automatischen Spracherkennung behandelt. Dieser Teil schließt mit einem kurzen Exkurs zur Thematik lokale (embedded) Spracherkennung versus Cloud Spracherkennung, um darzustellen, warum Cloud Spracherkennung für diese Arbeit unerwünscht war. Der zweite Abschnitt enthält Informationen über aktuell bekannte und verbreitete Frameworks und Software Development Kits (SDKs) im Bereich der Spracherkennung, sowie eine detailliertere Übersicht der Technologien von Microsoft.

2.1. Aufbau und Funktionsweise von Spracherkennungssystemen

In diesem Abschnitt wird dargelegt, welchem Schema der Aufbau einer automatischen Spracherkennung im Allgemeinen folgt (siehe *Abbildung 1*). Dabei lässt sich die Spracherkennung sehr generisch in zwei Schritte unterteilen, die Repräsentation des eingegangenen Sprachsignals und die Suche nach den Wörtern, die dieses Signal enthält. Die Suche basiert dabei auf einem akustischen, einem lexikalischen und einem Sprachmodell. Damit das System von Beginn an eine gute Erkennungsleistung liefern kann, muss vorab jedes einzelne Modell, welches im Rahmen der Spracherkennung eingesetzt wird, mittels Trainingsdatensätzen (Training Data) angelernt werden.

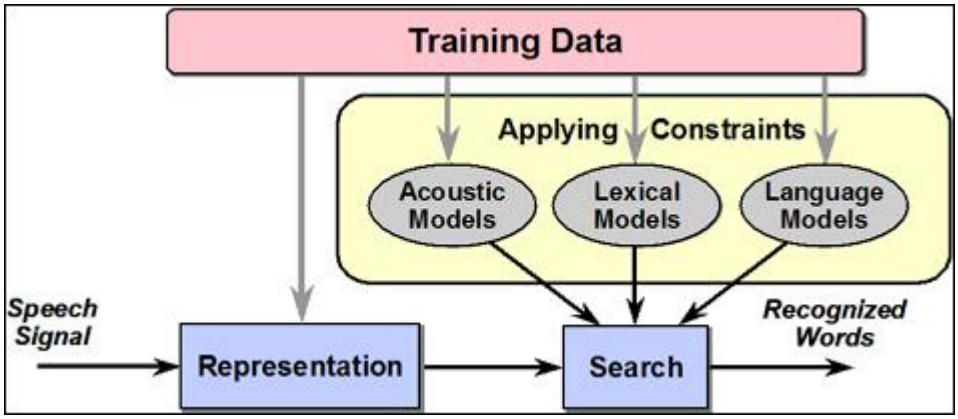


Abbildung 1: Die wichtigsten Komponenten und Bestandteile eines modernen Systems für die automatische Spracherkennung.
Der Ablauf wird durch eine gesprochene Eingabe gestartet, welche vom Mikrofon aufgenommen wird. Dieses aufgenommene Signal wird in analoger Form an das Spracherkennungssystem weitergeleitet, welches das Signal verarbeitet und die erkannten Worte als Ergebnis liefert [2].

Als erster Schritt wird eine Vorverarbeitung dieser Eingabe durchgeführt, welche dazu dient, das empfangene analoge Signal zu digitalisieren. Diese Transformation des Signals (Digitalisierung) erfolgt durch eine Abtastung und Quantisierung des Signals (siehe *Abbildung 2*).

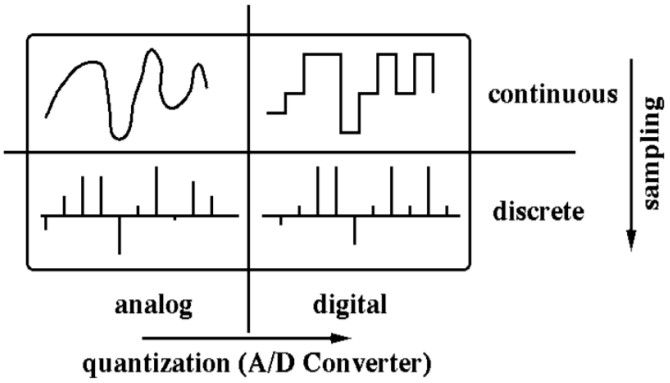


Abbildung 2: Sampling und Quantisierung eines Sprachsignals [3]

Die Abtastung, auch Sampling genannt, wird äquidistant in der Zeit durchgeführt. Das bedeutet, dass mit einer bestimmten Frequenz ein Messwert des analogen Signals erfasst wird. Bei der Abtastung des Signals wird das Signal zeitlich diskretisiert (Zeitdiskretisierung), das bedeutet, dass ein zeitkontinuierliches Signal in ein zeitdiskretes Signal umgewandelt wird. Bei der Abtastung des Signals muss außerdem das Nyquist-Shannon-Abtasttheorem berücksichtigt werden. Dieses Theorem besagt, dass für eine beliebig genaue Rekonstruktion eines Signals die Abtastfrequenz das Doppelte der höchsten Frequenz im Ursprungssignal betragen sollte [56].
Eine weitere Herausforderung beim Sampling ist der Aliasing Effekt. Bei diesem, auch aus der Bildverarbeitung bekannten Effekt, handelt es sich um Fehler und Störgeräusche, die beim Abtasten auf Grund einer zu geringen Abtastfrequenz entstehen. Um einem Aliasing Effekt entgegenzuwirken, kann die Abtastrate (Frequenz) erhöht werden beziehungsweise ein Anti-Aliasing Filter zur Limitierung der Signalbandbreite eingesetzt werden [3].
Bei der Quantisierung des Signals erfolgt die Diskretisierung der Werte des Signals zu den jeweiligen Abtastpunkten. Das bedeutet, es wird ein wertkontinuierliches Signal in ein wertdiskretes Signal umgewandelt. Wichtige Kennzahlen bei der Quantisierung eines Signals sind der durchschnittliche Quantisierungsfehler und das Signal-zu-Störgeräusch-Verhältnis (signal to noise ratio, kurz SNR). Die Performance einer Spracherkennung ist umso schlechter, je niedriger die SNR ist. Bei der Quantisierung des Sprachsignals ist außerdem die Wahl der sogenannte Abtasttiefe (Sampling Depth) ausschlaggebend für ein gutes Ergebnis. Sprachsignale bewegen sich üblicherweise in einem Bereich von 50 dB bis 60 dB und werden bei der automatischen Spracherkennung üblicherweise mit 16 Bit quantisiert.

Nach der Abtastung und Quantisierung wird ein weiteres Mal eine digitale Signalverarbeitung durchgeführt. In diesem Schritt wird ein Verfahren namens Merkmalsextraktion durchgeführt, bei dem die für die Spracherkennung relevanten Frequenzmerkmale aus dem Signal extrahiert werden. Die Durchführung dieses Prozesses hat mehrere Gründe, beispielsweise funktioniert das Gehör des Menschen auf Basis von Frequenzanalyse. Wie in vielen anderen Bereichen der Wissenschaft versucht man auch hier, sich an bestehenden Mechanismen der Natur zu orientieren. Aus der reinen „Time Domain Waveform“ könnten außerdem nur sehr schwer Informationen abgeleitet werden, beziehungsweise wäre sie deutlich schwerer zu le-

sen und zu verstehen. Durch die Frequenzanalyse kann daher die Signalverarbeitung vereinfacht werden und es kann die im Eingangssignal enthaltene phonetische Information erfasst (extrahiert) werden.

Das eigentliche Kernstück von modernen Spracherkennungssystemen ist die auf der Bayes'schen Entscheidungsregel basierende Fundamentalgleichung der automatischen Spracherkennung (siehe *Abbildung* 3) [4].

$$P(w \mid X) = \frac{\mathrm{P}(\mathrm{X} \mid w)\ \ \mathrm{P}(w)}{\mathrm{P}(\mathrm{X})}$$

Abbildung 3: Fundamentalgleichung der automatischen Spracherkennung [4]

w … wahrscheinlichste Wortkette
X … Merkmalsvektor
P(w) … a-priori Verteilung möglicher Wortfolgen
P (X) … Wahrscheinlichkeit des Merkmalsvektors
P(w | X)… a-posteriori Wahrscheinlichkeit
P(X | w)… bedingte Verteilungsdichte

Wenn die (bedingten) Wahrscheinlichkeiten bekannt sind, kann anhand der Bayes'sche Entscheidungsregel festgestellt werden, wie für ein Problem von entscheidungstheoretischer Art eine minimale Fehlerrate erreicht werden kann. Im Fall der Spracherkennung soll für die aus dem Eingangssignal stammenden Merkmalsvektoren ***X*** das wahrscheinlichste Wort $\hat{\mathbf{w}}$ aus einer Folge von geäußerten Wörtern w gefunden werden [5].

Beim ersten Teil der Fundamentalgleichung handelt es sich um das akustische Modell. Dieses Modell beschreibt dabei die Wahrscheinlichkeit des Merkmalsvektors ***X*** für ein gegebenes Wort w und stellt somit den Zusammenhang zwischen dem Merkmalsvektor und den im Vokabular enthaltenen Wörtern dar. Das Modell besteht meistens aus zwei Teilen, dem eigentlichen Akustikmodell und dem Aussprachewörterbuch (phonetischen Modell). Ein phonetisches Modell beschreibt Phoneme mittels Merkmalsvektoren [6].
Die Zerlegung von Wörtern in kleinere Einheiten – wie den Phonemen – hat vor allem den Grund, dass sowohl der Rechen- als auch Speicheraufwand für ganze Wörter enorm hoch ist. Der Muster-basierte Ansatz wurde neben Performancegründen auch deshalb durch andere Modellierungsprozesse ersetzt, weil er weder für die Sprecherinnen und Sprecher-unabhängige Spracherkennung, noch für die kontinuierliche Spracherkennung geeignet war.
Die bedingten Verteilungsdichten ***P(X | w)*** sind allerdings nicht exakt bekannt, sondern können nur angenähert werden. Hierfür werden üblicherweise Hidden Markov Modelle eingesetzt.
Mit einem Hidden Markov Modell wird die Struktur der gesprochenen Sprache in einem integrierten und konsistenten statistischen Modell-Framework abgebildet. Es werden aber auch die Variabilität des Sprachsignals und die daraus resultierenden spektralen Eigenschaften modelliert. All das wird mittels eines zweifachen stochastischen Prozesses realisiert. Das zentrale Element des Hidden Markov Modells ist die Markov Kette (siehe *Abbildung 4*). Diese besitzt ***n*** Zustände, sowie Übergangswahrscheinlichkeiten, mit welchen das System zu einem Zeitpunkt ***t*** in einen anderen Zustand wechselt. Im Fall der Spracherkennung repräsen-

tiert diese Kette eine linguistische Struktur, die aus einer Variation von Wörtern, einem sogenannten Trainigsset, besteht. [6] [7]

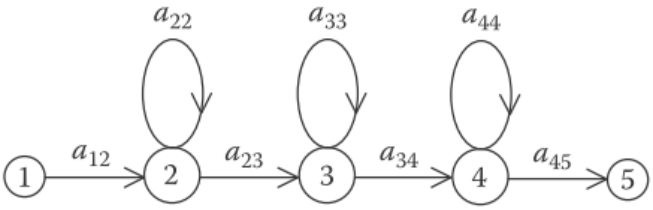


Abbildung 4: Markov Kette mit fünf Zuständen. Die Knoten 1 bis 5 stellen die möglichen Zustände in der Markov Kette dar. Die Verbindungen a^{ij} repräsentieren die Übergänge/Übergangswahrscheinlichkeiten zwischen den Zuständen i und j. In der Schleife a_{ii} des Knoten i verweilt das System bis zum nächsten Zustandswechsel. [6]

Der zweite Teil der Gleichung (siehe *Abbildung 3*) beinhaltet das sogenannte Sprachmodell, welches die Erkennung und das Verstehen der natürlichen Sprache verbessern soll. Um das zu ermöglichen, beinhaltet dieses Modell eine Wissensbasis, welche sich aus dem lexikalischen Wissen (Vokabeldefinition und Aussprache), der Syntax und Semantik (grammatikalische Regeln) und der Pragmatik (die kontextabhängige Bedeutung von Wörtern) der jeweiligen Sprachen zusammensetzt. Bei den Sprachmodellen wird zwischen zwei Ansätzen unterschieden, dem statistischen Sprachmodell und deterministischen Sprachmodell [6].
Mit statistischen Ansätzen wird versucht, die Wahrscheinlichkeit des Auftretens einer Wortsequenz **W** darzustellen. In der formalen Sprachtheorie wäre P(W) entweder 1.0, was bedeuten würde die Wortsequenz **W** ist ein gültiges Ergebnis, oder 0.0 für ein ungültiges Ergebnis. Da die gesprochene Sprache aber Regeln (Grammatik) besitzt, ist diese Theorie für diesen Anwendungsfall unbrauchbar.
Es besteht also nur die Möglichkeit, die Auftrittswahrscheinlichkeit einer Wortsequenz ***W*** anhand von A-priori-Wissen aus Trainingsdaten möglichst genau zu schätzen. State of the Art für diese Herangehensweise ist der Einsatz von N-Grammen. Bei N-Grammen handelt es sich um eine Form von Sprachmodellen, basierend auf Markov-Modellen, die dazu verwendet werden, das nächste Element (Phonem, Wort, etc.) in einer gesuchten Sequenz zu schätzen [8] [9].
Mit statistischen Sprachmodellen kann die natürliche Sprache vollständig beschrieben werden, was auch Eingaben ermöglicht, für die keine A-priori-Wahrscheinlichkeiten bekannt sind. Diese Modelle eigenen sich daher für Anwendungen mit großen Vokabularen oder freigesprochene Eingaben, ein klassisches Beispiel für einen solchen Anwendungsfall wäre ein Programm für Diktate [5].
Deterministische Sprachmodelle hingegen definieren über Grammatiken einen deutlich kleineren Satz an Regeln und Wortkombinationen und geben somit an, ob eine Wortfolge möglich oder unmöglich ist. Bei Grammatiken kann noch unterschieden werden in die deterministischen Grammatiken, welche eine formale Definition der Sprache darstellen und in die probabilistischen Grammatiken, welche die formale Definition generalisieren und wahrscheinlichkeitstheoretische Abhängigkeiten zwischen den Wortsequenzen ableiten. Deterministische Sprachmodelle bieten Entwicklerinnen und Entwicklern eine gute Möglichkeit, das Vokabular selbst zu definieren und Wörter wie Eigennamen oder Regeln für die natürliche

Sprache hinzuzufügen. Somit eigenen sich Grammatiken beispielsweise sehr gut für Applikationen mit Sprachsteuerung.

Das akustische Modell und das Sprachmodell (siehe *Abbildung 1*) bilden mit ihrem Wissen nun eine Basis für das tatsächliche Suchverfahren eines Spracherkennungssystems. Mit dem Suchverfahren soll nur im Suchraum (das heißt in allen möglichen Wortfolgen) jene Wortfolge gefunden werden, welche die höchste Wahrscheinlichkeit aufweist. Aus der Fundamentalgleichung der Spracherkennung (siehe *Abbildung 3)* ist nun ersichtlich, dass es sich hierbei um jenes Produkt aus dem akustischen Modell und dem Sprachmodell handelt, welches die höchste Wahrscheinlichkeit aufweist. Um diese Wahrscheinlichkeiten zu erhalten, müssen die Merkmalsvektoren klassifiziert werden. Da das aber vor allem bei größeren Vokabularen viel zu rechenintensiv und langsam wäre, müssen intelligente Algorithmen und Heuristiken eingesetzt werden, um möglichst effizient zu suchen [4].
Bei der Einzelworterkennung kann noch relativ trivial, beispielsweise mit einem simplen Forward-Algorithmus, die Wahrscheinlichkeit für ein Wort berechnet werden. Bei der kontinuierlichen Spracherkennung muss, wie bereits erwähnt, deutlich mehr auf Effizienz geachtet werden. Allgemein betrachtet muss hier sowohl die beste Wortfolge als auch die beste Zustandsfolge innerhalb der Wörter gefunden werden. Die Suche kann somit auch mit dem Zustandsraum-Suche-Paradigma (state search paradigm) beschrieben werden. Es gilt nun in diesem Zustandsgraphen den Pfad mit den geringsten Kosten vom Startzustand zum Endzustand zu finden [9] [10].
Bei diesem Zustandsgraphen (Gitter aus Zuständen) handelt es sich um eine Finite State Machine (FSM), zu welchen auch das Hidden-Markov-Model zählt.
Für diese Problemstellung ist zum momentanen Zeitpunkt in der Spracherkennung einer der verbreitetesten Ansätze die Viterbi-Suche. Mit dem auf Rekursion basierenden Viterbi-Algorithmus kann der wahrscheinlichste Pfad (in diesem Fall jener mit den geringsten Kosten) durch ein solches Gitter von Zuständen gefunden werden, dieser Pfad wird auch Viterbi-Pfad genannt. Dieser, der dynamischen Programmierung zugehörigen Algorithmus, kann als eine Art Netzwerk angesehen werden, welches Zustände zu bestimmten diskreten Zeitpunkten darstellt. Die Verbindungen zwischen diesen Zuständen repräsentieren die Zustandsübergänge und sind anhand ihrer Übergangswahrscheinlichkeit (maximum a posteriori Wahrscheinlichkeit) gewichtet. So wird bei jeder Iteration der Übergang mit der höchsten Wahrscheinlichkeit zum Pfad hinzugefügt bis der Endzustand erreicht ist.
Im konkreten Fall eines Hidden Markov Models werden beim Viterbi-Algorithmus in dem aus Markov-Ketten bestehenden Gitter die verborgenen Zustände des Modells unter Berücksichtigung einer beobachteten Sequenz von Worthypothesen betrachtet. Am Ende der Viterbi-Suche wird anhand der wahrscheinlichsten **n** Pfade eine Hypothese für die beste Wortfolge erstellt [8] [9] [11].

An diesem Aufbau von Spracherkennungssystemen hat sich im Allgemeinen über einen langen Zeitraum nichts mehr verändert. Hingegen an der Herangehensweise, die Modelle (für akustisches, lexikalisches und Sprachmodell) zu trainieren, schon. Da diese Modelle die Grundlage für die Leistungsfähigkeit einer Spracherkennung darstellen, ging vor allem im letzten Jahrzehnt der Trend in die Richtung, mittels Deep Learning Ansätzen und neuronalen Netzen große Datenmengen (Trainingsdaten) für die Modelle nutzen zu können. Diese sehr rechenintensiven Ansätze können unter anderem auch deshalb eingesetzt werden, weil die Spracherkennung immer häufiger in einer Cloud durchgeführt wird und daher auch erheblich mehr Ressourcen zur Verfügung stehen, als bei lokalen Systemen [9] [12].

Lokale versus Cloud-basierte Spracherkennung

Aufgrund der Performance, die ein wichtiger Bestandteil für die Verarbeitung der Datenmengen darstellt, sowie dem Speicherplatz, der beispielsweise für die unterschiedlichen Modelle (akustisches, lexikalisches, Sprachmodell) zur Spracherkennung benötigt wird, geht der Trend in den letzten Jahren immer mehr in Richtung Spracherkennung via Cloud. Mit einer Cloud-Lösung kann nicht nur eine performante Spracherkennung für den jeweiligen Audio-Input geliefert werden, sondern es können beispielsweise auch Deep Learning Ansätze für große Datenmengen (Big Data) eingesetzt werden. Das ermöglicht, die Modelle (akustisches, lexikalisches und Sprachmodell) stetig mit einer riesigen Menge an neuen Daten zu erweitern und zu verbessern und somit auch die Erkennungsleistung laufend zu steigern. Im Gegensatz dazu müssen lokale Spracherkennungen mit der Wissensbasis auskommen, die auf dem jeweiligen Gerät zu Verfügung steht. Diese Modelle können ausschließlich durch die Benutzung angelernt werden und sind damit auch sensitiver bezüglich der jeweilige Anwenderin oder dem Anwender [12] [83].

Cloud-Services für den Bereich der Spracherkennung zu nutzen wird außerdem dadurch begünstigt, dass die Internet-Verfügbarkeit und die Übertragungsraten in den letzten Jahren allgemein und darüber hinaus auch für mobile Geräte, enorm verbessert wurden.

Spracherkennung, die ausschließlich lokal auf dem jeweiligen Zielgerät durchgeführt wird, hat hingegen den Vorteil, dass keine Übertragungszeiten anfallen. Zudem werden Nutzerinnen und Nutzer dahingehend entlastet, dass die Bedenken hinsichtlich des Datenschutzes bei einer lokalen Lösung nichtig werden. Denn die Skepsis gegenüber der unwissentlichen Aufzeichnung der Daten ist groß – insbesondere wenn alles Gesprochene stets vom Mikrofon mitgehört und an die Cloud geschickt wird. Die damit verbundene Unsicherheit, wer sich letztlich Zugriff auf die persönlichen Daten in der Cloud verschaffen könnte, wird von vielen Nutzern und Nutzerinnen als unangenehm empfunden. Für die Endnutzerin oder den Endnutzer hat die lokale Spracherkennung vor allem bei mobilen Geräten den Vorteil, dass nicht nur weniger Energie sondern auch weniger Datenvolumen verbraucht wird [12] [83].

Um die Vorteile beider Varianten zu kombinieren, wird heutzutage meist eine Hybridvariante eingesetzt. Dazu werden kleine Sets mit einfachen Befehlen (Grammatiken) lokal ausgewertet und durchgeführt, während komplexere Anfragen an die Cloud geschickt werden. Ein Beispiel dafür sind lokal ausgewertete Kommandos, welche die Cloud-basierte Spracherkennung aktivieren („OK Google“, „Hey Siri“, „Alexa“…).

Eine weitere Möglichkeit, die zwei vorgestellten Varianten zu vereinen, zielt auf ein möglichst gutes Ergebnis hinsichtlich der Erkennungsrate ab. Dazu wird in einem ersten Schritt ausschließlich eine lokale Spracherkennung durchgeführt. Liefert dieser Vorgang keinen zufriedenstellenden Confidence-Wert, wird in einem zweiten Schritt eine Spracherkennung in der Cloud getriggert.

Für die Aufgabenstellung dieser Arbeit ist eine Cloud-basierte Spracherkennung nicht erwünscht, beziehungsweise geeignet, da in vielen Pensionistenhäusern nach wie vor keine oder nur schlechte Internetanschlüsse für die Bewohnerinnen und Bewohner zur Verfügung stehen. Das könnte zu regelmäßigen Ausfällen oder langen Wartezeiten bei der Spracherkennung führen. Des Weiteren hat zumeist vor allem diese Zielgruppe deutlich mehr Vorbehalte und Bedenken, was den Datenschutz betrifft und ist dementsprechend skeptisch gegenüber Technologien, die ständig Daten erfassen und in eine Cloud hochladen.

2.2. Überblick bekannter Frameworks und Software Development Kits

Durch die Fortschritte im Bereich der Spracherkennung und der damit einhergehenden Verbreitung dieser Technologie über ein breites Anwendungsspektrum, gibt es inzwischen Produkte und Lösungen, um Spracherkennung mit einem verhältnismäßig geringen Aufwand in neue Software einzubinden. Diese setzen sich aus einer Vielzahl an Frameworks, Services und Software Development Kits (SDKs) zusammen, die von unterschiedlichen Unternehmen oder Universitäten entwickelt wurden und teils auch kostenlos genutzt werden können (oder sogar als Open Source bereitgestellt werden). Im Folgenden wird ein Überblick über diese Entwicklung gegeben, der in Form einer Auswahl an bekannten Frameworks und SDKs im Bereich der Spracherkennung präsentiert wird.

CMUSphinx
Das CMUSphinx Toolkit (oft auch nur Sphinx genannt) wurde an der Carnegie Mellon University entwickelt. Sphinx ist circa seit dem Jahr 2000 (in der Version Sphinx 2) Open Source und frei verfügbar. Seit der ersten Version wird bei Sphinx auf Hidden Markov Models für akustische Modelle, sowie N-Grams für statistische Sprachmodelle gesetzt. Bei CMUSphinx handelt es sich um mehrere Tools und Engines zur Spracherkennung: der Sphinxbase, Sphinx4, Sphinxtrain und Pocketsphinx. Sphinx4 ist die aktuelle Version des anpassbaren (adjustierbaren) „Speech Recognizers“ des Toolkits und in Java geschrieben. SphinxTrain enthält Tools, um ein akustisches Modell (Acoustic Model) zu trainieren. Pocketspinx ist eine „lightweigt Recognizer Library“, mit welcher die Einbindung von Spracherkennung in Applikationen für die unterschiedlichsten Plattformen möglich sein soll. Ein Beispiel dafür sind Apps für Android Smartphones. Der letzte Teil des CMUSphinx Toolkits, namens Sphinxbase [13], stellt die Basis-Bibliothek (Library) dar und wird beispielsweise von Pocketsphinx benötigt. Für CMUSphinx sind aktuell 12 verschiedene Sprachen für akustische Modelle und Sprachmodelle verfügbar [14] [15].

Dragon Naturally Speaking SDKs
Einer der Branchenführer im Bereich der Spracherkennung ist schon seit längerer Zeit die Firma Nuance Communications Inc.. Nach ersten Softwarelösungen für das automatische Diktieren, wurde 1997 die erste kommerzielle Version von Dragon Naturally Speaking veröffentlicht. Die Software befindet sich mittlerweile in der Version 15 und wird, in einigen unterschiedlichen Varianten, in den verschiedenen Branchen eingesetzt. Seit circa 2006 werden von Dragon Communications auch Client und Server SDKs zur Verfügung gestellt, um Entwicklerinnen und Entwicklern von Software für Windows Betriebssysteme die Möglichkeit zu geben, Dragon Naturally Speaking in ihre Programme einzubinden. Inzwischen sind auch SDKs für bestimmte Anwendungen für die Betriebssysteme Linux und Mac OS verfügbar. Für all diese SDKs müssen allerdings im Allgemeinen Lizenzen bei Nuance erworben werden [16] [18]. Auch für mobile Plattformen gibt es inzwischen eine sogenannte Mobile SDK, mit welcher sowohl für Android, als auch iOS (und Windows Phone) Apps mit der Dragon Spracherkennung erweitert werden können. Diese SDK steht kostenlos zur Verfügung, wenn die Entwicklerin oder der Entwickler am Nuance Mobile Developer-Program (NMDP) teilnimmt [17]. Nach Fertigstellung der App kann für 90 Tage, ebenfalls kostenlos, der Nuance-Service für die Cloud-basierte Spracherkennung genutzt werden. Anschließend müssen für alle Services von Nuance Lizenzen erworben werden. Unterstützt werden aktuell bis zu 86 unterschiedliche Sprachen und Dialekte, wobei beispielsweise für die Dragon Mobile SDK momentan nur Englisch (amerikanisch und britisch), Spanisch, Französisch, Deutsch, Italienisch und Japanisch für die Speech-to-Text-Erkennung verfügbar sind [19].

Web Speech API

Bei der Web Speech API handelt es sich um eine Spezifikation, die von der Speech API Community Group der W3C festgelegt wurde. Diese Schnittstellen-Definition dient als Grundlage für die Implementierung von Spracherkennungen in Browsern, beziehungsweise in Webanwendungen und stellt selbst keine Spracherkennung zur Verfügung. Es handelt sich bei der Web Speech API also nur um eine definierte Schnittstelle, über welche jedes beliebige Service zur Spracherkennung (lokal oder online) eingebunden werden kann. Der Standard sieht für die Umsetzung der Schnittstelle die Programmiersprache JavaScript vor. Bisher wurde die Web Speech API lediglich für den Google Chrome Browser und den Mozilla Firefox umgesetzt - allerdings in beiden Fäll nicht vollständig [20].

Google Chrome

Seit der Google Chrome Browser Version 25 ermöglicht Google Entwicklerinnen und Entwicklern die Integration und Nutzung einer Spracherkennung über die Web Speech API. Die in JavaScript geschriebene API zur Sprachsynthese und Spracherkennung kann allerdings nur für Web-Anwendungen eingesetzt werden. Google nutzt hierbei ein eigenes Spracherkennungsservice, das auch bei Google Assistant (ehemals Google Now), beziehungsweise Google Voice Search eingesetzt wird. Dieses Service wird in diesem Kontext ausschließlich Serverseitig ausgeführt, was bedeutet, dass eine lokale (embedded) Spracherkennung nicht möglich ist. Eine Nutzung von Grammatiken ist in der aktuellen Version (63) von Google Chrome ebenfalls nicht möglich [21].

Mozilla Firefox

Im Mozilla Firefox ist die Web Speech API seit der Version 44 verfügbar. Da die meisten modernen Kommunikationsgeräte bereits über eine eigene Spracherkennung verfügen (zum Beispiel Cortana bei Windows 10, Siri bei iOS etc.), wird bei dieser Umsetzung der Schnittstelle auf die Spracherkennung des jeweiligen Geräts, auf dem die Applikation ausgeführt wird, zurückgegriffen. Außerdem ist es auch möglich für die Spracherkennung Grammatiken im JSpeech Grammar Format (JSGF) zu definieren. Momentan ist allerdings noch keine kontinuierliche Spracherkennung implementiert. Das bedeutet, die Entwicklerin oder der Entwickler müsste einen Workaround für eine kontinuierliche Spracherkennung implementieren, oder es wäre notwendig, dass die Anwenderin oder der Anwender die Spracherkennung nach jedem Ergebnis neu startet [22] [23] [24].

Microsoft

Seit circa Mitte der 1990er Jahre bietet Microsoft Entwicklerinnen und Entwicklern von Windows-Programmen die Möglichkeit, Spracherkennung und Sprachsynthese über eine Schnittstelle in die eigenen Anwendungen einzubinden. In den letzten 20 Jahren gab es eine Vielzahl an Weiterentwicklungen und Veränderungen in diesem Bereich. Zum aktuellen Zeitpunkt gibt es von Microsoft drei Systeme beziehungsweise Schnittstellen, die eine lokale Spracherkennung und Sprachsynthese ermöglichen:

- Microsoft Speech API - SAPI (Speech Application Programming Interface)
- Microsoft Speech Platform[1] (Microsoft Speech Server)
- Windows Speech Recognition

Da in dieser Arbeit Spracherkennungstechnologien von Microsoft eingesetzt werden sollen, wird in den folgenden Abschnitten (*2.2.1.* Microsoft Speech API (SAPI) bis *2.2.3.* Windows Speech Recognition) noch genauer auf diese eingegangen [25] [26].

Cloud Service APIs

Nachdem die Spracherkennung via Cloud Services schon sehr häufig von den unterschiedlichen Herstellern eingesetzt wird, werden auch einige APIs zu diesen Diensten angeboten. Zum Zeitpunkt der Recherche für diese Arbeit wurden solche APIs gerade erst veröffentlicht, beziehungsweise von den Herstellern für die nahe Zukunft angekündigt. Beispiele dafür wären die Google Cloud Platform API, Microsoft Speech API oder auch Alexa Voice Service API.

Zum Abschluss dieser Übersicht soll noch ergänzend erwähnt werden, dass es darüber hinaus noch andere Open Source Tools gibt, wie etwa das HTK[2] (Hidden Markov Model Toolkit) der Camebridge University, oder das an unterschiedlichen japanischen Universitäten entwickelte Julius[3]. Beide Toolkits wurden ursprünglich für die Forschung entwickelt, können aber auch problemlos in Anwendungen für unterschiedliche Betriebssysteme eingebunden werden. Es gibt aber auch Toolkits mit User Interfaces, wie jenes namens Simon[4]. Dieses kann sowohl unter Linux, als auch Windows verwendet werden und seine Spracherkennung basiert auf den bereits erwähnten Technologien CMUSphinx, HTK und Julius.

2.2.1. Microsoft Speech API (SAPI)

Microsoft SAPI war von Anfang an dafür konzipiert, Spracherkennung und Sprachsynthese für Desktop-Programme nutzbar zu machen. Bei dieser Speech API gilt es, grundsätzlich zwischen den Versionen 1-4 und den Versionen ab 5.0 zu unterscheiden. Die Versionen 1-4 wiesen größtenteils dieselbe Basis auf und wurden ausschließlich weiterentwickelt. Für die Version 5 fand hingegen ein komplettes Redesign der API statt [27] [33].
SAPI 1 bot, auf einem noch sehr primitiven Level, eine Schnittstelle für eine direkte Spracherkennung und Text-To-Speech Synthese, sowie für Voice Commands an. Diese APIs waren direkte Schnittstellen zu den Control Engines und konnten von Windows Anwendungen eingebunden werden. SAPI 1 wurde 1995 veröffentlicht und konnte unter den Windows Betriebssystemen Windows 95 und Windows NT verwendet werden.
Mit SAPI 3 wurde 1997 eine (noch sehr eingeschränkte) Form der Spracherkennung für das Diktieren von Text veröffentlicht, wobei diese Version noch kein kontinuierliches Diktieren unterstützte, sondern nur diskrete Spracheingaben [27].
Die 1998 veröffentlichte Version 4.0 brachte umfangreichere Verbesserungen und unterstützte auf einem wesentlich höheren Level Voice Commands, Voice Dictation (inklusive

[1] Da es sich bei der „Microsoft Speech Platform“ um einen Produktnamen handelt, sei darauf hingewiesen, dass nachfolgend sowohl englische Schreibform „Platform“ (im Zusammenhang mit Produktnamen), als auch die deutsche Version „Plattform“ verwendet wird.
[2] http://htk.eng.cam.ac.uk/
[3] http://julius.osdn.jp/en_index.php / https://github.com/julius-speech/julius
[4] https://simon.kde.org/

kontinuierlicher Spracheingabe), Voice Talk und Direct Speech Recognition. Auch für die Sprachsynthese wurden umfangreichere Updates veröffentlicht [27].
Wie bereits erwähnt, wurde für die im Jahr 2000 veröffentlichte Version 5.0 des SAPI ein komplettes Redesign der Speech SDK durchgeführt. Ziel dieses Redesigns war es, die Windows Applikation, die auf das SAPI zugreift, von der Speech Engine zu entkoppeln. Der Ansatz für diese Separation war, dass alle Aufrufe des SAPI über eine Runtime sapi.dll geroutet werden und diese bieten ein Set an Schnittstellen (sowohl für die Applikation, als auch für die Speech Engine) an. Der Sinn dieser Vorgehensweise ist beispielsweise, dass 3rd-Party Entwickler auch eigene Engines zur Spracherkennung und Sprachsynthese entwickeln und integrieren können. Unter der Bedingung, dass die Interface Definitions von SAPI eingehalten werden, kann dementsprechend die ursprüngliche Speech Engine ausgetauscht werden, ohne dass für die Applikation, die SAPI einbindet, Änderungen am Interface oder am Zugriff auf die Speech Funktionalitäten nötig werden. In *Abbildung 5* ist eine Übersicht der Architektur von SAPI 5 zu sehen, in der diese Kapselung klar zu erkennen ist [27] [29].

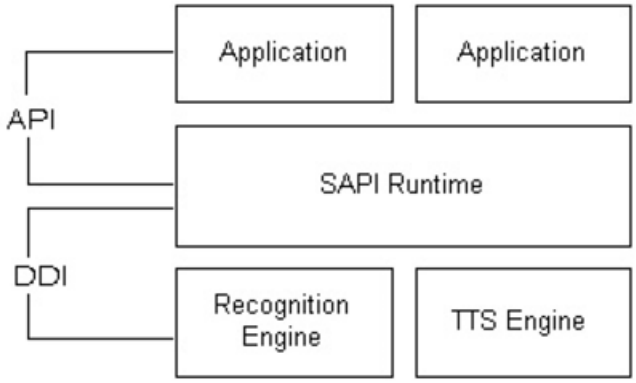


Abbildung 5: Architektur von SAPI in der Version 5.0. Auf der obersten Ebene befinden sich die Applikationen, welche über ein Application Programming Interface (API) mit der SAPI Runtime – auf der darunterliegenden Ebene – verbunden sind. Auf der tiefsten Ebene der Architektur befinden sich die Recognition Engine, sowie die TTS Engine, welche über ein Device Driver Interface mit der SAPI Runtime kommunizieren [28].

Die wichtigsten Features bei SAPI 5.0 [29] [30]:

- Grammatiken und benutzerspezifisches Lexikon:
  Mit Grammatik Objekten wird es ermöglicht, Wörter festzulegen, die erkannt werden sollen, falls sie in einer Spracheingabe enthalten sind. Über die Markup Language XML können diese Wörter (Grammatiken) spezifiziert oder auch dynamisch zur Laufzeit erzeugt werden.
  Über das benutzerspezifische Lexikon können Wörter und Ausdrücke, sowie eine bestimmte Aussprache von Wörtern zum Standard-Lexikon der Speech Engine hinzugefügt werden.

- Recognizer (Spracherkenner):
  In SAPI 5.0 werden für die Spracherkennung sogenannte *Recognizer* Objekte eingesetzt. Diese Objekte zur Spracherkennung können für Desktop-Applikationen auch in einem eigenen Prozess laufen. Das bietet die Möglichkeit, dass der Recognizer gleichzeitig von mehreren Applikationen genutzt werden kann, und ist dementsprechend ressourcensparender.

Es ist aber ebenso möglich, ein Recognizer Objekt explizit nur für eine einzige Applikation nutzbar zu machen. Damit hat lediglich eine spezifizierte Anwendung Kontrolle über diesen sogenannten *In-Proc-Recognizer*.

- Voice Objects:
  Mit Voice Objekten wird bei SAPI 5.0 die Sprachsynthese durchgeführt. Der Prozess der Sprachsynthese mittels dieser Voice Objekte wird über eine Markup Language (allerdings nicht der XML) gesteuert.

- Audio Schnittstellen:
  Mit den Audio Interfaces kann die Spracheingabe über Mikrofone, Audioausgaben über Lautsprecher, sowie Audio-Input von Wave Files durchgeführt und verarbeitet werden.

Die Version 5.0 der Speech API wurde das erste Mal 2001 mit Windows XP ausgeliefert und war auch kompatibel mit Windows 98 und Windows NT 4.0 aufwärts. Nach der Veröffentlichung war SAPI 5.0 vorerst eine reine COM API.
Mit der Ende 2001 ausgelieferten Version 5.1 wurden auch APIs für Visual Basic, Script-Sprachen und Managed Code[5] hinzugefügt.
SAPI 5.2 bot erstmals die Möglichkeit, SRGS (Speech Recognition Grammar Specification) und SSML (Speech Synthesis Markup Languages) zu verwenden. In dieser 2004 veröffentlichten Version waren auch Performance-Verbesserungen und Server Features enthalten.
Mit Windows Vista wurde SAPI 5.3 ausgeliefert, welches als Teil der Windows SDK nun vollständig in das Betriebssystem integriert wurde. Außerdem enthielt diese Version neue Engines für die Spracherkennung und Sprachsynthese und unterstützte den W3C XML-Standard für SRGS und SSML. Neben der Unterstützung von kontextfreien Grammatiken durch die API war es nun auch möglich, durch Skripte (z.B. geschrieben in JavaScript) semantische Interpretationen für Grammatiken hinzuzufügen, beziehungsweise durchzuführen [31].
Die aktuelle Version 5.4 des SAPI wurde erstmals mit Windows 7 ausgeliefert. Kompatibel zu dieser Version sind auch Windows 8 und Windows 10 sowie Windows Server 2008. Aktuell werden standardmäßig acht verschiedene Sprachen für die Spracherkennung, sowie drei Sprachen für die Sprachsynthese unterstützt. Für eine Nutzung muss für die Speech Engine vorab ein Training mit der Anwenderin oder dem Anwender durchgeführt werden und es werden Audio-Dateien im Format 16-bit unterstützt. SAPI kann über den Namespace *System.Speech* eingebunden werden. Durch das User-spezifische Anlernen von Trainingsdatensätzen ist es allerdings nicht möglich, eine solche SAPI-Implementierung direkt weiter zu verteilen [28] [32].

2.2.2. Microsoft Speech Platform

Die Microsoft Speech Platform wurde dafür konzipiert, Entwicklerinnen und Entwicklern von Windows-Server-Anwendungen die Möglichkeit zu bieten, über diverse Interfaces Spracherkennung und Sprachsynthese in ihre Programme einzubinden. Diese Schnittstellen sind in der Speech Platform API zusammengefasst. Die Microsoft Speech Platform setzt sich im Wesentlichen aus der Application Runtime (stellt die eigentlichen Speech Funktionalitäten zur Verfügung) und einer API zur Steuerung dieser Application Runtime zusammen. Diese

[5] https://msdn.microsoft.com/en-us/library/windows/desktop/bb318664(v=vs.85).aspx

sind einerseits von der Windows Speech Runtime und andererseits von der zuvor erläuterten Speech API (SAPI) abgeleitet. Ein weiterer wesentlicher Bestandteil sind die Runtime Languages. Diese sind allerdings weder in der Microsoft Speech Platform Runtime 11 noch in der Microsoft Speech Platform SDK 11 enthalten und müssen daher, passend zur jeweiligen Version, gesondert heruntergeladen und den Speech Engines zur Verfügung gestellt werden. Die Runtime Languages enthalten alle wesentlichen Informationen, die von den Speech Engines für die Spracherkennung und Sprachsynthese benötigt werden. Dies beinhaltet unter anderem ein Sprachmodel und ein Akustikmodel. Die eigentliche Funktionalität für alle Sprachoperationen wird von der Application Runtime zur Verfügung gestellt. Der soeben geschilderte Aufbau ist in der nachfolgenden *Abbildung 6* überblicksmäßig dargestellt [33] [34] [35].

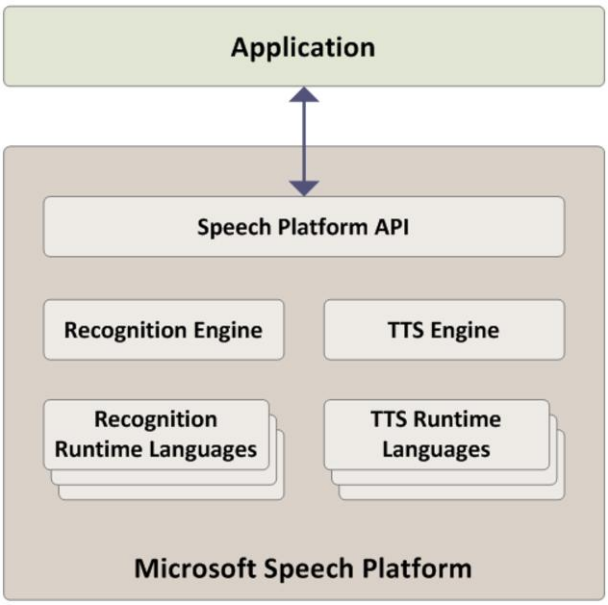


Abbildung 6: Komponenten der Microsoft Speech Platform mit der Speech Platform API als Schnittstelle zur Applikation, sowie den Engines und Runtime Languages für die Spracherkennung und Sprachsynthese (Text To Speech – TTS) [34].

Die Microsoft Speech Platform ist eigentlich nicht für den klassischen Gebrauch in Desktop-Applikationen entwickelt worden, sondern für automatische Sprachmenüs und Sprachkommandos, die dann beispielsweise für Telefonvermittlungsanwendungen auf Microsoft Windows Servern laufen. Die Speech Platform ist dementsprechend dafür ausgelegt, viele unterschiedliche Sprecherinnen und Sprecher verstehen zu können, anstatt bei einer bestimmten Anwenderin oder einem Anwender eine möglichst hohe Erkennungsrate aufzuweisen. Für den Einsatz bei Server-Anwendungen wird im Gegensatz zu SAPI auch nur eine Audio-Auflösung von 8-Bit eingesetzt. Momentan stehen 26 verschiede Sprachen für die Spracherkennung und Sprachsynthese zur Verfügung, die allerdings, wie bereits erwähnt, alle gesondert – passend zur Version der Microsoft Speech Platform Runtime und SDK – hinzugefügt werden müssen. Diese Vorgehensweise hat den Vorteil, dass eine Applikation, die die Microsoft Speech Platform einbindet, komplett „redistributable“ ist, ohne dass auf unterschiedlichen Geräten oder für unterschiedliche Anwenderinnen oder Anwender Anpassungen notwendig wären [34] [36].

Die API der Plattform kann über den Namespace *Microsoft.Speech* eingebunden werden. Durch die Ableitung der Microsoft Speech Platform von SAPI, können ein Großteil der Funktionen und Schnittstellen des Namespaces *Microsoft.Speech* exakt gleich genutzt werden wie bei *System.Speech* von SAPI. Das ist sehr nützlich, wenn diese beiden APIs ausgetauscht werden sollen, kann bei der Entwicklung aber auch zu Verwechslungen führen, da die Schnittstellen der Namespaces nicht zu 100% gleich sind.

2.2.3. Windows Speech Recognition

Windows Speech Recognition basiert ebenfalls auf der SAPI von Microsoft und dem Microsoft Speech Recognizer 8.0 (siehe *2.2.1. Microsoft Speech API (SAPI*)), der erstmals in Windows Vista zum Einsatz kam. Die Windows Speech Recognition wurde speziell als eine Speech API für die Applikationen der von Microsoft entwickelten Universal Windows Platform (UWP) konzipiert. Die Windows Runtime dieser Plattform besitzt eine andere Architektur als jene, die für die bisherigen Windows Programme genutzt wurde und wurde erstmals unter Windows 8 eingeführt. Die Plattform ermöglicht die Nutzung von UWP-Applikationen auf den unterschiedlichsten Gerätetypen, vom Smartphone bis zum gewöhnlichen Stand-PC. Für diese unterschiedlichen Einsatzbereiche waren weder SAPI, noch die Microsoft Speech Platform geeignet, weshalb eine eigenständige Speech API benötigt wurde [39].

Dieses neue Speech API kann mit dem Namespace *Windows.Media.SpeechRecognition* in UWP-Applikationen eingebunden werden und wird seit Windows 10 direkt in das Betriebssystem integriert ausgeliefert. Da Windows Speech Recognition eine Weiterentwicklung zu den bisherigen Speech APIs von Microsoft darstellt, wird auch hier ein Deep Neural Network Acoustic Model zur Spracherkennung eingesetzt. Wie bereits von der Microsoft Speech Platform etabliert, ist die Spracherkennung nicht nur mit Internetverbindung möglich, sondern auch offline. Die Spracherkennung ohne Internetverbindung funktioniert allerdings nur mit einer vordefinierten Speech Recognition Grammar Specification (SRGS), welche mittels eines Voice Command Definition (VCD) Files vorgenommen wird. So ist beispielsweise beim Windows-Sprachassistenten Cortana, welcher die Windows Speech Recognition API nutzt, offline nur ein gewisser Satz an Kommandos verfügbar. Die Online-Spracherkennung erfolgt über das Microsoft Speech Recognition Service (Cloud-Service), bei welchem es sich um eine Weiterentwicklung des Bing Speech Services (ebenfalls ein Produkt von Microsoft) handelt (siehe *Abbildung 7*) [37] [38].

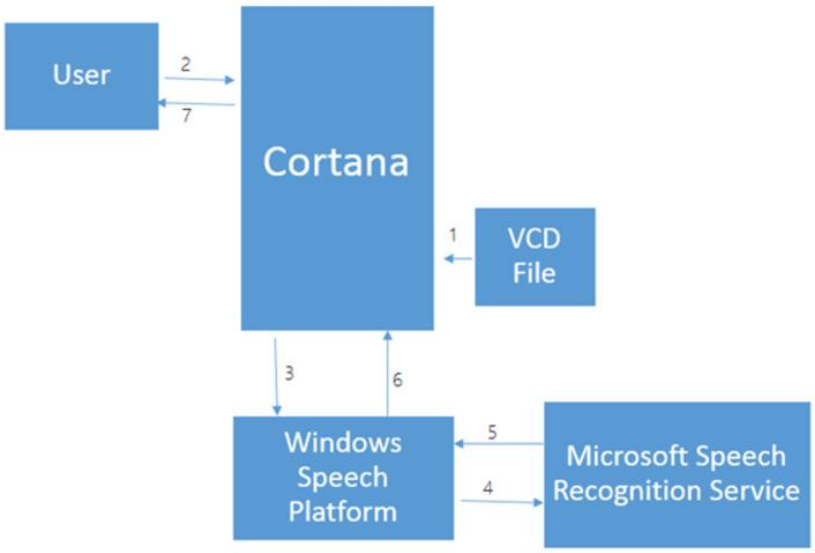


Abbildung 7: Funktionsweise der Windows Speech Recognition im Kontext von Cortana: (1) Beim Start von Cortana wird das Voice Command Definition (VCD) File geladen. (2) Der User kann Kommandos aus dem VCD File nutzen. (3)-(6) Die Spracheingabe wird an die Windows Speech Platform (lokal auf dem Gerät) und an das Microsoft Speech Recognition Service (Cloud) übermittelt. Aus den Ergebnissen der lokalen und online Spracherkennung wird ein Resultat ermittelt und an Cortana übergeben. (7) Cortana führt basierend auf dem identifizierten Resultat eine Aktion aus (Rückmeldung an den User oder Start einer Anwendung etc.) [39].

Aktuell werden von der Windows Speech Recognition API acht verschiedene Sprachen unterstützt, welche built-in mitgeliefert werden und keine zusätzliche Installation erfordern. Das Interface besitzt außerdem drei verschiedene Modi (*Listening*, *Sleeping*, *Off*), die den Status des Speech Recognizers repräsentieren. *Listening* bedeutet, dass die Spracherkennung aktiv ist und auf eine Spracheingabe wartet. *Sleeping* heißt, dass sich der Recognizer in einem Schlummerzustand (Standby) befindet und nur mit dem expliziten Start-Befehl aktiviert werden kann. Der Zustand *Off* bedeutet, dass die Spracherkennung gestoppt wurde und auf keine Sprachkommandos mehr reagiert [39].

3. Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen zu Themen und Methoden dargelegt, die für die Entwicklung und Implementierung des praktischen Teils dieser Arbeit relevant sind.

3.1. Requirements Engineering

Dieser Abschnitt bildet das Fundament für die im nachfolgenden Kapitel *4. Analyse und Zielsetzung* durchgeführten Prozesse. Aus diesem Grund werden im Folgenden die Begrifflichkeiten und Abläufe geklärt, die eingesetzt wurden und im Bereich der Softwareentwicklung dem Stand der Technik entsprechen.

Von den Autoren Dick, Hull und Jackson wird der Begriff Requirements Engineering wie folgt definiert:

"Requirements engineering: the subset of systems engineering concerned with discovering, developing, tracing, analyzing, qualifying, communicating and managing requirements that define the system at successive levels of abstraction" [40].

Trotz leicht variierender Definitionen ist Kernaussage, dass im Rahmen des Requirements Engineering alle Anforderungen, die von Seiten der Stakeholder an das System gestellt werden, erfasst und analysiert, sowie verwaltet werden.

Dieser Prozess lässt sich im Allgemeinen in die folgenden vier Bestandteile aufschlüsseln:
- Anforderungserhebung
- Anforderungsanalyse
- Anforderungsspezifikation
- Anforderungsbewertung (-validierung)

Bei der Durchführung einer **Anforderungserhebung** werden im Rahmen von Gesprächen mit den Stakeholdern alle Anforderungen in natürlicher Sprache ermittelt und ausgewertet. Das bedeutet, die Stakeholder formulieren die Anforderungen in eigenen Worten und benötigen kein Wissen über das Fachvokabular des jeweiligen Anwendungsbereichs. In der anschließenden **Anforderungsanalyse** erfolgt eine Bewertung und Klassifizierung der bisher gesammelten Informationen. Das beinhaltet beispielsweise die Überprüfung auf die Vollständigkeit der erhobenen Anforderung. Im Rahmen der **Anforderungsbeschreibung** werden, basierend auf den beiden vorherigen Schritten, die Anforderungen in einer konsistenten Form dokumentiert, zum Beispiel in einem Pflichtenheft. In der Softwareentwicklung sind aber auch Anwendungsfälle eine gängige Form Anforderungen zu beschreiben. Die **Anforderungsbewertung** ist ein Bestandteil des Requirements Engineerings, der nicht zwingend durchgeführt werden muss. In diesem Schritt können zu einem späteren Zeitpunkt die dokumentierten Anforderungen erneut kontrolliert und falls notwendig auch angepasst werden. Solche Änderungen können allerdings große Auswirkungen auf den Entwicklungsprozess und das ganze Projekt haben, wodurch vorab eine Vorgehensweise für potenziell eintretende Änderungen geplant werden sollte [40] [41].

3.1.1. Requirements (Anforderungen)

Der Begriff Requirements repräsentiert in der Softwareentwicklung Eigenschaften und Bedingungen, die von dem System (der Software) erfüllt werden müssen. Die ausgearbeiteten Anforderungen sollen dokumentieren, was die Stakeholder von dem System erwarten und verlangen, beziehungsweise was die Software leisten muss, um diese Bedürfnisse zu erfüllen [42].

*"**requirement:** A statement that identifies a product or process operational, functional, or design characteristic or constraint, which is unambiguous, testable or measurable, and necessary for product or process acceptability (by consumers or internal quality assurance guidelines)"* [43].

Bei Anforderungen werden grundsätzlich drei Kategorien unterteilt:
- Funktionale Anforderungen
- Nicht-funktionale Anforderungen
- Domänenanforderungen

Hierbei umfassen die **funktionalen Anforderungen**, jene Funktionalitäten, welche die Software liefern soll. Das beinhaltet Dienste und Services, die das System anbieten soll, welche Eingaben, Verarbeitungsschritte oder Ausgaben dazu benötigt werden, sowie das Verhalten des Systems in bestimmten Situationen. Die **nicht-funktionalen Anforderungen** beschreiben gewünschte Eigenschaften der Software, die über die funktionalen Anforderungen hinaus gehen und dementsprechend keine Vorgaben an die Dienste und Services des Systems beschreiben. Zu nicht-funktionalen Anforderungen zählen beispielsweise bestimmte Qualitätsmerkmale und Leistungsanforderungen, sowie Ansprüche an die Benutzbarkeit des Systems [44] [46].
Eine Kategorie, die im Gegensatz zu den funktionalen und nicht-funktionalen Anforderungen nicht in allen Definitionen vorhanden ist, sind die **Domänenanforderungen** (Rahmenbedingungen). Diese Form von Anforderungen beinhalten Bedingungen und Einschränkungen (das können sowohl funktionale, als auch nicht-funktionale Anforderungen sein), die von dem Environment in dem sich das System befindet definiert werden. Domänenanforderungen haben wesentliche Auswirkungen auf die Entwicklung des Softwaresystems, beziehungsweise das ganze Projekt und sind nicht oder nur schwer beeinflussbar sind [45] [47].

3.1.2. Stakeholder

Der in den vorherigen Abschnitten bereits mehrmals benutzte Begriff Stakeholder wurde von Freeman in den 1980er Jahren wie folgt definiert:

"Stakeholder: An individual, group of people, organisation or other entity that has a direct or indirect interest (or stake) in a system" [40].

Diese Definition ist allerdings sehr allgemein und bezieht sich auf das strategische Management. Für das Requirements Engineering im Bereich der Softwareentwicklung stellen Stakeholder jene Personen oder Organisationen dar, die einen Einfluss auf das System haben, mit

der Software in Verbindung stehen oder in irgendeiner Form von ihr betroffen sind. Stakeholder sind somit der wichtigste Einflussfaktor für den Prozess des Requirements Engineerings, sowie für die Anforderungen eines Systems [40].

Stakeholder werden in zwei Gruppen unterschieden:
- Primäre Stakeholder
- Sekundäre Stakeholder

Zu den **primären Stakeholdern** zählen jene Interessensgruppen, die direkten Einfluss auf die Anforderungen an das System haben. Dazu gehören beispielsweise Entwickler, Anwender, Auftraggeber, Geldgeber/Investoren, strategische Partner und Management (Geschäftsführung, Abteilungsleitung etc.). Die **sekundären Stakeholder** sind jene Personen und Organisationen, die indirekten Einfluss auf den Prozess und die Anforderungen haben, beziehungsweise nur in indirekter Weise vom System betroffen sind. Zu dieser Gruppe zählen zum Beispiel Qualitätssicherung und Testbetrieb, Gesetzgeber, Maintenance, Medien, Lieferanten, oder auch Mitbewerber [48].

3.1.3. Usability Testing

Der Begriff Usability wird von der International Organization for Standardization (ISO) wie folgt definiert (Standard ISO 9241-11):

"*The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use*" [50].

Im Rahmen des Usability Testings soll genau diese Eigenschaft eines Systems empirisch erhoben, beziehungsweise evaluiert werden. In anderen Worten heißt das, dass das System auf die, für die Anwenderinnen und den Anwender, wichtigsten Aufgaben und Bedienungsabläufe getestet wird, beziehungsweise die Benutzerinnen und Benutzer beim Umgang mit dem System und der Durchführung von Tasks beobachtet werden. Dieses Verfahren kann somit als Kennzahl für die Gebrauchstauglichkeit einer Software gesehen werden [49].

Es gibt zwei unterschiedliche Typen des Usability Testings:
- formative Tests
- summative Tests

Formative Tests werden noch während der Entwicklung des Systems in zyklischen Abständen durchgeführt. Sie zielen darauf ab, Probleme frühzeitig zu erkennen und beheben zu können. Im Gegensatz zu den formativen Tests werden **summative Tests** nach der Fertigstellung des Systems durchgeführt. Sie dienen hauptsächlich dazu zu erheben, ob die Requirments erfüllt wurden und können somit auch als Qualitätskontrolle eingesetzt werden [49].

3.2. Test Driven Development

Der Prozess des Test Driven Developments (TDD) sieht vor, dass kein Code für die zu entwickelnde Software geschrieben wird, bevor ein Test für die jeweilige Funktionalität geschrieben wurde. Dieser Entwicklungsprozess wird vor allem bei der agilen Softwareentwicklung eingesetzt und existiert bereits in einigen leicht unterschiedlichen Ausprägungen. Grundsätzlich bestehen beim TDD keine Vorgaben, ob die Tests manuell oder automatisch ausgeführt werden müssen. Inzwischen gibt es allerdings eine Vielzahl an Frameworks, die automatische Tests ermöglichen und dem Entwicklungsteam somit eine enorme Zeitersparnis ermöglichen. Bekannte Beispiele für solche Frameworks wären zum Beispiel JUnit, Selenium, Jasmine oder auch Mocha [51] [52].

Tests für diese Form der Softwareentwicklung können beispielsweise nach ihrem Umfang unterschieden werden. Für diese Arbeit waren die folgenden Typen relevant:

- Unit Testing
- Component Testing
- Integration/System Testing

Hierbei gibt es in der Fachliteratur, unter anderem auch in Abhängigkeit von der Programmiersprache, unterschiedliche Ansichten bei der Definition der einzelnen Typen, beziehungsweise ihren Grenzen (Übergängen) [52] [53].

Unit Testing

Units sollen den kleinsten möglichen Teil einer Software darstellen, der getestet werden kann (zum Beispiel Klassen oder Methoden). Diese Tests sollen in kurzen iterativen Zyklen geschrieben werden und es ermöglichen, ausschließlich und isoliert, die jeweilige Unit zu testen. Erfüllt der anschließend implementierte Code den Test, wird ein neuer Test für die nächste zu entwickelnde Unit erstellt. Ändern sich die Requirements werden zuerst ausschließlich alle betroffenen Unit-Tests, entsprechend der neuen Zielsetzung, geändert. Bei jenen Tests, die anschließend fehlschlagen, muss für den Code der Unit ein Refactoring (Überarbeitung) durchgeführt werden, bis alle Tests erneut erfolgreich durchlaufen werden [52] [53].

Component Testing

Als Komponente werden größere Teile der Software, die mehrere Units beinhaltet, angesehen. Auch bei diesen Tests wird die Komponente isoliert behandelt und es gibt keine Abhängigkeiten zu andern Teilen des Systems. Um Komponenten und ihre Schnittstellen wirklich isoliert testen zu können und die Ergebnisse nicht zu beeinflussen, ist es oft notwendig für andere Teile des Systems Platzhalter (Stubs) oder Testtreiber zu implementieren. Es soll somit die Funktionalität der Komponente überprüft werden, beispielsweise wird ein bestimmter Input übergeben und anschließend überprüft, ob die Komponente den erwarteten Output liefert [52] [53] [54].

Integration/System Testing

Integrations- oder Systemtests stellen die Ebene über den beiden zuvor beschriebenen Arten von Tests dar. Hier sollen mehrere, beziehungsweise alle Teile (Komponenten) eines Systems getestet werden. Das bedeutet, es werden keine isolierten Tests durchgeführt, sondern das Augenmerk liegt darauf, ob alle Komponenten der Software wie erwartet zusammenarbeiten und die gewünschte Performanz liefern. Zum Umfang von Systemtests gehört es auch mehrere unterschiedliche Systeme zu testen, beispielsweise das Frontend und das Backend einer Web-Anwendung [54] [55].

3.3. Design Pattern

Im folgenden Abschnitt werden Design Pattern und Besonderheiten der Programmiersprachen beschrieben, die für die Umsetzung des Treibers im Kapitel *5. Realisierung* eingesetzt werden. Die ausgewählten Konzepte sind ein wichtiger Baustein bei der Entwicklung und Implementierung, da sie zum einen den Stand der Technik repräsentieren und zum anderen wesentlich zur Qualität und dem Design, respektive der Architektur der Software, beitragen.

JavaScript

JavaScript unterscheidet sich von den klassischen objektorientierten Programmiersprachen dadurch, dass es nicht klassenbasiert ist. Bei Java, werden beispielsweise Objekte in Instanzen und Klassen unterteilt und deren Vererbung erfolgt über die Klassenhierarchie. JavaScript hingegen kennt in dieser Form keine Klassen oder Namensräume und unterscheidet nicht zwischen Typen von Objekten. JavaScript nutzt stattdessen einen Prototypen-Mechanismus für die Vererbung. Dadurch ist es möglich, dynamisch Eigenschaften und Methoden zum Objekt hinzuzufügen.
Diese Unterscheidung ist sehr wesentlich und muss für die Architektur und Implementierung einer Software beachtet werden. Für die Softwareentwicklung mit JavaScript haben sich einige unterschiedliche Design Pattern etabliert, welche unter anderem darauf abzielen, ein Klassen- und Namensräume-ähnliches Verhalten, zu erzielen. So gehört beispielsweise die Modularisierung, bekannt unter dem Namen Module Design Pattern, zum „State of the Art“ im Bereich der JavaScript-Programmierung. Hierbei soll sich die Anwendung aus möglichst eigenständigen, und voneinander unabhängigen, Modulen zusammensetzten. Diese lose Kopplung zwischen den Modulen oder Komponenten ermöglicht es, den Code gut und übersichtlich zu strukturieren. Ein solches Modul stellt eine sogenannte Immediately-Invoked-Function-Expression (IIFE) dar, wie sie zur Veranschaulichung in *Listing 1* zu sehen ist. Mit dieser Vorgehensweise besitzt das Modul einen privaten Scope und hat somit die Möglichkeit, „private“ und „public“ Variablen und Funktionen erzeugen zu können. Über das Return-Objekt der IIFE kann gesteuert werden, welche Variablen und Funktionen dieses Moduls von außen zugreifbar sein sollen [57] [58] [59].

```
//IIFE die ein Modul darstellt
(function() {
	//an dieser Stelle können private Variablen und Funktionen deklariert werden

	//return der IIFE/des Moduls
	return {
		//an dieser Stelle können public Variablen und Funktionen deklariert werden
	}
}) ();
```

Listing 1: Immediately-Invoked-Function-Expression (IIFE)

Die allgemeine Definition der Module Pattern ist allerdings sehr oberflächlich und mehrdeutig, was den Umgang mit „private“ und „public“ Variablen und Funktionen angeht, was sie für den praktischen Einsatz eher ungeeignet macht. Durch exaktere Definitionen und Verfeinerungen entstanden mehrere Formen und Variationen dieses Design Pattern, wobei Revealing Module Pattern eine sehr populäre Variation darstellt. Das Ziel der Revealing Modul Pattern ist es, eine Form von Datenkapselung zu erreichen und nur bestimmte Variablen und Funktionen des Objekts von außen zugreifbar zu machen (siehe *Listing 2*). [57] [58] [59]

```
var revealingModule = (function() {
	function firstPublicMethod() {
		alert( ‚This is a public method‘ );
	}

	function secondPublicMethod() {
		privateMethod();
	}

	function privateMethod() {
		alert( ‚This is a private methode‘);
	}

	// wird dieses Object instanziert können die im Return zurückgegebenen Funktionen
	// als public Methoden verwendet werden
	return {
		revealingMethodOne: firstPublicMethod,
		revealingMethodTwo: secondPublicMethod
	};
})();

//Sample calls:
//Output of this function call will be “This is a public method”
revealingModule.revealingMethodOne();
```

```
//Output of this function call will be "This is a private method"
revealingModule.revealingMethodTwo();

//Function will be undefined
revealingModule.privateMethod();

//Function will be undefined
revealingModule.firstPublicMethod ();
```

Listing 2: JavaScript Revealing Module Pattern

Durch die Einhaltung dieses Design Patterns wird nicht nur die Lesbarkeit und Übersichtlichkeit des Codes verbessert, sondern es herrscht auch automatisch eine durchgehende Konsistenz in den geschriebenen JavaScript-Files (Scripts).
Außerdem können durch den Einsatz von Module Patterns und/oder Revealing Module Patterns Anforderungen und Ziele in kleinere Teilmengen unterteilt werden, womit die Komplexität und der Schwierigkeitsgrad bei der Entwicklung sinken. Diese Charakteristiken sorgen für eine geringere Komplexität, sowie eine bessere Nachvollziehbarkeit des Codes und führen in Kombination mit einer leichteren Austauschbarkeit der Komponenten und Module zu einer besseren Wartbarkeit der Software [57] [58] [59].

Ein weiteres Entwicklungsmuster, welches an dieser Stelle erwähnt werden sollte, sind die Prototype Design Pattern (siehe *Listing 3*). Wie bereits erwähnt kennt JavaScript keine klassische Vererbung, ermöglicht aber dennoch ein objektorientiertes Arbeiten. Hierfür gibt es das Keyword *Prototype*, mit welchem ein Objekt erzeugt werden kann, für welches Variablen und Funktionen deklariert werden können. Wird der Konstruktor dieses Objekts aufgerufen, wird eine Kopie (ein sogenannter shallow clone) erzeugt, die eine neue Instanz des Objekts repräsentiert. Diese neue Objekt-Instanz besitzt nun dieselben Eigenschaften wie sein Original, bietet aber auch die Möglichkeit, zusätzliche Variablen und Funktionen hinzuzufügen oder originale Variablen und Funktionen zu überschreiben [57] [58] [59].

```
//Erzeugen eines "Prototype-Objects"
var workflow = function(wfId) {
        var id = wfId;
        var steps = [];
}

//Definieren von Funktionen des Prototypes
workflow.prototype.addStep = function(step) {
        this.steps.push(step);
};

workflow.prototype.addName = function(name) {
        this.name = name;
};
```

Listing 3: JavaScript Prototype Pattern

Auch Prototype Pattern gibt es in der Variante Revealing Prototype Patterns (siehe *Listing 4*), welche ebenfalls die Möglichkeit bietet, über das Closure (den Funktionsabschluss) festzulegen, welche Variablen und Funktionen „private“ und welche „public“ sind.

```
var workflow = function(wfId) {
	var id = wfId;
	var steps = [];
}

workflow.prototype = function() {

	var addStep = function(step) {
		this.steps.push(step);
	};

	var addName = function(name) {
		this.name = name;
	}

	//im Return können die public Funktionen des Objekts definiert werden
	return {
		wfAddStep: addStep,
		wfAddName: addName
	}
}
```

Listing 4: JavaScript Revealing Prototype Pattern

Weitere bekannte JavaScript Design Pattern wären beispielsweise noch Singleton, Factory, Observer oder Facade Pattern. Auf diese wird im Rahmen dieser Arbeit allerdings nicht weiter eingegangen [57] [58] [59].

4. Analyse und Zielsetzung

Dieses Kapitel beinhaltet, aufbauend auf dem im Kapitel *3.1. Requirements Engineering* beschriebenen Konzept des Requirements Engineering die Erhebung und Analyse der Anforderungen und Zielsetzungen dieser Arbeit.

4.1. Erhebung der Anforderungen und Zielsetzungen

Die Thematik dieser Arbeit behandelt die folgende Frage:
Wie sieht eine geeignete Lösung aus, um ein User-Interface, am Beispiel des Local User Interface (LUI), um einen Treiber zur Bedienung mittels einer lokalen (embedded) Spracherkennung von Microsoft, zu erweitern? Wie kann diese Erweiterung implementiert werden und inwieweit werden die identifizierten Anforderungen erfüllt?

Aus dieser Fragestellung lässt sich die Zielsetzung für diese Arbeit wie folgt ableiten:
Die Bedienung der Plattform LUI soll so optimiert werden, dass diese ausschließlich mittels Spracheingaben gesteuert werden kann. Das bedeutet, es soll eine Erweiterung (ein Treiber) entwickelt werden, die es der Anwenderin oder dem Anwender ermöglicht, nach dem Start des LUI ohne jegliche Eingabe durch Maus oder Tastatur alle zur Verfügung gestellten Funktionalitäten der Software nutzen zu können. Das beinhaltet sowohl die Navigation durch die verschiedenen Seiten, als auch die Nutzung unterschiedlicher Programme, die in das LUI integriert sind, wie zum Beispiel das Abspielen des Radios oder eines Hörbuchs oder auch das Nutzen des Internet Browsers. Auch die Verwendung von Sonderfunktionen, wie etwa das Absetzen eines Hilferufs (Notruf), soll ausschließlich durch Sprachkommandos möglich sein.

Die Steuerung des LUI durch Sprachkommandos soll dabei möglichst intuitiv sein und auch für ungeübte oder unerfahrene Anwenderinnen oder Anwender kein Problem darstellen. Ein Sprachkommando soll, wie bei den meisten gängigen Spracherkennungen, durch das Aussprechen eines (frei wählbaren) vorab definierten Schlüsselworts („Keyword“) gestartet werden. Während der kompletten Verwendung des LUI ist ausschließlich das Userinterface des LUI zu sehen, sodass die Erweiterung zur Sprachsteuerung keine optischen Auswirkungen auf die ursprüngliche Benutzeroberfläche des LUI hat. Ein User-Interface des Treibers könnte ansonsten eventuell für Irritationen oder Verwirrung sorgen und es soll sich auch für Anwenderinnen und Anwender, die das LUI bereits kennen, durch den Treiber nichts ändern. Da das LUI für das Betriebssystem Windows entwickelt wurde, zählt es auch zur Zielsetzung, im Rahmen dieser Arbeit, bei der Spracherkennung möglichst auf eine Technologie von Microsoft zu setzen. Da eine frei verfügbare Internetverbindung mit guten Übertragungsraten für die Bewohnerinnen und Bewohner von manchen Pensionistenwohnhäusern nicht vorausgesetzt werden kann, soll der Fokus auf eine lokale (embedded) Spracherkennung gelegt werden.

Diese Zielsetzung bedeutet für die Implementierung, die im Rahmen dieser Arbeit durchgeführt werden soll, die Entwicklung eines Treibers zur Bedienung des LUI ausschließlich mittels Sprachkommandos. Diese Treiber-Software muss über eine Spracherkennung sowie eine Logik zur Verarbeitung von Spracheingaben verfügen, welche eine möglichst natürliche und alltagstaugliche Bedienung des LUI erlauben. Das Hauptaugenmerk soll dabei auf dem

Teil des Treibers liegen, der die Kommunikation (mit dem LUI oder anderen Softwarekomponenten), sowie die Logik zur Verarbeitung aller in diesem Kontext auftretenden Nachrichten beinhaltet, was auch eine leichte Austauschbarkeit der Spracherkennung ermöglichen soll. Der Treiber soll außerdem möglichst Sprecherinnen- und Sprecher-unabhängig sein und für neue Anwenderinnen und Anwender keinen unzumutbar hohen Zeitaufwand für die Konfiguration oder ein umfangreiches Anlernen der Spracherkennung bedeuten.

Eine weiterführende Zielsetzung der Arbeit geht über die rein technische Erweiterung der bestehenden LUI-Software hinaus und bezieht sich auf die barrierefreie Nutzung des Programms für alle Menschen. Der entwickelte Treiber zur Spracherkennung soll nämlich nicht nur als Komfortfeature dienen, sondern richtet sich insbesondere an Menschen, denen es aus unterschiedlichen körperlichen Beeinträchtigungen nicht oder nicht mehr möglich ist, einen Computer mittels Touchscreen, Maus oder Tastatur bedienen zu können. Dieser Zielgruppe soll durch die Erweiterung des LUI dazu verholfen werden, das Programm ohne Einschränkungen und in seinem vollen Funktionalitätsumfang bedienen zu können.

4.2. Anforderungsanalyse

Um die im vorherigen Abschnitt erhobenen allgemeinen Anforderungen und Zielsetzungen bezüglich der qualitativen Maßstäbe der Softwareentwicklung umsetzen zu können, wurden im Rahmen einer Analyse alle wesentlichen Punkte identifiziert und konkretisiert. Die Anforderungen für die Entwicklung des Treibers wurden dabei in funktionale und nicht-funktionale Anforderungen aufgeschlüsselt. Im folgenden Abschnitt werden alle Anforderungen eingehend dargestellt.

Funktionale Anforderungen:

- Beim Start des LUI soll der Treiber inklusive aller zur Spracherkennung benötigten zusätzlichen Softwarekomponenten automatisch gestartet werden. Diese Startup-Prozedur soll ohne eine zusätzliche Eingabe oder Aktion durch die Anwenderin oder den Anwender geschehen. Das gleiche gilt auch für das Beenden des LUI – es soll jeder Teil der Treibersoftware automatisch beendet werden.

- Der Treiber soll während der kompletten Dauer der Nutzung im Hintergrund laufen und für die Anwenderin oder den Anwender nicht bemerkbar ins LUI integriert sein.

- Alle wichtigen Einstellungen und Konfigurationen für den Treiber müssen über die Konfigurationsdatei des LUI durchgeführt werden können. Das bedeutet, dass es eine Möglichkeit geben muss, bestehende Parameter der Settings verändern beziehungsweise neue hinzufügen zu können. Zusätzliche spezifische Konfigurationsmöglichkeiten an anderen Stellen, die ausschließlich für den in dieser Arbeit entwickelten Treiber gedacht sind, sind unerwünscht.

- Eine von der Anwenderin oder dem Anwender getätigte Spracheingabe muss evaluiert und auf seine Gültigkeit geprüft werden. Entspricht die gesprochene Eingabe einem verfügbaren Kommando des LUI, müssen die notwendigen Daten an das LUI

übermittelt werden, damit die gewünschte Aktion vom LUI durchgeführt werden kann.

- Für die Sprachkennung soll möglichst eine Technologie, beziehungsweise SDK, von Microsoft eingesetzt werden. Des Weiteren ist es erwünscht, dass die Spracherkennung lokal durchgeführt wird und auch ohne Internetverbindung funktionsfähig ist.

- In einem Fall, in dem der Treiber für die Spracherkennung vorübergehend oder dauerhaft nicht funktionsbereit ist, soll dies für die Anwenderin oder den Anwender ersichtlich gemacht werden.

- Der Treiber soll unterschiedliche Anwendungen zur Spracherkennung unterstützen. Um das zu gewährleisten, muss ein Interface zur Verfügung gestellt werden, welches garantiert, dass diesen Anwendungen alle notwendigen Informationen übermittelt werden.

Nicht-funktionale Anforderungen:

- Der Kern (Hauptteil) des Treibers soll möglichst generisch sein. Das bedeutet, der Treiber soll so implementiert werden, dass er kompatibel mit unterschiedlichster Software zur Spracherkennung ist und sein Programmablauf unabhängig von deren Umsetzung ist.

- Für die Anbindung von unterschiedlichen Applikationen zur Spracherkennung muss ein gut definiertes und dokumentiertes Interface zur Verfügung gestellt werden. Die Qualität dieser Dokumentation sollte es anderen Entwicklerinnen und Entwicklern problemlos ermöglichen, eigene Anwendungen in das Treiber-Modul zu integrieren.

- Um Erweiterungen zu implementieren, Wartungsarbeiten durchführen oder Anpassungen an geänderte oder neue Anforderungen vornehmen zu können, soll der Treiber strukturiert implementiert und der Quellcode ausführlich dokumentiert werden.

- Der Treiber soll möglichst robust sein und einen Dauerbetrieb ermöglichen. Der Treiber soll dementsprechend auftretende Fehler handhaben können und beispielsweise nicht in einen inkonsistenten Zustand oder Deadlock gelangen. Das gilt sowohl für die Kommunikation mit dem LUI und der Anwendungen zur Spracherkennung als auch für interne Abläufe der Treiberkomponenten.

- Die im Rahmen dieser Arbeit eingesetzte Applikation zur Spracherkennung soll auch bei unterschiedlichen Sprecherinnen und Sprechern eine möglichst stabile Performance aufweisen. Des Weiteren soll durch Erkennungs-/Antwortzeiten und Übertragungszeiten die Usability des LUI im Vergleich zur Bedienung mittels herkömmlicher Eingabemethoden (zum Beispiel Maus oder Touchscreen) nicht wesentlich beeinträchtigt werden. Dementsprechend soll eine praxisnahe Bedienung des LUI mittels Spracherkennung ermöglicht werden.

- Es soll die Qualität der in dieser Arbeit untersuchen Lösung zur Spracherkennung evaluiert werden. Auch die korrekte Funktionsweise des Treibers soll eingehend getestet und dokumentiert werden. Das gilt ebenso für die erarbeitete Gesamtlösung -

bestehend aus dem LUI und den Software-Komponenten des Treibers (das heißt der Anwendung zur Spracherkennung und der Treiberkomponente für Verarbeitung und Kommunikation).
    - Es sollen Test-Cases implementiert werden, um die Qualität des Treibers überprüfen und belegen zu können.

    - Für die Qualität der Spracherkennung soll eine zweite Applikation als Referenz evaluiert und ein Vergleich der Ergebnisse durchgeführt werden.

- Für die Erreichung der gesetzten Ziele soll keine kostenpflichtige Software eingesetzt werden.

4.3. Anwendungsfälle

Um zu Beginn des Entwicklungsprozesses einer Software eine Übersicht über die teils detaillierten Anforderungen zu behalten, werden in der Praxis Anwendungsfälle erstellt. Die Unified Modeling Language (UML) bietet die Möglichkeit diese Anwendungsfälle anschaulich zu visualisieren. Dabei werden keine Details der technischen Implementierung oder der Programmiersprachen berücksichtigt, sondern das UML-Diagramm dient ausschließlich zur Darstellung der wichtigen Funktionen der Software, sowie der Beziehung in welcher diese zueinander stehen.

In dem in *Abbildung 8* dargestellten Anwendungsfalldiagramm (Use Case Diagram) sind, vereinfacht gesagt, zwei unterschiedliche Haupt-Anwendungsfälle zu sehen, die sich jeweils aus weiteren, granularen Anwendungsfällen zusammensetzten.
Im ersten Anwendungsfall wird die Start-Up-Prozedur dargestellt, bei der die Applikation durch die Anwenderin oder den Anwender gestartet wird. Genau genommen wird zuerst das LUI gestartet, welches im Rahmen des Ladevorgangs wiederum den Start des Treibers triggert. Das Laden des Treibermoduls inkludiert den Start der Kommunikationskomponente und der Applikation zur Spracherkennung. Ein weiterer untergeordneter Anwendungsfall, der zur Start-Up-Prozedur gehört, ist die Konfiguration des Moduls zur Spracherkennung. Diese erfolgt anhand der von der Anwenderin oder dem Anwender gesetzten Parameter in der Konfigurationsdatei des LUI. Die entsprechenden Konfigurationen erhält das Treiber-Modul direkt nach seinem Start. Der Anwendungsfall „Treiber starten“ inkludiert außerdem die Funktion aktuelle Kommandos (Menu-Entries) des LUI zu verarbeiten und zu speichern, um Vergleichsoperationen für die Spracherkennung durchführen zu können.
Im zweiten großen Anwendungsfall des Diagramms wird der Prozess einer Spracheingabe durch die Anwenderin oder den Anwender abgebildet. Das Erkennen der Spracheingabe inkludiert auch die Verarbeitung dieser Eingabe. Wird vom Spracherkennungsmodul eine Spracheingabe erkannt, erfolgt eine dementsprechende Verarbeitung dieses Inputs. Wird anschließend anhand der Liste der aktuell zur Auswahl stehenden Kommandos eine gültige Eingabe erkannt, wird dieses an das LUI gesandt.

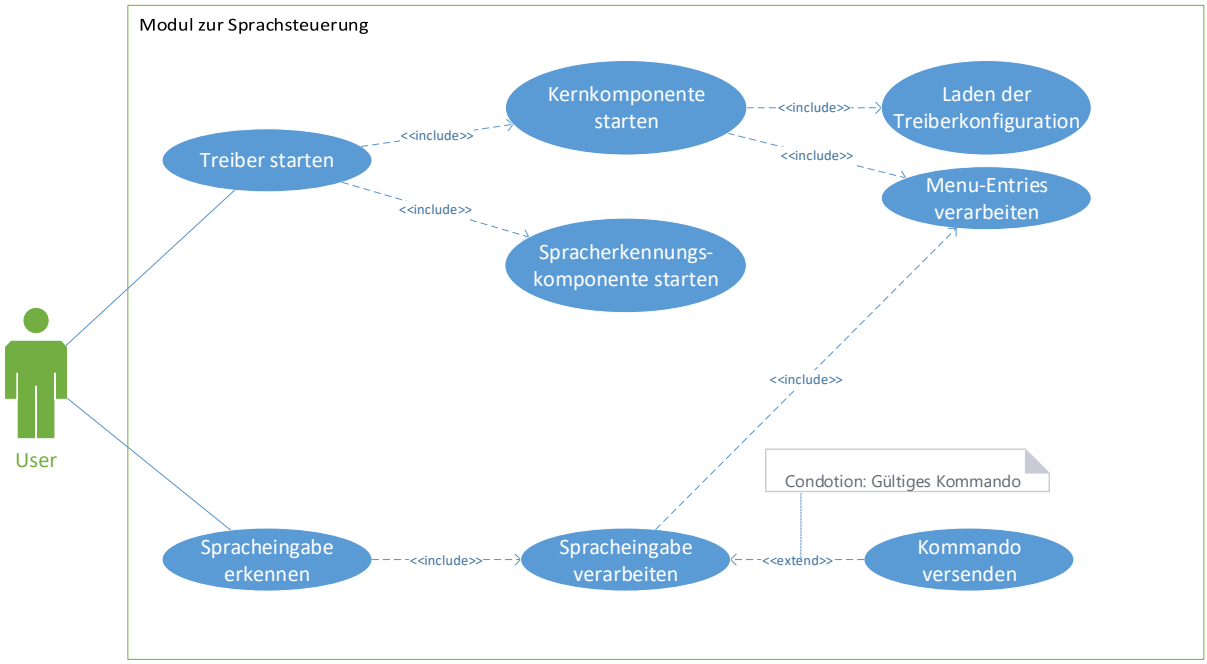


Abbildung 8: Anwendungsfalldiagramm für den Treiber zur Spracherkennung

5. Realisierung

Im folgenden Teil dieser Arbeit wird unter Berücksichtigung der bestehenden Anforderungen und Limitierungen (siehe Kapitel *4. Analyse und Zielsetzung*), im Detail beschrieben, wie die Software entwickelt und realisiert wurde. Dieser Abschnitt beinhaltet neben der Architektur, den Prozessen und Phasen der Entwicklung, sowie Testung der Software (beziehungsweise ihrer Komponenten), auch die Definition der Interfaces. Somit dient dieses Kapitel nicht nur als Dokumentation der Interfaces, sondern auch als Leitfaden für Entwicklerinnen und Entwickler einer Software, die mit diesem Treiber kommunizieren will.
Vorab ist für das Verständnis der Leserinnen und Leser noch einmal hervorzuheben, dass es sich hierbei um eine Software handelt, bei der es nicht vorgesehen ist, dass sie selbst eine GUI besitzt. Obwohl sowohl die Kernkomponente des Treiber-Moduls, sowie die implementierte Applikationen zur Spracherkennung auch Stand-Alone laufen könnten, war es weder Ziel dieser Arbeit, noch sinngemäß für diese Applikationen, GUIs zur Verfügung zu stellen, die dann einen Teil der Benutzeroberfläche des LUI überdecken würden. Alle Softwarekomponenten für die Sprachsteuerung des LUI laufen dementsprechend wie ein Treiber für das LUI und für die Anwenderin oder den Anwender ist nur das Userinterface des LUI sichtbar. Die Steuerung des Treibers erfolgt komplett über das LUI, beziehungsweise die gesetzten Einstellungen in der Konfigurationsdatei (des LUI). Für die Benutzerin oder den Benutzer ist somit zu keinem Zeitpunkt ersichtlich, dass im Hintergrund eine zusätzliche Software läuft, welche für die Verarbeitung der Sprachkommandos zuständig ist.

Dieses Kapitel gliedert sich in das Konzept und die Funktionsweisen der Software, die eingesetzten Technologien, die Entwicklung des Prototypens, sowie das Testing.

5.1. Konzept und Funktionsweisen der Software

Die Aufgabenstellung für den zu entwickelten Treiber legte von Beginn an das Hauptaugenmerk darauf, eine möglichst generische (universelle) Lösung zu wählen. Der Treiber besitzt aus diesem Grund eine Komponente, die in ihrer Funktion als Kern, beziehungsweise Basis, der Software dient und eine weitere Komponente, die für die Spracherkennung zuständig ist. Diese beiden Komponenten sind über eine Schnittstelle lose miteinander gekoppelt, was eine Austauschbarkeit der Spracherkennungstechnologie ermöglicht. Damit dieses Konzept funktioniert, wurden im Rahmen des Entwicklungsprozesses des Treibers klare Schnittstellendefinitionen (Interface Definitions) erstellt.
Diese Herangehensweise begründet sich damit, dass sich die unterschiedlichen Frameworks und Programme zur Spracherkennung nach wie vor in einer stetigen Entwicklung befinden und zumeist noch nicht die gewünschten Resultate erzielen.

Der bereits angesprochene Kern der Treibersoftware wird, in weiterer Folge, als Kernkomponente[6] bezeichnet und ist sowohl für die Kommunikation zwischen allen Komponenten zuständig, als auch für die Verarbeitung und Weiterleitung aller Nachrichten. Zu der Kommunikation zwischen dem LUI und dem Treiber zählt neben dem Empfangen von Nachrichten

[6] Im weiteren Verlauf auch Kommunikationskomponente genannt

auch die Funktion Nachrichten zu senden. Die vom LUI empfangenen Nachrichten werden vom Treiber, vor allem auf bestimmte Ereignisse, wie beispielsweise Kommandos (LUI und Treiber intern als Menu-Entries deklariert) einer aufgerufenen Seite überprüft und gegebenenfalls weiterverarbeitet. Die vom Treiber an das LUI retournierten Nachrichten beinhalten eine erkannte Spracheingabe, aber auch Fehlerfälle oder besondere Ereignisse, über welche die Benutzerin oder der Benutzer informiert werden soll. Ein solcher Fehlerfall wäre beispielsweise, dass vorübergehend die Verbindung zu der Spracherkennungssoftware abgebrochen ist und momentan keine Spracherkennung zur Verfügung steht. Die Kommunikationskomponente besitzt des Weiteren die Funktion Request-Nachrichten an das LUI zu senden. Diese Nachrichten können einen Request für die aktuellen Menu-Entries (momentan verfügbaren Kommandos), oder die im LUI geladenen Treiber-Konfigurationen (Settings) enthalten.
Über die Schnittstelle zwischen der Kernkomponente und der jeweiligen für die Spracherkennung verantwortlichen Software findet ebenso eine bidirektionale Kommunikation statt, die der Kommunikation zwischen dem LUI und der Kernkomponente ähnlich ist. Hierbei werden in die eine Richtung (Kernkomponente zu Spracherkennungsapplikation) die vorverarbeiteten, zur Auswahl stehenden Kommandos an die Spracherkennung gesandt. In die entgegengesetzte Richtung werden erkannte Kommandos zurück an die Kernkomponente geschickt.

Die Spracherkennungskomponente des Treibers (siehe *Abbildung* 9) ist für die eigentliche Spracherkennung zuständig. Dieser Softwareteil kann unter Berücksichtigung der Interfaces jederzeit ausgetauscht werden und die im Rahmen dieser Arbeit implementierte Applikation zur Spracherkennung ersetzen.

Abbildung 9 zeigt das allgemeine Konzept des Treibers für die Spracherkennung, beziehungsweise eine Übersicht der Softwarekomponenten des gesamten Systems (inklusive dem LUI), sowie die Interfaces, über welche die Komponenten miteinander kommunizieren.
In dieser Übersicht sind nicht nur die einzelnen Teile des gesamten Systems abgebildet, sondern auch, in welcher Beziehung sie zueinander stehen und wie die Kommunikationskanäle verlaufen. Bei den, dem Treiber zugehörigen Komponenten, ist die Kommunikationskomponente ein Fixbestandteil, der die gesamte Nachrichtenflüsse steuert und verarbeitet. Die im Rahmen dieser Arbeit implementierte Applikation zur Spracherkennung hingegen ist, wie bereits angesprochen, austauschbar. Es ist zwar verpflichtend, dass eine Software für diese Aufgabe in das Modul integriert wird, damit der Treiber funktionieren kann, beziehungsweise eine Spracherkennung möglich ist, es gibt aber grundsätzlich keine Vorgaben oder Einschränkungen, um welches Programm es sich dabei handeln muss.
Die Entscheidungen, für die finale Architektur dieses Treibers, sowie die Details der Implementierung und dem Vorgehen werden in den folgenden Abschnitten dieses Kapitels genauer erläutert.

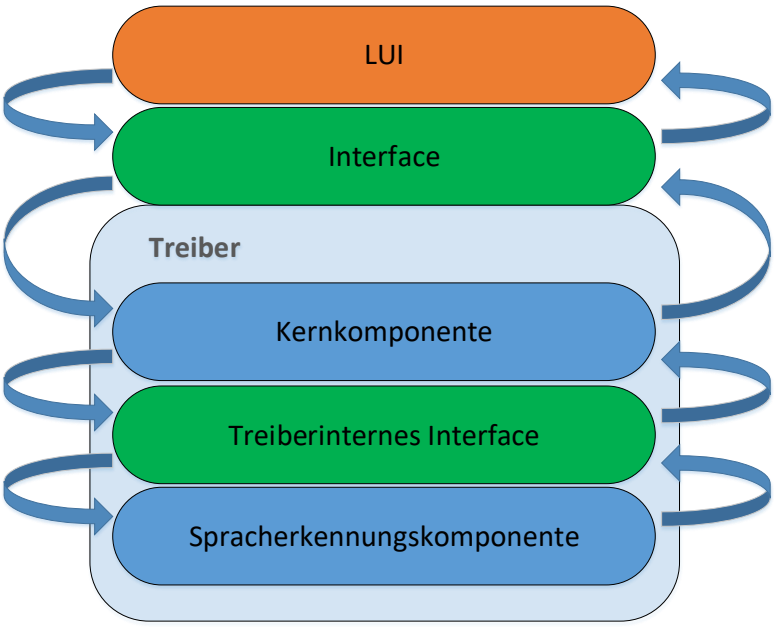


Abbildung 9: Übersicht des ganzen Systems inklusive Komponenten und Schnittstellen. Auf der linken Seite ist der Daten- und Informationsfluss vom LUI zum Treiber (bis hin zur Spracherkennungskomponente) zu sehen. Auf der rechten Seite ist der Daten- und Informationsfluss in die entgegengesetzte Richtung abgebildet.

5.2. Eingesetzte Technologien

In diesem Abschnitt wird auf Softwaretechnologien und -plattformen eingegangen, die für die Entwicklung des Treibers zur Steuerung des LUI mittels Spracherkennung wesentlich waren beziehungsweise für die Implementierung eingesetzt wurden. Es werden die Eigenschaften bereits bestehender Technologien erklärt, die sie für die Verwendung des Spracherkennungstreibers geeignet machen, wie auch in der Implementierung ersichtlich werden wird. Außerdem soll damit eine klare Trennung zwischen der eigens entwickelten Software und jener von Drittanbietern erfolgen. Die im Rahmen dieser Arbeit implementierte Software wird im Kapitel *5.3. Entwicklung eines Prototypen* beschrieben.

5.2.1. Softwareplattformen und Frameworks

Für die Implementierung der Komponenten des Treibers wurden unterschiedliche Softwareplattformen eingesetzt. Dies war notwendig, da sich die Aufgaben, beziehungsweise Funktionen, der Komponenten wesentlich voneinander unterscheiden. Die Kernkomponente basiert auf der Node.js Plattform, die Spracherkennungskomponente baut auf der .NET Plattform auf.

Node.js

Bei Node.js handelt es sich um ein JavaScript Runtime Environment, welches die von Google entwickelte JavaScript Engine V8 verwendet.

"Node.js is a platform built on Chrome's JavaScript runtime for easily building fast and scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices" [60].

Node.js ist sowohl für die Entwicklung von serverseitigen Anwendungen, als auch von Netzwerk-Applikationen geeignet. Das Framework kann somit für serverseitiges Scripting eingesetzt werden und dort beispielsweise Inhalte von dynamischen Webseiten erstellen, bevor diese an den Client, also den Web Browser des Users, gesendet werden. Node.js ist ein Cross-Plattform Runtime Environment, das bedeutet, dass Applikationen von Node.js sowohl auf Microsoft Windows, Linux als auch OS X ausgeführt werden können. Die I/Os (Inputs/Outputs) aller Node.js APIs sind asynchron, das heißt, dass beispielsweise ein mit Node.js ausgeführter Server nicht direkt auf die Antwort (Return Data) eines Requests wartet. Der Node.js Library liegt ein Single-Thread Modell mit Event-Loops zu Grunde. Treffen Return-Daten ein, wird bei der nächsten Event-Loop ein Notification Event der Node.js API, von welcher der Request gekommen ist, ausgelöst. Auf diese Art und Weise kann der Server die Return Daten dem jeweiligen Request zuordnen (Event Driven Approach).
Aus dem asynchronen Verhalten der Software ergibt sich ein sehr vorteilhaftes Feature, sie ist dadurch nämlich „Non-Blocking". Das bedeutet, dass der Programmablauf nicht auf Grund von fehlenden Return Daten Gefahr läuft zum „Stillstand" (Deadlock) zu kommen. Der asynchrone I/O, zusammen mit dem Event Driven Approach, sorgen außerdem für eine hohe Skalierbarkeit von Node.js Applikationen [61][63].
Node.js ist eine Open Source Library und somit nicht nur frei erhältlich, sondern der Source Code ist auch frei einsehbar. Das ermöglich es jedem/jeder beliebigen Softwareentwickler/in eigene Module für das Node.js Framework zu schreiben, was bereits dazu geführt hat, dass es eine Vielzahl an Modulen gibt. Diese Module können beispielsweise als Packages über den NPM (Node Package Manager[7]) heruntergeladen und installiert werden, wodurch sie automatisch zu den Node.js Basismodulen hinzugefügt werden. Durch dieses Feature können die Funktionalitäten des Frameworks leicht erweitert werden, was die Entwicklung von Node.js Anwendungen extrem vereinfacht. Bei der Node Package Manager Registry handelt es sich um ein öffentliches Repository von Open-Source Codes in Form von Modulen oder Packages. Ein solches Package ist ein Directory, welches mehrere JavaScript-Files enthalten kann und eine Datei mit Metadaten enthält, die es ermöglicht Updates und Bugfixes nachzuvollziehen. Findet eine Entwicklerin oder ein Entwickler bei der Benutzung eines Packages einen Bug, kann, durch das Konzept des NPMs, von dieser Person sofort ein Bugfix implementiert und als Versionsupdate in den NPM hochgeladen werden. Durch diese Vorgehensweise steht auch anderen Developern die neue Version wesentlich schneller zur Verfügung, da nicht zuerst ein Fehler-Report erstellt werden muss und anschließend die Fehlerbehebung durch die ursprüngliche Entwicklerin oder den Entwickler durchgeführt werden muss. Eine Alternative zum Node Package Manager ist außerdem das Paketverwaltungstool Bower [61] [62] [63].

[7] https://www.npmjs.com

Zusammenfassung und Gliederung der Features des Node.js-Frameworks:

- Asynchron und Event Driven
- Sehr performant
- Single Threaded, aber trotzdem hochgradig skalierbar
- Kein Buffering
- Erweiterbar (durch Module)
- MIT Lizenz und Open Source

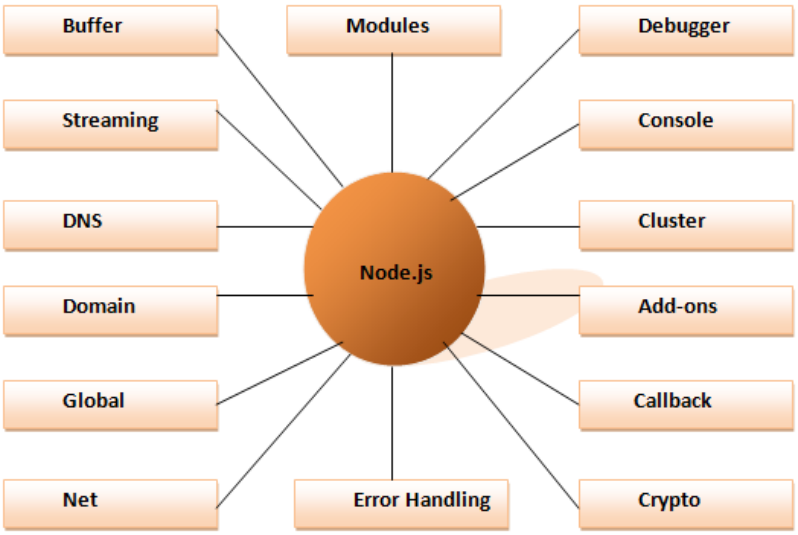


Abbildung 10: Die wichtigsten Komponenten des Node.js Frameworks [63]

.NET Framework

Das .NET Framework ist ein sogenanntes Runtime Execution Environment, welches einen Teil der von Microsoft entwickelten Software-Plattform .NET darstellt. Das .NET Framework ist dafür konzipiert, die Entwicklung von Anwendungen (Programmen), sowie deren Ausführung zu ermöglichen. Die Hauptbestandteile des .NET Frameworks sind eine Klassenbibliothek und die Common Language Runtime (CLR). Des Weiteren werden diverse Services und APIs (Application Programming Interface) zur Verfügung gestellt. Die Klassenbibliothek stellt eine große Auswahl an getestetem und wiederverwendbarem Code dar, der von den Entwicklerinnen und Entwicklern in die eigene Software eingebunden, beziehungsweise in ihr aufgerufen, werden kann. Die CLR ist eine sogenannte Execution Engine, sie führt die Anwendungen aus oder genauer gesagt managed sie während der Laufzeit [64] [65] [66].

Das .NET Framework unterstützt außerdem eine Vielzahl an verschiedenen Programmiersprachen. Dies wird vor allem dadurch ermöglicht, dass diese „höheren“ Programmiersprachen zum Zeitpunkt des Kompilierens zuerst in eine Common Intermediate Language übersetzt werden und erst zur tatsächlichen Laufzeit erfolgt vom .NET Runtime Environment die Übersetzung in die Maschinensprache der jeweiligen Plattform [64] [65].

Ein weiterer erwähnenswerter Teil des .NET Frameworks ist ASP .NET. Bei ASP .NET handelt es sich um ein Web Application Framework, was bedeutet, dass es für die Entwicklung von Webanwendungen, Webservices oder Webseiten gedacht ist [69].

Das .NET Framework bietet aber auch die Möglichkeit, durch das Einbinden von NuGet Packages die Bibliothek des Frameworks zu erweitern. Über den Paketmanager NuGet (der beispielsweise in Visual Studio integriert ist) können Erweiterungen und Updates für die Bibliothek heruntergeladen und installiert werden, ohne dass das .NET Framework selbst neu installiert oder upgedatet werden muss [68]. Solche Packages können von Microsoft selbst entwickelt worden sein, so werden zum Beispiel OOB Pakete (Out-of-Band-Releases) veröffentlicht, um neue Funktionen so schnell wie möglich zur Verfügung stellen zu können. Aber auch Pakete von Dritt-Anbietern – wobei darauf geachtet werden sollte, ob die Quelle (Urheber) vertrauenswürdig ist, da praktisch jeder ein Paket erstellen und in NuGet hochladen kann – stehen zur Verfügung. Die Einbindung von NuGet Packages ermöglicht eine schnelle und einfache Erweiterung der Funktionalitäten des .NET Frameworks und somit auch der Software, die mit ihm entwickelt wird [64] [66].

Zusammenfassung und Gliederung der wichtigsten Features des .NET Frameworks:

- Konsistente und objektorientierte Programmierumgebung
- Speicherverwaltung
- Threadverwaltung
- Codeausführungsumgebung wird bereitgestellt
- Umfangreiche Klassenbibliothek
- Versionskompatibilität und parallele Ausführung mehrerer CLR Versionen
- Sprachinteroperabilität

Im Folgenden ist zur Veranschaulichung eine übersichtsmäßige Darstellung der wichtigsten Features des .NET Frameworks aus der Microsoft Dokumentation zu sehen:

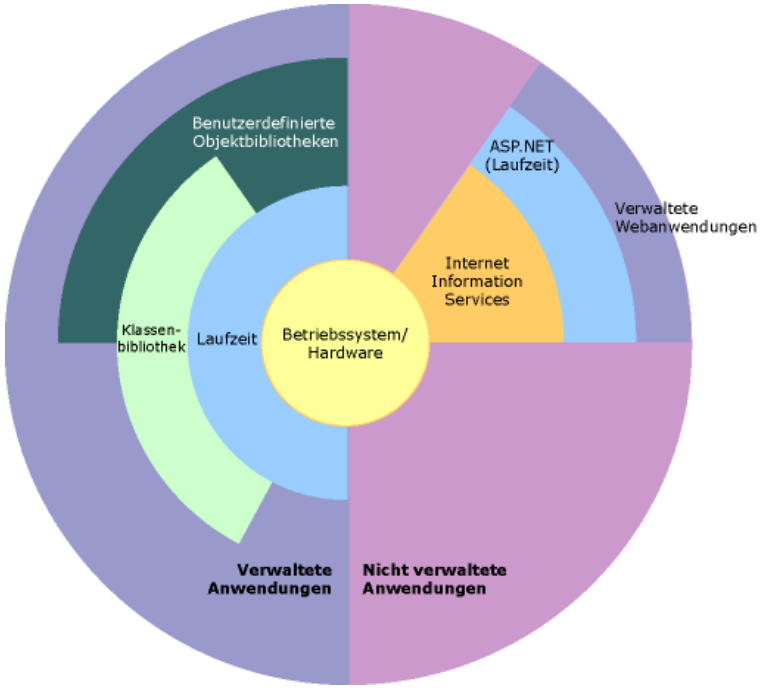


Abbildung 11: Die wichtigsten Features des .NET Frameworks, dargestellt mit dem Betriebssystem als Mittelpunkt und den Komponenten in den darüber liegenden Ebenen [67].

Im Rahmen dieser Arbeit wurde für die Entwicklung der Anwendung zur Spracherkennung die Version v4.5.2 des .NET Frameworks eingesetzt [64] [65].

5.2.2. Kommunikationsprotokolle und -technologien

Dieser Abschnitt geht auf die technischen Details für die Kommunikationsprotokolle und -technologien ein, die für die Schnittstellen zwischen dem Treiber und dem LUI, sowie zwischen der Kernkomponente und der Applikation zur Spracherkennung eingesetzt wurden.

Webserver

Im Allgemeinen hat ein Webserver (oder auch HTTP-Server genannt) die Aufgabe, bestimmte Inhalte und Dienste einem oder mehreren Clients über das Internet, oder das Intranet zur Verfügung zu stellen. Die Kommunikation zwischen dem Server und dem Client erfolgt dabei über das Hypertext Transfer Protocoll (HTTP, oder HTTPS). In der Regel werden Webserver dazu genutzt, Inhalte von Websites auf Anfrage des Browsers (Client-, oder HTTP -Request) zur Verfügung zu stellen (siehe *Abbildung 12*). Diese Inhalte können von statischen HTML-, CSS- und Bild-Dateien bis zu dynamisch erzeugten Inhalten gehen. Auf jeden HTTP-Request eines Clients antwortet der Webserver mit einem HTTP-Response, welcher auf jeden Fall einen HTTP-Status-Code enthält, aber auch Daten für den Client beinhalten kann. Wie bereits erwähnt, werden zur Übertragung zumeist das HTTP-Protokoll, sowie die Netzwerkprotokolle IP (Internetprotokoll) und TCP (Transmission Control Protocol) verwendet. Ein am weitesten verbreiteter Open Source Webserver ist der Apache Tomcat[8] [70] [71].

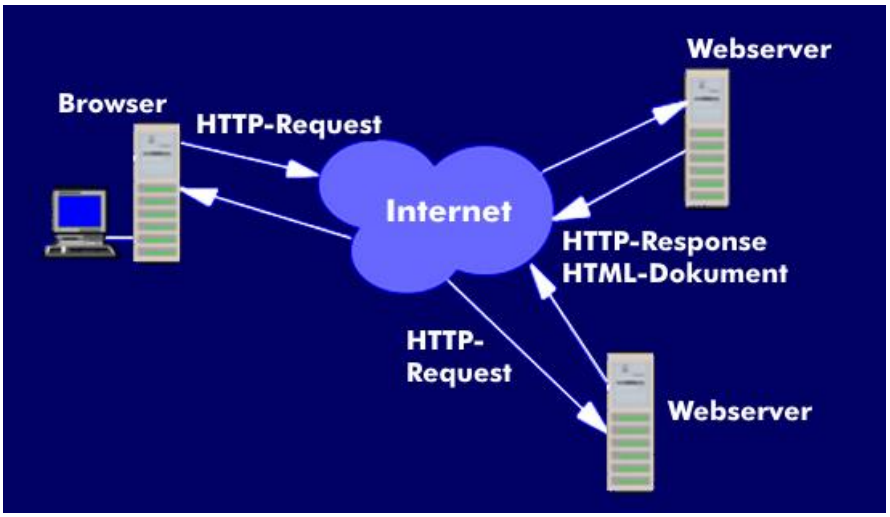


Abbildung 12: Übersicht über die Kommunikation zwischen Webservern und Client im Kontext Internet [71].

[8] https://tomcat.apache.org/

Für Test- und Entwicklungszwecke werden Webserver aber auch oft lokal eingesetzt. Für diesen Anwendungsfall ist beispielsweise das Tool XAMPP[9] sehr beliebt und weit verbreitet.

WebSocket

WebSockets basieren, wie auch HTTP-Verbindungen, auf dem TCP-Netzwerkprotokoll und ermöglichen ebenfalls das Versenden und Empfangen von HTTP-Requests und Responses. Bei der mittels des WebSockets ermöglichten bidirektionalen Kommunikation bleibt die TCP-Verbindung allerdings geöffnet (Upgrade), wohingegen bei normalen HTTP-Request nach dem Response die Verbindung wieder geschlossen wird. Bei einem WebSocket-Protokoll wird zu Beginn der Kommunikation mittels des sogenannten Handshake-Verfahrens eine Verbindung zwischen Server und Client aufgebaut. Bei einem WebSocket-Handshake wartet der Server auf eine Client-Request über einen Standard TCP Socket. Hierfür wird vom Client ein Standard-HTTP-Request mit einem WebSocket-Key an den Server gesandt. Erhält der Server eine gültige Verbindungsanfrage, wird mit dem Client-Key ein Server-Key erzeugt und mit einer Handshake-Response an den Client gesendet. Mit dem Versenden des Handshake-Response wurde der Handshake erfolgreich durchgeführt und es ist ein HTTP Upgrade erfolgt. Nun ist die Verbindung zwischen Server und Client offen und es ist eine Two-Way Communication (Full Duplex) möglich. Das heißt, es können von beiden Seiten Daten sowohl versendet, als auch empfangen werden (siehe *Abbildung 13)* [70] [72].

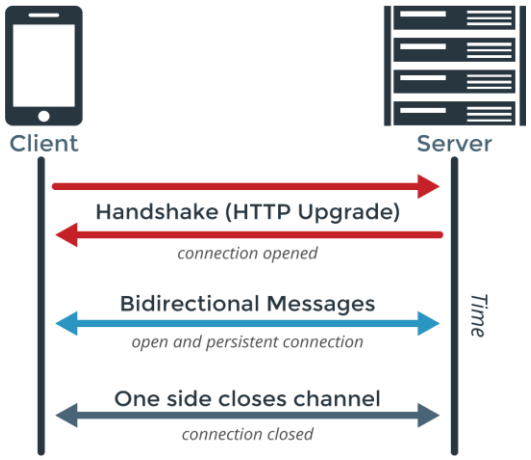


Abbildung 13: Verbindungsaufbau und Kommunikation zwischen einem WebSocket-Server und einem WebSocket-Client [72].

Ein WebSocket-Server kann zum gleichen Zeitpunkt mit einer Vielzahl an Clients über das WebSocket-Protokoll verbunden sein. Das Schließen des Kommunikationskanals (der Verbindung) kann sowohl durch den Server als auch durch den Client erfolgen.
Durch das WebSocket-Protokoll ergibt sich gegenüber einer normalen HTTP-Verbindung der wesentliche Vorteil, dass der Server immer Nachrichten an den Client senden kann, auch ohne einen vorangegangenen Request [70] [72].

[9] https://www.apachefriends.org/de/index.html

5.3. Entwicklung eines Prototypen

Als einer der wichtigsten Aspekte, der für die Entwicklung eines Treibers für die Sprachsteuerung des LUI identifiziert wurde, war, dass es sich nicht um eine maßgeschneiderte Lösung für das aktuelle LUI-Environment handeln soll. Das bedeutet, es soll eine Softwarearchitektur gewählt werden, die möglichst unspezifisch ist und somit auch mit zukünftigen Versionen des LUI kompatibel ist, sowie in keinem Konflikt mit anderen Treibern steht.
Des Weiteren wurde für diese Arbeit das Ziel gesetzt, die Treibersoftware möglichst unabhängig von dem verwendet Programm zur Spracherkennung zu gestalten, das bedeutet, dass auch hier keine proprietären Lösungen gewünscht sind. Dieser Ansatz soll die Möglichkeit offen lassen, bei zukünftigen Projekten, eine andere Software zur Spracherkennung einzusetzen, welche potentiell besser zu dem jeweiligen Anwendungsfall passt, oder eine erheblich bessere Erkennungsrate aufweisen kann.

Der Prozess der Ausarbeitung der Architektur (und des Konzepts) des Treibers unterteilt sich in die folgenden Schritte:

1. Auswahl des Kommunikationskanals
2. Schnittstellen/Interface Definition
3. Software-/System-Architektur
4. Implementierung des Prototypen
5. Testing
6. Treiberintegration

Diese Schritte werden iterativ durchlaufen und anhand dieses Prozesses werden somit neben dem Entwickeln eines Konzepts und der Architektur auch die Implementierung und Testung des Treibers durchgeführt. Durch diese Vorgehensweise wird gewährleistet, dass nicht nur die Anforderungen und Ziele (siehe *4. Analyse und Zielsetzung*) erfüllt werden, sondern auch rasch auf mögliche Fehler und Erkenntnisse eingegangen werden kann.

5.3.1. Auswahl des Kommunikationskanals

Bevor eine genaue Architektur des Treibers bestimmt werden kann, ist es essentiell, einen Kommunikationskanal zwischen dem LUI und dem Treiber festzulegen. Bei der Auswahl des Kommunikationskanals bestehen für diese Arbeit durch die bereits implementierten Kommunikationsschnittstellen des LUI bestimmte Vorgaben und es war nicht Teil der Aufgabenstellung dieser Arbeit, die Möglichkeiten in diese Richtung auszuweiten.

Wie in *Abbildung 14* zu sehen, besitzt das LUI bereits einen Pipe-Treiber und einen REST-Treiber. Wie im folgenden Abschnitt noch genauer erklärt wird, ermöglichen diese Treiber einen Datenaustausch sowohl über Pipes als auch das HTTP-Protokoll.

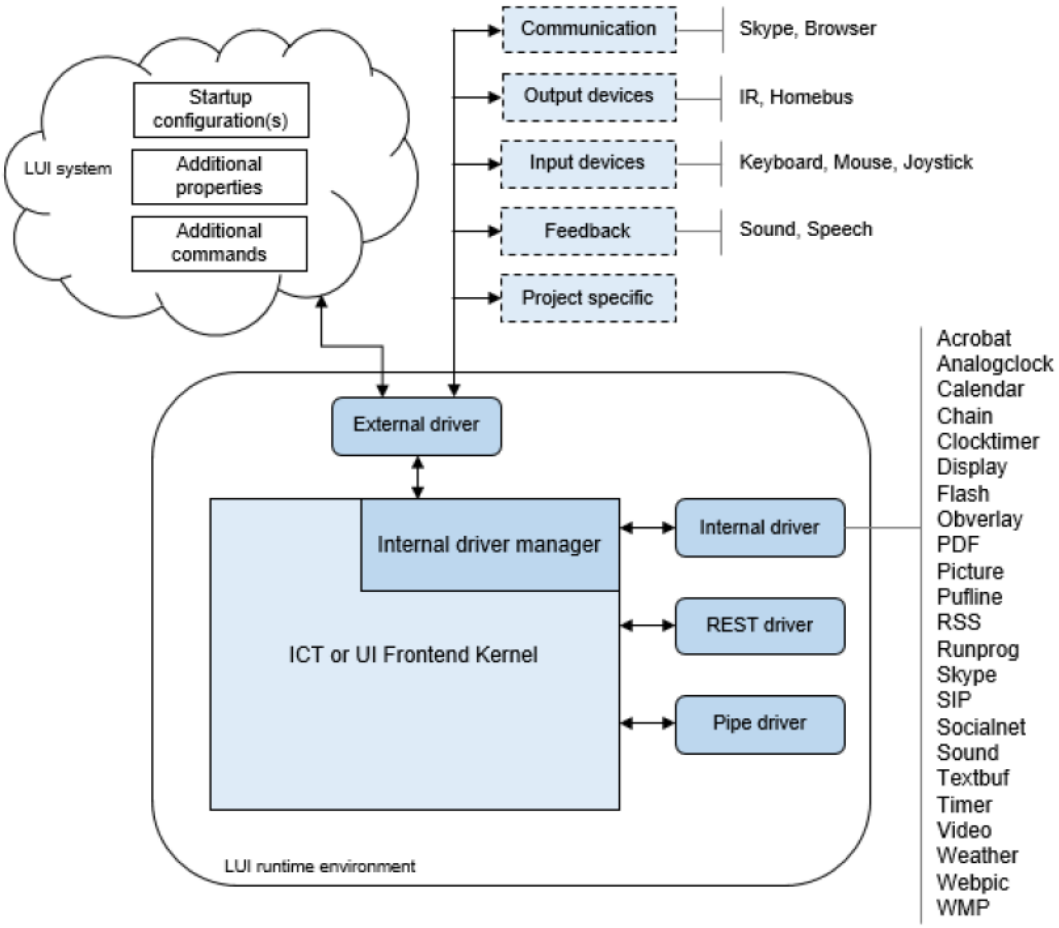


Abbildung 14: Übersicht das LUI runtime environment, inklusive aller Treiberschnittstellen.

Named Pipes

Pipes dienen im Allgemeinen dazu, um Informationen von einem Prozess zu einem anderen zu übermitteln. Die bei diesem Vorgang erzeugte „Leitung“ (Pipe) besitzt in der Regel einen Buffer für die Daten und überträgt meistens einen Bit- oder Byte-Stream. Grundsätzlich lässt sich zwischen zwei Arten von Pipes unterscheiden, den „anonymous Pipes“ und den „named Pipes“. Der LUI-Pipe-Driver implementiert die „named Pipes“, welche nach dem FIFO (first in, first out) Verfahren funktionieren und im laufenden System fortbestehen, selbst wenn der aufrufende Prozess nicht mehr existiert. Die Daten einer „named Pipe“ können außerdem von jedem Prozess gelesen werden, der eine Autorisierung besitzt und den Namen der Pipe kennt [73].

Die beim LUI-Pipe-Driver implementieren Named Pipes werden vom Windows Development Center wie folgt definiert:

“A named pipe is a named, one-way or duplex pipe for communication between the pipe server and one or more pipe clients. All instances of a named pipe share the same pipe name, but each instance has its own buffers and handles, and provides a separate conduit for client/server communication. The use of instances enables multiple pipe clients to use the same named pipe simultaneously.

Any process can access named pipes, subject to security checks, making named pipes an easy form of communication between related or unrelated processes.
Any process can act as both a server and a client, making peer-to-peer communication possible. As used here, the term pipe server refers to a process that creates a named pipe, and the term pipe client refers to a process that connects to an instance of a named pipe. The server-side function for instantiating a named pipe is ***CreateNamedPipe****. The server-side function for accepting a connection is* ***ConnectNamedPipe****. A client process connects to a named pipe by using the* ***CreateFile*** *or* ***CallNamedPipe*** *function.*
Named pipes can be used to provide communication between processes on the same computer or between processes on different computers across a network. If the server service is running, all named pipes are accessible remotely. If you intend to use a named pipe locally only, deny access to NT AUTHORITY\NETWORK or switch to local RPC" [73].

RESTful-API

Die RESTful API (Representation State Transfer Application Program Interface) basiert auf einem Architekturstil, beziehungsweise Architekturansatz, welcher auf eine HTTP-Kommunikation zwischen verteilten Computersystemen ausgerichtet ist. Aus diesem Grund wird die RESTful API auch oft RESTful Web Service genannt. Dabei werden HTTP-Requests eingesetzt, um die, in den *RFC 2616* (Request for Comments der Internet Engineering Task Force) standardisierten und definierten HTTP-Anfragemethoden „GET", „PUT", „POST" und „DELETE", ausführen zu können [74]. Mit diesen Methoden können Daten geändert oder aktualisiert, Ressourcen abgerufen oder erstellt, sowie gelöscht werden. Ein REST-Service funktioniert auf Basis einer Client-Server-Architektur. Bei dieser Architektur stellt ein Server gewisse Services (Dienste) zur Verfügung, die von einem Client angefragt werden können. Das REST-Protokoll ist komplett zustandslos („stateless"), das bedeutet, dass weder Serverseitig noch Client-seitig Zustandsinformationen zwischen zwei Anfragen (Requests) gespeichert werden. Jeder Request ist somit in sich geschlossen und enthält lediglich die vom Server und Client benötigten Informationen. Eine weitere wichtige Anforderung, die bei der Implementierung eines REST Services umgesetzt werden muss, ist die Definition einer einheitlichen Schnittstelle. Das vereinfacht die Architektur erheblich und ermöglicht somit auch eine einfache Nutzung der Schnittstelle. REST-Systeme sind außerdem in mehreren Ebenen („Layers") aufgebaut. Für den Client macht dies keinen Unterschied, da er mit der angebotenen Schnittstelle kommuniziert und nicht weiß, ob es sich um den „End-Server" handelt, oder welche Ebenen sich eventuell noch hinter der Schnittstelle befinden. Durch diese Architektur wird es ermöglicht, Systeme skalierbar zu machen, und es können beispielsweise auch einfach Security-Maßnahmen eingebaut werden [75].

Grundsätzlich wären beide vom LUI zur Verfügung gestellten Kommunikationskanäle für die Implementierung des Treibers zur Spracherkennung geeignet. Bei einem direkten Vergleich lässt sich, auf Grund der zu diesem Zeitpunkt erhobenen Anforderungen und Überlegungen hinsichtlich Implementierung und Performance, keine eklatanten Vor- oder Nachteile einer der beiden Technologien ausmachen. Ausschlaggebend waren an dieser Stelle vor allem die Überlegungen, welche Frameworks und Plattformen für die Treiberimplementierung geeignet sind, beziehungsweise welche Schnittstellen eingesetzt werden sollen. Da für diesen Zweck der Einsatz einer RESTful API angedacht war, ist es naheliegend und auch für die allgemei-

ne Architektur einfacher und übersichtlicher, diese Technologie ebenfalls für die Kommunikation mit dem LUI einzusetzen.
Im Folgenden wird aus diesem Grund nur mehr jene Schnittstelle des LUI betrachtet, die im Rahmen dieser Arbeit relevant war. Somit steht hier das Interface der REST-API im Vordergrund, da über dieses die Kommunikation zwischen dem LUI und dem Treiber abgewickelt wird.

5.3.2. Interface Definitions

Moderne IT-Systeme bestehen in der Regel aus mehreren separaten Komponenten, welche untereinander, aber auch über ihre eigenen Grenzen hinaus, mit anderen Systemen und Komponenten kommunizieren. Diese Kommunikation ist notwendig, um systematische Datenübertragung und Datenverarbeitung zu ermöglichen und findet an bestimmten, vordefinierten Stellen in einer Software statt. Diese Stellen werden im Allgemeinen Schnittstellen oder Interfaces genannt. Unterschieden wird hier zwischen Hardwareschnittstellen, Softwareschnittstellen und Benutzerschnittstellen. Da in dieser Arbeit nur Softwarekomponenten implementiert wurden, ist im Folgenden unter dem Begriff Interface (Schnittstelle) immer eine Softwareschnittstelle zu verstehen.

Vor der Erstellung einer konkreten (detaillierten) Architektur für die Implementierung eines Prototypen, mussten im ersten Schritt die Interfaces (Schnittstellen), basierend auf dem erstem Entwurf des Konzepts der Treibersoftware (siehe *5.1. Konzept und Funktionsweisen der Software*), definiert werden.
Um eine zuverlässige Kommunikation zwischen den einzelnen Softwarekomponenten zu gewährleisten, wurden die Interfaces genau definiert und im folgenden Abschnitt niedergeschrieben. Diese Vorgehensweise dient ebenfalls dazu, die Wartung (Maintainability) des Systems zu verbessern, beziehungsweise zu erleichtern. Das bedeutet, dass Updates, Fehlerbehebung, sowie der Austausch von Komponenten ermöglicht werden, ohne im gesamten System weitere Änderungen vornehmen zu müssen, oder Fehlfunktionen zu riskieren. Außerdem ermöglichen und erleichtern die klar definierten Interfaces anderen Softwareentwicklerinnen und -entwicklern die Anbindung von eigener Software.

Für die im Rahmen dieser Arbeit zu entwickelnde Software wurden im Wesentlichen an zwei unterschiedliche Stellen Interfaces definiert:

1) Schnittstelle zwischen dem LUI und dem Treiber
2) Schnittstelle zwischen der Kommunikationskomponente und der Spracherkennungskomponente
    2.1) Schnittstelle über einen WebSocket
    2.2) REST-Schnittstelle

Bei dem Interface zwischen der Kommunikationskomponente und der Spracherkennungskomponente wurden im Rahmen der Implementierung zwei unterschiedliche Interfaces (REST und WebSocket) eingesetzt. Auf diese Entscheidung, beziehungsweise Vorgehensweise wird im Kapitel *5.3.3. Softwarearchitektur und Workflow* noch genauer eingegangen. Um die Übersichtlichkeit und Struktur dieser Arbeit zu fördern, sind trotzdem bereits in diesem Abschnitt die Interface Definitions aller Schnittstellen enthalten.

1) Interface zwischen dem LUI und dem Treiber für die Spracherkennung:

Die aktuelle, bereits im LUI implementierte REST-Schnittstelle, ermöglicht es dem LUI einerseits Kommandos von anderen Programmen oder Systemen zu empfangen, und andererseits selbst Nachrichten zu verschicken. Hierfür ist in der Konfigurationsdatei des LUI ein Port für das Empfangen von Nachrichten – beziehungsweise Kommandos – definiert, der „listenport“ genannt wird. Für das Versenden von Nachrichten durch das LUI kann die URI (Uniform Resource Identifier) des Empfängers festgelegt werden. Auf die Konfiguration dieser Schnittstelle wird im Kapitel *5.5.1. Treiberkonfiguration* genauer eingegangen.

Im Folgenden wird das REST-Interface zwischen dem LUI und dem Treiber dokumentiert. Diese Dokumentation enthält einerseits die Darstellung der Nachrichten die vom LUI an den Treiber gesandt werden, und andererseits alle Informationen, die an das LUI retourniert werden.

- Empfangen von Nachrichten vom LUI
  Das Empfangen von Nachrichten vom LUI wird in dieser Darstellung anhand einer neuen Liste von Kommandos repräsentiert, die beim Aufruf einer neuen LUI-Seite gesendet werden.

Titel	Receive entrylist from LUI
URL	http://127.0.0.1:12345
Method	POST
Data Params	**Structure/Model:** <?xml version="1.0" encoding="utf-8"?> <command type="broadcast"> <message>newmenu</message> <data> <id>*current_menu_id*</id> <startentry> *startentry_id*</startentry> <start>…</start> <entry> <id> *object_id*</id> <text> *object_text*</text> <speech>*object_speech_command*</speech> <fireenable> *true*|*false*</fireenable> </entry> </data> </command> **Example:** <command type="broadcast"> <message>newmenu</message> <data> <id>M_Main</id> <startentry>E_Commands</startentry> <start>True</start> <entry> <id>E_RSS</id> <text>Lade Nachrichten ...</text>

	<speech/> <fireenable>True</fireenable> </entry> <entry> <id>E_alarm</id> <text>Hilfe!</text> <speech/> <fireenable>True</fireenable> </entry> </data> </command>
Success Response	Code: 200 Content: { }
Error Response	none
Notes	Messages from the LUI will be processed by the driver if they contain the following tags: • <message>newmenu</message> • <message>curmenu</message> • <message>driverunload</message> • <message>changelistenstate</message> • <message>getlistenstate</message> • <driver>driversettings</driver>

Tabelle 1: Interface Definition - Receive entrylist from LUI

- Senden von Nachrichten an das LUI
  Das Senden von Nachrichten vom Treiber an das LUI wird anhand eines (spezifischen) Kommandos dargestellt, welches vom LUI ausgeführt werden soll.

Titel	Send command to LUI
URL	http://127.0.0.1:45678 (or another in the configs specified port)
Method	POST
Data Params	**Structure/Model:** <?xml version="1.0" encoding="utf-8"?> <command type="command_type" token=”token_chars” sendid=”send_id” > <action>action_type</action> <data> command_id </data> </command> Example: <command type="internal" token="011235" sendid="tz"> <action>clickentry</action>

	<data> E_back </data> </command>
Success Response	Code: 200 Content: { }
Error Response	Common HTTP-Error-Codes
Sample Call	var responseCommand = '<command type="internal" token= "011235" sendid="tz"><action> clickentry </action><data>E-back</data></command>'; var options = { host: '127.0.0.1', port: '45678', method: 'POST', headers: { 'Content-Type': 'application/xml', 'Content-Length': responseCommand.length } }; req.write(responseCommand); req.end();
Notes	The following actions will be sent from the driver to the LUI: • <action>broadcastcurrentmenu</action> • <action>getdrvconfig</action> • <action>clickentry</action> • <action>displaymessage</action> • <action>getlistenstate</action> • <action>errorcallback</action>

Tabelle 2: Interface Definition - Send command to LUI

2. Interface zwischen der Kommunikationskomponente und der Spracherkennungs-komponente:

Diese Schnittstelle wird in der Implementierung dieser Arbeit in zwei unterschiedlichen Ausprägungen umgesetzt: Einerseits mit einem WebSocket-Server und andererseits mit einem HTTP-Server (siehe *5.3.3. Softwarearchitektur)*.

2.1. Interface Definitions WebSocket-Server:

Titel	Connect/send data to the WebSocket-Client (Speech Recognition Application)

URL	The WebSocket-server receives the required connection-data from the client when the client requests a connection. Further data can only be sent after the HTTP upgrade.
Method	SEND
Data Params	**Structure/Model:** data : JSON-String **Example (JSON decoded):** **//entrylist** var data = { entryList: { Lui: Internet, Lui: Hilfe, Lui: zurück } }; **//settings** var data = { settings: { token: 012345, confidenceThreshold: 0,55, commandWord: lui, language: de } }; **//listenState** var data = { listenState: { listenState: false } };
Success Response	Code: 200
Error Response	A close event is raised and provides an error message containing the closure reason and an error code.
Sample Call	var content = { entryList: { Lui: Internet, Lui: Hilfe, Lui: zurück } }; var type = „entryList“; var WebSocketServ = require('websocket').server; // server for the webSocket var server = http.createServer(function (request, response) { });

	server.listen(56789, function () { console.log('WebSocket running at http://127.0.0.1:' + 56789); }); websServer = new WebSocketServ({ httpServer: server }); websServer.on('request', function (request) { var connection = request.accept(null, request.origin); wsSend = function (content, type) { connection.send(JSON.stringify({content}), function e(error) { console.error('Send error occurred! Error message: ' + error); } }); };
Notes	The "Kernkomponente" (Node.js Applikation) sends the current Entrylist and Settings to the WebSocket-Client when they are available. Additionally the WebSocket-Client can request them with the messages „get-entrylist" and „get-settings".

Tabelle 3: Connect/send data to the WebSocket-Client (Speech Recognition Application)

Titel	Receive commands from the WebSocket-Client (Speech Recognition Application)
URL	http://127.0.0.1: 56789
Method	SEND
Data Params	**Structure/Model:** data received as JSON-String **Examples (JSON decoded):** *//client-request for the current menu-entries* var data = "get-entrylist"; *//client-request for the settings* var data = "get-settings"; *//recognized command id from client* var data = clickentry: "E_information"; *//display message from client* var data = display-message: "Message content"; *//error from client* var data = error-callback: "Error message";

Success Response	Code: 200
Error Response	A close event is raised and provides an error message containing the closure reason and an error code.
Sample Call	var content = {“click-entry”: “E_back”}; var WebSocketClient = require('websocket').client; client.on('connect', function(connection) { console.log('WebSocket-client connected to http://127.0.0.1:56789'); wsSend = function (content) { connection.sendUTF(JSON.stringify({content}), function e(error) { console.error('Send error occurred! Error message: ' + error); } } client.connect(‘http://127.0.0.1:56789');
Notes	The WebSocket-Server differentiate between the following requests: • “get-entrylist” • “get-settings” • “click-entry” • “display-message” • “error-callback”

Tabelle 4: Receive commands from the the WebSocket-Client (Speech Recognition Application)

2.2. Interface Definition HTTP-Server:

Titel	Send data to the REST-Client (Speech Recognition Application)
URL	Only sends responses to the request origin
Method	POST
Data Params	**Structure/Model:** data : JSON-String **Example (JSON decoded):** **//entrylist** var data = { entryList: { Lui: Internet, Lui: Hilfe, Lui: zurück } };

	//settings var data = { settings: { token: 012345, confidenceThreshold: 0,55, commandWord: lui, language: de } };
Success Response	Code: 200
Error Response	Common HTTP-Error-Codes
Sample Call	var content = { entryList: { Lui: Internet, Lui: Hilfe, Lui: zurück } }; response.writeHead(200, { 'Content-Type': 'application/json' }); response.write(JSON.stringify(content)); response.end();
Notes	The REST-Client can request the entrylist and the settings with the messages „get-entrylist" and „get-settings".

Tabelle 5: Send data to the (Speech Recognition Application)

Titel	Receive messages from the REST-Client (Speech Recognition Application)
URL	http://127.0.0.1: 54321
Method	POST
Data Params	**Structure/Model:** data : JSON-String **Example (JSON decoded):** *//client-request for the current menu-entries* var data = „get-entrylist"; *//client-request for the settings* var data = „get-settings"; *//recognized command id from client* var data = „E_back";

Success Response	Code: 200
Error Response	Common HTTP-Error-Codes
Sample Call	http.open("POST", 'http:// 127.0.0.1:54321, true); http.setRequestHeader("Content-type"; “application/json; charset=UTF-8”;); sendContent = JSON.stringify(„Lui Internet“); http.send(sendContent);
Notes	The HTTP-Server differentiate between the following requests: • “get-entrylist” • “get-settings” • “click-entry” • “display-message” • “error-callback”

Tabelle 6: Receive messages from the REST-Client (Speech Recognition Application)

5.3.3. Softwarearchitektur und Workflow

Die Architektur einer Software stellt die wesentliche Grundlage und Basis für die Entwicklung einer Software dar. Durch die Softwarearchitektur werden alle im System vorhandenen Komponenten und Teilsysteme strukturiert beschrieben, sowie deren Funktion und Hierarchie definiert. Darüber hinaus wird festgelegt in welcher Beziehung sich die Systembestandteile zueinander befinden und welche Interaktionen stattfinden. Dieses Vorgehen dient unter anderem dazu, nicht-funktionale Anforderungen einzuarbeiten, sowie das gewünschte Maß an Qualität sicherzustellen. Die Architektur bestimmt somit das Design einer Software und ist daher maßgeblich dafür verantwortlich, ob ein System in weiterer Folge über die gewünschte Funktionalität verfügt.

Im folgenden Abschnitt wird die Vorgehensweise beschrieben, mit der die Softwarearchitektur erstellt wurde, sowie auf welchen Entscheidungen sie basiert. Des Weiteren wird auch dargelegt, welche Erkenntnisse im Verlauf des iterativen Entwicklungsprozesses des Prototypen Änderungen, beziehungsweise ein Umdenken, bei der Architektur erforderlich gemacht haben.
Abschließend wird in diesem Abschnitt der Standard-Workflow der Software skizziert.

Basierend auf der Analyse, welche Anforderungen an einen Treiber für die Bedienung des LUI mittels Sprachkommandos gestellt werden (siehe Kapitel *4. Analyse und Zielsetzung*),

den erhobenen Use Cases und dem erstellten Konzept (siehe Kapitel *5.1. Konzept und Funktionsweisen der Software*), wurden an dieser Stelle die Designentscheidungen für die Softwarearchitektur getroffen. Der erste Schritt in diesem Prozess war, anhand der Use Cases und identifizierten Hauptfunktionalitäten eine grobe Struktur für den Treiber zu entwerfen. Infolgedessen ergab sich die Fragestellung, ob der Treiber aus einer Software mit mehreren Modulen, oder aber aus mehreren eigenständigen Softwarekomponenten bestehen soll.
Der Vorteil einer „Ein-Komponenten-Lösung“ wäre, dass keine weiteren Kommunikationskanäle zum Spracherkennungsmodul benötigt würden. Das bedeutet, dass während der Laufzeit der Software, nach dem Empfang von Nachrichtigen vom LUI, die Informationen ohne weitere Übertragungszeit sofort für die Spracherkennung (oder Spracherkennungsroutine) zur Verfügung stehen würden. Dieser Fall tritt allerdings nur dann ein, wenn das mitgelieferte Modul zur Spracherkennung auch tatsächlich verwendet wird. Da der Treiber aber unterschiedliche Applikationen zur Spracherkennung unterstützen soll, wäre in jedem Fall ein weiteres Interface (API) notwendig. Diese Lösung würde es erheblich verkomplizierteren, das integrierte Sprachmodul in Zukunft zu warten oder zu verbessern.
Für die Realisierung der gewünschten Software als nur eine Komponente (mit mehreren Modulen) wäre es außerdem notwendig, ein Framework oder eine Plattform zu finden, welche alle Anforderungen umsetzen könnte. Das würde bedeuten, dass sowohl die gesamte Logik für Kommunikation und Verarbeitung, als auch die Spracherkennung realisierbar sein müssten, mit der Einschränkung, keine zu proprietäre Lösung zu entwickeln.

Im Vergleich dazu, könnte bei der Implementierung von mehreren Komponenten, eine Kernkomponente nur für die Kommunikation und deren Verarbeitung eingesetzt werden. Das würde bedeuten, dass nur diese Komponente des Treibers direkt mit dem LUI kommuniziert und darüber hinaus über ein Interface verfügt, das es ermöglicht, eine zusätzliche Komponente zur Spracherkennung anzubinden. Dadurch wäre sowohl bei der Auswahl des Frameworks und der Entwicklungsumgebung für die Kernkomponente eine höhere Flexibilität gegeben, als auch bei der Entwicklung von Komponenten zur Spracherkennung.

Wie oben ausgeführt, lässt sich bereits bei einer relativ oberflächlichen Analyse und Betrachtung der Problemstellung erkennen, dass eine komponentenbasierte Softwareentwicklung für den Treiber zur Sprachsteuerung des LUI zielführender und besser geeignet ist. Deswegen wurde die Entscheidung getroffen, eine Software bestehend aus mehreren Komponenten, mit klar definierten Interfaces, zu entwickeln. Des Weiteren wurde diese Herangehensweise durch die durchgeführten Recherchen zu möglichen Plattformen, sowie Frameworks, die für die Entwicklung des Treibers geeignet wären, untermauert. Da keine Technologie gefunden wurde, die alle Anforderungen auf einmal optimal abdecken konnte, war es durch die Trennung auf mehreren Komponenten möglich, die jeweils am besten geeignetsten Technologien einzusetzen. Das entspricht auch der Zielsetzung einer möglichst flexiblen und generellen Lösung. Beim Einsatz von nur einer Komponente hätte dieses Ziel eine erheblich größere Herausforderung dargestellt.
Durch den Einsatz von logisch und funktionell in sich geschlossenen Komponenten und die eindeutige Definition von Interfaces, inklusive ihrer Dokumentation, wurde gewährleistet, dass unterschiedliche Teile der Software (Komponenten) ausgetauscht werden können, ohne die allgemeine Funktionsfähigkeit des Systems zu gefährden. Zugleich kam dieser Art der Architektur auch schon bei der Entwicklung eine große Bedeutung zu. So konnten, je nach Funktionalität der jeweiligen Komponente, die Zielsetzungen und Anforderungen schon zu Beginn der Entwicklung viel granularer und vor allem unabhängig von anderen Komponen-

ten, betrachtet werden. Indem Interfaces vorab definiert und dokumentiert wurden, war es außerdem möglich, die Komponenten unabhängig voneinander zu entwickeln und in Folge auch zu testen, ohne ein komplett funktionierendes System, oder funktionierendes Programm zu benötigen. Dies ermöglichte eine einfachere und bessere Optimierung der einzelnen Komponenten.

Für die folgenden Abschnitte werden die Begriffe „gesamtes System“ und „Gesamtlösung“ dermaßen definiert, dass sie alle Komponenten beinhalten, welche für das fertige Produkt benötigt werden. Das bedeutet, dass neben dem Treiber und dessen Komponenten auch das LUI zu diesem System gezählt wird, obwohl nur die anderen beiden Komponenten im Rahmen dieser Arbeit entwickelt wurden. Dementsprechend umfasst die Architektur des gesamten Systems im Wesentlichen die folgenden Teilsysteme und Komponenten:

- LUI
- Treiber für die Spracherkennung:
    - Kommunikationskomponente (Node.js Applikation)
    - Applikation zur Spracherkennung

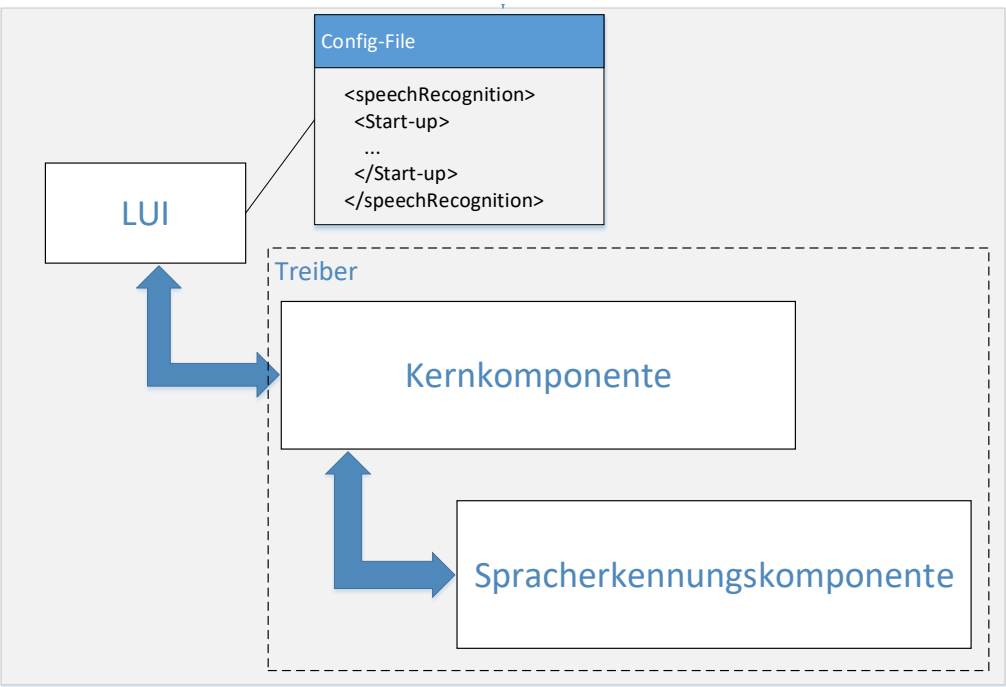


Abbildung 15: Architektur des gesamten Systems, bestehend aus dem LUI (das beim Start das Config-File lädt), der Kern- und Spracherkennungskomponente des Treibers. Außerdem sind durch Pfeile die Schnittstellen bzw. Kommunikationswege dargestellt.

An dieser Stelle sei erwähnt, dass der erste Entwurf des Prototypen ausschließlich eine einzelne Schnittstelle von der Kernkomponente des Treibers zur Applikation für die Spracherkennung vorsah (siehe *Abbildung 9*). Diese Schnittstelle wurde als ein REST-Interface umgesetzt und war vorerst auch nur dafür konzipiert, Spracheingaben von der Spracherkennungssoftware zu empfangen, auf ihre Gültigkeit zu prüfen und zu verarbeiten. Auch ein Versenden von Menu-Entries und Settings an die Spracherkennungssoftware wurde über HTTP-Requests problemlos realisiert. Eine wesentliche Schwachstelle war allerdings, dass bei einer REST-Schnittstelle keine dauerhafte Verbindung aufgebaut wird. Somit war für die

Kernkomponente nicht direkt erkennbar, ob und wann die Spracherkennungssoftware eventuell nicht mehr zur Verfügung stand oder nicht mehr funktionsbereit war. Zu diesem Zeitpunkt wurde allerdings die Entscheidung getroffen, beziehungsweise die Anforderung definiert, dass die Kernkomponente zu jeder Zeit die Information besitzen muss, ob die Spracherkennung verfügbar ist, um gegebenenfalls die Anwenderin und den Anwender umgehend darüber informieren zu können. Um an dieser Stelle keine Lösung zu implementieren, die unnötig Rechenleistung benötigt oder HTTP-Traffic erzeugt, musste ein Redesign für das Interface stattfinden und die REST-Schnittstelle ersetzt werden.
Nach einer Recherche wurde im Rahmen des schlussendlichen Einsatzes für diese treiberinterne Schnittstelle das WebSocket-Protokoll gewählt und implementiert. Dabei wird eine dauerhafte Verbindung zwischen dem WebSocket-Server (der Kernkomponente) und dem WebSocket-Client (Spracherkennungssoftware) erzeugt. Beim Abbruch der Verbindung oder Auftreten von Fehlern, wird die Kernkomponente des Treibers umgehend benachrichtigt und kann dementsprechende Aktionen einleiten, wie beispielsweise die Anwenderin oder den Anwender über eine im LUI eingeblendete Nachricht zu informieren.

Da die ursprüngliche REST-Schnittstelle aber schon funktionsfähig implementiert war, wurde sie als „Development-Interface“ in der Kernkomponente belassen. Das bedeutet, bei der entsprechenden Konfiguration in den Treibereinstellungen des LUI könnte für Entwicklungs- und Testzwecke einfach und schnell eine neue Applikation zur Spracherkennung über das REST-Interface angebunden werden. Durch die Dokumentation dieses Interfaces ist auch ein Refactoring oder Umbau in künftigen Arbeiten möglich.

Die endgültige Architektur, wie sie schlussendlich auch im Treiber implementiert ist, ist in der nachfolgenden *Abbildung 16* zu sehen.

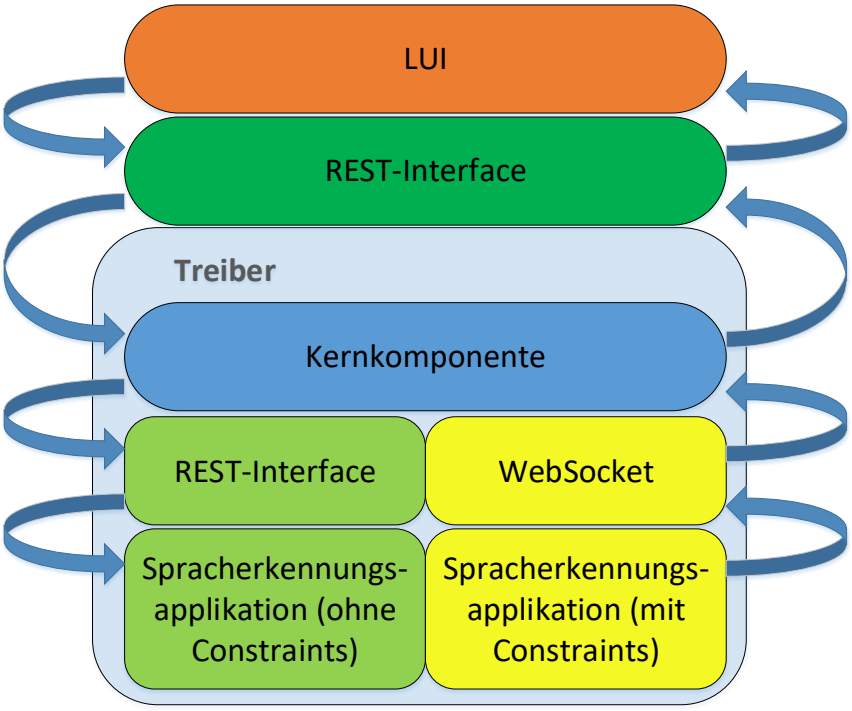


Abbildung 16: Zweiter Entwurf der Übersicht des ganzen Systems inklusive seiner Komponenten und Schnittstellen, mit den zwei treiberinternen Schnittstellen REST und WebSocket. Auf der linken Seite ist der Daten- und Informationsfluss vom LUI zum Treiber (über die jeweilige treiberinterne Schnittstelle hin zur Spracherkennungsapplikation) zu sehen. Auf der rechten Seite ist der Daten- und Informationsfluss in die entgegengesetzte Richtung zu sehen.

Workflow:

Sobald von der Anwenderin oder dem Anwender das LUI gestartet wird, wird über die gesetzten Einstellungen in der Konfigurationsdatei des LUI (siehe *Listing 14*) der Treiber gestartet. Auf Grund der gewählten Architektur für den Treiber bedeutet das, dass sowohl der Start der Kernkomponente (Kommunikationskomponente) getriggert wird, als auch der Start der Spracherkennungssoftware. Anschließend werden vom LUI zuerst die Settings und dann die Menu-Entries der aktuellen Seite des LUI, an den Treiber, beziehungsweise die Kernkomponente des Treibers gesendet. Die Settings sind für den Treiber deshalb wichtig, weil an dieser Stelle auch diverse Einstellungen, die den Treiber und seine Komponenten betreffen, vorgenommen werden können. Nach Bedarf werden der Kernkomponente die verarbeiteten Settings und Menu-Entries an die Applikation zur Spracherkennung weitergeleitet. Anschließend können von der Anwenderin oder dem Anwender Spracheingaben getätigt werden und sobald ein gültiges Sprachkommando erkannt wird, wird dieses von der Spracherkennungsapplikation an die Kernkomponente gesandt. Diese bringt das Kommando in die, für das LUI verständliche, Syntax und leitet es an das LUI weiter, welches wiederum die gewünschte Aktion ausführt (siehe *Abbildung 17*).
Dieser grobe Überblick über die Funktionalität des Systems wird in den nachfolgenden Abschnitten – der Dokumentation der einzelnen Komponenten – vor allem in Bezug auf Abläufe und mögliche Fehlerfälle, noch ins Detail vertieft.

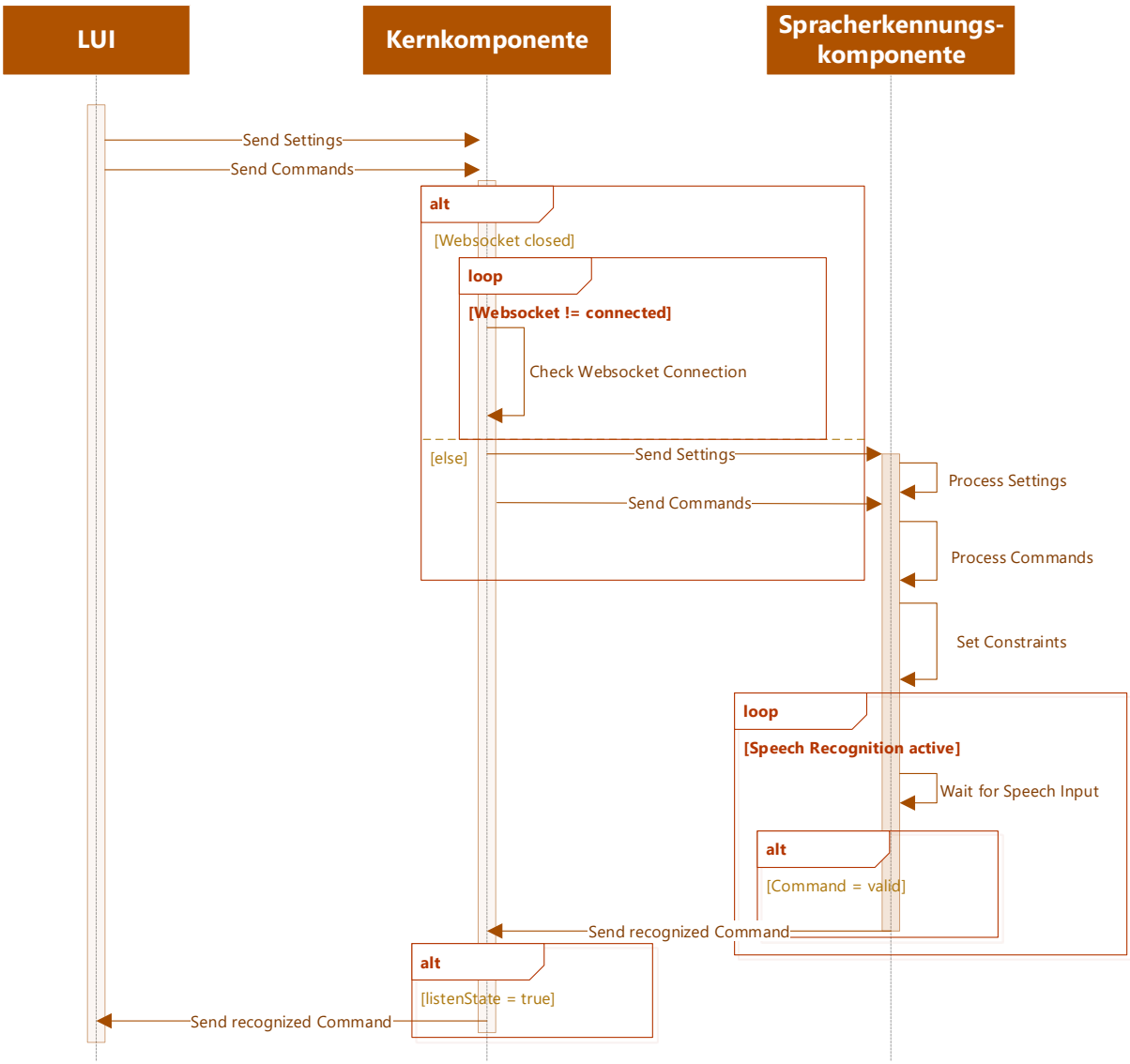


Abbildung 17: Sequenzdiagramm vom Standard-Workflow des Treibers

5.3.4. Prototyp – Komponenten im Detail

In diesem Kapitel wird auf die Implementierungsdetails des Prototypen beziehungsweise der einzelnen Komponenten des Prototypen eingegangen. Mit Ausnahme des LUI, welches im Rahmen des Gesamtsystems ebenfalls als Komponente definiert wurde, sind alle Komponenten selbst entwickelt und implementiert worden. In den jeweiligen Abschnitten der selbst entwickelten Komponenten werden anhand der implementierten Methoden die exakten Programmabläufe beschrieben und dokumentiert. Des Weiteren wird auf Details zur Implementierung der eingebundenen Frameworks und des genutzten Fremdcodes eingegangen.
Das LUI wird der Vollständigkeit halber in diesem Abschnitt ebenfalls beschrieben, dabei handelt es sich, im Gegensatz zu den anderen Komponenten, allerdings nur um eine kurze Zusammenfassung der wichtigsten Funktionen und Erläuterung jener Teile, die für die Umsetzung des Gesamtsystems relevant waren.

5.3.4.1. LUI

Beim Local User Interface (kurz LUI) handelt sich um jenes Programm, für das im Rahmen dieser Arbeit ein Treiber zur Bedienung und Steuerung mittels Spracheingabe implementiert wurde. Für die finale Implementierung, sowie Tests, wurde die Version *0.21.1.11* genutzt.
Das LUI wurde als Benutzeroberfläche entwickelt, die eine erleichterte Bedienung (barrierefreie Bedienung) für bestimmte Funktionen und Programme ermöglichen soll. Zu den Hauptfunktionen dieses Programms zählen eine Telefon- und Videotelefonfunktion, Inhalte zur Unterhaltung (beispielsweise diverse Spiele, eBücher, Radio, Hörbücher) und Informationsmedien (Internet-Browser, Wetter-Information, Handbuch und Soziale Netze). Ein weiteres essentielles Feature des LUI ist der Notruf-Button, mit dem von nahezu jeder Seite des LUI ein Hilferuf an eine vorab konfigurierte Nummer gesendet werden kann.
Im Folgenden ist ein Screenshot der Hauptseite des LUI zu sehen:

Abbildung 18: Screenshot des grafischen User Interfaces (GUI) einer LUI-Seite

Für weitere Details und Informationen wird auf die Dokumentation und Anleitung des LUI verwiesen [76] [77].

5.3.4.2. Kernkomponente

Die Node.js Applikation stellt die Kernkomponente – also genauer gesagt den Dreh- und Angelpunkt – des entwickelten Treibers dar.

Bei der Kernkomponente (Kommunikationskomponente) der Treiber-Software hat das mit Node.js ausgeführte Script im Allgemeinen die Aufgabe, die gesamte Kommunikation zwischen allen Komponenten zu regeln, beziehungsweise für eine Form von Routing (Verteilung) der Daten und Informationen zu sorgen.

Für die Kommunikation mit dem LUI fiel die Wahl auf den Einsatz eines HTTP-Servers (siehe *5.3.3. Softwarearchitektur und Workflow*). Da seitens des LUI die Interfaces (siehe *5.3.2. Interface Definitions*) so ausgelegt sind, dass über einen Port Nachrichten versandt und nur über einen anderen Port Kommandos empfangen werden, war es notwendig, zwei getrennte Webserver-Instanzen für diese Kommunikation einzusetzen. Das bedeutet, in der Kernkomponente wurden zwei HTTP-Server implementiert, wobei einer der beiden die Aufgabe hat Nachrichten zu verarbeiten, die über die Schnittstelle vom LUI eingehen, und der andere, Nachrichten in die entgegengesetzte Richtung an das LUI zu senden.
Wie bereits im Abschnitt *5.3.3. Softwarearchitektur und Workflow* erklärt wurde, musste festgestellt werden, dass eine weitere HTTP-Server-Instanz für die Schnittstelle zu einer Spracherkennungssoftware nicht ausreichend war (weitere Details dazu werden im folgenden Abschnitt der Dokumentation der Kernkomponente erläutert). Da die Webserver-Instanz aber grundsätzlich funktionsfähig ist und vor allem für Test- und Entwicklungszwecke leicht nutzbar ist, wurde diese Schnittstelle nicht aus der Anwendung entfernt, kommt aber im Standard-Programmablauf nicht zu Einsatz.
Aus diesem Grund wurde zusätzlich ein weiteres Interface implementiert. Dieses kommuniziert über einen WebSocket-Server und ist für den praktischen Einsatz und eine dauerhafte Nutzung gedacht. Dabei ermöglicht dieses zweite Interface den Applikationen zur Spracherkennung den Aufbau einer dauerhaften Verbindung zum WebSocket-Server der Node.js Applikation.

Zusammengefasst bedeutet das, die Kernkomponente des Treibers besitzt sowohl eine REST- als auch eine WebSocket-Schnittstelle für die Anbindung einer Spracherkennungssoftware. Der Treiber kann aber ausschließlich optional eine Anbindung an eine der beiden bereits erwähnten Interfaces der Kernkomponente nutzen (siehe *Abbildung 16*). Für den Einsatz bei Anwenderinnen und Anwendern ist aber lediglich das WebSocket-Interface konzipiert.

Entwurf

Wie bereits im vorangegangenen Absatz angeschnitten wurde, stellte es sich zu Beginn der Entwicklung der lokalen Node.js-Applikation (Kernkomponente) als eine der größten Herausforderungen heraus, ein Konzept zu erstellen, welches zuverlässig und performant mit einer Kombination aus asynchroner Kommunikation an den externen Schnittstellen und einer möglichst synchronen Datenverarbeitung und Weiterleitung an den internen Schnittstellen umgehen kann. Aus diesem Grund wird anschließend noch genauer auf die Beweggründe für die letztliche Implementierungsentscheidung eingegangen.

Da ein Nachrichtenaustausch sowohl mit dem LUI, als auch mit Applikationen zur Spracherkennung stattfinden soll, stellte sich die Frage, ob an dieser Stelle eine Nutzung der gleichen Schnittstellen (Ressourcen) ausreichend, beziehungsweise sinnvoll ist. Das Interface zum LUI (siehe Kapitel *5.3.2. Interface Definitions*) ist bereits durch das LUI vorgegeben und muss dementsprechend mit einem HTTP-Server implementiert werden, beziehungsweise muss es sich aus den bereits angeführten Gründen um zwei Webserver-Instanzen handeln (Kapitel *5.3.3. Softwarearchitektur und Workflow*). Wobei die Wiederverwendung einer dieser beiden Webserver-Instanzen für die Kommunikation mit zwei unterschiedlichen Komponenten (LUI und Spracherkennungskomponente) schon bei der technischen Umsetzung eine Herausforderung und große Fehlerquelle darstellen würde. Das machte diese Variante ungeeignet für eine Implementierung.
Wie in Kapitel *5.3.3. Softwarearchitektur und Workflow* beschrieben, wurde Umsetzung der Schnittstelle mittels eines WebSocket-Protokolls realisiert. Hier wird eine dauerhafte Verbindung zwischen WebSocket-Server und WebSocket-Client aufgebaut, aber es ist trotzdem eine bidirektionale, persistente und asynchrone Kommunikation möglich. Beide Seiten werden außerdem über Events informiert, wenn sich der Verbindungsstatus ändert und können dementsprechend reagieren.

Nach dieser Entscheidung stellte sich die Frage, in welchem Daten-Format die Informationen zwischen der Kernkomponente und der Spracherkennungskomponente ausgetauscht werden sollten. Nachdem diese Schnittstelle völlig unabhängig von jener, zwischen der Kernkomponente und dem LUI war, bot es sich an, ein anders Daten-Format, als das vom LUI verwendete XML-Format zu nutzen. Der Grund dafür war, dass das XML-Format neben den tatsächlichen Daten auch ihre komplette Datenstruktur überträgt und eher für eine übersichtliche textuelle Darstellung oder grafische Benutzeroberflächen geeignet ist, als für die effiziente Übertragung und Verarbeitung in Anwendungen. Für dieses Anwendungsgebiet ist eines der gängigsten Daten-Formate das JSON-Format (JavaScript Object Notation). JSON-Daten lassen sich wesentlich einfacher und schneller parsen als XML-Daten und JSON-Daten-Typen besitzen ein „1:1 Mapping“ auf die gängigen primitiven Daten-Typen (String, Integer, Array etc.).

Schon während der Recherchen und den ersten Entwürfen für den Treiber und seine Komponenten ergab sich außerdem die Fragestellung, in welcher Form der Treiber implementiert werden könnte, um möglichst vielen unterschiedlichen Anwendungen für die Spracherkennung eine Anbindung zu ermöglichen. Da das Node.js-Framework nicht alle erforderlichen Funktionalitäten zur Verfügung stellt, war es an dieser Stelle notwendig, die Plattform um Packages zu erweitern, welche die gewünschten Methoden beinhalten. Durch das Open Source Konzept sind eine Vielzahl an Packages erhältlich, wobei oft mehrere verschiedene annähernd die gleichen Funktionalitäten anbieten. Auch Packages, die sich nicht mehr im Beta Status befinden, bergen unter Umständen die Gefahr von Bugs oder unvollständigem Error-Handling. Eine zumindest oberflächliche Recherche im Vorfeld ist daher unabdingbar, um negative Überraschungen und zusätzlichen Zeitaufwand, die eventuell durch diesen Fremdcode verursacht werden könnten, zu vermeiden.
Eine sehr bekannte, sowie umfangreiche und mächtige Erweiterung für das Node-Framework ist das Modul mit dem Namen „Express“. Dieses weist aber durch die vorhandene Anzahl von Funktionen, die Großteils im Rahmen dieser Arbeit nicht benötigt wurden, eine unnötige Komplexität auf, weshalb sich der Autor gegen einen Einsatz dieses Frameworks entschied. Einerseits war es wichtig, dass unbeabsichtigte „Fehlfunktionen“ durch eventuelle, durch das Package überlagerte Funktionen, vermieden werden sollten. Des Wei-

teren kam es bereits bei ersten Implementierungsversuchen eines HTTP-Servers beim Empfang von LUI-Nachrichten zu unerwarteten Ergebnissen (Schwierigkeiten), welche bei der Benutzung einer vom Node-Basis-Framework angebotenen Methode nicht eintraten. Daraus ließ sich schließen, dass die optimale Lösung wäre, ausschließlich Packages zu verwenden, welche möglichst exakt die gewünschten, beziehungsweise benötigten, Funktionalitäten lieferten, aber nicht mehr Komplexität als notwendig aufwiesen.

Im Folgenden Abschnitt werden die für die Entwicklung der Node.js Applikation in dieser Arbeit eingesetzten Packages, beziehungsweise Module aufgelistet und beschrieben. Soll ein Modul in ein Script eingebunden werden, kann dieses über das Keyword „require“ und den Namen des Moduls als Parameter, beispielsweise einer Variablen zugewiesen werden. Diese Variable besitzt nun eine Referenz auf das Modul und es können somit die von diesem Modul exportierten Funktionen (public functions) aufgerufen werden. Ein Beispiel für das Einbinden eines Packages wäre:

```
//Initialisierung der Variablen http mit dem http-Modul
var http = require('http');

//Aufruf der createServer-Methode des http-Moduls
var server = http.createServer(function (request, response) {
        …
}).listen(8080);
```

Listing 5: Beispiel für das Einbinden eines Node.js Packages/Modules

Zu allen built-in Features (Packages) des Node.js Frameworks gibt es auf der Homepage (nodejs.org) eine Dokumentation, welche nicht nur die Funktionsweise und wesentlichen Methoden der Module erklärt, sondern auch Aufschluss über den Entwicklungsstatus gibt. So wird bei Node.js die folgende Klassifizierung für Features verwendet:

Stability: 0 - Deprecated This feature is known to be problematic, and changes are planned. Do not rely on it. Use of the feature may cause warnings. Backwards compatibility should not be expected.
Stability: 1 - Experimental This feature is subject to change, and is gated by a command line flag. It may change or be removed in future versions.
Stability: 2 - Stable The API has proven satisfactory. Compatibility with the npm ecosystem is a high priority, and will not be broken unless absolutely necessary.

Tabelle 7: Stability-Klassifizierung der Module in Node.js [81]

Der Entwicklungsstatus von Features ist deshalb wichtig, da es ständig Weiterentwicklungen am Framework gibt, die auch Module betreffen können, welche bereits den Zustand „Stability 2“ hatten.
Im Rahmen dieser Arbeit wurden ausschließlich Node.js Features und Packages eingesetzt, die den Zustand „Stability 2“ hatten.

Nachfolgend werden die Module beziehungsweise Packages vorgestellt, die bei der Implementierung des Prototypen der Node.js Applikation eingesetzt wurden. Hierbei sollen die Beschreibungen lediglich einen Überblick geben und nur jene Methoden der Packages, die auch im Zuge dieser Arbeit eingesetzt wurden, werden ausführlicher beschrieben [82].

- Modul *http*[10]:
  Bei dem *http* Modul handelt es sich um ein built-in Module von Node.js, was bedeutet, dass es direkt nach dem Setup des Node.js-Frameworks in ein Script eingebunden und verwendet werden kann. Das *http* Modul ermöglicht die Kommunikation, beziehungsweise das Übermitteln von Daten, über das Hyper Text Transfer Protocol (HTTP). Zu den für diese Arbeit relevanten wichtigen Funktionen zählen das Erstellen eines HTTP-Servers (Webserver), das Versenden von Client-Requests und das Erzeugen von Sockets.
  - Funktion „http-Server“
    Für den HTTP-Server kann ein Listen-Port definiert werden, über den Nachrichten empfangen und Daten auf direktem Weg auch wieder retour geschickt werden können.
  - Funktion „Client-Request“
    Der Client-Request ermöglicht das Senden von Nachrichten (Requests) an einen potenziellen Client. Um einen solchen Request verschicken zu können, ist es notwendig, einen Header mit der Host-Adresse und dem Port zu setzten.
  - Funktion „http-Agent“
    Mit dem http-Agent des HTTP-Moduls können HTTP-Clients verwaltet werden. Das bedeutet nicht nur, dass die HTTP-Verbindungen überwacht und geregelt werden, sondern auch, dass die Möglichkeit besteht, eine Verbindung zu HTTP-Clients wiederherstellen zu können. Genauer gesagt besitzt der HTTP-Agent eine Warteschlange von Anfragen, sowie einem zugehörigen Host und Port. Für die Abarbeitung jeder dieser Anfragen wird jeweils ein einzelner Socket wiederverwendet. Durch die Option ‚keepAlive‘ kann festgelegt werden, ob nach der Verarbeitung der Anfrage die Socket-Verbindung gespeichert werden soll, was eine zukünftige Wiederverwendung ermöglichen würde.

- Modul *events*[11]:
  Das Node.js Framework basiert unter anderem auf dem Konzept der Events und besitzt demnach ein sogenanntes Event-Driven Behaviour. Es werden also periodisch Events ausgelöst („emitted“), die dafür sorgen, dass sogenannte ‚Listeners‘ (Function Objects) aufgerufen werden. In einem Listener kann eine Methode definiert werden, welche beim Aufruf des Listeners und somit als Reaktion auf das Event, ausgeführt wird. Daher ist auch dieses Modul built-in und ermöglicht das Erstellen von eigenen Events, sowie deren Handhabung. Dazu zählen im Allgemeinen neue Event-Emitter, das Erstellen und Entfernen neuer Events, sowie das Hinzufügen und Entfernen von Event-Listener. Grundsätzlich können mehrere Listener für das gleiche Event definiert werden. Die Abarbeitung erfolgt dabei in jener Reihenfolge, in der sie im Script registriert wurden. Im Gegensatz zur asynchronen Auslösung („Emission“) eines Events, ist die Ausführung der für dieses Event definierten Listener inklusive all ihrer

[10] https://nodejs.org/api/http.html
[11] https://nodejs.org/api/events.html

Funktionen synchron.

- Modul *cluster*[12]:
  Mit dem Modul *cluster* ist es möglich, eine Vielzahl von „Child“-Prozessen zu erzeugen, um den sogenannten „Workload“ zu verteilen. Die vom „Master“-Prozess abgezweigten Prozesse werden in diesem Modul „Worker“ („Child“-Prozesse) genannt. Worker laufen simultan zueinander und ermöglichen es, ein Node.js Script, oder Teile davon, mehrfach parallel auszuführen (Multithreading). Aus diesem Grund ist es auch problemlos möglich, Worker zur Laufzeit beliebig zu beenden und neue zu erzeugen, ohne dabei andere Prozesse negativ zu beeinflussen. „Child“-Prozesse können sich außerdem Server Ports teilen („gemeinsam nutzen“), via IPC (Interprocess Communication) mit ihren „Parent“-Prozessen kommunizieren und beispielsweise bei Serveranwendungen Server Handles untereinander hin- und herschicken. Dieses Modul ist deshalb sehr interessant für die Entwicklung von Node.js Applikationen, da Node.js kein Multithreading unterstützt und nur auf einem Single Thread läuft. Moderne CPUs besitzen üblicherweise aber mehrere Rechenkerne und unter Umständen sogar noch weitere virtuelle Kerne in Form von Threads. Mit dem *cluster* Package können Software-Developer diese technischen Möglichkeiten besser nutzen und erheblich performantere Applikationen entwickeln.

- Modul *os*[13]:
  Auch das *os* Package ist ein built-in Modul und bietet unterschiedlichste Funktionen an, um während der Ausführung einer Node.js Applikation auf Informationen über das Betriebssystem des Computers zuzugreifen. So gibt es beispielsweise Funktionen, um herauszufinden, welche Anzahl an CPUs gerade zur Verfügung steht oder um welche CPU Architektur es sich bei dem aktuellen Computer handelt. Weitere Beispiele für Informationen, die über Methoden dieses Moduls abrufbar sind, sind der Hostname, der Pfad des Home Directory, das Network Interface oder mit welchem Betriebssystem die Applikation ausgeführt wird. Solche Informationen können unter Umständen für eine fehlerfreie und möglichst performante Ausführung einer Softwareanwendung wesentlich sein.

- Modul *xml2js*[14]:
  Bei dem Module *xml2js* handelt es sich um ein externes, beziehungsweise öffentliches Package. Wie der Name bereits vermuten lässt, enthält es eine Methode, um Daten im XML-Format in JavaScript-Objekte zu konvertieren. *Xml2js* unterstützt aber auch eine bi-direktionale Umwandlung, das heißt, es ist nicht nur das Parsen von XML zu JavaScript möglich, sondern auch die entgegengesetzte Richtung von JavaScript zu XML.

[12] https://nodejs.org/api/cluster.html
[13] https://nodejs.org/api/os.html
[14] https://www.npmjs.com/package/xml2js

```
var parser = new xml2js.Parser();

var xml = '<testTag> any content </testTag>'

parseString(xml, {trim: true}, function (err, result) {
        // result = {testTag: “any content”}
});

var obj = {name: "Max", Surname: "Mustermann", age: 33};

var builder = new xml2js.Builder();
var xml = builder.buildObject(obj);

//xml = <name> Max </name> <surname> Mustermann </surname> <age> 33
</age>
```

Listing 6: Implementation sample of the "xml2js" Package

- Modul *utf8*[15]:
  Für die Übertragung von Daten wird einer der am weitesten verbreiteten Zeichen-Kondierungs-Standards namens UTF-8 (Universal Character Set Transformation Format) verwendet. Mit dem Modul ‚utf8‘ ist es möglich, Daten UTF-8-Encoding, oder -Decoding zu betreiben.

```
var utf8 = require('utf8');

utf8.encode('\xA9');

utf8.decode('\xC2\xA9');
```

Listing 7: Implementation sample of the "utf8" Package

- Modul *lower-case*[16]:
  Dieses Package enthält eine Methode, um alle in einem String enthaltenen Buchstaben in Kleinbuchstaben umzuwandeln. Wird der „lowerCase“ Funktion entweder ein leerer String, der Wert „null“ oder „undefined“ übergeben, wird ein leerer String retourniert.

[15] https://www.npmjs.com/package/utf8
[16] https://www.npmjs.com/package/lower-case

```
var lowercase = require('lower-case');

var tmpString = lowerCase(null);                // tmpString = ""
var tmpString = lowerCase('STRING');            // tmpString =  "string"
var tmpString = lowerCase('STRING', 'tr');      // tmpString =  "string"
```

Listing 8: Implementation sample of the "lower-case" Package

- Modul *string*[17]:
  Im Gegensatz zu vielen anderen populären JavaScript Frameworks besitzt das Node.js Framework keine Methoden zur Handhabung oder Manipulation des Datentyps String. Das über den NPM erhältliche Modul *string* ist eine JavaScript Library, die genau für diesen Zweck entwickelt wurde. Mit dem Modul *string* werden alle nativen JavaScript Methoden für den Typ String importiert, wobei an Stelle eines nativen JavaScript Strings ein Objekt von den Methoden des Moduls *string* zurückgegeben wird. Das bedeutet eine leicht veränderte Handhabung beim Funktionsaufruf, wie in den folgendem Implementierungsbeispielen zu sehen ist (*Listing 9*, *Listing 10*).

  Im Rahmen dieser Arbeit wurden vor allem die Methoden trim() und strip() dieses Moduls verwendet:

  - Die Funktion trim() entfernt alle Leerzeichen am Anfang und am Ende eines Strings.

```
var S = require('string');

S('hello ').trim().s;           //'hello'
S(' hello ').trim().s;          //'hello'
S('\nhello').trim().s;          //'hello'
S('\nhello\r\n').trim().s;      //'hello'
S('\thello\t').trim().s;        //'hello'
```

Listing 9: Implementation sample of the customized trim-function of the "string" Package

  - Die Funktion strip(string1, string2, …) entfernt alle Zeichen, beziehungsweise Zeichenketten die im übergebenen String vorkommen.

```
var S = require('string');

S('1 2-3_4 5_6-7__89--0').strip(' ', '_', '-').s;       //'1234567890'
S('Hello big round world!').strip('big', 'round').s;    //'Hello world!'
```

Listing 10: Implementation sample of the customized strip-function of the "string" Package

[17] https://www.npmjs.com/package/string

Implementierung

Beim der Implementierung der in JavaScript geschriebenen Kernkomponente des Treibers wurde die Software hinsichtlich ihrer Funktionen in mehrere Teile und Files unterteilt. Jeder dieser Teile wird als Modul bezeichnet, wobei im Rahmen dieser Arbeit jedes Modul auch in einem eigenen (gleichnamigen) File gespeichert wurde. Dieses Vorgehen soll der Übersichtlichkeit und Austauschbarkeit dienen. Wie in der nachfolgenden *Abbildung 19* zu sehen ist, gliedert sich die Node.js-Applikation (Kernkomponente) in die folgenden Module:

- lui_driver_cluster.js
- http_server.js
- util_proc.js
- util_mapping.js
- util_modules.js
- util_string.js

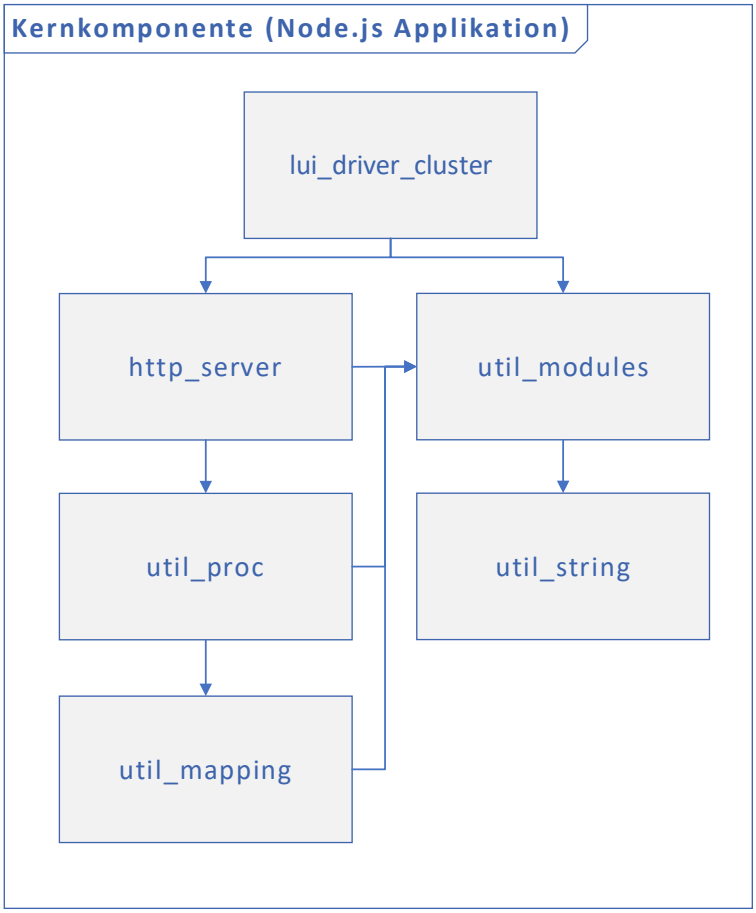


Abbildung 19: Übersicht der Module der Kernkomponente (Node.js Applikation). Die Anordnung der Module repräsentiert von oben nach unten den hierarchischen Aufbau innerhalb dieser Treiberkomponente. Ein Pfeil in Richtung eines anderen Moduls steht für die Zugriffsrechte auf diesen Teil der Software.

Allgemeiner Programmablauf dieser Komponente

Da der Programmablauf mit asynchron eintreffenden Nachrichten und Events eine gewisse Komplexität darstellt, wird der Prozess nachfolgend anhand von zwei Flussdiagrammen visualisiert. Im ersten Diagramm (siehe *Abbildung 20*) sind alle wesentlichen Vorgänge in der Kernkomponente abgebildet, die vom Start der Applikation, über das Empfangen von Nachrichten vom LUI, bis hin zum Senden von Nachrichten an den WebSocket-Client durchlaufen werden. In dem zweiten Diagramm (siehe *Abbildung 21*) ist zu sehen, welche Abläufe in Gang gesetzt werden, wenn der WebSocket-Server in der Kernkomponente vom WebSocket-Client der Spracherkennungskomponente eine Nachricht empfängt.

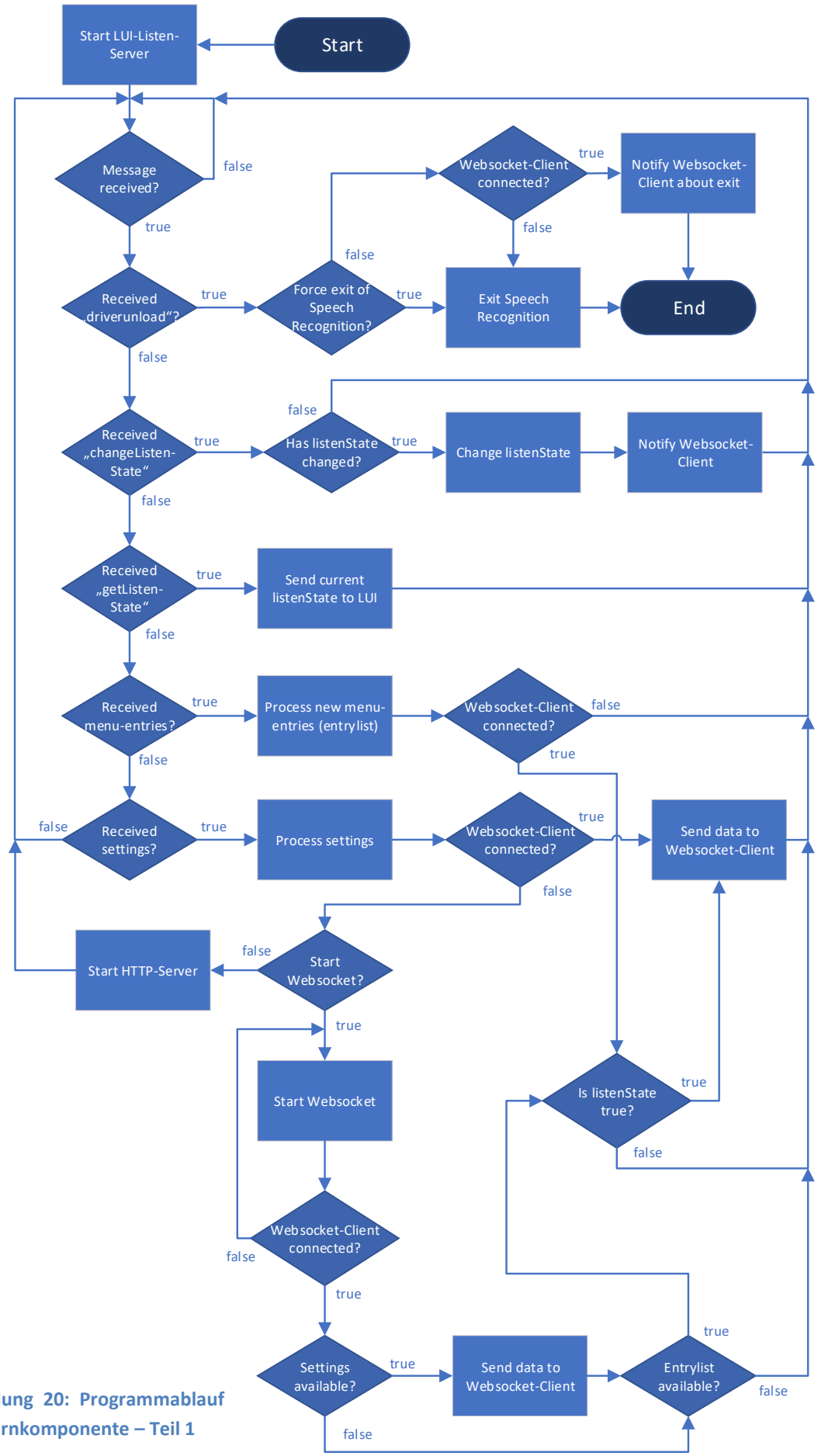


Abbildung 20: Programmablauf der Kernkomponente – Teil 1

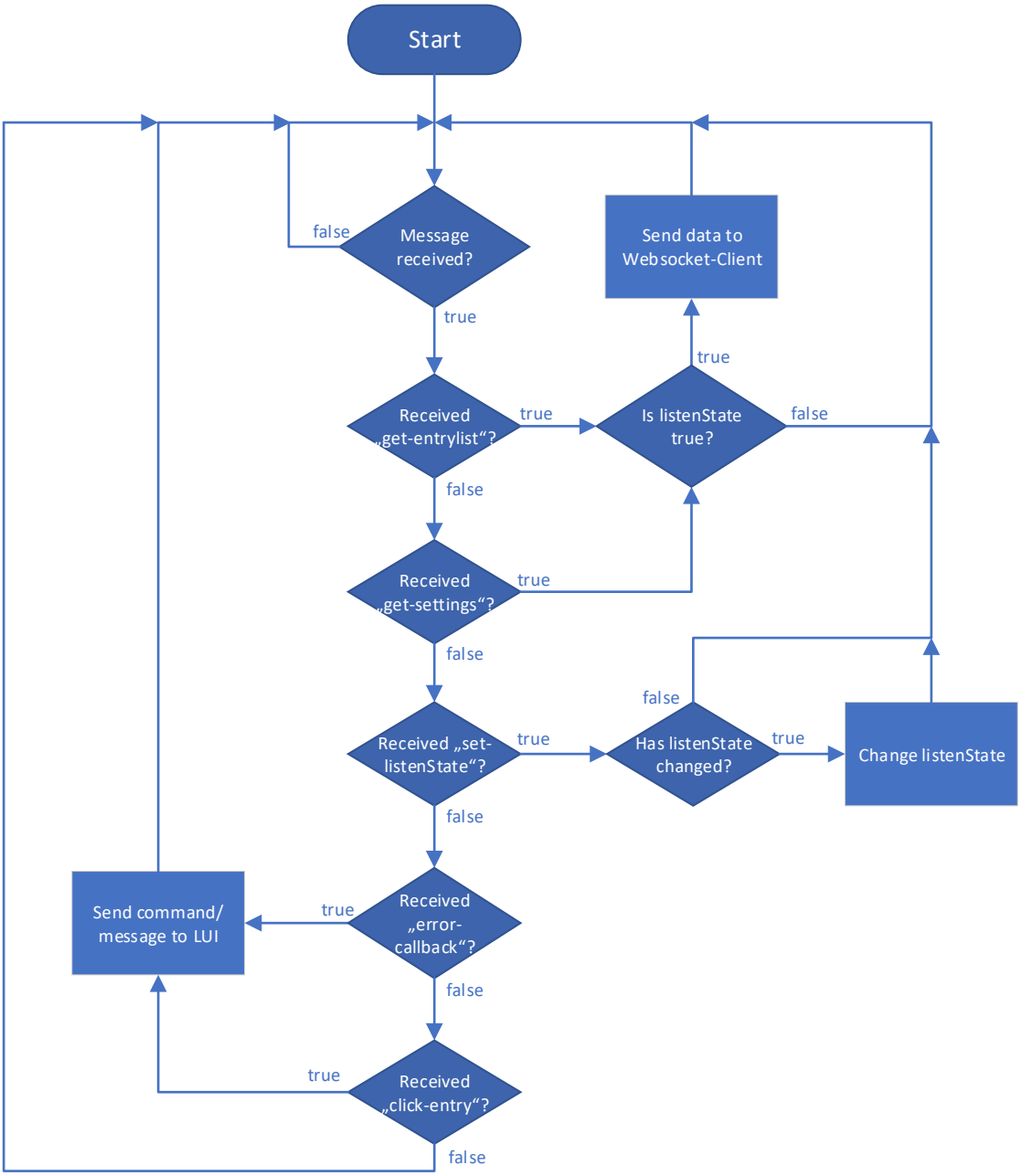


Abbildung 21: Programmablauf der Kernkomponente – Teil 2

Für den Start dieser Komponente wird durch das vom LUI aufgerufene Batch-File das Modul *lui_driver_cluster* ausgeführt (siehe *5.5.2. Treiber Lifecycle)*. Dieses Script startet auch das Debug- und Error-Logging, welche in Text-Files im Root-Ordner der Applikation gespeichert werden (siehe Kapitel *5.6. Logging*). Anschließend wird in einem Sub-Prozess die eigentliche Kommunikations- und Verarbeitungslogik dieser Komponente aufgerufen. Dabei wird zuerst im Modul *http_server* der Webserver gestartet, mit dem Nachrichten vom LUI empfangen werden können. Falls noch keine Settings oder Menu-Entries empfangen wurden, wird in weiterer Folge über einen zweiten Webserver der jeweilige Request an das LUI geschickt.
Treffen Settings vom LUI ein, werden diese in ein Objekt gespeichert und es wird damit die Schnittstelle zur Spracherkennungssoftware gestartet. Das Konzept dieser Arbeit sieht hier-

bei vor, dass dafür ein WebSocket-Server eingesetzt wird. Wenn erfolgreich eine Verbindung zu einem WebSocket-Client hergestellt wird, wird ein Event getriggert, das dafür sorgt, dass die Settings an diesen Client gesendet werden.
Ist in der Zwischenzeit ebenso eine Entrylist (Menu-Entries) vom LUI an die Komponente gesandt worden, wird diese auch an die Applikation zur Spracherkennung übermittelt, sofern der treiberinterne Status „listenState“ gleich „true“ ist. Der Status Listenstate kennzeichnet, ob sich die Spracherkennung im Zustand „aktiv“ oder „inaktiv“ befindet. Dieser Zustand kann abhängig von der Treiberkonfiguration über Sprachkommandos und/oder durch eine Nachricht vom LUI verändert werden.
Bei der Entrylist handelt es sich um ein Objekt, dessen Properties als Key den Namen des jeweiligen Menu-Entries und als Value deren Kommando-String (ID der LUI-Entries) enthalten. Die Zerlegung der XML-Attribute und das Erstellen des Entrylist-Objekts erfolgt im Modul *util_proc*. Nach jeder empfangenen und verarbeiteten Entrylist wird außerdem ein Event getriggert, welches bei bestehender WebSocket-Connection die neuen Menu-Entries an die Spracherkennungssoftware übermittelt.
Geht im WebSocket-Server eine Nachricht vom Client ein, gibt es eine Reihe an vordefinierten Requests, die akzeptiert werden. So können die Treiber-Settings, die aktuelle Entrylist oder der aktuelle Listenstate nachgefragt werden. Der WebSocket-Server akzeptiert aber auch Texte (zum Beispiel Hinweise oder Warnungen), die in weiterer Folge in der Benutzeroberfläche des LUI angezeigt werden sollen. Bei entsprechender Konfiguration des Treibers ist es außerdem möglich, dass der WebSocket-Client Error-Messages an die Kernkomponente übermittelt.
Die Applikation zur Spracherkennung kann aber auch ein erkanntes Kommando an die Kernkomponente übermitteln. Hierbei wird zwischen einem Sprachkommando zum Verändern des Listenstates und Kommandos für das LUI unterschieden. Ein Kommando für das LUI wird über die Webserver-Instanz, zum Versenden von Nachrichten, an das LUI übermittelt. Auch hier erfolgt die vorherige Überprüfung auf den aktuellen Listenstate der Spracherkennung und eine Weiterleitung des Kommandos erfolgt nur bei aktiver Spracherkennung. Die Logik für die Aktivierung und Deaktivierung der Spracherkennung ist in der Kernkomponente implementiert, da die Steuerung zentral und möglichst nah an der Schnittstelle zum LUI erfolgen soll. Außerdem muss dadurch diese Logik nicht für jede Applikation zur Spracherkennung neu umgesetzt werden. Der Listenstate wird über ein internes Event der Kernkomponente gesteuert, welches unabhängig von der Quelle des Aufrufs immer gleich ausgelöst wird. Beim Start des Treibers ist die Spracherkennung automatisch aktiv („listenState = true“).

Wird beispielsweise zu Test- oder Entwicklungszwecken als Interface zur Applikation für die Spracherkennung die REST-Schnittstelle ausgewählt, wird anstatt des WebSocket-Servers ein weiterer HTTP-Server gestartet. In diesem Fall werden nicht explizit Settings oder Menu-Entries versandt, sondern müssen bei Bedarf vom REST-Client nachgefragt werden („get-entrylist“ und „get-settings“). Trifft eine Message mit einer erkannten Spracheingabe ein, wird diese, abhängig von der Konfiguration und dem Listenstate, direkt an das LUI weitergeleitet, oder zuerst validiert. Soll der Input validiert werden, werden die Daten der Spracheingabe an das Modul *util_proc* übergeben, welches gemeinsam mit dem Modul *util_mapping* ermittelt, ob ein gültiges Kommando enthalten ist. Wird ein Kommando gefunden, wird es an das LUI weitergeleitet, indem es an die Webserver-Instanz für das Senden von Nachrichten an das LUI übergeben wird.

Module der Komponente im Detail

- Modul *lui_driver_cluster.js*:
  Das JavaScript-File *lui_driver_cluster* beinhaltet die „Main-Funktion“ dieser Softwarekomponente und es handelt sich dabei auch um jenes Skript, welches beim Programmstart als erstes aufgerufen wird. In diesem File wird ein Clustering-Mechanismus des, über den NPM (Node Package Manager) bezogenen, Third-Party-Moduls (Packages) „cluster“ eingesetzt. Mit dem Modul „cluster“ wird zu Beginn des Skripts ein Master-Cluster erzeugt, in welchem in weiterer Folge das gesamte Programm abläuft und welches das Logging startet (siehe Kapitel *5.6. Logging*).
  Die eigentliche Hauptaufgabe des Master-Clusters ist aber ein Error-Handling zu ermöglichen und in einem bestimmten Maß für Ausfallsicherheit der Komponente zu sorgen. Das funktioniert über das Management von sogenannten Worker-Clustern. Bei einem Worker-Cluster wird ein Unterprozess erzeugt, in dem jeder beliebige Code ausgeführt werden kann. Diese Worker-Cluster können bewusst beendet werden oder beenden sich beim Eintreten eines Errors beziehungsweise Fehlerfalls von selber. In beiden Fällen wird im Master-Cluster das Event „exit“ ausgelöst. Anhand des Fehler-Codes, der dem Event „exit als Parameter mitgegeben wird, wird entweder sofort ein neuer Worker-Cluster erzeugt oder die gesamte Applikation explizit beendet. Das bewusste Beenden der Applikation wird durch einen selbst erzeugten Fehler-Code (es wurde der Code „123“ gewählt) in Folge des Beendens des LUI eingeleitet.

  Nach dem Initialisieren des Master-Clusters wird umgehend ein Worker-Cluster abgezweigt. In jedem neu erzeugten Worker-Cluster – unabhängig davon, ob er direkt nach dem Programmstart oder nach dem Beenden eines anderen Worker-Cluster erzeugt wurde – wird nach der Cluster-Initialisierung die Methode „runDriver“ des Modules *http_server* aufgerufen. Diese Methode wrappt („beinhaltet“) den Kern, beziehungsweise den Hauptteil der Logik dieser Komponente, womit die eigentliche Aufgabe der Kernkomponente des Treibers im Subprozess Worker-Cluster ausgeführt wird. Auf die Details und den exakten Inhalt der Funktion „runDriver“ wird im Abschnitt des Moduls *http_server* eingegangen.

  Der Vorteil dieser Herangehensweise ist, dass der Treiber nicht erneut vom LUI gestartet werden muss, denn das würde auch einen Neustart des LUIs erfordern. Außerdem geht dieser Vorgang in der Regel so schnell vonstatten, dass es der Anwenderin oder dem Anwender nicht auffallen sollte, dass der Treiber kurzzeitig nicht zur Verfügung stand.
  Die Nutzung des Clusterings in diesem Modul, beziehungsweise dieser Komponente, entspricht allerdings nicht ganz dem eigentlichen Konzept, das hinter dem Third-Party-Modul „cluster“ steckt. Laut der Dokumentation[18] von „cluster“ war die eigentliche Idee das Node.js Framework, welches nur Single Threading unterstützt, um die Möglichkeit zu erweitern, mehrere Threads erzeugen zu können. Somit könnten die Fähigkeiten, beziehungsweise Ressourcen von modernen Computern, die meist schon mehrere Prozessorkerne besitzen, besser genutzt werden. Das bedeutet, dass mit dem Clustering ermöglicht werden sollte, anhand von unterschiedlichen Worker-Clustern mehrere Prozesse parallel ablaufen zu lassen. In dieser Implementierung hingegen, werden die Worker-Cluster ausschließlich sequenziell ausgeführt und nur dafür genutzt, dass bei

[18] https://nodejs.org/api/cluster.html

möglichen Fehlern der Hauptprozess nicht gestoppt wird.

- Modul *http_server.js:*
  Im Modul *http_server* wird die Fähigkeit von Modulen genutzt, eigene Attribute exportieren zu können und sie somit für andere Module verfügbar zu machen. Von *http_server* wird das Attribut Namens „runDriver“ exportiert, welches als Funktion definiert ist. Diese Funktion fungiert als Wrapper des gesamten Codes, der sich in diesem Modul befindet. Das bedeutet alle Funktionen und Variablen vom Script „http-server“ existieren im Scope der Funktion „runDriver()“. Wird „runDriver()“ aufgerufen, werden demnach mit einem einzigen Funktionsaufruf von außen alle Funktionen von *http_server* ausgeführt. Außer auf „runDriver()“ kann auf die Funktionen und Variablen von *http_server* allerdings von außen nicht direkt zugegriffen oder ein Wert verändert werden.

  Im Folgenden ist zuerst eine Übersicht über wichtige Funktionen und Events des Moduls zu sehen, bevor anschließend auf die Abläufe in der Methode „runDriver()“ eingegangen wird.

  Mit Hilfe dieser Events soll mit asynchron ablaufenden Prozessen und asynchron eintreffenden Daten umgegangen werden können, das bedeutet, dass asynchrone Systemzustände (unterschiedliche Zustände in den einzelnen Systemkomponenten) vermieden werden sollen.

  - Events:
    - clientConnection
      Das Event „clientConnection“ wird vom WebSocket ausgelöst, wenn ein erfolgreicher Handshake mit einem Client zustande gekommen ist. Beim Auslösen dieses Events wird die Funktion „setClientConnected“ aufgerufen.
    - newProcessedEntryList
      Das Event „newProcessedEntryList“ wird nach dem Empfangen und Verarbeiten einer neuen Entrylist ausgelöst. Beim Auslösen dieses Events wird die Funktion „setNewProcessedEntryList“ aufgerufen.
    - startSrServer
      Das Event „startSrServer“ wird von dem HTTP-Server in der Funktion „luiServer“ ausgelöst. Beim Auslösen dieses Events wird die gleichnamige Funktion „startSrServer“ aufgerufen.
    - setListenState
      Das Event „setListenState“ wird aufgerufen, wenn ein Kommando zum Verändern des aktuellen Listenstates vom LUI oder dem WebSocket-Client empfangen wird. Der Listenstate gibt an, ob die Spracherkennung momentan aktiv („listenState = true“) oder inaktiv („listenState = false“) ist. Beim Auslösen dieses Events wird die Funktion „triggerListenState“ aufgerufen.

    Im Folgenden ist ein Implementierungsbeispiel anhand des Events „clientConnection“ zu sehen:

```
eventEmitter.on('newProcessedEntryList', setNewProcessedEntryList);
```

Listing 11: Implementation sample of an event in Node.js

- Functions:
  - „setClientConnected()“
    Diese von dem Event „clientConnection“ getriggerte Funktion überprüft, ob bereits Settings vom LUI empfangen wurden, sowie ob eine aktuelle Entrylist (Liste von Menu-Entries) zur Verfügung steht. Diese beiden Überprüfungen sind unabhängig voneinander und falls die jeweilige Bedingung zutrifft, werden entsprechend die Daten Settings oder Entrylist als Parameter an die Funktion „wsSend()“ des WebSockets übergeben.
  - „setNewProcessedEntryList()“
    Diese von dem Event „newProcessedEntryList“ getriggerte Funktion überprüft, ob eine Connection zu einem WebSocket-Client besteht. Ist das der Fall, wird die aktuelle Entrylist als Parameter an die Funktion „checkSpeechListenState“ übergeben. Diese retourniert – abhängig vom Listenstate und der Konfiguration von Sprachkommandos für die Aktivierung und Deaktivierung der Spracherkennung – eine Entrylist, die an die WebSocket-Funktion „wsSend()“ übergeben wird. Des Weiteren erfolgt eine Überprüfung, ob bereits Settings vom LUI erhalten wurden, ist das nicht der Fall, wird die Methode „getSettings()“ aufgerufen.
  - „checkSpeechListenState(list)“
    „checkSpeechListenState“ wird von der Funktion „newProcessedEntryList“ aufgerufen und erhält als Übergabeparameter („list“) die aktuelle Entrylist. Anhand des Listenstates wird überprüft, ob die Spracherkennung aktiv oder inaktiv ist. An dieser Stelle wird unterschieden, ob eine Sprachsteuerung (*das heißt eine Beeinflussung des Listenstates über Sprachkommanods*) für die Aktivierung und Deaktivierung der Spracherkennung konfiguriert wurde. Wurden keine Sprachkommandos für diesen Anwendungsfall konfiguriert, wird bei aktiver Spracherkennung ausschließlich die aktuelle Entrylist an den Aufrufer zurückgegeben und bei inaktiver Spracherkennung eine leere Liste retourniert.
    Soll über Sprachkommandos der Listenstate geändert werden können, werden bei aktiver Spracherkennung die aktuelle Entrylist um ein Kommando zur Deaktivierung ergänzt und zurückgeliefert. Bei inaktiver Spracherkennung wird ausschließlich ein Kommando zur Aktivierung zurückgegeben.
  - „errorCallback(type, message)“
    An diese Funktion können im Fehlerfall sowohl der „Fehlertyp“, als auch eine Fehlernachricht übergeben werden. Wurde in der Treiberkonfiguration eine Callback-Funktion definiert, die Fehler an das LUI weiterleiten soll, werden die Daten der Übergabeparameter („type“ und „message“) in ein Objekt gespeichert und an die Funktion „clientResponse“ übergeben, welche sie dann an das LUI übermittelt.
  - „triggerListenState(param)“
    Der dieser Funktion übergebene Parameter „param“ beinhaltet den Zustand zu dem der Listenstate geändert werden soll. Entspricht der Listenstate bereits dem Zustand des Übergabeparameters erfolgt keine weitere Aktion. Andernfalls wird der Listenstate entsprechend „param“ geändert. Wird die Spracherkennung von inaktiv („listenState = false“) auf aktiv („listenState = true“) geändert, wird – falls eine Verbindung zum WebSocket-Client besteht – sowohl der neue Listenstate, als auch die aktuelle Entrylist an die Sprach-

erkennung geschickt. Wird die Spracherkennung auf inaktiv gesetzt, erfolgt bei bestehender Websocket-Verbindung ausschließlich die Benachrichtigung über den neuen Zustand.
Falls konfiguriert wurde, dass eine Aktivierung und Deaktivierung der Spracherkennung auch mittels Sprachkommandos möglich sein soll, werden außerdem jeweils das Kommando für die Aktivierung oder Deaktivierung der Spracherkennung an den Websocket-Client übermittelt (das erfolgt über den Aufruf der Funktion „setNewProcessedEntryList()“).

- „startSrServer()“
  Diese von dem Event „startSrServer“ getriggerte Funktion überprüft, welcher Connection-Type in den von dem LUI übergebenen Settings gesetzt wurde. Handelt es sich um den Connection-Type „WebSocket“, wird die Funktion „websocketServer()“ aufgerufen, andernfalls wird die Funktion „srServer()“ ausgeführt.
- „luiServer()“
  In dieser von „startLuiListenServer“ aufgerufenen Funktion wird der HTTP-Server mit der statischen URI http://127.0.0.1:12345 gestartet. Dieser Server empfängt alle vom LUI versendeten Nachrichten und enthält die Logik für die unterschiedlichen Typen von Nachrichten. Relevant sind dabei die Typen „driverunload“, „newmenu“ und „curmenu“, welche im XML-Tag „message“ übergeben werden, sowie der im Config-File definierte Name des Treibers, der im XML-Tag „driver“ übergeben wird.
  Beendet sich das LUI, sendet es „driverunload“, was bewirkt, dass an dieser Stelle der Kernkomponente, abhängig von dem Setting „forceSrExit“ aus dem Config-File, eine Exit-Message an die Applikation zur Spracherkennung gesendet oder direkt ein Windows-Shell-Command ausgeführt wird, um die Applikation zu beenden. Anschließend erfolgt das Beenden der Kernkomponente selbst.
  Die Message-Typen „newmenu“ und „curmenu“ haben zur Folge, dass der Daten-Inhalt an die Methode „processEntryList()“ des Moduls „utilProc“ übergeben werden. Wird eine gültige Entrylist zurückgeliefert, wird das Event „newProcessedEntryList“ ausgelöst.
  Wird das erste Mal eine Nachricht mit dem XML-Tag „driver“ und dem statischen Namen „speechRec“ empfangen, werden die Settings entsprechend den, in den „relevantSettings“ des Config-Files definierten Tag-Namen, verarbeitet. Im Anschluss wird das Event „startSrServer“ ausgelöst.
- „startLuiListenServer()“
  Diese Funktion wird durch den Aufruf der Wrapper-Funktion „runDriver()“ getriggert. Sie enthält einen try-catch-Block, bei dem im try-Teil die Funktion „luiServer“ aufgerufen wird. Tritt ein Fehler auf, während die Funktion „luiServer()“ ausgeführt wird, wird dieser im catch-Teil mit dem übergebenen Error geloggt.
- „getSettings()“
  Diese Funktion wird ebenfalls durch den Aufruf der Wrapper-Funktion „runDriver()“ getriggert. Ihre Ausführung erfolgt direkt nach der Funktion „startLuiListenServer()“. Falls noch keine Settings empfangen wurden, erfolgt über den Aufruf der Funktion „clientResponse()“ ein Request für die Settings des LUI.

- „getCurrentMenu()“
  Diese Funktion wird durch den Aufruf der Wrapper-Funktion „runDriver()“ getriggert. Ihre Ausführung erfolgt direkt nach der Funktion „startLuiListenServer()“ und „getSettings()“. Falls noch keine Entrylist existiert, beziehungsweise keine Menu-Entries requested wurden, erfolgt über den Aufruf der Funktion „clientResponse()“ ein Request der aktuellen Menu-Entries des LUI.
- „srServer()“
  Diese Funktion wird von der Methode „startSrServer()“ aufgerufen. Sie startet den HTTP-Server „speechRecServer“, der als REST-Schnittstelle optional zum WebSocket als Interface für Applikationen zur Spracherkennung dienen kann.
- „websocketServer()“
  Diese Funktion wird von der Methode „startSrServer()“ aufgerufen. Sie startet und beinhaltet den WebSocket-Server, was auch die Verarbeitung von Nachrichten vom WebSocket-Client einschließt. Nach dem erfolgreichen WebSocket-Handshake wird das Event „clientConnection“ ausgelöst. Client-Messages werden abhängig von ihrem Content wie folgt behandelt[19]:
  - Message-Content „get entrylist“: Ruft die Funktion „wsSend()“ auf, mit der aktuellen Entrylist im JSON-Format als Übergabeparameter.
  - Message-Content „get settings“: Ruft die Funktion „wsSend()“ auf, mit den Settings im JSON-Format als Übergabeparameter.
  - Sonstiger Messages-Content: Es wird davon ausgegangen, dass es sich um ein erkanntes Kommando handelt und es wird die Funktion „clientResponse()“ mit dem Message-Content als Übergabeparameter aufgerufen.

  Die Funktion „wsSend()“ ist ebenfalls Teil des WebSocket-Servers. Sie kann von jeder Stelle in diesem Modul aufgerufen werden und es können ihr die beiden Parameter „content“ und „type“ übergeben werden. „Content“ ist mandatory, während „type“ optional ist und nur in das Log-File geschrieben wird. Die Funktion übermittelt den übergebenen Content an den WebSocket-Client.
- „clientResponse()“
  Diese Funktion kann von jeder Stelle in diesem Modul aufgerufen werden und beinhaltet einen HTTP-Server, um Nachrichten an das LUI zu senden. Die Funktion muss mit den zwei Parametern „action“ und „str“ aufgerufen werden. Der Parameter „action“ ist mandatory und enthält die Aktion, die im LUI ausgeführt werden soll. Es gibt vier unterschiedliche Aktionen:
  - „clickentry“: Auswählen/klicken eines Buttons auf der LUI-Seite
  - „displayMessage“: Anzeigen einer Overlay-Nachricht im LUI
  - „broadcastcurrentmenu“: Request der aktuellen Menu-Entries
  - „getdrvconfig“: Request der LUI-Settings
  - „errorcallback“: Übermitteln von Fehlermeldungen an das LUI
  - „getListenState“: Übermitteln des aktuellen Listenstate an das LUI

  Der Parameter „str“ enthält den Content, der übermittelt werden soll. Bei „action“ muss angegeben werden, um welche Aktion (Art von Nachricht) es sich

[19] Implementierungsdetails sind der in diesem Abschnitt in weiterer Folge enthaltenen Interface-Definition zu entnehmen.

handelt. Gemäß der Interface Definitions wird zwischen „clickentry“, „displaymessage“, „broadcastcurrentmenu“, „getdrvconfig“, „errorcallback“ und „getlistenstate“ unterschieden.

Nach der detaillierten Beschreibung der Events und Methoden dieses Moduls folgt nun eine Beschreibung des Programmablaufs innerhalb von *http_server* (siehe *Abbildung 20*).
Durch den, von einem neuen Work-Cluster des Moduls „lui_driver_cluster“ getriggerten Aufruf der Methode „runDriver()“, erfolgt eine Initialisierung und Definition der Variablen und Funktionen, welche sich innerhalb dieser Methode befinden. Im Zuge dessen wird beispielsweise auch der Listenstate auf seinen initialen Wert „true“ gesetzt. Im Anschluss an den Initialisierungsvorgang wird die Funktion „startLuiListenServer()“ ausgeführt. Diese Funktion dient ausschließlich dem Aufruf der Funktion „luiServer()“, die den HTTP-Server beinhaltet, der Nachrichten vom LUI empfängt. Die weitere Ausführung des Scripts dieses Moduls führt die Funktionen „getSettings()“ und „getCurrentMenu()“ aus. Diese schicken über die Funktion „clientRequest()“ einen Request für Settings oder die aktuellen Menu-Entries an das LUI, falls diese noch nicht empfangen wurden. Die Methode „clientRequest()“ führt einen HTTP-Server aus, der für das Senden von Nachrichten an das LUI genutzt wird.
Werden über die REST-Schnittstelle des HTTP-Servers von „luiServer()“ Daten empfangen, erfolgt eine Überprüfung auf deren Inhalt. Wie in der vorangegangen Beschreibung der Funktion erläutert wurde, wird zwischen den Nachrichten-Attributen „driverunload“, „curmenu“, „newmenu“, „changelistenstate“, „getlistenstate“ und dem selbst definierten Identifikator des Treibers – „speechRec“ – unterschieden, um die jeweilige weitere Verarbeitung der Daten einzuleiten. Durch die asynchrone Kommunikation mit dem LUI kann es nun zu unterschiedlichen Szenarien, beziehungsweise unterschiedlicher Reihenfolge der weiteren Abläufe kommen. Treffen zuerst neue Menu-Entries ein, werden diese an die Methode „processEntryList()“ des Moduls „utilProc“ übergeben. Die Details dieser Verarbeitung werden im folgenden Abschnitt in der Dokumentation des Moduls *util_proc* erläutert. Wird ein Objekt mit einer gültigen Entrylist zurückgeliefert, wird das Event „newProcessedEntryList“ ausgelöst, dessen Funktion „setNewProcessedEntryList()“ bei aktivierter Spracherkennung („listenState = true“) und einer bestehenden Verbindung zu einem WebSocket-Client die Entrylist an die Applikation zur Spracherkennung schickt. Treffen das erste Mal Settings ein, erfolgt eine Verarbeitung dieser Settings und es wird das Event „startSrServer“ ausgelöst, welches den Server für die, im Config-File des LUI ausgewählte, Schnittstelle startet.
In der aktuellen Implementierung und mit der in dieser Arbeit verwendeten Applikation zur Spracherkennung ist es vorgesehen, einen WebSocket als Schnittstelle zu verwenden, und es wird somit über die Funktion „startSrServer()“ ein WebSocket gestartet. Geht nun bei diesem WebSocket-Server ein Connection-Request durch einen WebSocket-Client ein, wird er bei dieser Konfiguration automatisch akzeptiert, was bedeutet, dass ein Handshake und somit ein HTTP Upgrade durchgeführt wird. Ab diesem Zeitpunkt ist die WebSocket-Verbindung offen und es könnten Nachrichten empfangen und gesendet werden. Bevor aber tatsächlich Nachrichten empfangen werden, wird durch den erfolgreichen Verbindungsaufbau noch das Event „clientConnection“ getriggert. Die von dem Event ausgelöste Funktion „setClientConnected“ sendet, falls vorhanden, die aktuellen Settings und Entrylist (unter Berücksichtigung des aktuellen Listenstates) mittels der WebSocket-Funktion „wsSend()“ an den WebSocket-Client. Die WebSocket-

Funktion „wsSend()“ dient ausschließlich dazu, den ihr übergebenen Content an den WebSocket-Client zu übermitteln.
Nachrichten vom WebSocket-Client werden unter den, in den bereits im Abschnitt *5.3.2. Interface Definitions* für das Empfangen von Daten festgelegten Kriterien, akzeptiert. Bei den empfangenen Nachrichten wird zwischen dem Content „get-entrylist“, „get-settings“, „click-entry“, „display-message“ und „error-callback“ unterschieden. Wird „get-entrylist“ oder „get-settings“ empfangen, werden die aktuelle Entrylist oder Settings an den WebSocket-Client gesandt. Ist die Spracherkennung zu diesem Zeitpunkt deaktiviert, wird die Entrylist allerdings nicht an den WebSocket-Client übermittelt. Wird „display-message“ oder „error-callback“ vom WebSocket-Server empfangen, wird der Inhalt dieser Nachrichten für eine Benachrichtigung in der Benutzeroberfläche, beziehungsweise als Fehlermeldung an das LUI weitergeleitet. Eine Nachricht deren Object-Key „click-entry“ ist, wird als erkanntes Sprachkommando interpretiert. In Folge dessen wird davon ausgegangen, dass der Object-Value das Sprachkommando beinhaltet. Da für Kommandos, die über den WebSocket empfangen werden, der Workflow so definiert ist, dass die Validierung der Spracheingabe in der Applikation zur Spracheingabe erfolgt, wird in der Kernkomponente dahingehend keine Überprüfung mehr durchgeführt. Es wird also davon ausgegangen, dass es sich um ein gültiges Kommando handelt, wodurch bei entsprechendem Listenstate („true“) diese Eingabe sofort an die Funktion „clientRequest()“ übergeben wird.
Die Methode „clientRequest()“ enthält den HTTP-Server für das Senden von Nachrichten an das LUI. Dieser Methode können zwei Paramater übergeben werden, wobei der erste („action“) verpflichtend ist und der zweite („str“) abhängig von der angegeben „action“ optional ist. Handelt es sich um eine gültige Aktion, wird der jeweilige XML-String erzeugt und die Nachricht an das LUI geschickt.

Wie einleitend bei der Dokumentation der Kernkomponente bereits erläutert, existiert neben dem WebSocket-Server auch noch ein Webserver, der ebenfalls als Schnittstellen zur Applikation zur Spracherkennung dienen kann. Dieser, eigentlich nur mehr als Development-Schnittstelle gedachte HTTP-Server, kann bei den entsprechenden Einstellungen im Config-File beim Aufruf der Funktion „startSrServer()“ anstatt des WebSockets gestartet werden.
Diese Variante unterscheidet sich in ihrem Ablauf dahingehend von jener mit dem WebSocket, dass bei entsprechender Konfiguration (siehe *5.5.1. Treiberkonfiguration*) auch die komplette Logik für den Entscheidungsprozess – ob eine Spracheingabe ein gültiges Kommando enthält, oder nicht – von der Kernkomponente, beziehungsweise deren Modulen übernommen wird. Das heißt, dass neben der Verarbeitung und Aufbereitung der Nachrichten vom LUI, bei dieser Variante, auch eine Verarbeitung der von der Spracherkennungsapplikation empfangenen Nachrichten stattfindet.
Wie der Interface Definition des HTTP-Servers entnommen werden kann (siehe Kapitel *5.3.2. Interface Definitions*), erwartet der Webserver, dass ein im JSON-Format codierter String übermittelt wird. Dabei wird an dieser Stelle der Kernkomponente davon ausgegangen, dass von der Applikation zur Spracherkennung ein String mit der erkannten Spracheingabe übermittelt wird.
Diese empfangenen Daten werden an die Funktion „triggerInputProcessing()“ des Moduls „util-proc“ übergeben. In diesem Modul erfolgt die Verarbeitung der Input-Daten, beziehungsweise unter Zuhilfenahme des Moduls *util_mapping,* eine Prüfung auf gültige Sprachkommandos. Auf diese Module wird in der weiteren Folge dieses Kapitels noch detaillierter eingegangen.

Wenn ein gültiges Sprachkommando gefunden wird, wird dieses als Return-Value an jene Stelle in der Webserver-Logik zurück geliefert, an der der Funktionsaufruf stattgefunden hat. Das weitere Prozedere ist dann äquivalent zu der Variante mit einem WebSocket-Server und es wird der Funktion „clientResponse()“ das erkannte Kommando übergeben.

- Modul *util_modules.js:*
  Im JavaScript-File mit dem Namen *util_modules* werden alle Packages eingebunden, welche für diese Node.js-Komponente (Kernkomponente) des Treibers für die Spracherkennung verwendet werden. Die Vorgehensweise für die Einbindung der Packages ist im „exports“-Objekt des Moduls Attribute zu definieren, denen über das Keyword „require“ die gewünschten Packages zugewiesen werden (siehe *Listing 12*).
  Der Sinn, alle Packages in diesem Modul einzubinden und sie dann allen Modulen zur Verfügung zu stellen (genauer gesagt sie dadurch „public“ zu machen), liegt darin, nicht in jedem Modul, beziehungsweise JavaScript-File, die benötigten Packages einzeln einbinden zu müssen, sondern diese überall mit dem gleichen Aufruf verwenden zu können. Diese Vorgehensweise hat außerdem noch den Vorteil, dass Packages nur an einer Stelle ausgetauscht oder neu hinzugefügt werden können. Nachfolgend ist ein Implementierungsbeispiel für das Hinzufügen eines Packages zu diesem Modul zu sehen.

```
module.exports = {
        //load modules
        http: require('http')
};
```

Listing 12: Implementation sample of the export functionality of a Node.js module

- Modul *util_proc.js*:
  Wie bereits angeschnitten, hat das Modul *util_proc* mehrere Aufgaben. Es wird sowohl eine Verarbeitung neu eingetroffener Menu-Entries durchgeführt, als auch ein Vergleich zwischen einer getätigten Spracheingabe und den verfügbaren Kommandos gemacht, sofern die dementsprechende Konfiguration für diese Applikation gewählt wurde. Die drei wesentlichen Methoden, die von diesem Modul exportiert werden sind „getEntryList()“, „processEntryList()“ und „triggerInputProcessing()“.
  - Functions:
    - „processEntryList()“:
      Diese Funktion ist primär dafür gedacht, das als Übergabeparameter erhaltene Array aus Objekten von Sprachkommandos in eine Datenstruktur zu konvertieren, die für eine weitere Verarbeitung besser und leichter handhabbar ist. Dabei fiel die Wahl auf den Datentyp Object, wobei jede Property des Objekts einen Menu-Entry repräsentiert. Der Key einer Object-Property enthält dabei entweder den Text- oder den Spracheintrag eines Menu-Entries (Elements) des LUI und als Value die ID dieses Menu-Entries. Nachdem es zu einem Menu-Entry im LUI sowohl einen Text-Eintrag, als auch beliebig viele Speech-Einträge geben kann, kann es vorkommen, dass die ID eines Menu-Entries mehrmals als Value zu einem Key gespeichert wird. Das ist aber durchaus erwünscht, da somit anhand der Spracheingabe sofort die Menu-Entry-ID verfügbar ist. Diese wird wiederum benötigt, um die gewünschte Aktion im LUI ausführen zu lassen. Zu der Liste an gültigen

Sprachkommandos wird ein Entry aus der erhaltenen Liste an Menu-Entries allerdings nur hinzugefügt, wenn der Entry zumindest einen Text-Eintrag besitzt. Erst anschließend wird überprüft, ob es noch Speech-Einträge zu diesem Menu-Entry gibt.

- „triggerInputProcessing()“:
  Die Funktion „triggerInputProcessing()“ wird aufgerufen, nachdem der Webserver für Spracherkennungsapplikationen eine Spracheingabe erhält. Diese Spracheingabe wird als Parameter an diese Funktion übergeben, welche zuerst überprüft, ob eine Liste an gültigen Sprachkommandos zur Verfügung steht. Ist das der Fall, wird die Spracheingabe und die Liste der gültigen Sprachkommandos der Funktion „getCommand()“ des Moduls *util_mapping* übergeben. Diese Methode retourniert an ihren Aufrufer entweder die ID des Elements, zu dem das Kommando der Spracheingabe gehört, oder „null“, falls die Suche keinen Treffer ergab. Dieses Ergebnis wiederum wird an den ursprünglichen Aufrufer der Funktion „triggerInputProcessing()“ zurückgeliefert.
- „getEntryList()“:
  Durch den Aufruf der Funktion „getEntryList()“ wird die aktuelle Liste der gültigen Sprachkommandos als Rückgabewert retourniert.

- Modul *util_mapping.js*:
  Die Suche nach einem gültigen Kommando in der übergebenen Spracheingabe, findet in der Funktion „compareCommands()“ statt. Im ersten Schritt des Vergleichsprozesses wird die Spracheingabe auf das im Config-File hinterlegte Kommandowort durchsucht. Falls das Kommandowort in der Spracheingabe nicht gefunden wird, handelt es sich um keine gültige Eingabe. Wird das Kommandowort gefunden, werden die darauf folgenden Einträge in der Spracheingabe mit den Einträgen der verfügbaren Kommandos für die aktuelle Seite im LUI verglichen. Hierbei werden allerdings nur dann mehrere Wörter als Spracheingabe in Betracht gezogen, falls es aktuell gültige Kommandoeinträge gibt, die eine entsprechende Anzahl an Wörtern aufweisen. Wird ein Kommando als valide eingestuft, wird es von der Funktion „compareCommands()“ als Rückgabeparameter an die aufrufende Funktion zurückgegeben. Gibt es beim Abgleich keinen gültigen Treffer, wird „null“ an die aufrufende Methode „triggerInputProcessing()“ retourniert.

- Modul *util_string.js*:
  Das Package „string“ liefert eine Vielzahl an Methoden zur String-Manipulation. Bei manchen dieser Methoden können oder müssen gewisse Parameter übergeben werden, die für die jeweilige Operation relevant sind. Ein Beispiel dafür ist die Methode „strip()“. An diese Funktion können jene Zeichen übergeben werden, die aus dem jeweiligen String entfernt werden sollen. Damit nicht in jedem Modul ein ganz spezifischer Aufruf erfolgen muss, wurde hierfür eine generische Methode im Modul *util_string* implementiert, an welche sowohl der String, als auch die zu entfernenden Zeichen, als Parameter übergeben werden. Ein weiterer Grund für diese Modularisierung war, dass das Package „string“ keinen String zurückliefert sondern ein Objekt. Das bedeutet, dass nicht nur die Funktion zur Manipulation des Strings aufgerufen werden muss, sondern anschließend der String aus dem Ergebnisobjekt extrahiert werden muss. Für eine bessere Lesbarkeit sowie eine einfachere Fehlerprävention war es sinnvoll, diesen Ablauf an ei-

ner bestimmten Stelle zu implementieren. So kann bei jedem Aufruf die von nativen JavaScript-String-Methoden gewohnte Syntax beziehungsweise Vorgehensweise genutzt werden. Das Modul enthält die modifizierten Methoden „strip“, „stripChars“, „trim()“ und split()“.

5.3.4.3. Spracherkennungskomponente

Die Spracherkennungskomponente hat im Rahmen dieser Arbeit die Aufgabe, Spracheingaben beziehungsweise Kommandos von der Anwenderin oder dem Anwender zu erkennen und zu validieren. Die Überprüfung der Gültigkeit des Kommandos erfolgt mittels Abgleich zwischen der Spracheingabe und den aktuellen Menu-Entries des LUI. Bei einem validen Kommando wird dieses an die Kernkomponente geschickt. Für die Kommunikation mit der Kernkomponente fungiert die Applikation zur Spracherkennung als WebSocket-Client und erzeugt nach dem Programmstart eine Verbindung mit dem WebSocket-Server der Kernkomponente.

Entwurf

Im Rahmen des ersten Entwurfs der Spracherkennungskomponente wurde eine Windows UWP Application implementiert, in der die Windows Speech API für die Spracherkennung eingebunden war. Dabei handelte es sich zu Beginn um möglichst simple Anwendungen, welche über eine REST-Schnittstelle mit der Kernkomponente kommunizierte und die Resultate der Spracherkennung zurücklieferte.
Für erste Funktionstests, wurde auch eine Logik für das Erstellen von Constraints für die Spracherkennung implementiert. Diese Constraints basierten auf den Menu-Entries, die vom LUI über die Kernkomponente an die Spracherkennungskomponente übermittelt wurden. Bei diesen Tests mit den anderen Komponenten trat allerdings eine unerwartete Problematik mit dieser Applikation zur Spracherkennung auf, die leider einen Technologiewechsel erforderlich machte.
Konkret musste bei diesen Tests in Kombination mit der Kernkomponente und dem LUI festgestellt werden, dass die Applikation zur Spracherkennung nur funktioniert, wenn sie im Vordergrund läuft. Die Recherche ergab, dass die Ursache für dieses Problem der von Microsoft festgelegte Lifecycle für UWP Apps war. Im Detail bedeutet das, dass die Windows Runtime die UWP Apps in einen „Idle“, beziehungsweise „Suspended“ Zustand versetzt. Sobald diese App also in den Hintergrund tritt, werden die von ihr benötigten Ressourcen freigegeben, um diese für Apps, die momentan im Vordergrund ausgeführt werden, zur Verfügung stellen zu können. Das ermöglicht den Einsatz dieser UWP Apps auf den unterschiedlichsten Gerätetypen (auch mobilen Geräten wie Smartphones und Tablets). Grundsätzlich bestünde die Möglichkeit, beispielsweise mit Background Tasks, Extended Execution oder Activity Sponsored Execution Teile der Logik für die Spracherkennung weiterhin auszuführen, trotzdem sich die UWP App nicht mehr im Vordergrund (im Fokus) befindet. Nachdem aber im Zustand „Suspended“ beispielsweise der Zugriff auf das Mikrofon und dessen Input-Daten entzogen wird, würden auch solche Services oder Prozesse das Problem nicht lösen, beziehungsweise umgehen. In der Folgenden *Abbildung 22* aus der Microsoft Dokumentation ist der Lifecycle einer UWP App skizziert [78] [79] [80].

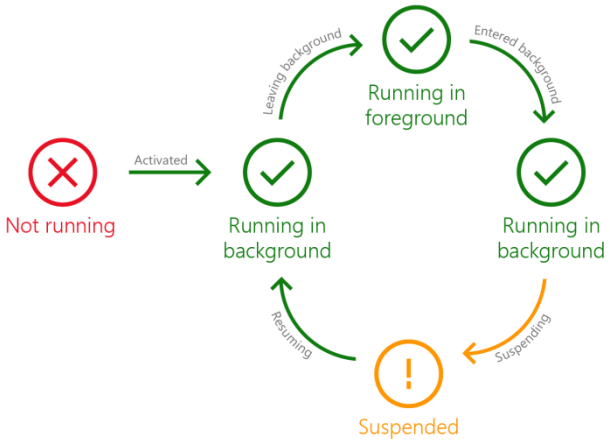


Abbildung 22: Lifecycle einer UWP App [79]

Im Rahmen des Treibers für die Sprachsteuerung des LUI ist es allerdings absolut unerwünscht, dass sich eine Anwendung im Vordergrund, also vor der Benutzeroberfläche des LUI, befindet. Eine weitere Idee, die Windows Speech API trotzdem einsetzen zu können, war, sie in eine Anwendung zu integrieren, die auf der .NET Plattform basiert. Recherchen ergaben, dass das zu diesem Zeitpunkt von Seiten Microsofts nicht vorgesehen ist und auch nicht unterstützt wird. Das bedeutete in weiterer Folge, dass es keine zuverlässige Methode gab die Windows Speech API in einer sauberen, beziehungsweise zuverlässigen Art und Weise in eine Anwendung der .NET Plattform zu integrieren. Aus technischer Sicht liegt diese Problematik daran, dass die Universal Windows Platform (UWP) und ihre Apps Teil des Windows Ecosystems sind und es sich somit um eine gänzlich andere Plattform handelt, zu welcher vor allem in diesem konkreten Fall keine Kompatibilität besteht. Auf Grund dessen wurde entschieden die UWP App mit der Windows Speech API ausschließlich im Rahmen der Evaluierung dieser Arbeit einzusetzen und sie dort mit jener Technologie zu vergleichen, die mit der auf dem .NET Framework basierenden Applikation eingesetzt wird.

Da entsprechend der Zielvorgaben eine Microsoft-Technologie für die Spracherkennung eingesetzt werden sollte, fiel basierend auf den Recherchen aus Kapitel *2.2. Überblick bekannter Frameworks und Software Development Kits*) die Entscheidung auf die Microsoft Speech Platform. Im ersten Entwurf einer Applikation, basierend auf der .NET Plattform, sowie der Microsoft Speech Platform, wurde erneut eine REST-Schnittstelle implementiert. Aus den bereits beschriebenen Gründen (Kapitel *5.3.3. Softwarearchitektur und Workflow*) erfolgte für dieses Interface allerdings ein Umdenken und es wurde ein WebSocket-Client eingesetzt.
Die endgültige Umsetzung der Komponente zur Spracherkennung nutzt dementsprechend das WebSocket-Protokoll für die Kommunikation, sowie die Microsoft Speech Platform für die Spracherkennung, für welche Grammatiken, basierend auf den Menu-Entries des LUIs, erstellt werden.

Implementierung

Bei der Komponente, basierend auf einer Spracherkennung mit der Microsoft Speech Platform, handelt es sich um eine Executeable-Datei (exe-Date), die in der Programmiersprache C-Sharp geschrieben wurde. Um Funktionen der .NET Plattform und der Microsoft Speech

Platform nutzen zu können, war es notwendig, alle benötigten Funktionen zu Beginn dieser Anwendung zur Spracherkennung einzubinden. Das erfolgt über die Direktive „using“, gefolgt von dem entsprechenden Namen des gewünschten Namespaces. In C-Sharp erfüllen Namespaces den Zweck, eine Art Container für Klassen darzustellen. Das ermöglicht nicht nur einen einfacheren und übersichtlicheren Zugriff auf die Klassen und ihre Methoden, sondern auch einen besser (klarer) abgegrenzten Scope. Für die Implementierung dieser Komponente war es notwendig, die folgenden Namespaces einzubinden:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.IO;
using System.Globalization;
using Microsoft.Speech.Recognition;
using System.Net.WebSockets;
```

Listing 13: In die C# Applikation zur Spracherkennung eingebundene Namespaces

- System:
  Bei *System* handelt es sich im .NET Framework um einen Core-Namespace. *System* enthält alle fundamentalen, beziehungsweise Basis-Klassen. Mit dem Einbinden dieses Namespaces wird somit die Arbeit mit den häufigsten Datentypen, aber auch Events, Interfaces oder Exceptions erheblich vereinfacht.
  Um sowohl das Schreiben des Codes zu vereinfachen, also auch eine bessere Lesbarkeit zu garantieren, wurden auch noch weitere Sub-Namespaces von *System* zu Beginn des implementierten Codes direkt eingebunden.
  - System.Collections.Generic:
    Dieser Namespace umfasst Klassen und Interfaces für generische Collections[20]. Gegenüber nicht-generischen Collections bieten sie den Vorteil einer hohen Typensicherheit und einer besseren Performance. Im Rahmen dieser Implementierung werden mit diesem Namespace vor allem Dictionary-Collections erzeugt.
  - System.Text:
    Mit *System.Text* können ASCII- und Unicode-Zeichen dargestellt werden, sowie Byteblöcke und Zeichenblöcke ineinander konvertiert werden. Vor allem die Funktion für das Codieren und Decodieren von Bytes, entsprechend dem UTF-8 Standard, war wesentlich für die implementierte WebSocket-Kommunikation der Spracherkennungsapplikation.
  - System.Threading und System.Threading.Tasks:
    Alle Funktionen, die notwendig sind, um in einem Programm Multi-Threading nutzen zu können, werden vom Namespace *System.Threading* zur Verfügung gestellt. *System.Threading.Tasks* ist ein Sub-Namespace, der Typen zur Verfügung stellt, mit denen es möglich ist, Code asynchron und parallel auszuführen. Bei dieser Implementierung wurde vor allem der einfache Typ *Task* (be-

[20] https://msdn.microsoft.com/de-de/library/system.collections.generic(v=vs.110).aspx

sitzt keinen Return Value) für asynchrones Senden und Empfangen von Nachrichten eingesetzt.

- System.IO:
  System.IO ermöglicht das Lesen und Schreiben von Datenstreams und Dateien. Mit diesem Namespace werden die Zeichen des über den WebSocket eingehenden Bytestreams in der Applikation zur Spracherkennung eingelesen.
- System.Globalization:
  Dieser Namespace enthält Klassen und Funktionen für alle „kulturbezogenen" Informationen („culture-related information"). Das betrifft beispielsweise die Sprache, Formatierungsmuster für das Datum, Währung, Land und Region etc.. Dieser Namespace war vor allem für die Umsetzung der Spracherkennung wichtig, da der Speech Engine die gewünschte Sprache (Runtime Language) für die Spracherkennung zugewiesen werden muss.

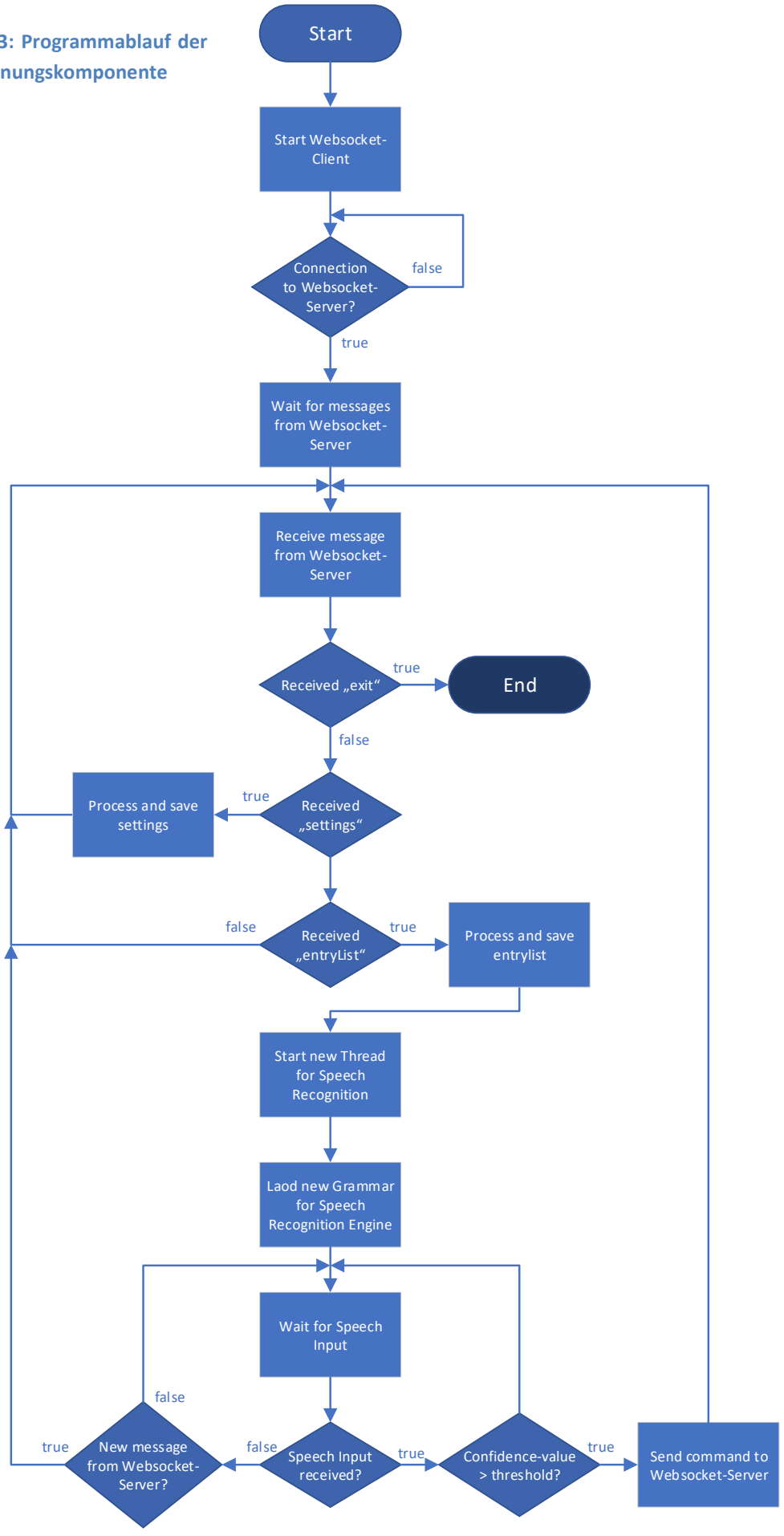
- Microsoft.Speech.Recognition:
  Dieser Namespace ermöglicht die Spracherkennung mit allen von der Microsoft Speech Platform zur Verfügung gestellten Funktionen. Auf die Details dieser Plattform wurde bereits bei der Erläuterung der eingesetzten Technologien ausführlich eingegangen (siehe Kapitel 4.4.2.).

- System.Net.Websockets:
  System.Net.Websockets umfasst alle Funktionen die benötigt werden, um eine Kommunikation über das WebSocket-Protokoll abzuwickeln. Für diese Implementierung war vor allem die Klasse *ClientWebSocket* von Bedeutung.

Allgemeiner Programmablauf dieser Komponente

Vor der Beschreibung des Programmablaufs werden zuerst mittels eines Flussdiagramms die Prozesse dargestellt, die in dieser Komponente durchlaufen werden. Das Diagramm (siehe *Abbildung 23*) zeigt dabei den Start der Komponente, sowie die Vorgänge beim Eintreffen von Nachrichten vom WebSocket-Server und beim Erkennen von Spracheingaben.

Abbildung 23: Programmablauf der Spracherkennungskomponente



Die auf der Spracherkennungstechnologie Microsoft Speech Platform basierende Anwendung namens *MicrosoftSpeechRecognition* wird durch die Kernkomponente (Node.js Anwendung) des Treibers gestartet. Für diese Applikation wird ebenfalls ein Logging für bestimmte Ereignisse und Fehlerfälle durchgeführt (siehe Kapitel *5.6. Logging*).
In der Main-Methode der Klasse „Program“ wird direkt nach dem Start des Programms eine neue Instanz der Klasse „speechRecThread*“* erzeugt. Dieses Objekt wird in die Klasse *Globals* gespeichert und steht mittels seiner Akzessoren (Getter und Setter) global in der kompletten Anwendung zur Verfügung.
Wie im folgenden Abschnitt (Klassen und Funktionen der Komponente) in der Beschreibung von „speechRecThread“ noch genauer erklärt wird, wird in dieser Klasse eine Instanz einer vom Recognizer-Objekt abgeleiteten Speech Engine erzeugt. Dieser Speech Engine wird die gewünschte Runtime Language zugewiesen und in weiterer Folge dynamisch eine Grammatik geladen, welche sich aus dem Kommandowort (welches in den von der Kernkomponente empfangenen Settings definiert ist) und der aktuellen Entrylist (Menu-Entries) zusammensetzt.
In der Main-Methode der Klasse „Program“ wird direkt nach dem Start des Programms der asynchrone Task *Connect()* aufgerufen, wobei diesem Task die URI des WebSocket-Servers übergeben wird. *Connect()* erzeugt als erstes einen neuen WebSocket-Client. Solange sich dieser WebSocket-Client nicht im Status „open“ befindet, wird in einer Schleife versucht mit der übergebenen URI eine Verbindung zum WebSocket-Server herzustellen. An dieser Stelle ist die Applikation intern gestartet und wartet auf eine Antwort des WebSocket-Servers, bevor Nachrichten empfangen werden können und eine Spracherkennung stattfinden kann.
Findet ein HTTP-Upgrade durch den erfolgreichen Handshake mit dem Server statt, wurde eine WebSocket-Connection aufgebaut und der WebSocket befindet sich nun im Status „open“. Dieser Status triggert den Aufruf des Tasks *Receive(),* der als Parameter die aktuelle WebSocket-Instanz übergeben bekommt. Ändert sich der Zustand des WebSockets zu irgendeinem Zeitpunkt (zu einem anderen Zustand als „open“), wird der Task *Receive()* abgebrochen und erneut versucht, mit einer neuen WebSocket-Client-Instanz eine Verbindung zum WebSocket-Server aufzubauen.

Der soeben erwähnte Task *Receive()* ist nun mittels der übergebenen WebSocket-Instanz in der Lage den Input-Stream zu lesen und zu verarbeiten. Treffen Daten von der Kernkomponente des Treibers - die in dieser Implementierung den WebSocket-Server darstellt - ein, werden alle Daten-Junks bis zum Ende der Übertragung in ein Buffer-Array gespeichert. Ist die gesamte Nachricht eingegangen, erfolgt eine UTF-8-Decodierung und Aufruf der Methode *processBuffer()* mit der Nachricht als Übergabeparameter.
In der Methode *processBuffer()* erfolgt die Verarbeitung der Daten. Dieser Methode wird außerdem ein Dictionary übergeben, in welches valide Daten nach ihrer Verarbeitung gespeichert werden sollen. Im ersten Schritt wird überprüft, ob die empfangene und JSON decodierte Nachricht ein Objekt mit dem Namen „settings“, „entryList“ oder „exit“ enthält. Wird „exit“ empfangen, führt das zu einem umgehenden Beenden der Anwendung.
Handelt es sich um eine neue Entrylist oder Settings, wird für jeden Eintrag des Objekts ein Werte-Paar (bestehend aus Identifier/Key und Value) zu dem, der Methode übergebenen, Dictionary hinzugefügt, welches anschließend an den Aufrufer retourniert wird. Enthält die Nachricht weder einen Exit-Befehl, noch Settings oder eine Entrylist, wird sie ignoriert und der WebSocket-Client wartet erneut auf Daten, die vom WebSocket-Server übermittelt werden.

Entsprechend dem definierten Standard-Workflow werden zu Beginn jeder Kommunikation zuerst die Settings und anschließend neue Kommandos in Form einer Entrylist erwartet. Werden keine Settings empfangen, gibt es Default-Werte, die trotzdem einen reibungslosen Ablauf der Applikation ermöglichen. Enthält das von der Methode *processBuffer()* befüllte Dictionary eine Entrylist oder Settings, werden diese mittels Setter in der Klasse „Globals“ gespeichert. Ist eine neue Entrylist eingetroffen, wird, falls es einen aktiven Thread für die Spracherkennung gibt, dieser zuerst beendet. Anschließend wird ein neuer Thread erzeugt, dem die Instanz der Klasse „speechRecThread“ übergeben wird, in der in weiterer Folge die neue Entylist als Grammatik geladen wird. Ab diesem Zeitpunkt ist die Spracherkennung aktiv und wartet auf Spracheingaben.
Wird eine Spracheingabe erkannt, löst die Speech Engine des aktiven Threads das vordefinierte Event „SpeechRecognized“ aus und ruft die für dieses Event definierte Methode (*speechRecEngine_SpeechRecognized()*) auf. Diese, als Event-Handler fungierende Methode, bekommt von der Speech Engine ein Objekt der Klasse „SpeechRecognizedEventArgs“ (Teil der Klasse „Microsoft.Speech“) übergeben. Dieses Objekt enthält diverse Informationen zur erkannten Spracheingabe, beispielsweise neben den erkannten Worten auch deren Confidence-Value[21].
Nun wird zuerst überprüft, ob beim übergebenen Resultat der gewünschte Schwellwert des Confidence-Value erreicht wurde. Ist das nicht der Fall, wird dieser Input ignoriert und die Spracherkennung wird wieder gestartet. Wurde ein entsprechender Confidence-Value erzielt, erfolgt eine Überprüfung, ob die mittels der vordefinierten Grammatik erkannte Spracheingabe auch wirklich einem Eintrag des aktuellen Entrylist entspricht. Existiert der entsprechende Eintrag in der Entrylist, bleibt die Spracherkennung im Standby-Zustand, ohne vorerst den ganzen Thread zu beenden. Anschließend erfolgt ein Aufruf des Tasks *Send()*, welcher das erkannte Kommando übergeben bekommt und es an den WebSocket-Server schickt. Treten bei dieser Übertragung keine Fehler auf, wird danach umgehend der Thread der Spracherkennung beendet. Ab diesem Zeitpunkt wartet die Applikation zur Spracherkennung ausschließlich auf das Eintreffen einer neuen Entrylist im Task *Receive()*, bevor der soeben beschriebene Ablauf von Neuem beginnen kann.

Obwohl der Standard-Use-Case des Treibers zur Spracherkennung nicht vorsieht, dass das LUI alternierend via Spracheingabe und anderen Eingabemethoden bedient wird, muss die Applikation zur Spracherkennung in der Lage sein, asynchron eintreffende neue Entrylists, die nicht durch eine Spracheingabe getriggert wurden, zu verarbeiten.
Bei einem solchen Szenario wird beim Empfangen neuer Daten vom WebSocket-Server, sofort der aktive Thread der Spracherkennung gestoppt. So wird verhindert, dass eventuell Kommandos aus einer Grammatik erkannt werden, die nicht mehr der aktuellen Seite des LUI entsprechen. Handelt es sich bei den neuen Daten nicht um eine Entrylist, wird der Thread mit der ursprünglichen Grammatik unverändert wieder gestartet. Dieses Szenario sollte theoretisch aber nicht eintreten können, da der WebSocket-Server nur zu Beginn Settings schickt und anschließend nur mehr Entrylists oder einen Exit-Befehl. Wird dennoch eine neue Entrylist empfangen, wird der gestoppte Thread beendet und es beginnt anschließend mit der Übergabe der Daten an die Methode *processBuffer()* die exakt gleiche Vorgehensweise wie im vorhergehenden Absatz erläutert.

[21] Der Confidence-Value dient bei Microsoft Speech Recognition Engines als Maß für die Wahrscheinlichkeit, dass das Resultat tatsächlich dem Input entspricht. Genauere Erklärung im Kapitel 6.

Klassen und Funktionen der Komponente

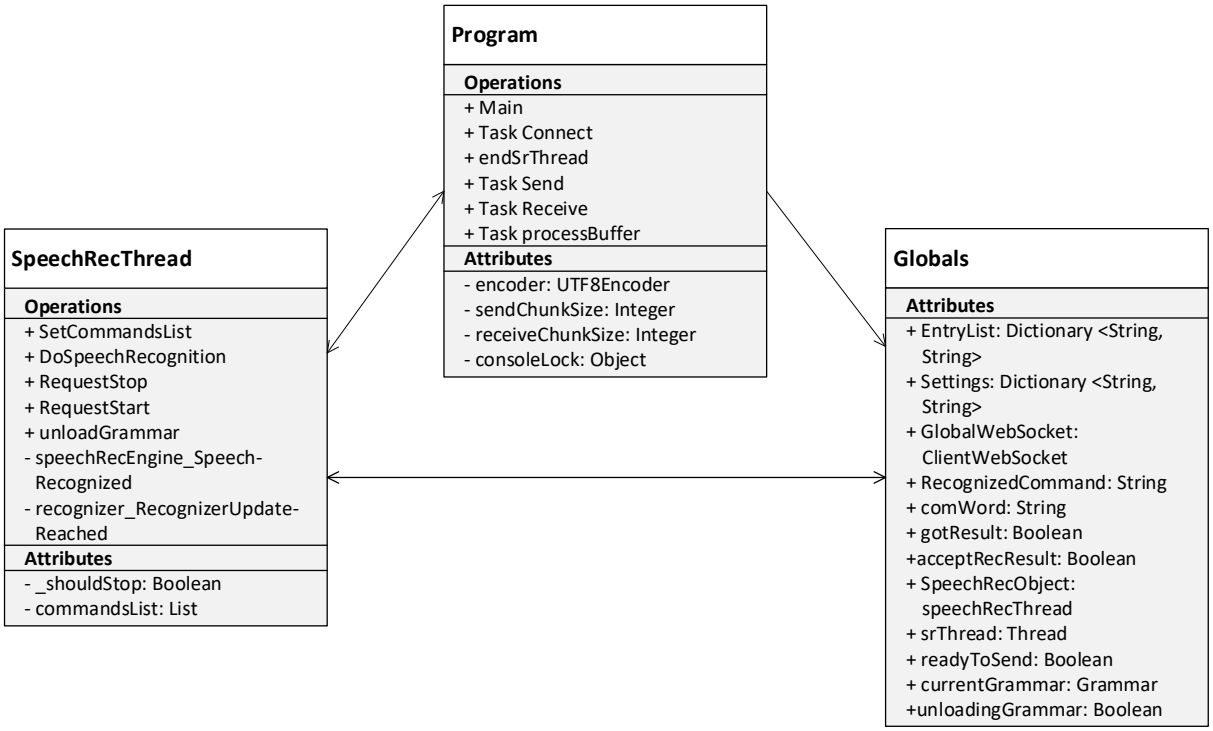


Abbildung 24: Klassendiagramm der C#-Anwendung für die Spracherkennung

Alle Klassen und Methoden dieser C-Sharp-Applikation befinden sich in einem eigenen Namespace, für den der Name „MicrosoftSpeechRecognition“ gewählt wurde. Nach der vorangegangen allgemeinen Beschreibung des Programmablaufs erfolgt nun eine detailliertere Darstellung aller Klassen und deren Methoden inklusive Beschreibung:

- Klasse „public static class Globals“*:*
  In der Klasse „Globals“ werden ausschließlich diverse Variablen deklariert. Das hat den Grund, dass bestimmte Werte während des asynchronen Programmablaufs an einer zentralen Stelle verfügbar sein müssen. Außerdem soll es somit möglich sein diese Variablen im Laufe der Programmausführung von verschiedenen Funktionen in unterschiedlichen Klassen des Programms aufrufen oder ihren Inhalt verändern zu können.

- Klasse „public class SpeechRecThread“:
  Diese Klasse repräsentiert einen (Speech) *Recognizer*. Sie enthält alle notwendigen Methoden und Events, die für die Spracherkennung im Rahmen der Treiber-Implementierung dieser Arbeit benötigt werden.
  - Methode *public void setCommandList(Dictionary<string, string> list):*
    Die Methode *setCommandList()* wird aus der Main-Klasse „Program“, und zwar im asynchronen private Task *Receive(),* aufgerufen. Ihre Funktion ist in der jeweiligen Instanz der Klasse „SpeechRecThread“, die eigene Entrylist vom Typ List<string>, mit den Keys des übergebenen Dictionary<string,

string> list zu ersetzen.

- Methode *public void unloadGrammar():*
  Die Methode *unloadGrammar()* hat, wie der Name der Funktion schon vermuten lässt, die Aufgabe, das aktuelle Grammar der Speech Recognition Engine der jeweiligen Instanz von der Klasse „SpeechRecThread“ zu entfernen (ein „Unload“ durchzuführen). Das ist notwendig bevor ein neues Grammar in die Engine geladen werden kann, da ansonsten die Einträge eines neuen Grammars nur zu den bestehenden hinzugefügt werden würden.

- Methode *public void DoSpeechRecognition():*
  Die Methode *DoSpeechRecognition()* enthält die eigentliche Logik zur Spracherkennung. Der Aufruf dieser Methode erfolgt aus der Main-Klasse „Program“ und zwar im asynchronen Task *Receive()*, wo vor dem Methodenaufruf ein neuer Thread erzeugt wird, dem eine Instanz der Klasse „SpeechRechThread“ übergeben wird.
  Beim Aufruf von *DoSpeechRecognition()* erfolgen die Initialisierung der Speech Recognition Engine und die Zuweisung der Sprache (Runtime Language), die erkannt werden soll. Außerdem werden der Input-Kanal für die Spracherkennung, sowie der *Recognize-Mode*[22] definiert. Zuletzt enthält *DoSpeechRecognition()* noch einen der wichtigsten Punkte für die Spracherkennung, die Initialisierung eines neuen Eventhandlers für erkannte Spracheingaben. Die gesamte Funktion zur Spracherkennung wird so lange in einer Schleife ausgeführt, bis sie explizit von außen durch das Verändern des Parameterwerts der Schleifen-Bedingung gestoppt wird.

- Methode *public void RequestStop():*
  Mit der Funktion *RequestStop()* kann in der jeweiligen Instanz der Klasse „SpeechRecThread“ die Schleife, in der die Spracherkennung läuft (der Ablauf der Methode *DoSpeechRecognition()*), gestoppt werden. Sie wird an unterschiedlichen Stellen im Programmablauf aufgerufen, beispielsweise wenn ein gültiges Kommando erkannt wird.

- Methode *public void RequestStart():*
  Mit der Funktion *RequestStart()* kann in der jeweiligen Instanz der Klasse „SpeechRecThread“ die Schleife, in der die Spracherkennung läuft (der Ablauf der Methode *DoSpeechRecognition()*), gestartet werden. Sie wird an unterschiedlichen Stellen im Programmablauf aufgerufen, beispielsweise wenn ein erkanntes Kommando nicht den gewünschten Confidence-Value aufweist und die Spracherkennung fortgesetzt werden soll.

- Methode *static void speechRecEngine_SpeechRecognized(*object sender, RecognizerUpdateReachedEventArgs e*):*
  Bei der Funktion *speechRecEngine_SpeechRecognized()* handelt es sich um den in der Methode *DoSpeechRecognition()* definierten Eventhandler. Die

[22] Im Recognize-Mode kann definiert werden, ob ein oder mehrere Spracheingaben hintereinander möglich sein sollen. Soll nur eine Spracheingabe am Stück möglich sein, stoppt die Spracherkennung nach einem erkannten Input automatisch und muss anschließend wieder explizit gestartet werden.

Funktion wird dementsprechend aufgerufen, wenn von der Speech Recognition Engine ein, dem Grammar entsprechendes, Ergebnis für eine Spracheingabe zurückgeliefert wird. In dieser Methode wird überprüft, ob das Ergebnis dem gewünschten Confidence-Value entspricht und es folgt eine erneute, letzte Überprüfung, ob das Kommando in der aktuellen Entrylist enthalten ist. Wird die Spracheingabe als gültig eingestuft, wird ein Versenden dieses Inputs an den WebSocket-Server mit der Übergabe des Kommandos an den Task *Send()* der Klasse „Program“ getriggert.

- Klasse „class Program“*:*
  Die Klasse „Program*“* enthält die Methode *static void Main()*. Dementsprechend befindet sich an dieser Stelle der Einstiegspunkt für Programmausführung. Des Weiteren umfasst diese Klasse noch alle für die Kommunikation über das WebSocket-Protokoll notwendigen Funktionalitäten.

  - Task *Public static async Task Connect():*
    Bei *Connect()* handelt es sich um einen Task, der nach seinem Aufruf asynchron ausgeführt wird. Der Aufruf wird einmal zu Beginn der Programmausführung in der Main Methode aufgerufen. Im dem Task wird ein neuer Client für eine WebSocket-Kommunikation erzeugt. Über die, als Task-Parameter übergebene URI, wird anschließend versucht, eine Verbindung zum WebSocket-Server herzustellen. Solange keine erfolgreiche Verbindung aufgebaut werden kann, versucht sich dieser Task in einer Schleife immer wieder erneut, mittels der übergebenen URI, mit dem WebSocket-Server zu verbinden. Wurde eine erfolgreiche WebSocket-Connection (WebSocket-Status „open“) erzeugt, erfolgt ein Aufruf des Tasks *Receive(),* dem als Übergabeparameter die aktuelle WebSocket-Instanz mitgegeben wird. Sobald der WebSocket vom Status „open“ in einen anderen Zustand wechselt, werden alle Prozesse gestoppt und es wird erneut versucht, mit einer neuen WebSocket-Client-Instanz eine Connection aufzubauen.

  - Methode *private static void endSrThread():*
    Die Methode *endSrThread()* ist ein Hilfsmittel, um innerhalb der Klasse „Program“ sowohl das aktuell geladene Grammar der „SpeechRecThread“ Instanz zu entfernen, als auch den Thread dieser Instanz zu stoppen. Dementsprechend wird diese Funktion an unterschiedlichen Stellen, teils auch zur Fehlerbehandlung, aufgerufen.

  - Task *Public static async Task Send():*
    Der asynchrone Task *Send()* wird ausschließlich von Methoden *speechRecEngine_SpeechRecognized()* der aktuellen Instanz des „SpeechRecThread“ aufgerufen. Nach dem Aufruf des Tasks wird die als Übergabeparameter mitgegebene erkannte Spracheingabe an den WebSocket-Server gesendet. Bei einem erfolgreichen Versenden der Spracheingabe, wird der Thread, der die aktuelle Instanz von „SpeechRecThread“ beinhaltet, beendet. Im Fehlerfall wird die Spracherkennung über die Methode *RequestStart()* der jeweiligen „SpeechRecThread“ Instanz erneut gestartet.

- Task *Public static async Task Receive(ClientWebSocket webSocket):*
  Bei *Receive()* handelt es sich ebenfalls um einen Task, der asynchron ausgeführt wird. Der Aufruf erfolgt aus dem Task *Connect()* (ebenfalls aus der Klasse „Program“) und es wird als Parameter die aktuelle WebSocket-Instanz übergeben. *Receive()* hat die Aufgabe, vom WebSocket-Server eingehende Nachrichten zu empfangen. Die Daten-Junks des Datenstreams werden dabei in einen Byte-Buffer (Buffer-Array) gespeichert, der nach vollendeter (erfolgreicher) Übertragung UTF-8-decodiert und als String gespeichert wird. Dieser String wird an die in Methode *processBuffer()* übergeben, welche die Daten in ein *Dictionary* vom Typ <string, string> speichert. Nach diesem Vorgang (dem erfolgreichem Ablauf von *processBuffer()*), erfolgt eine Unterscheidung zwischen einer eingetroffenen Entrylist (Menu-Entries) oder Settings. Während Settings global gespeichert werden, wird bei neuen Kommandos zusätzlich noch ein neuer Thread für die Spracherkennung erzeugt und als Parameter eine Instanz der Klasse „speechRecThread“ übergeben. Nach dem anschließenden Starten des Threads mittels der Methode *RequestStart(),* wird die übergebene Instanz der Klasse „speechRecThread“ ausgeführt. Diese führt die Spracherkennung mit einer Grammatik, basierend auf den neuen Menu-Entries (Entrylist), durch.

- Methode *private static void processBuffer(Dictionary<string, string> newList, string buffer):*
  Die Methode *processBuffer()* wird vom Task *Receive()* aufgerufen und verarbeitet die übergebenen Daten, die vom WebSocket-Server empfangen wurden. Zu Beginn erfolgt eine Überprüfung, um welche Art von Daten es sich handelt. Dabei werden Objekte mit dem Namen „entryList“, „settings“ oder „exit“ akzeptiert, während andere Daten in keiner Form weiter verarbeitet werden. Die Message „exit“ bewirkt, dass sich die Anwendung umgehend beendet. Handelt es sich um neue Menu-Entries oder Settings, werden diese in das übergebene Dictionary „newList“ gespeichtert. Die Entscheidung fiel auf diese Datenstruktur, da sie einfach handhabbar ist. Jedes Wertepaar, das dem Dictionary hinzugefügt wird, enthält als Identificator/Key den Key des neuen Menu-Entries, sowie als Value den Value des Menu-Entires. In dieser Form kann eine einfache und schnelle Suche auf im Dictionary enthaltene Kommandos durchgeführt werden.

5.4. Testing

In diesem Abschnitt wird dokumentiert, welche Vorgehensweise für das Testen, der in dieser Arbeit implementierten, Komponenten des Treibers zur Bedienung des LUI mittels Spracherkennung gewählt und umgesetzt wurde.

Da bei der Entwicklung das Hauptaugenmerk auf der Kernkomponente des Treibers und dessen Performanz und Robustheit lag, wurden für diesen Teil der Software die umfangreichsten Tests implementiert. Die richtige Erkennung von Spracheingaben, sowie die weitere Verarbeitung in der Komponente zur Spracherkennung wurde aufgrund des Umfangs und der Komplexität dieser Tests ausschließlich im Rahmen der Evaluierung der Spracherkennung und der Gesamtlösung durchgeführt (siehe *6. Evaluierung*), nicht durch automatische Unit-Tests.

Die Kernkomponente des Treibers wurde in Anlehnung an das Konzept des Test Driven Developments entwickelt. Dabei wurde in dieser Arbeit nach dem groben Entwurf der Kernkomponente, sowie nach Refactorings von Funktionen der Kernkomponente Unit-Tests (Test-Fälle) erstellt. Anhand dieser Test-Fälle war es nach einer neuen Implementierung oder einem Refactoring umgehend möglich, die korrekte Funktionsweise dieses Codes, beziehungsweise das Erreichen des gesetzten Ziels zu überprüfen. Die unterschiedlichen Testfälle wurden in einem Script implementiert, das automatisch ausgeführt werden kann und alle implementierten Testfälle durchführt. Dieses Script kann problemlos erweitert werden und ist auch ein wesentliches Instrument, um die Wartbarkeit, sowie zukünftige Erweiterungen der Kernkomponente, zu unterstützen. Das Script wurde ebenso wie die Kernkomponente des Treibers mit dem Framework Node.js erstellt.

Für das Test-Script wurde das Node.js Framework um die folgenden Module erweitert:

- Modul „mocha"[23]:
  Diese Modul wurde nicht für die eigentliche Entwicklung und Implementierung des Codes der Komponenten der Kernkomponente eingesetzt, sondern, um dessen Funktionsfähigkeit zu testen. Bei „mocha" handelt es sich um ein JavaScript Test-Framework, das eine Vielzahl an Funktionen zum Erstellen von Testfällen zur Verfügung stellt. Dieses Framework ermöglicht beispielsweise ein unkompliziertes Testen von asynchronen Programmabläufen, beziehungsweise asynchronen Schnittstellen wie REST-Interfaces. Außerdem können automatisch übersichtliche Dokumentationen über die durchgeführten Testfälle und deren Ergebnisse erstellt werden. Somit können wichtige Testparameter wie die Test-Coverage, Dauer der Tests, fehlgeschlagene Tests oder auch aufgetretene Error schnell erkannt und ausgewertet werden.

- Modul „chai"[24]:
  Das Modul „chai" ist eine sogenannte Assertion Library und ist sehr gut dafür geeignet, Test-Frameworks (wie „mocha") zu erweitern. Assertion Library bedeutet, dass Bedingungen (Behauptungen) erzeugt werden können, anhand deren Erfüllung überprüft werden kann, ob sich der getestete Code wie gewünscht verhält. Von „chai"

[23] https://mochajs.org/
[24] https://www.npmjs.com/package/chai / http://chaijs.com/

werden drei unterschiedliche Assertions zur Verfügung gestellt: „assert“, „expect“ und „should“. Auf die detaillierten Unterschiede der unterschiedlichen Assertion-Styles wird an dieser Stelle auf die Dokumentation von „chai“ verwiesen[25]. Im Rahmen der in dieser Arbeit durchgeführten Tests wurde „chai“ mit der Assertion-Methode „expect“ als Erweiterung zum Test-Framework „mocha“ eingesetzt.

- Modul „http-shutdown“[26]:
  Wie der Name dieses Moduls bereits suggeriert, bietet es die Funktion an HTTP-Server zu beenden. Die HTTP-Server des Moduls „http“ besitzen zwar eine Methode „exit“ um den Server zu beenden, allerdings wird dadurch nur verhindert, dass neue Connections aufgebaut werden können, sogenannte „keep-alive“ Connections bleiben aber nach wie vor aktiv. Mit „http-shutdown“ wird eine einfache und schnelle Möglichkeit zur Verfügung gestellt, um auch diese Connections zu beenden. Diese Funktion wurde bei den Testfällen im Rahmen der Unit-Tests der Kernkomponente genutzt.

Die Unit-Tests für die Kernkomponente wurden im File (Modul) „lui-driver-test.js“ erstellt. Unter einem Unit-Test ist in diesem Fall zu verstehen, dass ein bestimmter Teil der Komponente, der noch granularer ist als ein Modul, getestet wurde (beispielsweise die Funktionalität des WebSocket-Servers). Ein Test für einen solchen Teil einer Komponente kann dabei mehrere Test-Fälle enthalten. Mit „mocha“ wird ein Test mit Methode *describe()* definiert, so wie mit der Methode *it()* die einzelnen Test-Fälle festgelegt werden. Als Parameter für diese beiden Methoden sollten die Beschreibungen der jeweiligen Tests und Testfälle übergeben werden, da diese dann auch automatisch im Testbericht zu den jeweiligen Ergebnissen geschrieben werden. Dadurch wird der Testbericht deutlich übersichtlicher und lesbarer.
Das Mocha.js Test-Framework ist in seiner Verwendung und Funktionalität anderen bekannten Test-Frameworks, wie beispielsweise „Jasmine“, sehr ähnlich. Die automatischen Tests für die Kernkomponente können mit dem Batch-File „start_test“ durchgeführt werden und die Ergebnisse sind in dem Text-File „test-result.txt“ zu finden. Sowohl das Batch-File als auch das Text-File mit den Ergebnissen befinden sind im Root-Verzeichnis des Treibers.

Testfälle

Da der Schwerpunkt bei der Entwicklung des Treibers auf der Kernkomponente lag, die im Gegensatz zu der Applikation zur Spracherkennung nicht austauschbar ist, lag auch der Schwerpunkt der Unit-Tests auf dem Überprüfen der Funktionsfähigkeit der Kernkomponente.

Für die Kernkomponente wurden die folgenden Tests und Testfälle definiert:

- Testen des HTTP-Servers, der Nachrichten vom LUI empfängt
  - Der Server soll beim erfolgreichen Empfangen von Settings mit dem Status-Code 200 (Success) antworten.

[25] http://chaijs.com/guide/styles/#differences
[26] https://www.npmjs.com/package/http-shutdown

    - Der Server soll beim erfolgreichen Empfangen von neuen Menu-Entries mit dem Status-Code 200 (Success) antworten.
    - Der Server soll beim erfolgreichen Empfangen des Befehls zur Änderung des Listenstates mit dem Status-Code 200 (Success) antworten.
        - Wurde der Listenstate geändert, soll eine Display-Message an das LUI übermittelt werden.

- Testen des HTTP-Servers, der Nachrichten an das LUI sendet
    - Der Server soll nach dem Start ein Request für Settings an das LUI senden.
    - Der Server soll nach dem Request für Settings auch einen Request für die aktuellen Menu-Entries an das LUI senden.
    - Der Server soll nach dem Empfangen eines Requests des Listenstates den aktuellen Listenstate an das LUI übermitteln.

- Testen des WebSocket-Servers
    - Nach erfolgreichem Verbindungsaufbau soll der WebSocket-Server Settings an den WebSocket-Client schicken.
    - Nach den Settings soll der WebSocket-Server bei Verfügbarkeit die aktuellen Menu-Entries (Entrylist) an den WebSocket-Client senden.
    - Sendet der WebSocket-Client einen Request für die Entrylist, soll der Server diese, bei Verfügbarkeit, im Response übermitteln.
        - Bei aktiver Spracherkennung soll die aktuelle Entrylist übermittelt werden.
        - Bei deaktivierter Spracherkennung soll nichts übermittelt werden.
        - Bei aktiver Spracherkennung und konfiguriertem Sprachkommando für die Deaktivierung soll die Entrylist inklusive dem Sprachkommando zur Deaktivierung übermittelt werden.
        - Bei deaktivierter Spracherkennung und konfiguriertem Sprachkommando für die Aktivierung soll ausschließlich das Sprachkommando zur Aktivierung übermittelt werden.
    - Sendet der WebSocket-Client einen Request für die Settings, soll der Server diese bei Verfügbarkeit im Response übermitteln.
    - Sendet der WebSocket-Client ein erkanntes Kommando an den Server, soll die Kernkomponente dieses Kommando an das LUI weiterleiten, falls der Listenstate gleich „true“ ist.
    - Nach einer Veränderung des Listenstates soll dieser umgehend an den WebSocket-Client übermittelt werden.

Die Tests für die Komponente zur Spracherkennung wurden teils manuell und anhand der definierten Tests und Test-Fälle (einer Checkliste) durchgeführt, um die wesentlichen Funktionen im Zusammenhang mit der Kernkomponente zu überprüfen.

Für die Komponente zur Spracherkennung wurden die folgenden Tests und Testfälle definiert:

- Testen des WebSocket-Clients
    - Mittels Client-Request eine Verbindung zum WebSocket-Server erzeugen
    - Empfangen und decodieren von neuen Menu-Entries.
    - Empfangen und decodieren von neuen Settings.
    - Empfangen und decodieren des Listenstates.
    - Senden von erkannten Kommandos an den WebSocket-Server.
    - Nachfragen der aktuellen Entrylist.
    - Nachfragen der aktuellen Settings.
    - Nachfragen des aktuellen Listenstates.

- Testen der Logik und Spracherkennung der Applikation
    - Verarbeiten neuer Menu-Entries
    - Laden und entfernen von Grammatiken in die Speech Recognition Engine
    - Starten und stoppen der Spracherkennung

5.5. Treiberintegration

In diesem Abschnitt wird darauf eingegangen, wie der Treiber in das LUI-Environment integriert wurde. Diese Integration lässt sich in zwei Tasks unterteilen, für die eine Lösung ausgearbeitet werden musste. Dabei handelt es sich um die Treiberkonfiguration, die im Config-File des LUI vorgenommen werden muss, sowie um einen Startup- und Shutdown-Prozess für den Treiber.

5.5.1. Treiberkonfiguration

Da die Entscheidung gefällt wurde, dass für die Anbindung des Treibers an das LUI die REST-Schnittstelle genutzt werden soll (siehe Kapitel *5.3.2. Interface Definitions*), musste die entsprechende Konfiguration in den LUI-Settings (im File „startup.dcfg“) eingetragen werden. Dieses File ist entsprechend dem XML-Format (Extensible Markup Language) aufgebaut und UTF-8 codiert gespeichert, dementsprechend wurde auch die REST-Konfiguration vorgenommen. Für diese Arbeit wurde der Treiber wie folgt konfiguriert:

```
<restdriversettings>
   <driver name="speechRec">
      <active>true</active>
      <listenport>45678</listenport>
      <method>post</method>
      <url>http://127.0.0.1:12345</url>
      <token>011235</token>
      <exe>C:\LUI_speechrec_driver\start_nodejs.bat</exe>
      <connectionType>websocket</connectionType>
      <confidenceThreshold>0,55</confidenceThreshold>
      <commandWord>lui</commandWord>
      <language>de</language>
      <srProgram>MicrosoftSpeechRecognition.exe</srProgram>
      <forceSrExit>true</forceSrExit>
      <errorcallback>drivererror</errorcallback>
      <speechRecState>
         <speech>
            <activate>Spracherkennung aktivieren</activate>
            <deactivate>Spracherkennung deaktivieren</deactivate>
         </speech>
        <changelistenstate>setListenState</changelistenstate>
        <getlistenstate>getListenState</getlistenstate>
      </speechRecState>
      <relevantSettings>listenport, method, url, token, connectionType,
      confidenceThreshold, commandWord, language, srProgram,
      forceSrExit, errorcallback, speechRecState</relevantSettings>
   </driver>
</restdriversettings>
```

Listing 14: LUI REST-Driver-Settings

Ein Großteil dieser Konfiguration bezieht sich auf die Kernkomponente (Kommunikationskomponente) des Treibers, beziehungsweise ist für diesen Teil der Software wichtig. Da die Applikation zur Spracherkennung, wie bereits erläutert, austauschbar ist, muss von der Entwicklerin oder dem Entwickler dieser Applikation festgelegt werden, welche zusätzlichen Settings eventuell notwendig sind. Das Hinzufügen von weiteren Einstellungen wird nach der anschließenden, detaillierten Erklärung der Konfiguration aus *Listing 14* erläutert.

<driver name>

Hier wird der interne Name (das heißt die interne Repräsentation) des Treibers festgelegt. Dieser Tag ist verpflichtend und sein Wert darf nicht verändert werden.

<active>

Anhand dieses Settings wird der Treiber beim Start des LUIs entweder aufgerufen oder ignoriert. Valide Werte sind hierbei „true“ (bedeutet Treiber soll aufgerufen werden) oder „false“, wobei letzteres die Default-Einstellung ist.

<listenport>

An dieser Stelle wird jener Port festgelegt, über den das LUI Nachrichten, beziehungsweise Kommandos empfangen kann. Der Default-Wert für den Listenport des LUI ist mit „45678“ definiert.

<method>

Unter diesem XML-Tag wird die http-Methode definiert. Bei den zur Auswahl stehenden Methoden handelt es sich um:

- Post
- Put
- Patch

Für die Treiberkonfiguration dieser Arbeit wurde die Methode „Post“ ausgewählt. Allgemein lässt sich der wesentliche Unterschied zwischen Post, Put und Patch wie folgt erklären: „Put“ ist per Definition für das Ersetzen von ganzen Ressourcen an der angegebenen URI gedacht, also beispielsweise das Überschreiben eines ganzen Files. Die Methode „Post“ hingegen, soll dafür eingesetzt werden, bereits vorhandene Daten an der Ziel-URI zu aktualisieren. Die letzte zur Verfügung stehende Option namens „Patch“ bietet die Möglichkeit, ein Update auf einzelne Datenfelder durchzuführen. (Die Erläuterung tiefergehende Unterschiede übersteigt den Rahmen dieser Arbeit.)

<url>

Die URL, an welche die Nachrichten vom LUI gesendet werden soll, wird an dieser Stelle angegeben. Diese Einstellung ist nicht optional und für die Funktionsfähigkeit der REST-Schnittstelle unumgänglich. Dieser Wert darf außerdem nicht verändert werden.

<token>

Anhand des hier angegeben Tokens kann der Treiber identifiziert werden. Diese Konfiguration ist ebenfalls optional, unterliegt allerdings der Voraussetzung, dass der Token ausnahmslos einzigartig („unique“) sein muss, sich also von den Token anderer Treiber unterscheidet.

<exe>

Diese Einstellung ist optional und bietet die Möglichkeit eine Executeable oder ein Batch-File aufzurufen. Diese Option wird im Rahmen dieser Arbeit genutzt, um über ein Batch-File den entwickelten Treiber zur Spracherkennung, sowie die eingesetzte Applikation zur Spracherkennung, aufzurufen.

<connectionType>

Mit dieser Einstellung wird der Kommunikationskomponente des Treibers angegeben, welche Schnittstelle zur Spracherkennungssoftware genutzt werden soll. Grundsätzlich kann bei diesem Setting „websocket“, „webserver validate“ oder auch überhaupt nichts angegeben werden.

Die Treibersoftware prüft im ersten Schritt ausschließlich auf den Parameter „websocket“. Nur wenn der explizite Wert „websocket“ (siehe *Listing 14*) gesetzt wird, erfolgt die Akti-

vierung des WebSocket-Servers. Wird an dieser Stelle nichts oder etwas anderes außer „websocket“ angegeben, wird ausschließlich die Rest-Schnittstelle aktiviert.
Bei der REST-Schnittstelle kann ausgewählt werden, ob Informationen, die von der Spracherkennung empfangen werden, validiert oder ausschließlich an das LUI weitergeleitet werden. Dafür erfolgt die Überprüfung auf den Wert "webserver validate". Wurde "webserver validate" in diesem XML-Tag angegeben, werden die Ergebnisse der Spracherkennung mit den verfügbaren Kommandos abgeglichen und nur bei einer Übereinstimmung an das LUI übermittelt.

<confidenceThreshold>
Der Confidence Threshold ist in diesem Kontext der Schwellwert, ab welchem eine erkannte Spracheingabe als gültig gewertet wird. Er ist dafür gedacht, dass für die jeweilige Implementierung der LUIs und der Spracherkennung einer, den äußeren Einflüssen, sowie der User-Gruppe entsprechenden Wert, eingestellt werden kann. Bei der Nutzung des WebSockets wird dieser Wert direkt an die Client-Software weitergeleitet und kann dementsprechend beliebig gewählt werden, beziehungsweise einer beliebigen Notation entsprechen. Wird nicht die WebSocket-Schnittstelle, sondern die REST-Schnittstelle genutzt, sollte es sich um einen Wert entsprechend dem Datentype Integer oder Double (mit einem Komma als Trennzeichen für die Dezimalzahl) handeln.

<commandWord>
Diese Einstellung ist optional. Das Command Word kann gesetzt werden, um eventuell die Erkennungsrate der Spracherkennung zu verbessern. Wird auf diese Weise ein Command Word gesetzt, wird bei der Nutzung der REST-Schnittstelle beispielsweise erst auf ein gültiges Kommando geprüft, wenn in der Spracheingabe zuvor das Command Word enthalten ist.

<language>
Die Sprache kann optional gesetzt werden. Die Kommunikationskomponente selbst benötigt diese Setting allerdings nicht und leitet das Kommando im Fall einer bestehenden WebSocket-Verbindung ausschließlich direkt an den Client weiter.

<srProgram>
Unter dem Tag „srProgram“ wird der exakte Name (inklusive Datei-Endung der Software) angegeben, mit welchem die Spracherkennung durchgeführt werden soll. Diese Angabe benötigt der Treiber, um die Anwendung gegebenenfalls starten, beziehungsweise beenden zu können. Für einen korrekten Programmablauf der Kernkomponente ist diese Einstellung optional, allerdings muss die Software für die Spracherkennung selbstständig starten, falls dieses Setting nicht genutzt wird. Das auszuführende Programm für die Spracherkennung muss im Root-Folder der Node.js-Applikation abgelegt werden, oder der Pfad muss relativ zu diesem Root-Folder angegeben werden.

<errorcallback>
An dieser Stelle der Konfiguration kann angegeben werden, unter welchem „Action-Name“ der Treiber Fehlermeldungen an das LUI weiterleitet. Diese Einstellung ist nicht verpflichtend, wird aber stark empfohlen, um ein korrektes Error-Handling und Fehleranalysen zu ermöglichen.

<speechRecState>

Dieser Tag beinhaltet alle Einstellungen, die im Zusammenhang mit dem Listenstate des Treibers, beziehungsweise der Spracherkennung getätigt werden können. „speechRecState“ enthälte die „Sub-Settings“ „speech“, „changelistenstate“ und „getlistenstate“.

<speech>

Um eine Aktivierung und Deaktivierung der Spracherkennung mittels Sprachkommandos zu ermöglichen, muss dieser Tag angegeben werden. Die jeweiligen Sprachkommandos werden unter „activate“ und „deactivate“ spezifiziert. Es ist auch möglich, nur für eine der beiden Aktionen ein Sprachkommando anzugeben. Wird beispielsweise nur „deactivate“ spezifiziert, kann die Spracherkennung über das angegebene Kommando zwar deaktiviert werden, die Aktivierung muss aber über einen anderen Weg erfolgen. In diesem Fall muss allerdings der komplette Tag „activate“ aus der Konfiguration gelöscht werden, sonst könnte es unter Umständen passieren, dass Leerzeichen als Kommando interpretiert werden.

<activate>

Hier kann das Sprachkommando angegeben werden, mit dem die Aktivierung der Spracherkennung erfolgen soll. In der Beispiel-Konfiguration in *Listing 14* wäre das „Spracherkennung aktivieren“.

<deactivate>

Hier kann das Sprachkommando angegeben werden, mit dem die Deaktivierung der Spracherkennung erfolgen soll. In der Beispiel-Konfiguration in *Listing 14* wäre das „Spracherkennung deaktivieren“.

<changelistenstate>

An dieser Stelle wird definiert, mit welcher Message das LUI den Listenstate verändern kann. In der Beispiel-Konfiguration in *Listing 14* wäre das die Message „setListenState“. Als Daten der Nachricht wird ein Boolean-Wert erwartet, der den Zustand repräsentiert, zu dem der Listenstate wechseln soll.
Beispielaufruf zum Verändern des Listenstates:

```
<command type="broadcast">
  <message>setListenState</message>
  <data>true</data>
</command>
```

<getlistenstate>

Unter „getlistenstate“ wird die Message spezifiziert, unter der der aktuelle Listenstate nachgefragt werden kann. Mit dem „Message-Name“ als Action und dem aktuellen Listenstate als Daten erfolgt dann die Response vom Treiber an das LUI.
Beispiel-Request:

```
<command type="broadcast">
  <message>getListenState</message>
</command>
```

Beispiel-Response:

```
<command type=internal">
  <action>getListenState</action>
  <data>true</data>
</command>
```

<forceSrExit>
Diese Konfiguration ist verpflichtend und muss mit den Parametern „true“ oder „false“ angegeben werden. Dies legt fest, wie die Kernkomponente beim Beenden des LUI mit der Applikation zur Spracherkennung verfahren soll. Bei „true“ beendet die Kernkomponente die native Software zur Spracherkennung selbst, während bei „false“ die Applikation zur Spracherkennung nur über ein Beenden benachrichtigt wird und selbstständig für das weitere Vorgehen und Beenden verantwortlich ist.

<relevantSettings>
Der Eintrag „relevant Settings“ ist verpflichtend; an dieser Stelle müssen die exakten Tag-Namen jener Settings angegeben werden, die vom Treiber benötigt, beziehungsweise genutzt werden sollen. Das bedeutet, dass alle Einstellungen die verpflichtend sind, sowie jene die beispielsweise für die Software zur Spracherkennung benötigt werden, hier angegeben werden müssen.

Die hier erläuterte Konfiguration kann noch um weitere Settings ergänzt werden, dafür muss ausschließlich ein neuer Tag hinzugefügt, und dessen Tag-Name unter „relevantSettings“ ergänzt werden. Diese Konfiguration wird dann über die Kommunikationskomponente der Applikation zur Spracherkennung zur Verfügung gestellt.

Zur Vollständigkeit sei erwähnt, dass für die Konfiguration von Treibern für das LUI überdies noch weitere Einstellungen verfügbar wären, die im Rahmen dieser Implementierung allerdings nicht verwendet wurden, beziehungsweise nicht relevant waren und daher nicht näher erläutert werden. Weitere Details zu den umfangreichen Einstellungsmöglichkeiten, wie beispielsweise dem Setzten von Umgebungsvariablen, sind im Manual des LUI zu finden [76].

5.5.2. Treiber Lifecycle

Da der Treiber nur lose mit dem LUI gekoppelt ist, muss explizit dafür gesorgt werden, dass dieser immer funktionsfähig bleibt. Dazu zählt auch ein dem LUI entsprechendes Verhalten, also ein gleichzeitiges Starten und Beenden. In diesem Abschnitt werden die Lifecyle-Prozesse des Treibers zur Sprachsteuerung dokumentiert, die exakt dieses Verhalten gewährleisten sollen.

Startup des Treibers

Wie bei der Erklärung des Konfigurations-Files des LUI bereits angeschnitten wurde, kann ein Pfad zu einer Datei (Executable, Batch etc.) angegeben werden, wodurch diese Datei beim Start des LUI automatisch aufgerufen wird. Für die Treibersoftware wurde für das Startup-Prozedere ein simples Batch-File geschrieben, welches bei seinem Aufruf die Kernkomponente (Node.js Applikation) startet. Da die Spracherkennung austauschbar ist und über die Config-Datei festgelegt werden kann, welche verwendet werden soll, wartet die Kernkomponente anschließend auf diese vom LUI übergebenen Settings. Über den in den Settings übergebenen Pfad, beziehungsweise Namen der Spracherkennungsapplikation (wird unter dem Tag <srProgram> übergeben), kann anschließend auch diese Komponente gestartet werden. Bei einem erfolgreichen Start aller Treiberkomponenten wird die Anwenderin oder

der Anwender mittels einer Nachricht, in der Benutzeroberfläche des LUI, über die Funktionsfähigkeit der Sprachsteuerung informiert.

Runtime-Verhalten des Treibers

Bricht im laufenden Betrieb die Verbindung zur Spracherkennung ab, wird von der Kernkomponente diese Instanz der Spracherkennungskomponente beendet (falls sie trotzdem noch aktiv sein sollte). Somit wird verhindert, dass mehrere Applikationen zur Spracherkennung gleichzeitig laufen, beziehungsweise eine fehlerhafte Instanz niemals beendet wird. Über diesen Verbindungsabbruch wird die Anwenderin oder der Anwender erneut mittels einer Nachricht informiert. Anschließend wird die Spracherkennungskomponente neu gestartet und bei einem erfolgreichen Verbindungsaufbau eine Benachrichtigung in der Benutzeroberfläche des LUI angezeigt.

Shutdown des Treibers

Wird das LUI beendet, bekommt der Treiber eine Benachrichtigung dieses Ereignisses übermittelt. Abhängig von der gewählten Treiber-Konfiguration (unter dem XML-Tag <forceSrExit>), wird beim Eintreffen dieser Nachricht zuerst die Spracherkennungskomponente und anschließend die Kernkomponente des Treibers beendet.
Sollte das LUI unerwartet, beziehungsweise irregulär, beendet werden, wird keine Information über diesen Vorgang an die Kernkomponente übermittelt. Es muss allerdings trotzdem sichergestellt werden, dass der Treiber infolge dessen beendet wird. Da über die REST-Schnittstelle keine dauerhafte Verbindung besteht, wird von der Kernkomponente in einem konfigurierbaren Intervall ein simpler Request an das LUI geschickt. Wenn das LUI noch aktiv ist, wird auf jeden Fall eine Antwort mit einem Status-Code retourniert. Wird keine Antwort mehr empfangen, wie beispielsweise durch reguläres Beenden, werden alle Treiberkomponenten geschlossen.

5.6. Logging

Um die Wartbarkeit der Software zu erleichtern, beziehungsweise den Programmablauf und aufgetretene Fehler nachvollziehen zu können, erfolgen für alle Treiberkomponenten laufend Aufzeichnungen in Form eines Loggings. Das Logging der Kernkomponente unterteilt sich in das Aufzeichnen von diversen Informationen zum Programmablauf im File „log-debug_$currentDate.txt“, sowie das Erfassen von Fehlern (Errors) im File „log-error_$currentDate.txt“ („$currentDate“ fungiert hier als Platzhalter für das automatisch generierte Datum des Tages, an dem das File erstellt wurde). Bei jedem neuen Programmaufruf wird der aktuelle Zeitstempel in die Logging-Files geschrieben. Diese beiden Files sind im Root-Folder des Treibers zu finden.

Bei der im Rahmen dieser Arbeit implementierten Spracherkennungskomponente wird im Allgemeinen das gleiche Prinzip für das Logging verfolgt. Informationen zum Programmablauf werden in das File „ms-log-debug.txt“ gespeichert. Zur Runtime aufgetretene Fehler werden in das File „ms-log-error.txt“ geschrieben. Zusätzlich gibt es auch noch ein weiteres File mit dem Namen „ms-log-result.txt“; hier werden alle Ergebnisse der Spracherkennung aufgezeichnet. Alle drei Text-Files sind im Root-Ordner des Treibers zu finden und zu Beginn des Programmablaufs wird in alle Files der aktuelle Zeitstempel eingetragen.

6. Evaluierung

Dieses Kapitel umfasst die Evaluierung des im Rahmen dieser Arbeit entwickelten Treibers für die Bedienung des LUI mittels Spracherkennung. Das umfasst sowohl die Kernkomponente des Treibers als auch die Anwendungen zur Spracherkennung.
Dabei wurden im ersten Teil die im Zuge der Entwicklung des Treibers erstellten Applikationen zur Spracherkennung mit zwei der von Microsoft angebotenen Speech APIs (Microsoft Speech Platform und Windows Speech) getestet.
Des Weiteren wurde im zweiten Teil der Evaluierung die erarbeitete und implementierte Gesamtlösung des Treibers für die Spracherkennung für das LUI untersucht. Im Gegensatz zu den Unit-Tests, die im Rahmen des vorherigen Kapitels durchgeführt wurden, handelt es sich hierbei um Integrations-Tests. Das bedeutet, dass der komplette Treiber hinsichtlich seiner Funktionsweise, sowohl aus technischer Sicht, als auch aus dem Blickwickel von Anwenderinnen und Anwendern getestet wurde.

6.1. Zielsetzung der Evaluierung

Bei der Evaluierung der beiden Speech APIs von Microsoft sollen die Erkennungsraten[27] unter verschiedenen Störeinflüssen, sowie mit unterschiedlichen Anwenderinnen und Anwendern getestet und verglichen werden. Da die API der Microsoft Speech Platform die ältere Technologie ist und die ursprünglich für diese Arbeit konzipierte Windows Speech API ersetzt, soll der direkte Vergleich zwischen den beiden Schnittstellen zur Spracherkennung im Vordergrund stehen, nicht die absoluten Werte beziehungsweise Erkennungsraten.
Bei diesem Vergleich soll neben der grundsätzlichen Erkennung der Spracheingaben besonders auf die Frequenz von „false-positives“ geachtet werden (das heißt, wie oft die Spracherkennung eine Eingabe fälschlicher Weise als ein gültiges Kommando, oder als ein anderes Kommando erkennt). Anhand, der in der folgenden Auflistung dargestellten Teile, dieser Evaluierung soll der Vergleich der beiden Speech APIs durchgeführt werden:

- Eingabe von Kommandos ohne additive Störungen
- Eingabe von Kommandos mit additiven Störungen
- Eingabe ausschließlich von Störungen

Im Rahmen der Evaluierung der Gesamtlösung (des kompletten Prototypen) soll die Funktionsweise des Treibers im Dauerbetrieb getestet werden. Konkret wird dabei auf die folgenden Punkte Augenmerk gelegt:

- Starten der Komponenten des Treibers
- Laden der Einstellungen aus dem Konfigurationsfile
- Empfangen und Verarbeiten der Kommandos (Menu-Entries) des LUI
    - Setzen der Constraints
- Übermittlung der erkannten Kommandos
- Verhalten bei Dauerbetrieb und längeren Standby-Zeiten
- Verhalten bei Fehlerfällen und Beenden oder Neustart des LUI

[27] In dieser Evaluierung wird unter Erkennungsrate der durchschnittliche Confidence-Value verstanden mit dem ein Kommando erkannt wird. Eine fehlerhafte Erkennung fließt dabei mit dem niedrigsten Confidence-Value – 0 – in die Ergebnisstatistik ein.

Die Ergebnisse aller durchgeführten Messungen und Tests sollen Aufschluss über die Alltagstauglichkeit des implementierten Treibers, sowie die möglichen Einsatzbereiche geben.

6.2. Vergleich der Microsoft APIs für die Spracherkennung

6.2.1. Setup für die Evaluierung

Für die Messungen im Rahmen dieser Evaluierung kamen die bereits im Laufe der Prototypen-Implementierung entwickelten Anwendungen für die Spracherkennung zum Einsatz. Dabei handelte es sich bei der Microsoft Speech Platform um jene Applikation, die auch als Komponente für die Gesamtlösung implementierte wurde.
Für die Windows Speech API wurde die Universal Windows Platform Application eingesetzt, die im ersten Konzeptentwurf für den Treiber zum Einsatz hätte kommen sollen, aber aus den bereits angeführten Gründen (Kapitel 5.2.4.), welche auch bei der Durchführung dieser Tests eine Herausforderung darstellten, nicht eingesetzt werden konnte.
Beide Programme sind so konfiguriert, dass sie sofort nach einem registrierten Input die Spracherkennung beenden und ein finales Ergebnis liefern, also keine kontinuierliche Spracheingabe zulassen. Des Weiteren wird ausschließlich mit einer offline Spracherkennung und vordefinierten Constraints gearbeitet. Die Constraints für die Spracherkennung enthalten immer den jeweiligen Kontext (die jeweiligen Kommandos) der LUI-Seite, die das zu testende Kommando enthält. Dieses Setup entspricht genau jener, der in der Gesamtlösung enthaltenen Komponente zur Spracherkennung, und dient auch dazu, die beiden Technologien zur Spracherkennung unter den gleichen Preconditions testen zu können.

Zur Aufnahme der Sound-Samples der Probandinnen und Probanden wurde das Mikrofon der externen Audio-Hardware Jabra Speak 410[28] genutzt. Mit der kostenfrei erhältlichen Audio-Software Audacity[29] wurden die Aufzeichnungen gespeichert, sowie in weiterer Folge zugeschnitten und den Test-Anforderungen entsprechend verarbeitet. Für die Audio-Aufnahmen saßen die Probandinnen und Probanden bei einem Tisch, auf dem das Mikrofon in einer Entfernung von circa 30 bis 40 Zentimeter vom Mund entfernt (schräg unterhalb des Kinns) aufgestellt wurde.
Um Qualitätsverluste durch das Abspielen und erneute Erfassen der aufgezeichneten Spracheingaben mittels eines Mikrofons zu vermeiden, wurde die ebenfalls frei erhältliche Software Voicemeeter eingesetzt. Dieses Programm kann in den Audio-Einstellungen des Betriebssystems als Output-Device konfiguriert werden und fungiert als virtueller Input (virtuelles Mikrofon). Somit ermöglicht es, den abgespielten Audio-Stream direkt wieder als Input-Stream zu nutzen, ohne eine tatsächliche (physische) Output- oder Input-Hardware einsetzen zu müssen.

[28] https://www.jabra.com.de/business/speakerphones/jabra-speak-series/jabra-speak-410
[29] http://www.audacity.de/

Für die Evaluierung wurden die folgenden 15 Kommandos getestet:

1. Lui Hilfe
2. Lui Unterhaltung
3. Lui Information
4. Lui lade Nachrichten
5. Lui zurück
6. Lui Internet
7. Lui Wetter
8. Lui Handbuch
9. Lui soziale Netze
10. Lui berühre mich
11. Lui Telefon
12. Lui Spiele
13. Lui Start
14. Lui Audio
15. Lui Tasten

6.2.2. Durchführung der Evaluierung

Wie bereits erwähnt, steht bei diesen Tests nicht die absolute Erkennungsrate im Vordergrund, sondern der Vergleich zwischen der Microsoft Speech Platform und der neueren Windows Speech API. Das soll Aufschluss darüber geben, ob es lohnenswert sein könnte, nach einer Möglichkeit zu suchen, um die Windows Speech API doch auch in diesem Kontext (als Komponente des Treibers für das LUI) nutzen zu können.

Für die Evaluierung wurden 15 Kommandos ausgewählt, die für die Navigation durch die Hauptfunktionen des LUI essentiell sind. Als Spracheingaben für die Tests wurden von acht verschiedenen Probandinnen und Probanden Aufnahmen der Kommandos gemacht. Es handelte sich dabei um vier männliche und vier weibliche Personen im Alter zwischen 23 und 62 Jahren. Jedes Kommando wurde pro Person drei Mal aufgezeichnet, um auch die Varianz in Aussprache und Betonung, die schon bei ein und derselben Person auftreten kann, zu berücksichtigen. Außerdem wurden zwei Tonspuren ausschließlich mit additiven Störgeräuschen aufgezeichnet. Unter additiven Störgeräuschen sind zum Beispiel Gespräche, Geräusche und sonstigen Störeinflüssen im Hintergrund, die nicht zur tatsächlichen Eingabe eines Kommandos gehören, zu verstehen. Eine dieser beiden Aufnahmen enthielt die Aufzeichnung einer Nachrichten-Fernsehsendung - und somit auch gesprochenen Text - während die andere ausschließlich Geräusche von Haushaltsgeräten, wie das Zuschlagen von Türen/Fenstern sowie Straßenlärm, der durch das offene Fenster kam, beinhaltete.

Für die Evaluierung wurde jede der Aufnahmen über die Software Audacity abgespielt und mittels des Programms Voicemeeter wieder als virtueller Input für die Spracherkennungen eingespielt. Dieser Vorgang war bei beiden Applikationen gleich, eine Unterscheidung beim Testablauf gab es nur bei den Anwendungen selbst.
Das in einer Executable-Datei ausgeführte Programm, das die Microsoft Speech Platform nutzt, konnte dauerhaft im Hintergrund laufen und gab automatisch alle Ereignisse in dem Konsolenfenster der Anwendung aus, beziehungsweise schrieb sie in eine Log-Datei (Text-Datei) im Root-Verzeichnis des Programms. Die Ergebnisse umfassten sowohl das Kommando, dass erkannt wurde, als auch den zugehörigen Confidence-Value.

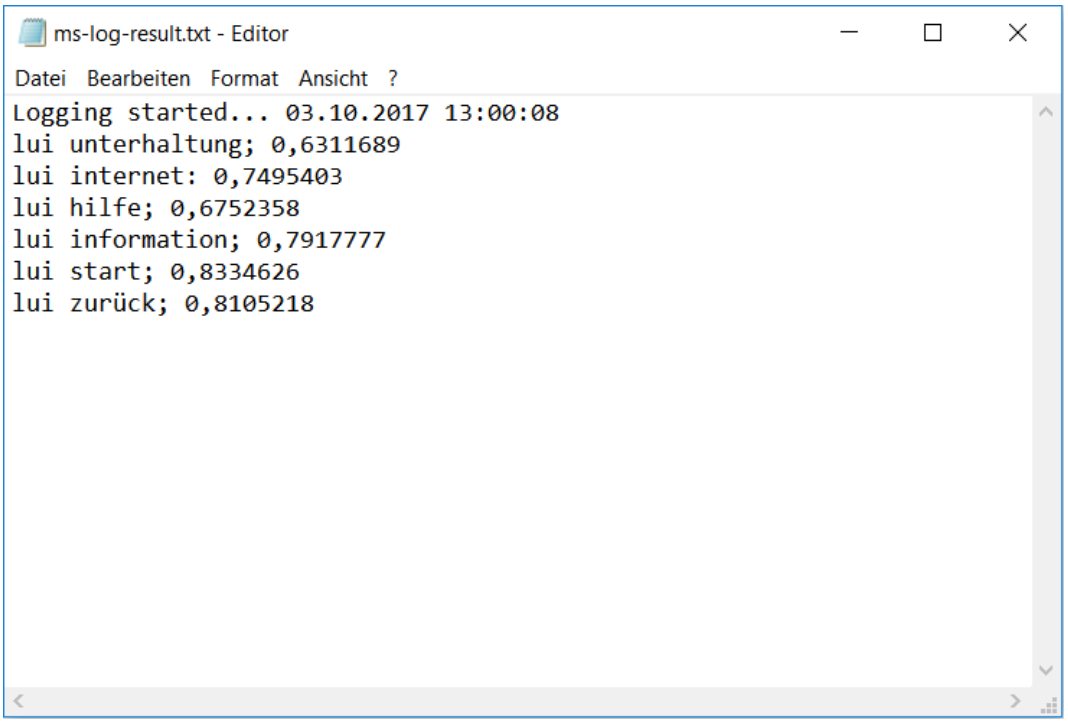


Abbildung 25: Screenshot des Log-Files mit den erkannten Ergebnissen der Spracherkennungsapplikation mit der Microsoft Speech Platform.

Die Universal Windows Platform Application mit der Windows Speech API geht hingegen in einen Standby-Zustand (Idle-Zustand), sobald sie nicht mehr im Vordergrund steht, beziehungsweise wird die Applikation von der Windows Runtime zu einem Idle-Zustand gezwungen. Somit werden alle von der Applikation genutzten Ressourcen wieder freigegeben und es ist nicht mehr möglich, auf den Audio-Input zuzugreifen, und somit ist auch kein Zugriff mehr auf den Audio-Input möglich. Diese Problematik musste manuell umgangen werden: Nach jedem Start des Abspielens einer Aufzeichnung wurde die Applikation wieder in den Vordergrund geholt und die Spracherkennung manuell gestartet. Dafür besitzt die UWP-App eine GUI (Graphic User Interface), die nicht nur zum Starten und Stoppen der Aufzeichnung dient, sondern auch ein Textfeld besitzt, in dem die erkannten Kommandos inklusive ihrem Confidence-Value ausgegeben, beziehungsweise angezeigt werden.

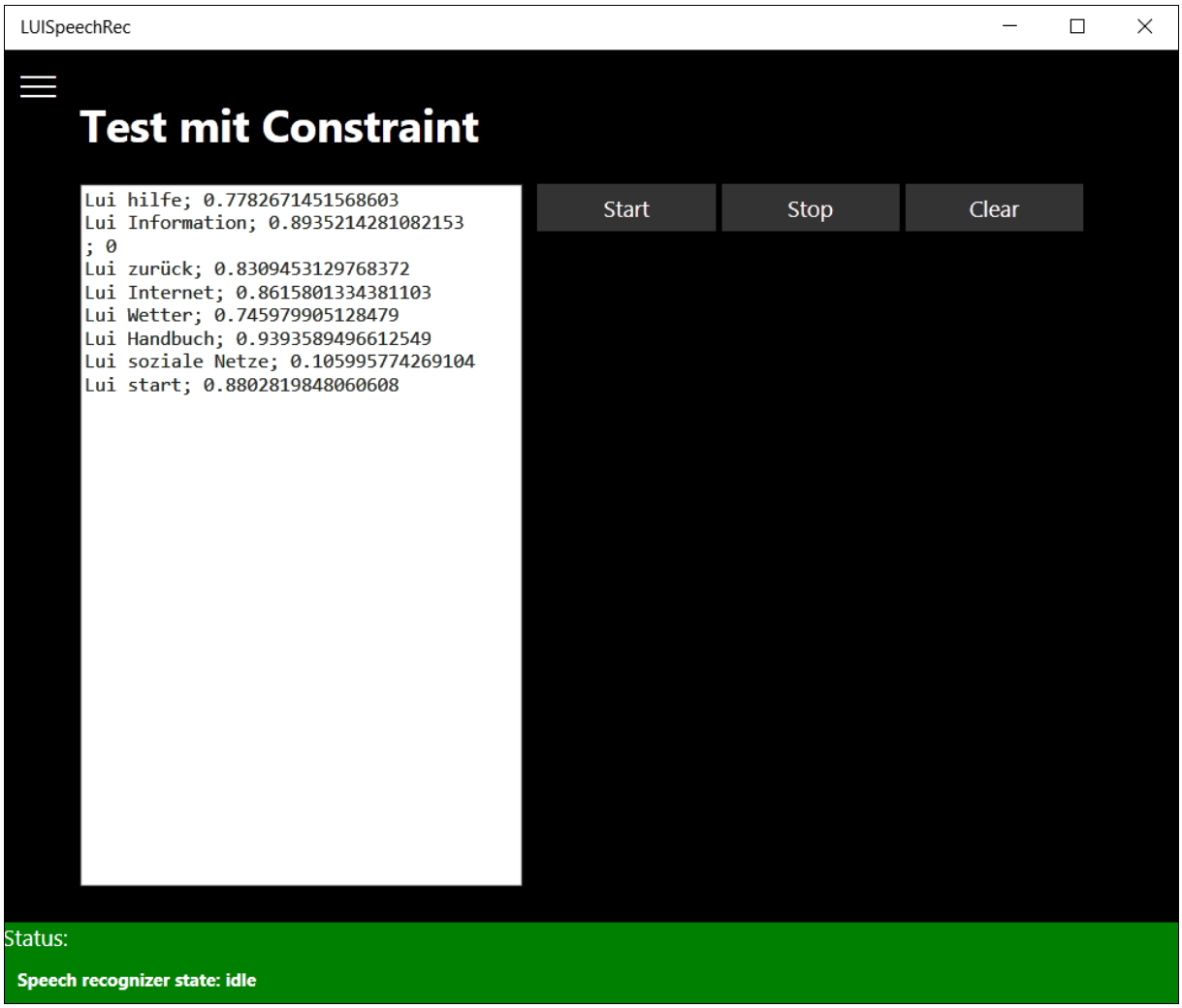


Abbildung 26: Screenshot der GUI der Windows Speech API App mit den Speech Recognition Ergebnissen

Sowohl bei der Microsoft Speech Platform, als auch bei der Windows Speech API, werden für die Ergebnisse der Spracherkennung Confidence-Values zwischen 0 und 1 zurückgeliefert. Ein Confidence-Value von 1 entspricht dabei dem höchst möglichen Wert und steht somit für das höchste Maß an Wahrscheinlichkeit, dass dieses Ergebnis – im Vergleich zu den anderen Ergebniskandidaten mit niedrigerem Confidence-Value – der tatsächlichen Spracheingabe entspricht. Diese Werte wurden für den Vergleich bei dieser Evaluierung herangezogen. Zu beachten ist allerdings, dass diese Skalierung laut der Dokumentation von Microsoft[30] nicht direkt mit Prozent-Angaben, wie sie oft bei anderen Speech-Recognition-Engines üblich sind, korreliert.

[30] https://msdn.microsoft.com/en-us/library/microsoft.speech.recognition.recognizedphrase.confidence(v=office.14).aspx

6.2.2.1. Ergebnisse der „normalen“ Kommandos:

Für diesen Teil der Evaluierung wurden ausschließlich die originalen Aufnahmen der Kommandos abgespielt, welche keine additiven Störgeräusche enthielten, sondern ausschließlich die Spracheingaben der Probanden und Probandinnen.
Es soll an dieser Stelle jedoch festgehalten werden, dass zum Teil starke Varianzen bei den Conficence-Values der Probandinnen und Probanden festgestellt werden konnten. Folgende Boxplots zeigen die Varianzen der gesprochenen Kommandos einer männlichen Testperson. Die Ergebnisse der Microsoft Speech Platform (*Abbildung 27*) sowie der Windows Speech API (*Abbildung 28*) werden wie folgt dargestellt:

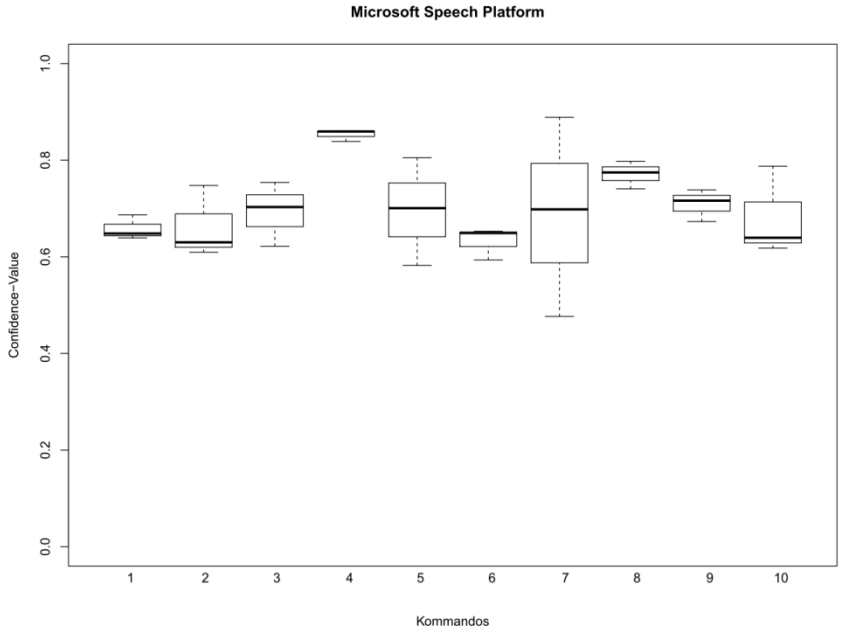


Abbildung 27: Boxplot der Confidence-Value-Varianzen bei einer einzelnen Testperson bei Verwendung der Microsoft Speech Platform Spracherkennung

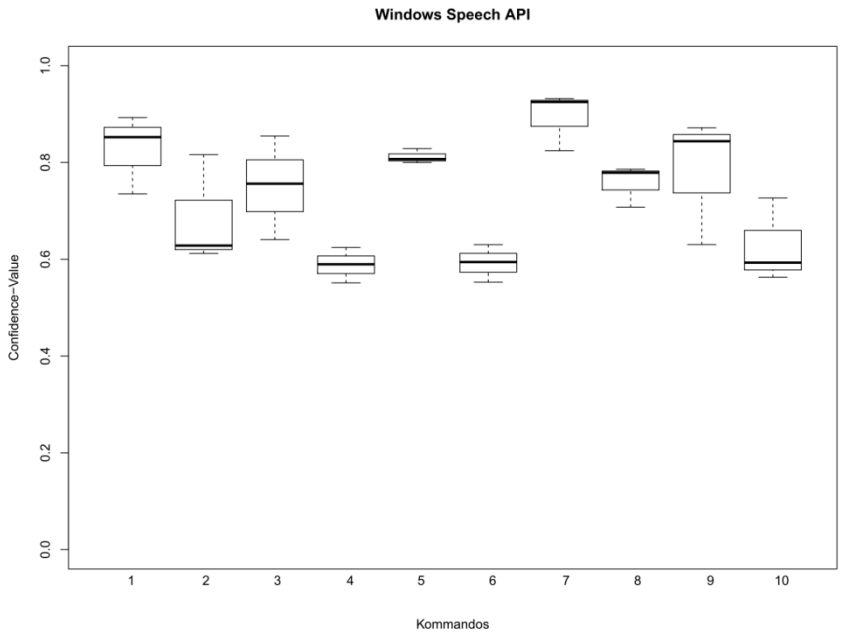


Abbildung 28: Boxplot der Confidence-Value-Varianzen bei einer einzelnen Testperson bei Verwendung der Windows Speech API Spracherkennung

Im nachfolgenden Diagramm ist eine Gegenüberstellung der Mittelwerte der Confidence-Values über alle Kommandos, gesprochen von allen Testpersonen, zu sehen. Hierfür wurden die drei unterschiedlichen Aufnahmen der ausgewählten 15 Kommandos pro Testperson, sowohl von der Microsoft Speech Platform als auch der Windows Speech API ausgewertet und für die Gesamtheit der Ergebnisse der Mittelwert berechnet.

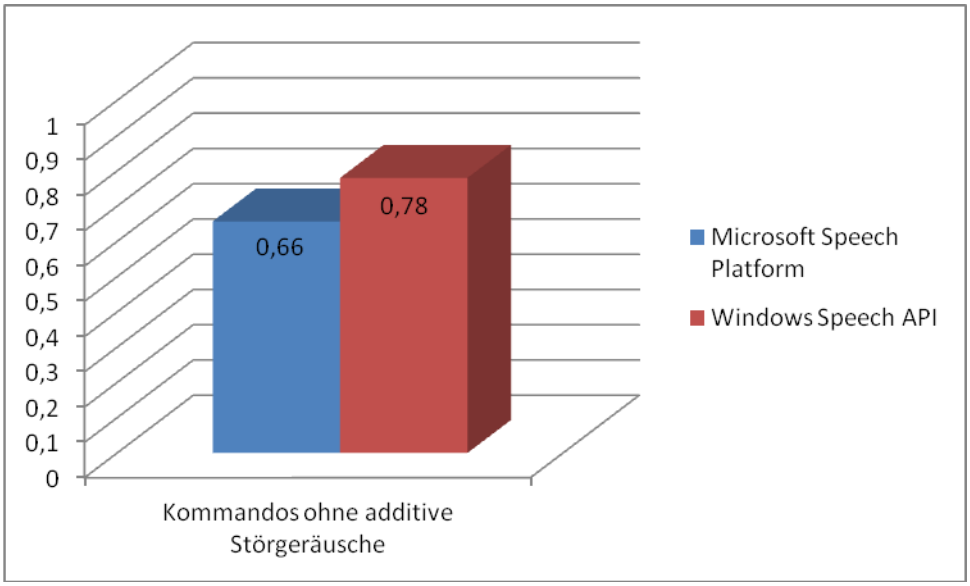


Abbildung 29: Diagramm der Confidence-Values von Kommandos ohne additive Störgeräusche

Wie im Diagramm zu erkennen ist, war die Windows Speech API, mit den gleichen Constraints und im selben Kontext, mit einem durchschnittlichen Confidence-Value von 0,78

zuverlässiger als die ältere Microsoft Speech Platform. Das spiegelte sich auch bei einzelnen Eingaben wieder, da die Windows Speech API auch bei allen Mittelwerten der einzelnen Kommandos einen höheren Confidence-Value aufweisen konnte als die Microsoft Speech Platform.

6.2.2.2. Ergebnisse der Kommandos in unterschiedlichem Kontext

In diesem Abschnitt sind die Ergebnisse von zwei unterschiedlichen Kommandos angeführt, die in unterschiedlichem Kontext getestet wurden. Dabei wurden die Erkennungsraten der Kommandos „Hilfe“ und „zurück“ einmal im Kontext der LUI-Seite „Unterhaltung“ und einmal im Kontext der LUI-Seite „Internet“ erfasst und analysiert. Auch bei diesen Tests wurden für ein repräsentatives Ergebnis für jedes Kommando die drei unterschiedlichen Aufnahmen der Probandinnen und Probanden abgespielt.
Diese Evaluierung wurde durchgeführt, um mögliche Verwechslungen zwischen Kommandos zu prüfen, beziehungsweise um zu testen, wie stark unterschiedlicher Kontext die Confidence-Values eines Kommandos beeinflussen kann.

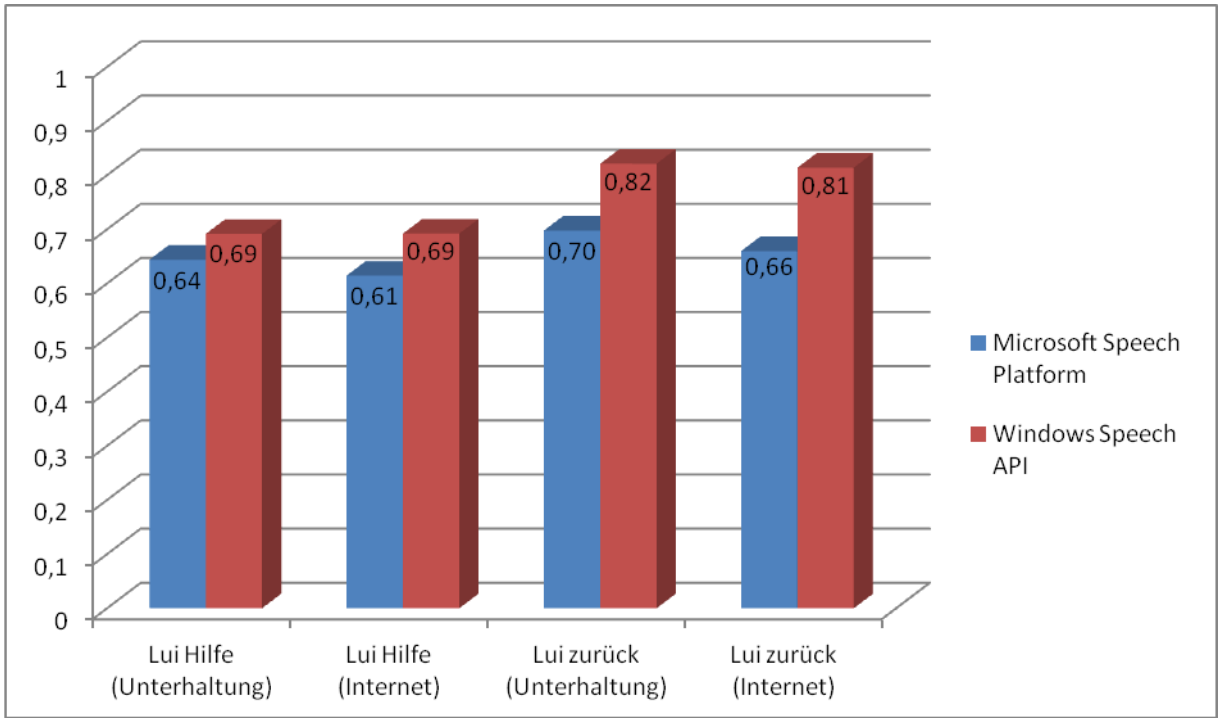


Abbildung 30: Diagramm der Confidence-Values von Kommandos in unterschiedlichem Kontext

Bei dieser Evaluierung konnte kein signifikanter Unterschied bei den Erkennungsraten der aufgenommenen Kommandos in unterschiedlichem Kontext festgestellt werden. Es kam weder zur Erkennung von einem falschen Kommando noch zu ergebnislosen Erkennungsversuchen. Allerdings sollte an dieser Stelle festgehalten werden, dass alle Probandinnen und Probanden bei der Aufnahme der Sprachkommandos um eine deutliche Aussprache bemüht waren. Auf Grund der Erfahrungen, die der Autor dieser Arbeit im Rahmen der Entwicklung gesammelt hat, liegt allerdings die Vermutung nahe, dass es bei leiser Aussprache oder stärkerem Dialekt leichter zu Verwechslungen kommen könnte.

6.2.2.3. Ergebnisse der Kommandos mit Hintergrundgeräuschen

Für die Darstellung von realistischen Szenarien wurden neben den Sprachkommandos von den Test-Teilnehmerinnen und Teilnehmern auch additive Störgeräusche aufgenommen. Für diese Tests wurden wiederum zwei unterschieden Arten der Hintergrundgeräusche gewählt. Einmal enthielten die additiven Störgeräusche gesprochene Worte, wodurch ein Szenario mit anderen Menschen im gleichen Raum oder in der Nähe simuliert werden soll. Für diese Evaluierung wurde eine mit mittlerer Lautstärke im Fernsehen laufende Nachrichtensendung aufgenommen. Das Mikrofon wurde für diese Aufnahme in einem Abstand von circa drei Metern zum Fernsehgerät aufgestellt.
Für die zweite Form der Hintergrundgeräusche wurden ausschließlich Geräusche aufgenommen, die üblicherweise in einem Wohnhaus vorkommen können. Diese Aufnahme beinhaltete Hintergrundlärm in Form von Haushaltsgeräten, wie beispielsweise ein laufender Geschirrspüler, aber auch das Zustoßen von Schreibtischladen, das schwunghafte Schließen von Fenstern, Papierrascheln oder gedämpfter Straßenlärm, der durch das offene Fenster hereindringt.
Für die Untersuchung der Auswirkungen von additiven Störgeräuschen wurden alle Kommandos mit beiden Tonspuren, die Hintergrundgeräusche simulieren, getestet. Das bedeutet, dass in Audacity einmal die Tonspur mit der Aufnahme der Fernsehsendung und einmal jene, die ausschließlich Geräusche enthielt, parallel zu der Tonspur der Kommandos der Probanden und Probandinnen hinzugefügt und abgespielt wurde.
Den Ergebnissen muss vorweggenommen werden, dass für die Tests der Beginn der Tonspuren mit den additiven Störgeräuschen mit dem Beginn der Sprachkommandos synchronisiert wurde. Dies musste durchgeführt werden, da die Microsoft Speech Platform fast ausnahmslos schon vor Beginn des tatsächlichen Sprachkommandos eines der Worte aus der Fernsehsendung als gültiges Kommando erkannte (siehe nachfolgende *Abbildung 31*). Das hätte einen direkten Vergleich der beiden APIs zur Spracherkennung verhindert. Da aber auch die Erkennungsleistung bei den tatsächlichen Sprachkommandos gegenübergestellt werden sollen, wurden mit der Synchronisierung die dementsprechenden Bedingungen geschaffen.
Ein weiteres Problem bei den aus der Nachrichtensendung fälschlicherweise erkannten Kommandos war, dass zumeist auch ein Confidence-Value zugewiesen wurde, der ausreichend war, um eine Eingabe, beziehungsweise Aktion, im LUI auszulösen. Im Gegensatz dazu erkannte die Windows Speech API bei der Fernsehsendung im Hintergrund zwar auch immer wieder Spracheingaben, allerdings wurde bei dem Set an Aufnahmen, welches für die Vorabevaluierung herangezogen wurde, kein einziger Treffer (Match) von dieser Speech Engine geliefert. Folglich lieferte die Windows Speech API als Ergebnis einen leeren String mit einem Confidence-Value von 0.

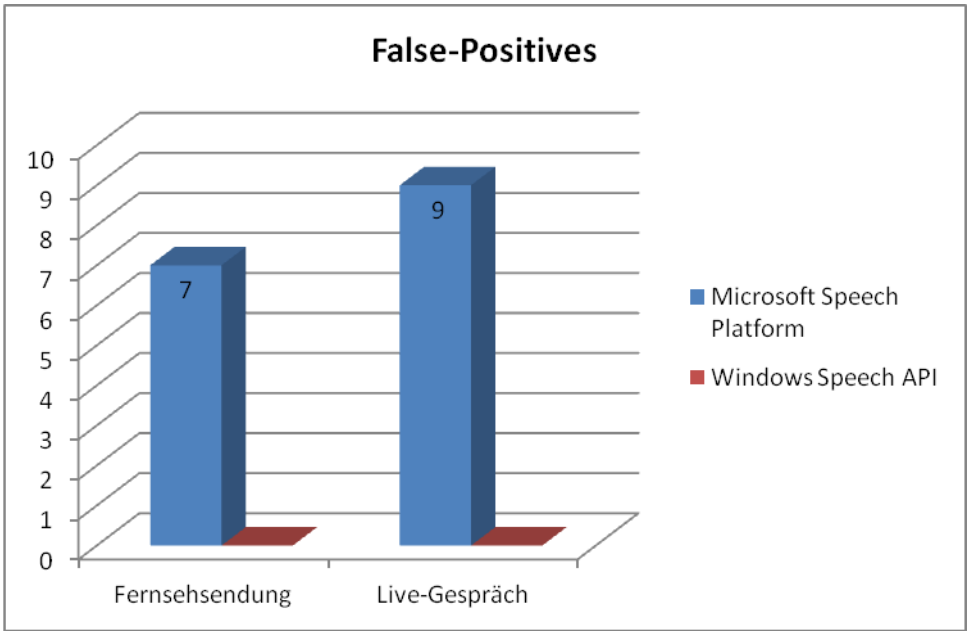


Abbildung 31: Diagramm der "False-Positive" Erkennung der Speech Recognition APIs

Die Tatsache, dass die Microsoft Speech Platform bei beliebiger kontinuierlicher Sprache im Hintergrund fälschlicherweise schon valide Eingaben erkannte, war ein deutliches Indiz dafür, dass diese Applikation für den Alltagsgebrauch schlechter geeignet ist. Des Weiteren wurden Kommandos häufig als gültig erkannt, obwohl das vorangestellte Commandword „LUI“ fehlte.

Im Weiteren wurde trotzdem noch ein direkter Vergleich gezogen, wie sich die Erkennungsraten der einzelnen Kommandos bei additiven Störgeräuschen verändern. Die Ergebnisse der Tests mit synchronisiertem Start der Tonspuren von Hintergrundgeräuschen und Sprachkommandos sind in dem nachfolgenden Diagramm zu sehen. Für einen übersichtlichen Vergleich wurden sowohl die Ergebnisse der Spracheingaben ohne Hintergrundgeräusche, als auch die Ergebnisse der beiden Tests mit den additiven Störgeräuschen, dargestellt.

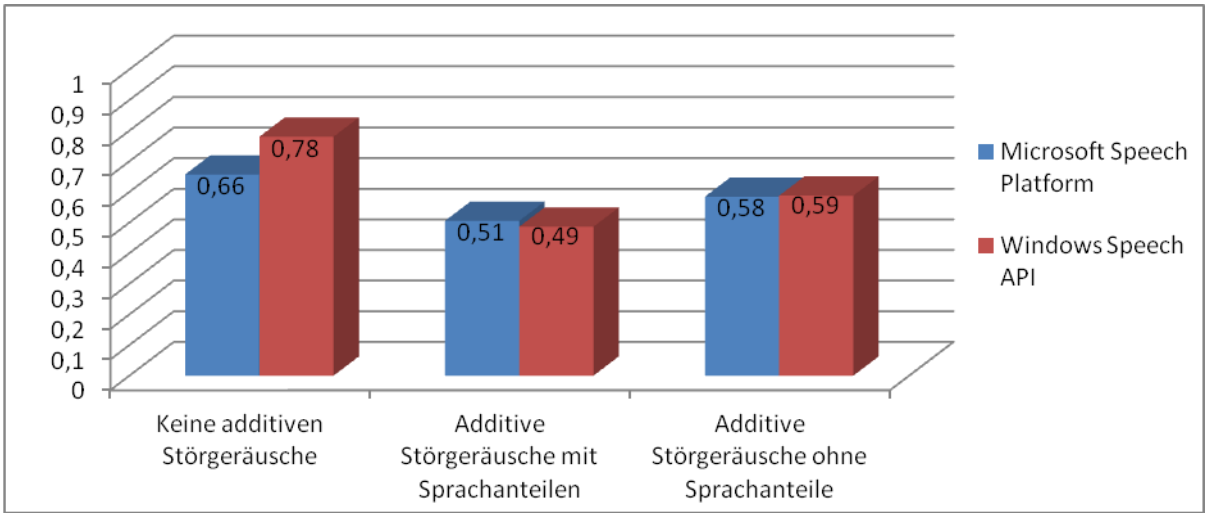


Abbildung 32: Übersicht der Confidence-Value-Mittelwerte bei unterschiedlichen äußeren Einflüssen

Anhand der Graphik ist zu erkennen, dass bei additiven Störgeräuschen das Niveau der Confidence-Values der Windows Speech API auf jenes der Microsoft Speech Platform sinkt. Nachdem die Windows Speech API bei den Spracheingaben ohne Hintergrundgeräusche höhere Confidence-Values aufweisen konnte, bedeutet das höhere Verluste in der Akkuratität der Erkennung von Kommandos mit additiven Störgeräuschen im Vergleich zur Microsoft Speech Platform. Wie bereits erwähnt wird bei dieser isolierten Evaluierung allerdings außer Acht gelassen, dass die Microsoft Speech Platform im Allgemeinen mit additiven Störgeräuschen, die im speziellen Sprachanteile enthalten, signifikant schlechter umgehen kann. Handelt es sich allerdings um allgemeinen Lärm ohne Anteile von menschlicher Sprache, liefern die Microsoft Speech Platform und die Windows Speech API Ergebnisse auf einem annähernd gleichen Niveau.

Existieren additive Störgeräusche, die synchron zu Spracheingaben laufen, waren die Confidence-Values geringfügig besser bei jenen Störgeräuschen, die keinen Sprachanteil enthielten. Allerdings waren speziell bei der Windows Speech API die Unterschiede zwischen den beiden unterschiedlichen Formen von Hintergrundgeräuschen im Vergleich zu dem Resultat ohne jegliche Hintergrundgeräusche innerhalb des Bereichs, der hinsichtlich der Schwankungsbreite, die schon bei ein und derselben Person auftreten kann, zu vernachlässigen.
Hingegen signifikant war, dass es je nach Störgeräusch Sprecherinnen und Sprecherabhängige Unterschiede gab. Für den Autor dieser Arbeit war dabei zu beobachten, dass frequenzabhängige Abweichungen auftraten. So waren bei eher hochfrequenten additiven Störgeräuschen die Erkennungsraten bei den weiblichen Probanden (deren Stimmen höhere Frequenzanteile aufweisen) durchgängig schlechter als bei männlichen Probanden (deren Stimmen niedrigere Frequenzanteile beinhalten) und ebenso in umgekehrter Hinsicht.

6.2.3. Zusammenfassung

Allgemein kann festgehalten werden, dass schon bei derselben Person, bei ein und demselben Kommando, Schwankungen beim Confidence-Value von bis 0,20 oder in Extremfällen auch mehr beobachtet werden konnten (*Abbildung 27* und *Abbildung 28*).
In Anbetracht dieser Varianzen sind die Ergebnisse der Spracherkennung bei additiven Störgeräuschen in einem vertretbaren Bereich. Hierbei gibt es allerdings signifikante Unterschiede zwischen der Microsoft Speech Platform und der Windows Speech API. Da die Microsoft Speech Platform eine erheblich höhere "false-positive" Rate aufweist und dazu neigt, bei jeglicher Sprache, die detektiert wird, ein Ergebnis zu liefern (auch wenn keine Kommandoworte enthalten sind), sollte, beziehungsweise muss ein höherer Confidence-Value gewählt werden, um eine akzeptable Alltagsnutzung gewährleisten zu können. Bei der Windows Speech API hingegen, wäre es deutlich unbedenklicher, einen niedrigeren Confidence-Value als Schwellwert zur Einstufung des Ergebnisses zu wählen, da in den Tests dieser API keine "false-positives" erkannt wurden. Dadurch kann mit dem Confidence-Value, der in der Config-Datei als Schwellwert gesetzt wird wesentlich flexibler auf Umgebungsgeräusche eingegangen werden. Die Windows Speech API wäre somit besser konfigurierbar für eine abwechselnde Nutzung der Spracherkennung von einer größeren Bandbreite unterschiedler Sprecherinnern und Sprecher.

Kontext-abhängige (unterschiedliche Kombinationen der Kommandos als Constraints) Schwankungen bei den Erkennungsraten konnten bei der Evaluierung, wenn überhaupt, nur in einem vernachlässigbaren Rahmen festgestellt werden und signifikante Ausreißer wurden nicht beobachtet.
Insgesamt kann festgehalten werden, dass sich die Windows Speech API als zuverlässiger herausstellte, besonders bei additiven Störgeräuschen, die Sprachanteile enthielten.

6.3. Evaluierung der Gesamtlösung des Treibers zur Spracherkennung für das LUI

Bei diesem Teil der Evaluierung wurde im Gegensatz zu den Unit-Tests nicht nur die Funktionsfähigkeit einzelner Komponenten getestet, sondern die entwickelte (Software-) Gesamtlösung. Das heißt, es wurde auch das Zusammenspiel aller Komponenten überprüft, sowie die Erfüllung von Qualitätskriterien, die für diese Software aufgestellt wurden. Dabei wurden allerdings keine eigentlichen Integrationstests durchgeführt, sondern anhand von Testprotokollen Abläufe abgearbeitet und deren Ergebnisse kontrolliert. Für die Nachvollziehbarkeit und Kontrolle der Tests wurden beispielsweise auch die Log-Files der einzelnen Komponenten des Treibers herangezogen. Mit Hilfe der in diesen Log-Files festgehaltenen Ereignisse und Ergebnisse konnten die Abläufe und Resultate der Tests belegt werden.

Die Evaluierung wurde entsprechend dem bereits beschriebenen Setup auf drei unterschiedlichen Geräten mit Windows 10 über einen Zeitraum von sieben Tagen durchgeführt. Dabei war das LUI über den gesamten Zeitraum aktiv und wurde von drei unterschiedlichen Personen wiederholt, gemäß dem Test-Protokoll Spracheingaben, bedient. Bevor das Testprotokoll ausführlicher dargestellt wird, werden die Voraussetzungen für die Durchführung der Tests festgehalten. Da das LUI selbst bislang noch nicht vollständig für die Spracherkennung eingerichtet wurde, fehlen für einige Buttons und Seiten des LUI noch Einträge, die als Kommandos für die Spracherkennung genutzt werden können. Das trifft vor allem auf Anwendungen zu, die ins das LUI integriert wurden, wie beispielsweise der Audio-Player oder diverse Spiele. Eine weitere Einschränkung war, dass bei den Menu-Entries der Seite „Audio“ eine fehlerhafte UTF-8 Codierung für den Eintrag des Radios Ö1 vom LUI versendet wurde und die Einträge dieser Seite deshalb nicht mehr decodiert werden konnten. An dieser Stelle kam die für solche Fälle in der Kernkomponente des Treibers implementierte Fallback-Entrylist zum Einsatz, welche die Kommandos „Hilfe“, „zurück“ und „Start“ enthält.

Test-Protokoll:

- Startvorgang
  ➔ LUI starten
    - Automatisches Starten der Kernkomponente des Treibers
    - Ist der initiale Listenstate gleich „true“?
    - Wurden Settings und Menu-Entries empfangen?
    - Wurde im LUI angezeigt, dass die Spracherkennung momentan nicht verfügbar ist?
    - Wurde die Komponente zur Spracherkennung gestartet?
    - Wurden Settings empfangen?
    - Wurden Menu-Entries empfangen?
    - Wurde die Spracherkennung gestartet und eine Grammatik geladen?
    - Wurde im LUI angezeigt, dass die Spracherkennung jetzt verfügbar ist?

- Bedienung/Steuerung mittels Spracheingaben
  Für jede Eingabe müssen die folgenden Checkpoints überprüft werden:
    - War die erkannte Spracheingabe tatsächlich von der Anwenderin oder dem Anwender?

- Wurde der Confidence-Threshold überschritten, falls ja wurde das richtige Kommando von der Applikation zur Spracherkennung versandt?
- Wurde das Kommando richtig an das LUI weitergeleitet?
- Wurden neue Menu-Entries von der Kernkomponente empfangen und an die Spracherkennungskomponente weitergeleitet?
- Wurde die Entrylist empfangen und eine neue Grammatik geladen?

Getätigte Eingaben:

➔ „Lui berühre mich“
➔ „Lui Unterhaltung“
➔ „Lui zurück“
➔ „Lui Information“
➔ „Lui Internet“
➔ „Lui Start“
➔ „Lui Telefon“
➔ „Lui Tasten“
➔ „Lui zurück“
➔ „Lui zurück“
➔ „Lui Unterhaltung“
➔ „Lui soziale Netze“
➔ „Lui Hilfe“
➔ „Nein keine Hilfe“

- Verändern des Listenstates im laufenden Betrieb
  - War die erkannte Spracheingabe tatsächlich von der Anwenderin oder dem Anwender?
  - Wurde der Confidence-Threshold überschritten? Falls ja, wurde das richtige Kommando von der Applikation zur Spracherkennung versandt?
  - Handelt es sich um ein Kommando zum Aktivieren oder Deaktivieren der Spracherkennung?
    - Falls ja:
      - Wurde der Listenstate korrekt verändert?
      - Wurde eine Display-Message an das LUI übermittelt?
      - Wurde der WebSocket-Client über den geänderten Status benachrichtigt?
      - Wurde entsprechend dem Status eine Entrylist an den WebSocket-Client übermittelt (falls konfiguriert mit den jeweiligen Sprachkommandos für die Aktivierung und Deaktivierung der Spracherkennung)?
  - Ist die Spracherkennung aktuell aktiviert („listenState = true“)?
    - Falls ja:
      - Wurde das Kommando richtig an das LUI weitergeleitet?
      - Wurden neue Menu-Entries von der Kernkomponente empfangen und an die Spracherkennungskomponente weitergeleitet?
      - Wurde die Entrylist empfangen und eine neue Grammatik geladen?

    - Falls nein:
      - Wurde keine weitere Aktion unternommen?

Getätigte Eingaben:

- ➔ „Lui berühre mich“
- ➔ „Lui Unterhaltung“
- ➔ „Lui Spracherkennung deaktivieren“
- ➔ „Lui zurück“
- ➔ „Lui Information“
- ➔ Lui Spracherkennung aktivieren“
- ➔ „Lui zurück“
- ➔ „Lui Information“
- ➔ „Lui Internet“
- ➔ *Befehl/Nachricht von Lui zur Deaktivierung der Spracherkennung*
- ➔ „Lui Start“
- ➔ „Lui zurück“
- ➔ *Befehl/Nachricht von Lui zur Aktivierung der Spracherkennung*
- ➔ „Lui Start“
- ➔ „Lui Telefon“
- ➔ „Lui Tasten“
- ➔ „Lui zurück“
- ➔ „Lui zurück“
- ➔ „Lui Hilfe“
- ➔ „Nein keine Hilfe“

- Beenden des LUI beziehungsweise in Folge des Treibers (der Gesamtlösung)
  - Wurde die Exit Nachricht in der Kernkomponente erkannt?
  - Wurde ein Exit der Applikation zur Spracherkennung eingeleitet bzw. diese in Folge beendet?
  - Wurde die Kernkomponente beendet?

Ergebnisse

Zunächst werden jene Ergebnisse dargestellt, die vor allem im Zuge der Gesamtlösung des Treibers zur Steuerung des LUI mittels Spracherkennung relevant sind. Darüber hinaus werden auch jene Ergebnisse dargestellt, die aufgrund äußerer Bedingungen in dieser Forschungsarbeit nicht veränderbar oder beeinflussbar waren. Nichtsdestotrotz wurden auch diese Ergebnisse bei der Evaluierung berücksichtigt und werden nachfolgend festgehalten.

In keinem der Testläufe auf den drei unterschiedlichen Geräten kam es im Zeitraum der Evaluierung zu Auffälligkeiten hinsichtlich des Programmablaufs der Treiberkomponenten. Das bedeutet, es sind keine Fehler aufgetreten, die zu einer Terminierung oder zu einem Fehlverhalten der Software (Kernkomponente oder Spracherkennungskomponente des Treibers) geführt haben, beziehungsweise mussten auch die implementierten Mechanismen zum Error-Handling nicht zum Einsatz kommen.
Die einzige Auffälligkeit die beobachtet werden konnte war, dass das LUI bei einem Test-Lauf nach circa fünf Tagen mit der Fehlermeldung „Das LUI User Interface funktioniert nicht mehr, das Programm musste beendet werden“ geschlossen wurde. Der Treiber hat sich bei diesem Zwischenfall jedoch wie vorgesehen ebenfalls beendet.

Die Testung der Reaktions- und Verarbeitungszeit der Treiberkomponenten bei der Spracheingabe verlief einwandfrei. Bei einem korrekten Kommando und dem erforderlichen Confidence-Value wurde das Kommando an das LUI umgehend versandt und ausgeführt, sodass für den Benutzer oder die Benutzerin keine Verzögerungen auftraten, die den Bedienungsfluss beeinträchtigen würden. Die rasche Verarbeitung der Kommandoeingaben lässt sich anhand des optisches sowie des akustisches Feedbacks des LUIs gut nachvollziehen.
Auch die Liste der neuen Kommandos wurde umgehend in der Applikation zur Spracherkennung geladen und stand direkt nach dem Erscheinen der neuen LUI-Seite für die Spracherkennung zur Verfügung. Das geschah sogar so schnell, dass eine geübte Anwenderin oder ein geübter Anwender praktisch gleichzeitig mit dem optischen Erscheinen der neuen LUI-Seite ein Kommando von dieser Seite aussprechen konnte und dieses erfolgreich ausgeführt wurde. Auch die Funktion zur Aktivierung und Deaktivierung der Spracherkennung (Veränderung des Listenstates) verhielt sich wie gewünscht, sowohl bei der Verwendung von Sprachkommandos, als auch bei Aufrufen über die REST-Schnittstelle (REST Calls).

Ein Hindernis, das auftauchte, war, dass das LUI beispielsweise auf der Seite des integrierten Internetbrowsers zeitweise einen Script-Fehler in einer Message-Box anzeigte, welche nur durch eine Bestätigung geschlossen werden konnte. Nachdem diese Nachricht nicht in einem Fenster des LUI sondern in einem allgemeinen System-Fenster angezeigt wird, ist die Bedienung mittels Spracheingabe jedoch nicht möglich. Das ist jedoch ein Mangel, der in der Software des LUI gelöst werden müsste und aus diesem Grund hier nicht weiter erforscht wurde.

Hinsichtlich der Qualität der Spracherkennung wurde im Verlauf der Testläufe beobachtet, dass leider selbst in einer geräuscharmen Umgebung teilweise schlechtere Erkennungsraten erreicht wurden als jene, die in den Resultaten der Evaluierung der Spracherkennungen zu sehen sind. Das ist zum einen vermutlich darauf zurückzuführen, dass bei regelmäßiger Nutzung im Alltag die Betonung und Deutlichkeit bei der Aussprache der Kommandos schwankt. Zum anderen könnten auch unterschiedlichen Entfernungen und Positionen zum Mikrofon von den Anwenderinnen und Anwendern dazu beigetragen haben, dass manche Eingaben schlechter erkannt wurden als andere. Außerdem traten über den Zeitraum der Testläufe auch immer wieder diverse, nicht beeinflussbare und in ihrer Lautstärke schwankende, Hintergrundgeräusche auf, die Einfluss auf die Erkennungsraten hatten.
Schlechte Erkennungsraten im Verlauf der Tests zeigten außerdem die Problematik auf, dass für die Anwenderin oder den Anwender teilweise unklar ist, ob das Sprachkommando nicht erkannt wurde oder ob die Spracherkennung möglicherweise nicht funktioniert. Im Rahmen dieser Arbeit wurde allerdings bewusst auf ein Feedback verzichtet. Das hat einerseits den Grund, dass es bereits im LUI ein akustisches Feedback beim Ausführen eines Kommandos gibt, wodurch ein weiteres akustisches Feedback bei nicht erkannten Eingaben für Verwirrung sorgen könnte. Andererseits wäre ein optisches Feedback nur über das LUI und dessen Display-Message-Funktion möglich, was beispielsweise bei Hintergrundgeräuschen oder ähnlichem dazu führen könnte, dass ständig Nachrichten angezeigt würden, wodurch das Arbeiten mit dem LUI mehr oder weniger unmöglich wäre.

Ein Problem, welches sich ebenfalls im Laufe der Tests herauskristallisierte, war, dass bei einem Hilfe-Ruf eine Seite im LUI aufgerufen wird, auf der dieser Hilferuf bestätigt werden muss. Dafür wird auch akustisch mittels einer Sprachausgabe nachgefragt, ob ein Hilferuf abgesetzt werden soll. Zu diesem Zeitpunkt ist die Spracheingabe aber schon wieder aktiv und erkennt diese Sprachausgabe des LUI als neue Spracheingabe. Eine ähnliche Proble-

matik ergibt sich bei der Telefonfunktion, bei der die Spracherkennung ständig mit Sprach-Input konfrontiert ist, wodurch sich die Wahrscheinlichkeit für fälschlicherweise erkannte Kommandos erhöht. An dieser Stelle müsste eine konzeptuelle Entscheidung getroffen werden, ob das LUI (falls der Treiber zur Spracherkennung aktiv ist) selbst keine Audio-Ausgaben tätigt oder solche Einzelfälle vom Treiber explizit anders behandelt werden sollen. Da nur das LUI alle notwendigen Informationen für diese Situationen zur Verfügung hat, könnte an dieser Stelle vom LUI beispielsweise die Funktion genutzt werden, den Listenstate des Treibers zu verändern. Das bedeutet mittels, eines Befehls (via REST Call) die Spracherkennung des Treibers zu deaktivieren, beziehungsweise sie zum passenden Zeitpunkt wieder zu aktivieren. Da es aber nicht Aufgabenstellung dieser Arbeit war, Änderungen im Progammablauf des LUI vorzunehmen, sollten diese Überlegungen als Anregungen für Weiterentwicklungen des Systems gesehen werden.

Zusammenfassend kann festgehalten werden, dass der implementierte Treiber zu Bedienung des LUI mittels Spracherkennung funktioniert und für unterschiedliche Sprecherinnen und Sprecher geeignet ist. Die Geschwindigkeit, mit der das LUI mit diesem Treiber gesteuert werden kann, beschränkt die Usability des LUI in keinster Weise. Es sind allerdings Verbesserungen bei der Komponente zur Spracherkennung von Nöten, um allgemein die Qualität der Spracherkennung zu verbessern, aber auch, um mit Sprache im Hintergrund besser umgehen zu können und die deshalb fälschlicherweise erkannten Kommandos zu vermeiden. Nach aktuellem Wissensstand ist es nicht möglich, auf den Quellcode des Speech Recognizers von Microsoft zuzugreifen, beziehungsweise solch tiefgreifende Veränderungen durchzuführen. Daher könnte ein Umstieg auf eine andere Technologie notwendig sein, um diese Qualitätsmerkmale erreichen zu können.
Des Weiteren sind auch Optimierungen seitens des LUI notwendig, um den vollen Funktionsumfang mittels Sprach-Kommandos zuverlässig zu gewährleisten.

7. Resümee und Ausblick

Im Rahmen dieser Arbeit wurde die Frage untersucht, ob es möglich ist, ein User-Interface am Beispiel des Local User Interface (LUI) um einen Treiber zur Bedienung mittels Spracherkennung zu erweitern und wie diese Erweiterung implementiert werden kann. Für die Überprüfung dieser Fragestellung wurde ein Treiber für das LUI entwickelt, der im Hintergrund läuft und die Spracherkennung, sowie die notwendige Logik für Verarbeitung von Menu-Entries enthält. Der eigens entwickelte Treiber besteht aus einer Kernkomponente und einer Spracherkennungskomponente. Der Treiber und seine Komponenten werden automatisch beim Öffnen des LUI gestartet und ausgeführt, beziehungsweise beenden sich selbstständig, wenn das LUI geschlossen wird. Während der gesamten Nutzung des LUI ist ausschließlich dessen GUI zu sehen - der Treiber zur Steuerung mittels Spracherkennung arbeitet komplett im Hintergrund. Das Starten und Beenden des Treibers, sowie andere Treiber-Einstellungen können zentral im Konfigurations-File des LUI getätigt werden.
Die Kernkomponente ist ein fixer Bestandteil des Treibers und übernimmt nicht nur die vollständige Kommunikation mit dem LUI, sondern auch mit der Spracherkennungsapplikation (Spracherkennungskomponente). Des Weiteren erfolgt in der Kernkomponente auch die Verarbeitung aller eingehenden Nachrichten und ggf. deren Weiterleitung, sowie die Kontrolle und Steuerung des aktuellen Status der Spracherkennung (aktiviert/deaktiviert). Um die Funktionsfähigkeit des gesamten Treibers zu gewährleisten, ist eine Spracherkennungskomponente zwingend notwendig. Welche Applikation für die Spracherkennung jedoch eingesetzt wird, ist optional und kann durch die klar definierten Interfaces schnell und einfach ausgetauscht werden. Steht keine Applikation zur Spracherkennung zur Verfügung oder ist diese Komponente vorübergehend nicht erreichbar (nicht funktionsfähig), wird die Anwenderin oder der Anwender über die GUI des LUI darüber informiert.
In der Implementierung dieser Arbeit wurde eine Spracherkennung basierend auf der Microsoft Speech Platform entwickelt, welche die aktuellen Menu-Entries des LUI als Grammar (Constraints) für die Speech Recognition Engine lädt. Im Rahmen der Tests (Unit-Test und Evaluierung) mit der Kernkomponente bewies diese aber auch, dass ihr Interface sehr generisch ist und die Möglichkeit bietet mit unterschiedlichsten Applikationen zur Spracherkennung gekoppelt zu werden. Dadurch können auch weitere Technologien zur Spracherkennung implementiert werden, deren Entwicklungsursprung nicht an Microsoft gebunden ist.

Anhand der durchgeführten Evaluierung der Gesamtlösung des Treibers konnte festgestellt werden, dass sich das LUI mittels der implementierten Technologien bedienen lässt. Hinsichtlich Eingabegeschwindigkeit konnten kaum Unterschiede zu andern Eingabetechnologien, wie einer Maus oder einem Touchscreen, ausgemacht werden.
Im Rahmen einer Evaluierung der Spracherkennungstechnologien wurde sowohl die Erkennungsrate der Microsoft Speech Platform als auch jene der Windows Speech API erhoben. Dabei wurde vor allem deutlich, dass die neuere Technologie der Windows Speech API erheblich besser mit additiven Störgeräuschen umgehen kann - insbesondere mit jenen, die Sprachanteile enthalten. Die Evaluierungsergebnisse deuten darauf hin, dass die Microsoft Speech Platform überwiegend versucht, jede Spracheingabe einer „Choice“ des Grammars zuzuordnen, anstatt ungültige Eingaben als solche auszuweisen. Dadurch werden Sprachkommandos möglich, an deren Anfang nicht das vordefinierte Kommandowort genannt wird. Das führt in weiterer Folge des Öfteren zu einer falschen Erkennung von Kommandos (false-

positives). Die Windows Speech API erkennt hingegen, dass es sich um eine ungültige Eingabe handelt und liefert ein dementsprechendes Resultat.
Die Erkenntnisse dieser Evaluierung liefern somit einen Anreiz für weitere Forschungsarbeiten und künftige Weiterentwicklungen auf dem Gebiet der Implementierung einer Spracherkennung mittels Windows Speech API für das LUI. Die Ergebnisse, welche im Rahmen dieser Arbeit gewonnen werden konnten, liefern aber auch eine wichtige Grundlage, um die Spracherkennung mit Microsoft Speech Platform weiterzuentwickeln und zu verbessern.
Bei der aktuellen Implementierung des Treibers würde der Autor empfehlen, eventuell ein Mikrofon einzusetzen, das einfach manuell deaktiviert werden kann, wenn keine Spracheingabe getätigt werden soll. Eine weitere Möglichkeit wäre die Spracherkennung zwar über Sprachkommandos deaktivieren zu können, die erneute Aktivierung aber ausschließlich über das LUI zu steuern (zum Beispiel mittels eines Buttons in der Benutzeroberfläche). Dies geht aus der Beobachtung hervor, dass sich die Microsoft Speech Platform bei Gesprächen im Hintergrund vermehrt fehlerhaft verhält, da diese oft fälschlicherweise als Kommandos erkannt werden (false-positives). Das führt dazu, dass ungewollt Befehle ausgeführt werden, die zur Folge haben können, dass im LUI Aktionen durchgeführt werden. Da die Applikation zur Spracherkennung mit dieser Technologie im Rahmen der Evaluierungen ansonsten aber gut funktioniert hat, wäre das ein interessanter Ansatzpunkt für Verbesserungen und Weiterentwicklungen, wie beispielsweise eine Verbesserung der „false-positive“ Rate.
Ein weiterer Vorschlag zur Weiterentwicklung der Applikation zur Spracherkennung wäre ein Konzept für ein User-Feedback zu entwickeln. Das bedeutet, dass die Anwenderin oder der Anwender eine Rückmeldung erhält, falls eine Eingabe nicht erkannt wurde oder sie nicht den gewünschten Confidence-Value erreicht hat.

Für eine reibungslose Bedienung des LUI mittels des Treibers zur Spracherkennung ist es außerdem notwendig, diverse Verbesserungen am LUI vorzunehmen. Der wichtigste Punkt wäre an dieser Stelle, für alle Seiten Menu-Entry-Einträge für die Spracherkennung zu ergänzen. Obwohl in der Datenstruktur des LUI hierfür schon bzgl. Des Vorbereitungen getroffen wurden, fehlen noch sehr viele Einträge – insbesondere für die, in das LUI integrierten Anwendungen, wie ein Audio-Player oder diverse Spiele – die noch ergänzt werden müssen.

Abschließend kann somit festgehalten werden, dass die eingangs definierte Forschungsfrage positiv beantwortet werden kann, da die Steuerung des LUI mittels Spracheingabe durch die Implementierung des Treibers im Zuge dieser Forschungsarbeit ermöglicht wurde. Die Ergebnisse, die dadurch erzielt wurden, lassen sich vor allem hinsichtlich der Geschwindigkeit, mit der das LUI durch den Treiber zur Spracherkennung bedient werden kann, mit klassischen Eingabegeräten wie Maus, Tastatur oder Touchscreen, vergleichen. Die generellen Ergebnisse der Spracherkennung mit der Microsoft Speech Platform erwiesen sich trotz der Erkennung von „false-positives“ ebenfalls als zufriedenstellend. Die Defizite, welche durch die Evaluierungen offengelegt wurden, bilden die Grundlage für weitere Forschungsarbeiten und tragen damit zur Weiterentwicklung und Professionalisierung der Bedienung eines User-Interface wie das LUI mittels Spracherkennung bei.

LITERATURVERZEICHNIS

[1] Hörl, J.; Kolland, F.; Majce, G. (2007): Hochaltrigkeit in Österreich; S. 23ff.; URL: http://www.bse-vienna.at/pdf/Hochaltrigkeit_in_Oesterreich.pdf#page=31 ; Stand: 01.02.2016

[2] Glass, J.; Zue, V. (2003): Major Components in a Speech Recognition System; Massachusetts Institute of Technology; URL: http://ocw.alfaisal.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-345Automatic-Speech-RecognitionSpring2003/CourseHome/index.htm ; Stand: 10.04.2016

[3] Schultz, T. (2013): The Big Picture Or The Components of Automatic Speech Recognition (ASR); Auszug aus den Unterlagen zur LV Multilinguale Mensch-Maschine Kommunikation; S. 13, S. 18; URL: http://csl.anthropomatik.kit.edu/downloads/vorlesungsinhalte/MMMK-PP02-BigPicture-SS2013.pdf ; Stand: 13.4.2016

[4] Schukat-Talamazzini, E. G. (1995): Automatische Spracherkennung; Grundlagen, statistische Modelle und effiziente Algorithmen; S.165 ff.; Wiesbaden: Springer Fachmedien

[5] Paulus, E. (1987): Mustererkennung 1987; 9.DAGM-Symposium Braunschweig, 29.9-1.10.1987, Proceedings; S. 118ff; Berlin, Heidelberg: Springer-Verlag

[6] Huang, X.; Deng, L. (2009): An Overview of Modern Speech Recognition; In: Indurkhya, N.; Damerau, F.J. (Hrsg): Handbook of Natural Language Processing; Second Edition; Kapitel 15, S. 339-367; Boca Raton: CRC Press

[7] Juang, B. H.; Rabiner, L. R. (2004): Automatic Speech Recognition; A Brief History of the Technology Development; Georgia Institute of Technology, Atlanta; Rutgers university and the University of California, Santa Barbara

[8] Euler, S. (2006): Grundkurs Spracherkennung; Vom Sprachsignal zum Dialog – Grundlagen und Anwendungen verstehen; Wiesbaden: Vieweg & Teubner Verlag

[9] Pfister, B.; Kaufmann, T. (2008): Sprachverarbeitung, Grundlagen und Methoden der Sprachsynthese und Spracherkennung; Kapitel 12: Statistische Spracherkennung, S. 327-368; Kapitel 13: Sprachmodellierung, S. 369-400; Berlin, Heidelberg: Springer-Verlag

[10] KIT (2011): Grundlagen der Automatischen Spracherkennung, Sprachmodellierung, Teil 3, Suche; Auszug aus den Unterlagen zur LV Grundlagen der Automatischen Spracherkennung; S. 18 ff.; URL: http://i13pc106.anthropomatik.kit.edu/fileadmin/cmu-uka-shared-files/S/2011-01-12.pdf ; Stand: 15.02.2017

[11] Ryan, M. S.; Nudd, G. R. (1993): The Viterbi Algorithm; Conventry; S. 2 ff.; University of Warwick: Department of Computer Science

[12] Huang, X.; Baker, J.; Reddy, R. (2014): A Historical Perspective of Speech Recognition; Communications of the ACM vol. 57, no. 1, S94; ACM

[13] CMUSphinx: Sphinxbase; URL: https://github.com/cmusphinx/sphinxbase ; Stand: 04.06.2016

[14] Sourceforge: CMU Sphinx Speech Recognition Tool Kit; Acoustic and Language Models; URL: https://sourceforge.net/projects/cmusphinx/files/Acoustic%20and%20Language%20Models ; Stand: 04.06.2016

[15] CMUSphinx: Overview of the CMUSphinx toolkit; URL: https://cmusphinx.github.io/wiki Stand: 04.06.2016

[16] Nuance Dragon for developers: Dragon Software Developer Kit – Integrieren Sie Dragon Spracherkennung in ihre Anwendung; URL: http://www.nuance.de/for-developers/dragon/index.htm ; Stand: 28.05.2016

[17] Nuance Dragon Mobile for developers: Nuance Developer Program – Dragon Mobile SDK; URL: http://www.nuance.de/for-developers/dragon-mobile-sdk/index.htm ; Stand: 28.05.2016

[18] Nuance for Developers: Homepage für Entwickler - Überblick Plattformen und Anwendungsbereiche; URL: http://www.nuance.de/for-developers/index.htm ; Stand: 26.05.2016

[19] Nuance: Nuance Recognizer Language Availability; URL: https://www.nuance.com/omni-channel-customer-engagement/voice-and-ivr/automatic-speech-recognition/nuance-recognizer/recognizer-languages.html ; Stand: 29.05.2016

[20] Shires, G.; Wennborg, H. (2012): Web Speech API Specification; URL: https://w3c.github.io/speech-api/speechapi.html ; Stand: 10.06.2016

[21] Shires, G.: Voice Driven Web Apps: Introduction to the Web Speech API; URL: https://developers.google.com/web/updates/2013/01/Voice-Driven-Web-Apps-Introduction-to-the-Web-Speech-API ; Stand: 10.06.2016

[22] MDN Web Docs: Web Speech API; URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Speech_API ; Stand: 10.06.2016

[23] MDN Web Docs: Using the Web Speech API; URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Speech_API/Using_the_Web_Speech_API ; Stand: 10.06.2016

[24] MDN Web Docs: SpeechRecognition.continuouse; URL: https://developer.mozilla.org/en-US/docs/Web/API/SpeechRecognition/continuous ; Stand: 10.06.2016

[25] Microsoft Developer Network: Speech Platforms; URL: https://msdn.microsoft.com/en-us/library/hh361571(v=office.14).aspx ; Stand: 11.06.2016

[26] Microsoft Developer Network: For Windows and Windows Server 2008; URL: https://msdn.microsoft.com/en-us/library/hh323805(v=office.14).aspx ; Stand: 11.06.2016

[27] Jones, K. A. (2004): Windows Speech Recognition Programming; Kapitel 2: Introducing Microsoft Windows Speech API and Microsoft Compatible Speech SDK Suites, S.27-60; Lincoln, Nebraska: iUniverse

[28] Microsoft Developer Network: Speech API Overview (SAPI 5.4); URL: https://msdn.microsoft.com/en-us/library/ee125077(v=vs.85).aspx ; Stand: 12.06.2016

[29] Microsoft Developer Network: Speech API (SAPI) 5.0 (Windows CE 5.0); URL: https://msdn.microsoft.com/de-de/library/ms897381.aspx ; Stand: 12.06.2016

[30] Microsoft Developer Network: Applications and Services Development (Windows CE 5.0); URL: https://msdn.microsoft.com/de-de/library/ms862084.aspx ; Stand: 12.06.2016

[31] Microsoft Developer Network: Microsoft Speech API (SAPI) 5.3; URL: https://msdn.microsoft.com/en-us/library/ms723627(v=vs.85).aspx ; Stand: 12.06.2016

[32] Microsoft Developer Network: Microsoft Speech API (SAPI) 5.4; URL: https://msdn.microsoft.com/en-us/library/ee125663(v=vs.85).aspx ; Stand: 12.06.2016

[33] Greenlee, M. (2012): Microsoft.Speech and System.Speech; Lync Development; URL: http://blog.greenl.ee/2012/09/06/microsoft-speech-system-speech/ ; Stand: 13.06.2016

[34] Microsoft Developer Network: Microsoft Speech Platform, Microsoft Speech Platform Overview; URL: https://msdn.microsoft.com/de-de/library/jj127858.aspx ; Stand: 13.06.2016

[35] Microsoft Developer Network: Microsoft Speech Platform SDK 11 Documentation; URL: https://msdn.microsoft.com/en-us/library/dd266409(v=office.14).aspx ; Stand: 13.06.2016

[36] Microsoft Developer Network: Microsoft Speech Platform; URL: https://msdn.microsoft.com/en-us/library/office/hh361572%28v=office.14%29.aspx ; Stand: 12.06.2016

[37] Windows Dev Center: Microsoft Spracherkennung; URL: https://developer.microsoft.com/de-de/windows/speech ; Stand: 12.06.2016

[38] Trufinescu, A.; Shahid, K. (2015): Cortana Extensibility and Speech Platform in Depth; Channel9, Microsoft; URL: https://view.officeapps.live.com/op/view.aspx?src=http%3a%2f%2fvideo.ch9.ms%2fsessions%2fbuild%2f2015%2f3-716.pptx ; Stand: 12.06.2016

[39] Arsurya, N. (2015): Integrating Cortana in your Universal Windows App using Visual Studio 2015; Microsoft Developer Network; URL: https://blogs.msdn.microsoft.com/arsurya/2015/07/06/integrating-cortana-in-your-universal-windows-app-using-visual-studio-2015/ ; Stand: 12.06.2016

[40] Hull, E.; Jackson, K.; Dick, J. (2011): Requirements Engineering; Third Edition; Kapitel 1: Introduction, S. 1-22; London: Springer Verlag

[41] Glinz, M. (2006): Requirements Engineering I, Grundlagen; Universität Zürich, Institut für Informatik; URL: https://files.ifi.uzh.ch/rerg/amadeus/teaching/courses/requirements_engineering_I_ws0607/Kapitel_01_Grundl.pdf ; Stand: 23.04.2016

[42] Radatz, J. et al. (1990): IEEE standard glossary of Software Engineering Terminology; New York: The Institute of Electrical and Electronics Engineers

[43] IEEE Computer Society (2005): ISO/IEC 26702:2007(E), IEEE Std 1220-2005, Systems engineering – Application and management of the systems engineering process; New York: The Institute of Electrical and Electronics Engineers

[44] Hohler, B.; Villinger, U. (1998): Normen und Richtlinien zur Prüfung und Qualitätssicherung von Steuerungssoftware; Informatik-Spektrum vol. 21, no. 2, S. 63-73; Berlin, Heidelberg: Springer Verlag

[45] Memon, A. (2010): Software Requirements, Descriptions and specifications of a system; Auszug aus den Unterlagen zur LV Software Engineering; University of Maryland, Dept. of Computer Science; URL: https://www.cs.umd.edu/~atif/Teaching/Spring2010/Slides/3.pdf ; Stand: 24.04.2016

[46] International Qualification Board for Business Analysis (2011): Standard glossary of terms used in Software Engineering, Version 1.0; URL: https://www.astqb.org/documents/Standard-glossary-of-terms-used-in-Software-Engineering-1.0-IQBBA.pdf ; Stand: 23.04.2016

[47] Pohl, K.; Böckle, G.; Van der Linden, F. J. (2005): Software Product Line Engineering; Kapitel 10: Domain Requirements Engineering, S. 193-216; Berlin, Heidelberg: Springer Verlag

[48] Sharp, H.; Finkelstein, A.; Galal, G. (1999): Stakeholder Identification in the Requirements Engineering Process; London: City University, University College

[49] Barnum, C. M. (2011): Usability Testing Essentials; Kapitel 1: Establishing the essentials, S.9 -23; Burlington: Elsevier

[50] Jokela, T. et al. (2003): The Standard of User-Centered Design and the Standard Definition of Usability: Analyzing ISO 13407 against ISO 9241-11, S. 53-60; Finland: Oulu University

[51] Beck, K. (2002): Test-Driven Development, By Example; Pearson Education

[52] Janzen, D.; Saiedian, H. (2005): Test-Driven Development: Concepts, Taxonomy, and Future Direction; IEEE Computer Society vol. 38, no. 9, S. 43-50

[53] Perez, R. (2010): Unit testing, component level testing and UI testing, what to use and when; Microsoft Developer; URL: https://blogs.msdn.microsoft.com/raulperez/2010/04/29/unit-testing-component-level-testing-and-ui-testing-what-to-use-and-when/ ; Stand: 30.04.2016

[54] Jéron, T. et al. (2000): Efficient object-oriented integration and regression testing; IEEE Reliability Society vol. 49, no.1, S. 12-25

[55] Zboray, V. (2012): Software Engineering und Projektmanagement, Integration und Test; Auszug aus den Unterlagen der LV Software Engineering und Projektmanagement 2.0 VO 2012WS; TU Wien

[56] Fritzsche, K. (2004): Das Abtasttheorem; Auszug aus den Unterlagen zur LV Mathematik für Elektrotechniker; S.17 ff.; URL: http://www2.math.uni-wuppertal.de/~fritzsch/lectures/met/sitn_33.pdf, Stand: 10.2.2017

[57] Osmani, A. (2012): Learning JavaScript Design Patterns; Sebastopol: O'Reilly Media

[58] Harmes, R.; Diaz, D. (2008): Pro JavaScript Design™ Patterns; Kapitel 5: The Singleton Pattern, S. 65-82; New York: Springer-Verlag

[59] Timms, S. (2014): Mastering JavaScript Design Patterns; Kapitel 2: Organizing Code, S. 23-42, Kapitel 3: Creational Patterns, S. 43-64; Birmhingham: Packt Publishing

[60] Node.js (2016): Description Node.js; URL: https://nodejs.org/en/ ; Stand: 01.05.2016

[61] Herron, D. (2013): Node Web Development, Second Edition; Birmhingham: Packt Publishing

[62] npmjs.com: NPM Documentation; URL: https://docs.npmjs.com/ ; Stand: 01.05.2016

[63] javaTPoint: What is Node.js; URL: https://www.javatpoint.com/what-is-nodejs ; Stand: 01.05.2016

[64] Thai, T. L.; Lam, H. (2001): .NET Framework Essentials, Third Edition; Beijing, Cambridge, Fornham, Köln, Paris, Sebastopol, Taipei, Tokyo: O'Reilly & Associates

[65] Troelsen, A.; Agarwal, V. V. (2010): Pro VB 2010 and the .NET 4 Platform; New York: Springer Verlag

[66] Microsoft Docs: Get started with the .NET Framework; URL: https://docs.microsoft.com/en-gb/dotnet/framework/get-started/ ; Stand: 22.10.2016

[67] Microsoft Docs: Overview of the .NET Framework; URL: https://docs.microsoft.com/en-gb/dotnet/framework/get-started/overview ; Stand: 22.10.2016

[68] Microsoft Docs: An introduction to NuGet; URL: https://docs.microsoft.com/en-gb/nuget/what-is-nuget ; Stand: 22.10.2016

[69] Microsoft Developer Network: ASP.NET Overview; URL: https://msdn.microsoft.com/en-us/library/4w3ex9c2.aspx ; Stand: 22.10.2016

[70] Abts, D.(2015): Masterkurz Client/Server-Programmierung mit Java, Kapitel 6, HTTP-Kommunikation im Web, Kapitel 7, Bidirektionale Kommunikation mit WebSocket; Wiesbaden: Vieweg + Teubner Verlag

[71] ITWissen.info: Webserver; DATACOM Buchverlag GmbH; URL: https://www.itwissen.info/Webserver-web-server.html ; Stand: 23.10.2016

[72] Jaitla, J. (2015): WebSockets vs REST: Understanding the Difference; URL: https://www.pubnub.com/blog/2015-01-05-websockets-vs-rest-api-understanding-the-difference/ ; Stand: 23.10.2016

[73] Windows Dev Center (2016): Named Pipes; Microsoft; URL: https://msdn.microsoft.com/de-de/library/windows/desktop/aa365590(v=vs.85).aspx ; Stand: 25.03.2017

[74] Fielding, R. et al. (1999): W3C Network Working Group; URL: https://www.w3.org/Protocols/rfc2616/rfc2616.txt ; Stand: 21.05.2016

[75] Hawke, S.: REST, RDF Simple Data Interface Protocol - Level Zero; W3C Semantic Web; URL: https://www.w3.org/2001/sw/wiki/REST ; Stand: 22.05.2016

[76] AAT (2008): ISUI Manual, LUI Configuration Manual; Version 0.1.0.; Centre for Applied Assistive Technologies: TU Wien

[77] AAT Homepage: ISUI; URL: http://www.aat.tuwien.ac.at/isui/index.html ; Stand: 15.02.2016

[78] Microsoft Docs: UWP Apps; URL: https://docs.microsoft.com/de-de/windows/uwp/get-started/whats-a-uwp ; Stand: 24.10.2016

[79] Windows Dev Center: Windows 10 universal Windows platform (UWP) app lifecycle; URL: https://docs.microsoft.com/en-us/windows/uwp/launch-resume/app-lifecycle ; Stand: 24.10.2016

[80] Microsoft Docs: UWP Introduction; URL: https://docs.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide ; Stand: 24.10.2016

[81] Node.js: Node.js Documentation – Stability Index; URL: https://nodejs.org/api/documentation.html#documentation_stability_index ; Stand: 01.05.2016

[82] Node.js: Node.js Documentation – Modules; URL: https://nodejs.org/api/modules.html ; Stand: 01.05.2016

[83] Mozzer, T. (2015): Deploying speech recognition locally versus the cloud; URL: http://www.embedded-computing.com/embedded-computing-design/deploying-speech-recognition-locally-versus-the-cloud ; Stand: 09.07.2016

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTING-Verzeichnis

ABKÜRZUNGSVERZEICHNIS

AAT	Angewandte Assistierende Technologien
AAL	Active and Assisted Living
APP	Application
HTK	Hidden Markov Model Toolkit
IIFE	Immediately-Invoked-Function-Expression
ISO	International Organization for Standardization
JSGF	JSpeech Grammar Format
LUI	Local User Interface
NMDP	Nuance Mobile Developer-Program
RDF	Resource Description Framework
TDD	Test Driven Development
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UWP	Universal Windows Platform
W3C	World Wide Web Consortium
XML	Extensible Markup Language