FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Spracherkennung im Browser

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsingenieurwesen Informatik

eingereicht von

Stefan Müller
Matrikelnummer 0928704

an der
Fakultät für Informatik der Technischen Universität Wien

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

Wien, 14.01.2015

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung

Stefan Müller
Veitingergasse 5
1130 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig 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.

Wien, am 14. Jänner 2015

Stefan Müller

Kurzfassung

Im Bereich der sprachgesteuerten Benutzerschnittstellen gab es in den letzten Jahren sehr große Fortschritte. Während sich sprachbasierte Anwendungen auf mobilen Endgeräten (z. B. Siri, Google Now etc.) und am Desktop bereits etabliert haben, sind Sprachsteuerung und -eingabe bei browserbasierten Benutzerschnittstellen aber noch sehr unüblich.

Im Rahmen der vorliegenden Arbeit werden die Grundlagen der automatischen Spracherkennung vorgestellt und Möglichkeiten für die Steuerung von browserbasierten Benutzerschnittstellen per Spracheingabe untersucht. Die Arbeit gibt einen Überblick über die neuen Möglichkeiten zur Audioaufnahme, -wiedergabe und -verarbeitung, welche im Umfeld von HTML 5 entstanden sind, und stellt konkrete technische Lösungsansätze für die Umsetzung einer automatischen Spracherkennung im Web-Kontext vor.

Der praktische Teil der Arbeit beschäftigt sich mit dem Entwurf, der Implementierung und der Evaluierung eines Prototyps für ein Sprachsteuerungsmodul. Es wird ein System vorgestellt, das moderne Webtechnologien – allen voran die Web Speech API – verwendet, um eine Sprachsteuerung zu realisieren. Das entwickelte System nutzt ein flexibles XML-Dateiformat zur Definition von Befehlen. Des Weiteren implementiert es eine phonetische Nachverarbeitung der vom Google Spracherkennungsservice gelieferten Ergebnisse, um die Gesamterkennungsleistung zu steigern. Im Zuge einer Evaluierung wurde die Erkennungsleistung des Systems unter verschiedenen Bedingungen erhoben.

Schlüsselwörter: *Sprachsteuerung, automatische Spracherkennung, Webtechnologien, HTML 5, JavaScript, Web Speech API, Softwareentwicklung*

Abstract

During the last few years there has been substantial progress in the area of voice-controlled user interfaces. While users have become used to voice-based applications on mobile devices (e.g. Siri or Google Now) or on the desktop, it is currently still very uncommon to see voice-controlled web applications.

In this master thesis fundamentals of automatic speech recognition are presented and options for the implementation of voice-controlled web interfaces are discussed. The thesis gives an overview of the new options for audio recording, playback and processing, which have been developed in the context of HTML 5, and investigates selected technical solutions for the implementation of automatic speech recognition in web applications.

The practical part of the thesis deals with the design, implementation and evaluation of a prototype for a voice control module. A system is presented that uses modern web technologies, most notably the Web Speech API, in order to implement voice control for web applications. The developed system uses a flexible XML file format to define commands. Additionally it implements a phonetic post-processing of the results provided by Google's automatic speech recognition service in order to improve the overall recognition performance. As part of the work, the recognition performance of the system has been evaluated under different conditions.

Keywords: *Voice Control, Automatic Speech Recognition, Web Technologies, HTML 5, JavaScript, Web Speech API, Software Development*

Inhaltsverzeichnis

Kapitel 1

Einleitung

In den hoch entwickelten westlichen Industrienationen führen die demographische Entwicklung und die damit einhergehenden sozialen Veränderungen zu einer stark alternden und zunehmend individualisierten Gesellschaft. Der stetig wachsende Anteil an älteren und alleinstehenden Menschen resultiert unter anderem in einem steigenden Bedarf an technischen Systemen, welche die Alltagstätigkeiten erleichtern und es den Menschen erlauben, möglichst lange und unabhängig im eigenen Wohnumfeld zu leben. Diese sogenannten altersgerechten Assistenzsysteme werden, neben Methoden und Konzepten, welche das alltägliche Leben älterer und auch benachteiligter Menschen unaufdringlich und situationsabhängig unterstützen, unter dem Oberbegriff Ambient Assisted Living (AAL) zusammengefasst.

Im Zuge des Projektes CongeniAAL wurde am Zentrum für Angewandte Assistierende Technologien (AAT) der TU Wien ein solches AAL-System entwickelt. Eine der zentralen Komponenten dieses Systems bildet ein Tabletcomputer, welcher über ein flexibles User Interface – Local User Interface oder kurz LUI genannt – die Interaktion zwischen Benutzerinnen bzw. Benutzern und System ermöglicht und diverse Funktionen – wie beispielsweise Videotelefonie, Darstellung von Vitalparametern, einfacher Internetzugriff, Erinnerungen für die Medikamenteneinnahme, Spiele zum Trainieren der kognitiven Fähigkeiten etc. – zur Verfügung stellt. Die Benutzerschnittstelle LUI wurde ursprünglich für Microsoft Windows-Betriebssysteme entwickelt und unterstützt zur besseren Barrierefreiheit, neben der Eingabe über den Touchscreen, auch die Möglichkeit der Steuerung mittels gesprochener Sprache.

Aktuelle Projekte am AAT beschäftigen sich mit der Erweiterung und Portierung der bestehenden Benutzerschnittstelle, sodass diese, neben Microsoft Windows-Systemen, auch auf anderen Software-Plattformen betrieben werden kann. Das zugrundeliegende Konzept sieht dabei die Verwendung moderner, durch das W3C (World Wide Web Consortium) standardisierter Webtechnologien (zum Beispiel HTML 5, CSS 3, JavaScript etc.) zur Implementierung einer offlinefähigen browserbasierten Benutzerschnitt-

stelle vor. Wie bisher soll auch die zukünftige Benutzerschnittstelle unter Verwendung automatischer Spracherkennung (Automatic Speech Recognition, ASR) mittels Sprache gesteuert werden können. Die bestehenden ASR-Komponenten sind, da sie auf der Microsoft Speech API basieren, plattformgebunden und können folglich nicht wiederverwendet werden. Die Sprachsteuerung muss deshalb neu implementiert werden.

1.1 Problemstellung und Zielsetzung

Im Bereich der sprachgesteuerten Benutzerschnittstellen gab es in den letzten Jahren sehr große Fortschritte. Besonders auf mobilen Endgeräten, wie Smartphones oder Tabletcomputer, haben Spracheingabe und -steuerung stark an Bedeutung gewonnen – wie Anwendungen wie Apples Siri oder Google Now zeigen. Während sich aber sprachbasierte Anwendungen auf mobilen Endgeräten und am Desktop (z. B. diverse Diktieranwendungen, integrierte Sprachsteuerung in Windows etc.) bereits etabliert haben, sind Sprachsteuerung und -eingabe bei browserbasierten Benutzerschnittstellen noch sehr unüblich. Abgesehen von Googles Voice Search[1], welche seit Mitte 2011 verfügbar ist, gibt es derzeit keine populären Beispiele für Webseiten, die gesprochene Sprache zum Zweck der Steuerung oder Eingabe nutzen.

Begründen lässt sich dies unter anderem mit den bisher sehr limitierten technischen Möglichkeiten, welche Webtechnologien boten, um solche Anwendungen zu realisieren. Mit HTML 5 werden nun aber zahlreiche neue Komponenten (z. B. die Web Audio API oder die Web Speech API) eingeführt, welche die Umsetzung von Spracheingabe und -steuerung in browserbasiertern Anwendung deutlich vereinfachen sollen.

Das Ziel der Arbeit ist es, zu untersuchen, wie die Umsetzung einer Sprachsteuerung für browserbasierte Benutzerschnittstellen unter Verwendung moderner Webtechnologien erfolgen kann. Zunächst soll dazu analysiert werden, welche technischen Lösungsansätze existieren und welche Eigenschaften diese besitzen. Basierend auf den erlangten Erkenntnissen kann dann jener Lösungsansatz ausgewählt werden, welcher am geeignetsten für die Implementierung eines Sprachsteuerungsmoduls für die LUI-Portierung erscheint.

Das Resultat der Arbeit soll ein funktionsfähiger, umfangreich getesteter und dokumentierter Prototyp eines Moduls für die Steuerung von browserbasierten Benutzerschnittstellen per Spracheingabe sein. Der Prototyp muss flexibel konfigurierbar sein und auf einfache Weise in verschiedene Anwendungen integriert werden können, sodass neben dem Einsatz in der Portierung von LUI auch die Verwendung in anderen, zukünftigen Anwendungen möglich ist. Die Spracherkennung soll eine möglichst hohe Erkennungsleistung bieten und der Prototyp darf ausschließlich unter Verwendung von Standard-

[1] Siehe `https://www.google.at`. Spracheingabe funktioniert aber nur mit dem Google Chrome Browser ab der Version 25.

Webtechnologien bzw. zukünftigen Standard-Webtechnologien umgesetzt werden. Weitere Anforderungen an das Modul müssen in einer im Zuge der Arbeit durchzuführenden Anforderungsanalyse identifiziert und genauer spezifiziert werden.

1.2 Aufbau der Arbeit

Nachdem im ersten Kapitel ein einleitender Überblick über die Problemstellung, Zielsetzung und den Aufbau der Arbeit gegeben wurde, werden in Kapitel 2 die theoretischen Grundlagen und der aktuelle Stand der Technik in den für die Arbeit relevanten Themenbereichen beschrieben.

In Kapitel 3 werden zunächst in einer Anforderungsanalyse die Anforderungen an den Prototypen definiert. Anschließend wird, basierend auf den ermittelten Anforderungen, ein geeigneter technischer Lösungsansatz für die Umsetzung des Prototyps ausgewählt.

Das Kapitel 4 dokumentiert die Umsetzung des Prototypen. Basierend auf dem gewählten Lösungsansatz wird zunächst ein Softwareentwurf erstellt, welcher wiederum die Basis für die Implementierung des Prototypen bildet. In weiterer Folge wird die Implementierung durchgeführt und das Resultat vorgestellt. Es wird Aufschluss über gewählte Technologien und Werkzeuge gegeben und anhand ausgewählter Quellcodeartefakte gezeigt, wie die Umsetzung der konzipierten Funktionen erfolgte.

Nachdem der Prototyp implementiert wurde, soll dieser einer umfassenden Evaluierung unterzogen werden. Es gilt dabei die Erkennungsleistung des Prototypen unter verschiedenen Bedingungen zu ermitteln. In Kapitel 5 werden die Zielsetzung, Durchführung und Ergebnisse dieser Evaluierung im Detail beschrieben.

Im abschließenden Kapitel 6 werden die Ergebnisse der Arbeit zusammengefasst und ein Fazit gezogen. Schlussendlich wird noch ein Ausblick auf zukünftige Entwicklungen sowie mögliche Erweiterungen und Verbesserungen gegeben.

Kapitel 2

Grundlagen und Stand der Technik

Im folgenden Kapitel werden grundlegende Konzepte und Begriffe, welche den wesentlichen Rahmen für die Betrachtung der im einführenden Kapitel beschriebenen Problemstellungen bilden, erläutert und der aktuelle Stand der Technik in den relevanten Themenbereichen beschrieben.

In Abschnitt 2.1 wird ein allgemeiner Überblick über den Themenbereich der automatischen Spracherkennung gegeben. Unter anderem wird dabei auf die Geschichte, die Anwendungsbereiche, den Aufbau und die Funktionsweise von Spracherkennungssystemen eingegangen und deren theoretische Grundlagen kurz beschrieben.

Der darauf folgende Abschnitt 2.2 nimmt auf die Webtechnologien HTML, CSS, JavaScript und XML Bezug. Diese Technologien werden im Zuge der Implementierung des Prototyps verwendet und sollen deshalb kurz vorgestellt werden.

Abschnitt 2.3 beschäftigt sich mit der Spracherkennung im Browser. Zunächst werden dazu die Möglichkeiten zur Audioaufnahme und -wiedergabe im Browser betrachtet. Anschließend werden konkrete technische Lösungsansätze zur automatischen Spracherkennung im Browser vorgestellt.

2.1 Automatische Spracherkennung

Die automatische Spracherkennung ist ein Teilgebiet der Mensch-Maschinen-Kommunikation und hat zum Ziel, das gesprochene Wort in eine entsprechende textuelle Repräsentation überzuführen, ohne dabei zwingend die Bedeutung oder Absicht des Gesprochenen zu verstehen. Sie stellt eine Vorstufe des automatischen Sprachverstehens dar [1, 2].

2.1.1 Geschichtlicher Überblick und Stand der Technik

Erste Versuche Spracherkennungssysteme zu entwickeln, wurden bereits in den 50er Jahren des 20. Jahrhunderts unternommen. Diese noch recht einfachen analogen Systeme basierten auf Zeitbereichsmerkmalen oder der Analyse der mittels analoger Filterbänke gewonnenen spektralen Resonanzen und konnten lediglich isolierte Ziffern oder einsilbige Wörter eines einzelnen Sprechers erkennen [3]. Die Ableitung der Laute durch Analyse ihrer Spektralform blieb bis zum Ende der 60er Jahre der vorherrschende Ansatz, lieferte aber nur mäßigen Erfolg [1].

In den 70er Jahren wurde begonnen, dynamische Programmierung für den Vergleich von Sprachmustern einzusetzen. Das Verfahren der dynamischen Zeitanpassung (engl.: Dynamic Time Warping, DTW) erlaubte den Vergleich von zeitlich verzerrten Mustern und ermöglichte die Umsetzung leistungsfähiger Einzelworterkenner, die auch sprecherunabhängig funktionierten. Von entscheidender Bedeutung für das Forschungsgebiet war das von der Advanced Research Projects Agency (ARPA) des US-amerikanischen Verteidigungsministerium ausgeschriebene Projekt SUR (Speech Understanding Research). Das daraus hervorgehende System Harpy konnte 1011 Einzelworte bei einer Fehlerrate von weniger als 5% erkennen. Bedeutend war aber vor allem dessen neuartige Architektur, welche – basierend auf einem mehrschichtigen modularen linguistischen Modell – richtungsweisend für zukünftige Spracherkennungssysteme sein sollte [3, 29].

Mit dem Übergang vom dynamischen Mustervergleich zu den statistischen Ansätzen, erfolgte Mitte der 80er Jahre ein Paradigmenwechsel, der den entscheidenden Durchbruch zur Erkennung von kontinuierlich gesprochener Sprache bringen sollte. Vor allem der Ansatz, Hidden-Markov-Modelle (HMM) zur Modellierung der Variabilität der Sprachsignale zu verwenden, erwies sich als äußerst erfolgreich und ist bis heute von zentraler Bedeutung [3]. Es war damit erstmals möglich, die komplexen Zusammenhänge zwischen Spracheinheiten in ein stochastisches Modell zu verpacken, dessen Parameter aus vorgegebenen Sprachproben geschätzt werden konnten. Durch die Verwendung der statistischen Ansätze gelang es im Laufe der 90er Jahre schließlich Systeme zu entwickeln, die hinreichend gute Leistung erzielten, um in kommerziellen, massenmarkttauglichen Produkten eingesetzt zu werden.

Obwohl moderne Systeme unter kontrollierten Bedingungen mittlerweile sehr hohe Erkennungsleistungen erreichen[1], stellt die automatische Spracherkennung unter schwierigen akustischen Bedingungen (z. B. Hintergrundgeräusche, Raumhall etc.) immer noch ein nicht zufriedenstellend gelöstes Problem dar. Seit der Jahrtausendwende konzentrieren sich Forschungsaktivitäten daher vermehrt auf Methoden zur Erhöhung der Robustheit von Spracherkennungssystemen unter schwierigen akustischen Bedingungen. Die

[1] Sie werden mit Genauigkeiten von bis zu 99% beworben. Siehe z. B. `http://shop.nuance.co.uk/store/nuanceeu/en_GB/pd/ThemeID.26429600/productID.306536600`.

Ansätze reichen dabei von der Verwendung von Mikrofonarrays [10] über diverse Filtermethoden (z. B. Wiener-Filter [11]) bis zu Adaptionen auf Modellebene [12]. Ein weiterer Lösungsansatz ist die bimodale Spracherkennung. Diese hat zum Ziel, Spracherkennungssysteme unter Zuhilfenahme einer zweiten Modalität – beispielsweise der Gestik oder Mimik – in ihrer Erkennungsleistung zu verbessern. In diesem Zusammenhang beschäftigt sich die aktuelle Forschung vor allem mit der Integration von visuellem Kontext in die Spracherkennung, also der sogenannten audiovisuellen Spracherkennung [13].

Ein weiterer Forschungsbereich, welcher viel Aufmerksamkeit erfährt, ist die akustisch Modellierung. Bereits Anfang der 90er Jahre wurde erstmals demonstriert, wie künstliche neuronale Netze und Hidden-Markov-Modelle sinnvoll verknüpft werden können [4]. In der Zwischenzeit wurden diese sogenannten hybriden Ansätze stark weiterentwickelt. Forschungsergebnisse der letzten Jahre zeigen, dass gegenüber klassischen HMM-Ansätzen mit hybriden Ansätzen – wie beispielsweise CD-DNN-HMM (Context-Dependent Deep Neural Network Hidden Markov Models) – wesentlich bessere Erkennungsleistungen erzielt werden können [5, 6]. Als sehr bedeutend haben sich auch die enormen Mengen an Sprach- und Textdaten sowie die Methoden, diese Daten zu sammeln, zu handhaben und damit die stochastischen Modelle zu trainieren, herausgestellt. Beispielsweise werden in [7] über 87.000 Stunden Sprachdaten, welche aus Mitschnitten der Google Voice Search stammen, zum Trainieren eines verteilten akustischen Modells verwendet. In [28] wird wiederum ein Sprachmodell mit einem Sprachkorpus bestehend aus 230 Milliarden Wörter trainiert. In beiden Fällen konnte die Erkennungsleistung dadurch signifikant verbessert werden.

2.1.2 Anwendungsgebiete und -beispiele

Die gesprochene Sprache ist für die Menschen eine der wichtigsten Methoden, um miteinander zu kommunizieren. Der Gedanke, diese auch als Mittel für die Mensch-Maschinen-Kommunikation einzusetzen, liegt daher nahe. Obwohl die Effizienz einer Kommunikationsform letztendlich von der jeweiligen Anwendung abhängt, bringt eine gesprochene Mensch-Maschinen-Kommunikation eine Reihe von Vorteilen mit sich, die sie als vielversprechende Verbesserung erscheinen lassen – beispielsweise eine höhere Informationsübertragungsrate, gesteigerte Bewegungsfreiheit oder die Verfügbarkeit von Händen und Augen für andere Aufgaben (für einen detaillierteren Überblick siehe [1, 14]).

Tatsächlich werden in der Mensch-Maschinen-Kommunikation automatische Spracherkennungssysteme schon für viele Aufgaben erfolgreich eingesetzt. In einer groben Einteilung lassen sich drei Anwendungsgebiete unterscheiden [8]:

Steuerung von Geräten: Grundsätzlich stellt die Steuerung von Geräten per Spracheingabe eine intuitive und komfortable Form der Bedienung dar und wird beispielsweise bei Smartphones (z. B. Sprachwahl, persönliche digitale Assistenten

wie Apples Siri, Microsofts Cortana oder Google Now etc.), Unterhaltungselektronik (z. B. Smart TVs) oder Computern (z. B. integrierte Sprachsteuerung in Windows) eingesetzt. Der Einsatz von Sprachsteuerung bietet sich aber insbesondere in jenen Fällen an, wo die Hände für andere Aufgaben benötigt werden bzw. die Augen auf etwas gerichtet bleiben müssen (z. B. Steuerung eines Operationsmikroskops oder Bedienung des Navigationssystems, Radios oder Telefons im Auto) oder die Verwendung herkömmlicher Eingabemethoden durch motorische Einschränkungen erschwert ist (z. B. kann älteren Menschen oder Menschen mit Behinderung die Bedienung von Rollstühlen, Betten oder Computer erleichtert werden).

Sprachdialogsysteme: Sprachdialogsysteme, oder auch IVR-Systeme (Interactive Voice Response), dienen der automatisierten Abwicklung natürlichsprachlicher Dialoge zwischen Mensch und Maschine. Häufig werden Sprachdialogsysteme im Zusammenhang mit dem Telefon eingesetzt, beispielsweise für automatisierte Auskunftssysteme (z. B. Wetter-, Börsen- oder Fahrplanauskunft), Buchungssysteme, Teleshopping, Telebanking, Televoting oder im Zusammenhang mit Call Centern (z. B. zur Anrufervorqualifizierung).

Diktiersysteme: Diktiersysteme ermöglichen die textuelle Erfassung fließend gesprochener Sprache („speech-to-text“). Sie sind für den Einsatz in einer relativ geräuscharmen Büroumgebung konzipierte Spracherkenner, welche ein sehr großes Vokabular (mehrere hunderttausend Wörter) besitzen. Die Systeme sind prinzipiell universell einsetzbar. Für bestimmte Berufsgruppen, wie beispielsweise Anwältinnen und Anwälte oder Ärztinnen und Ärzte, gibt es aber eigens angepasste Systeme, da diese ein entsprechendes Fachvokabular verwenden. Um maschinell zu dolmetschen, lassen sich Technologien für Spracherkennung und automatische Textübersetzung auch miteinander kombinieren.

Die Einschätzung, welche Erkennungsleistung als akzeptabel angesehen werden kann, ist stark von der jeweiligen Anwendung abhängig. Moderne Systeme, welche speziell für einen Anwendungsbereich entworfen wurden, besitzen eine hohe Erkennungsleistung, wenn sie in einer kontrollierten und störungsfreien Umgebung eingesetzt werden. Sollten aber ein oder mehrere Parameter dieser Konfiguration geändert werden, ist mit einer deutlichen Verringerung der Erkennungsleistung zu rechnen.

2.1.3 Herausforderung Spracherkennung

Dem Menschen fällt es – selbst unter schwierigen akustischen Bedingungen – verhältnismäßig leicht, gesprochene Sprache zu verstehen. Deshalb darauf zu schließen, dass maschinelle Spracherkennung ebenso einfach zu bewerkstelligen wäre, ist aber falsch.

Vielmehr ist die automatische Erkennung von Sprache eine äußerst schwierige Aufgabe, welche einen sehr interdisziplinären Charakter besitzt und deren Lösung daher Fachwissen verschiedenster Bereiche (z. B. Mustererkennung, Signalverarbeitung, Statistik, Akustik, Linguistik, Phonetik etc.) erfordert.

Grundlegende Schwierigkeiten der Spracherkennung

Ein großes Problem für die automatische Spracherkennung stellt die hohe *Variabilität* natürlicher, gesprochener Sprache dar. Verschiedene Aufnahmen ein und desselben Satzes werden sich immer unterscheiden, selbst wenn diese von derselben Person gesprochen wurden. Ein einfacher Mustervergleich zwischen einer Aufnahme und einer gespeicherten Referenz, kann somit kein zufriedenstellendes Ergebnis liefern. Begründet liegt die Variabilität in unterschiedlichen Sprechweisen (Tempo, Emotionen etc.), individuellen Sprechmerkmalen (Alter, Geschlecht etc.), habituellen Sprachmerkmalen (z. B. dialektale Färbung) und kontextuelle Aussprachevariationen (Sprachrhythmus, Verschleifungen etc.). Auch die akustische Umgebung (z. B. Hintergrundgeräusche) und die Eigenschaften des Aufnahmekanals (z. B. Typ und Position des Mikrofons) schlagen sich nieder [1, 3].

Neben der Variabilität der Sprache stellt die Ambiguität – also Mehrdeutigkeit – lautsprachlicher Äußerungen ein weiteres Problem dar. Mehrdeutigkeiten treten dabei auf allen sprachlichen Ebenen, beispielsweise im semantischen Bereich (z. B. das Wort „Bank“ kann ein Sitzmöbel oder ein Geldinstitut sein) oder in Form von Homophonen (z. B. „Lid“ und „Lied“, „Grat“ und „Grad“ etc.), auf und müssen durch den Spracherkenner ebenfalls berücksichtigt werden [1].

Weitere Probleme ergeben sich durch die *Kontinuität* und *Komplexität* der Sprache. Während erstere dafür sorgt, dass in einem Sprachsignal in der Regel keine Wortgrenzen auszumachen sind und die Wörter somit nicht einfach einzeln erkannt werden können, ist letztere dafür verantwortlich, dass die Spracherkennung hohe Anforderungen an Rechenleistung und Speicherkapazität stellt (u. a. aufgrund der großen Zahl an Kombinationen bei der Satzbildung) [1].

Problematik der Distant Speech Recognition

In heutigen Anwendungen der Spracherkennung (vgl. Abschnitt 2.1.2) wird versucht, durch die Verwendung von Headsets, Telefonhörern oder durch die kopfnahe Montage des Mikrofons, den Abstand zwischen der sprechenden Person und dem Mikrofon in einem optimalen Bereich zu halten („close-talk“). Dies liegt darin begründet, dass die automatische Spracherkennung mit entfernten Mikrofonen (engl.: Distant Speech Recognition, DSR) aktuell noch keine zufriedenstellend gelöste Problemstellung darstellt.

Durch den bei der DSR größeren Abstand zwischen der Signalquelle – also in der Regel dem Mund der Sprecherin oder des Sprechers – und dem Mikrofon, sinkt die

Leistung des Sprachsignals im Vergleich zu im Raum existierenden Störungen, wodurch das Signal-Rausch-Verhältnis geringer wird. Folgende Störungen lassen sich feststellen:

- *Additive Störungen*, die entstehen, wenn fremde Schallquellen das Signal additiv überlagern. Es wird unterschieden zwischen stationären additiven Störungen (z. B. Rauschen eines Lüfters) und instationären additiven Störungen (z. B. konkurrierende Sprecher).
- *Konvolutive Störungen*, die durch die Faltung des originalen Sprachsignals mit verzögerten Versionen des Ursprungssignals (z. B. Raumhall) entstehen.

Die Störeinflüsse verursachen einen Informationsverlust des Sprachsignals, sodass die Merkmalsvektoren des Sprachsignals von jenen des Trainingssignals abweichen. Grundsätzlich kann davon ausgegangen werden, dass automatische Spracherkennung mit abnehmendem Signal-Rausch-Verhältnis immer schwerer wird. Neben dem Signal-Rausch-Verhältnis hat aber auch die Art der Störungsquelle einen entscheidenden Einfluss auf die Erkennungsleistung (z. B. sind stationäre additive Störungen typischerweise leichter zu kompensieren als instationäre) [9].

2.1.4 Statistische Spracherkennung

Nachdem in der automatischen Spracherkennung zunächst verschiedene andere Ansätze verfolgt wurden (siehe Abschnitt 2.1.1), hat sich der statistisch orientierte Ansatz durchgesetzt. Dieser Ansatz betrachtet die Spracherkennung als Dekodierungsprozess. Ein Sprecher erzeugt zunächst durch die Artikulation einer Wortfolge $\boldsymbol{w}$ ein akustisches Signal $\boldsymbol{s}$. Dieses Signal wird aufgezeichnet und zum Erkenner übertragen, welcher aus dem digitalisierten Sprachsignal eine Folge von Merkmalsvektoren $\boldsymbol{X}$ extrahiert. Im Prozess der statistischen Dekodierung wird dann versucht, aus den Merkmalsvektoren $\boldsymbol{X}$ die geäußerte Wortfolge zu rekonstruieren. Das Ergebnis ist eine möglichst verlässliche Näherung $\hat{\boldsymbol{w}}$ der ursprünglichen gesprochenen Wortfolge $\boldsymbol{w}$. Abbildung 2.1 stellt diese Zusammenhänge in einem Kanalmodell dar.

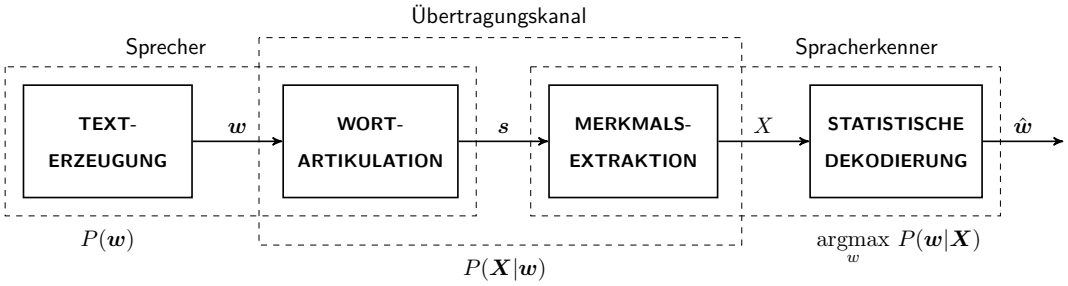


Abbildung 2.1: Kanalmodell der Spracherkennung nach [1, 3].

Formal ausgedrückt, ist das Ziel der statistischen Dekodierung diejenige Wortfolge $\hat{\boldsymbol{w}}$ zu finden, welche die A-posteriori-Wahrscheinlichkeit $P(\boldsymbol{w}|\boldsymbol{X})$ maximiert:

$$\hat{\boldsymbol{w}} = \underset{w}{\operatorname{argmax}}\ P(\boldsymbol{w}|\boldsymbol{X}) \tag{2.1}$$

Um eine optimale Entscheidung zu treffen, ist es demnach nötig die Wahrscheinlichkeit $P(\boldsymbol{w}|\boldsymbol{X})$ zu berechnen. Da diese aber schlecht zu berechnen ist, wird unter Verwendung des *Satzes von Bayes* die Gleichung umgestellt:

$$\hat{\boldsymbol{w}} = \underset{w}{\operatorname{argmax}}\ P(\boldsymbol{w}|\boldsymbol{X}) = \underset{w}{\operatorname{argmax}}\ \frac{P(\boldsymbol{w}) \cdot P(\boldsymbol{X}|\boldsymbol{w})}{P(\boldsymbol{X})} \tag{2.2}$$

Die Größe $P(\boldsymbol{X})$ im Nenner ist die Wahrscheinlichkeitsdichte der akustischen Merkmalsvektoren $\boldsymbol{X}$. Diese ist gegenüber $\boldsymbol{w}$ konstant und kann für die Maximierung daher vernachlässigt werden. Damit ergibt sich:

$$\hat{\boldsymbol{w}} = \underset{w}{\operatorname{argmax}}\ P(\boldsymbol{w}) \cdot P(\boldsymbol{X}|\boldsymbol{w}) \tag{2.3}$$

Durch die Umformung konnte das ursprüngliche Problem aus Gleichung 2.1 in zwei Teilprobleme zerlegt werden:

- $P(\boldsymbol{X}|\boldsymbol{w})$ ist die bedingte Wahrscheinlichkeit, die aussagt, wie wahrscheinlich es ist, die Merkmalsvektoren $\boldsymbol{X}$ zu beobachten, wenn die Wortfolge $\boldsymbol{w}$ aufgezeichnet wurde. Welche Merkmalsvektoren für welche Wortfolgen wie wahrscheinlich sind, wird durch das *akustische Modell* beschrieben.
- $P(\boldsymbol{w})$ ist die A-priori-Wahrscheinlichkeit für das Auftreten der Wortfolge $\boldsymbol{w}$. Sie bringt zum Ausdruck, welche der möglichen Wortfolgen des Vokabulars sinnvoll sind und wird als *Sprachmodell* bezeichnet.

2.1.5 Aufbau eines Spracherkennungssystems

Der im vorhergehenden Abschnitt vorgestellte statistische Ansatz ist heute dominierend, praktisch alle modernen Spracherkennungssysteme basieren darauf. Der Aufbau eines modernen Spracherkennungssystemes entspricht dem in Abbildung 2.2 dargestellten Blockdiagramm.

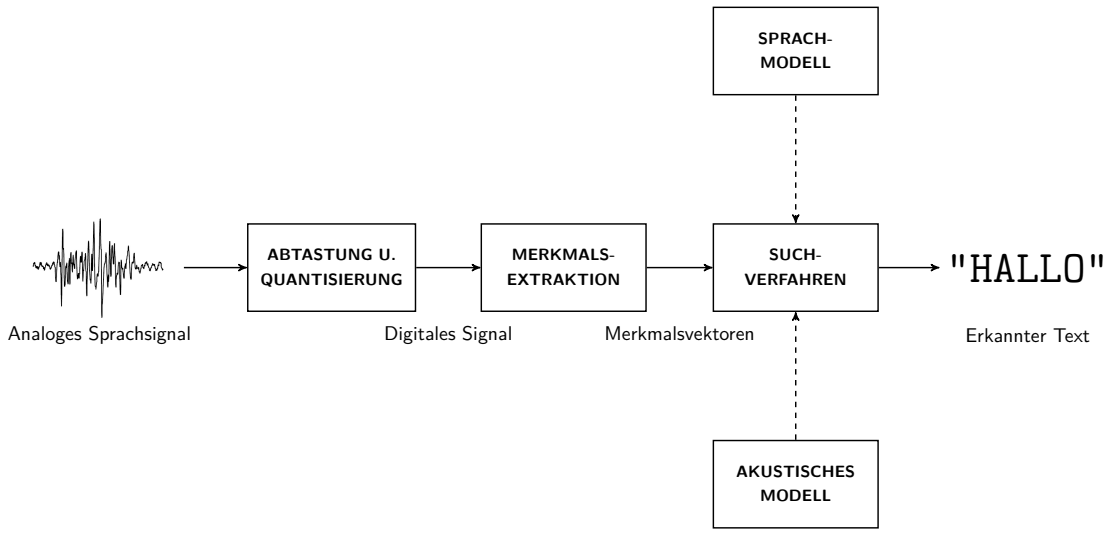


Abbildung 2.2: Aufbau eines Spracherkennungssystems (angelehnt an [3]).

Die einzelnen Komponenten werden im Folgenden überblicksartig beschrieben. Als vertiefende Literatur sei auf [1] und [3] verwiesen.

Abtastung und Quantisierung

Die beim Sprechen entstehenden akustischen Schallwellen werden mittels eines Mikrofons aufgenommen und in ein elektrisches Signal umgewandelt. Damit dieses analoge Signal durch das Spracherkennungssystem verarbeitet werden kann, muss es digitalisiert werden. Dazu wird es zunächst an äquidistanten Zeitpunkten abgetastet, um eine diskrete Folge von Werten zu gewinnen. Diese Werte werden anschließend quantisiert, wodurch das eigentliche digitale Signal entsteht.

Da sich die wesentlichen Informationen des Sprachsignals unterhalb von 8 kHz befinden, genügt gemäß des Shannonschen Abtasttheorems eine Abtastfrequenz von 16 kHz, um diese zu erfassen. Üblicherweise wird bei der Spracherkennung daher eine Abtastfrequenz von 16 kHz verwendet. Eine Ausnahme bilden Telefonanwendungen: Da über den analogen Telefonkanal maximal Frequenzen von 3,4 kHz übertragen werden können, wird bei diesen typischerweise eine Abtastfrequenz von 8 kHz verwendet.

Merkmalsextraktion

Am Eingang zur Merkmalsextraktion liegt nun ein digitalisiertes Sprachsignal vor. Die Aufgabe der Merkmalsextraktion ist es nun, die für die Spracherkennung relevanten Informationen aus diesem Signal zu extrahieren und gleichzeitig Störeinflüsse, also beispielsweise Informationsanteile die von der sprechenden Person oder der Sprechweise abhängig sind, herauszufiltern. Es erfolgt somit eine Reduktion der Datenmenge, deren Ergebnis eine Folge von mehrdimensionalen Merkmalsvektoren darstellt, die möglichst die typischen Charakteristika der zugrunde liegenden Sprachblöcke hervorheben sollen.

Die Gewinnung der Merkmalsvektoren erfolgt üblicherweise mittels der sogenannten Kurzzeitanalyse. Dabei wird ein Analysefenster in fester Schrittweite über das Sprachsignal bewegt. In dem vom Fenster definierten Bereich von ca. 5-30 ms Länge, kann Sprache als stationär angenommen werden, sodass dieser mittels diskreter Fourier-Transformation in den Frequenzbereich überführt werden kann. Für jeden Fensterbreich werden dann bestimmte Merkmale – wie zum Beispiel die Intensität der sprachrelevanten Frequenzen – ermittelt und als Merkmalsvektor dargestellt.

Akustisches Modell

Die Zusammenhänge zwischen den Merkmalsvektoren und den Wörtern des Vokabulars werden mit dem akustischen Modell beschrieben. Der dominierende Ansatz für die Realisierung des akustischen Modells beruht dabei auf Hidden-Markov-Modellen [2].

Bei einem Hidden-Markov-Modell handelt es sich um einen zweistufigen stochastischen Prozess. In der ersten Stufe kann der Prozess als ein Zustandsautomat, der aus einer endlichen Menge von Zuständen und den dazugehörigen Übergängen besteht, betrachtet werden. Der Übergang zwischen zwei Zuständen ist dabei durch die Zustandsübergangswahrscheinlichkeit spezifiziert. Zusätzlich ist für den Zustandsautomaten eine Folge von Beobachtungszeitpunkten definiert. Zu jedem dieser Beobachtungszeitpunkte nimmt der Zustandsautomat einen bestimmten Zustand ein und wechselt zum folgenden Zeitpunkt, entsprechend der festgelegten Zustandsübergangswahrscheinlichkeit, in einen neuen Zustand. Die zweite Stufe des Prozesses besteht darin, dass jeder Zustand in jedem Zeitschritt einen reellen Merkmalsvektor ausgibt. Jeder Zustand ist dazu mit einer sogenannten Emissionsverteilung für die Merkmalsvektoren verknüpft. Ein Beobachter sieht ausschließlich die erzeugten Merkmalsvektoren. Die Zustandsübergänge, welche durchlaufen worden sind, bleiben ihm aber verborgen (engl.: hidden).

Abbildung 2.3 stellt ein vereinfachtes Beispiel für ein Hidden-Markov-Modell zum Wort „haben“ dar. Die Modellierung erfolgt hier auf Basis von Phonemen[3]. Mittels der Übergänge zwischen den einzelnen Zuständen lassen sich unterschiedliche Sprechgeschwindigkeiten modellieren. Schnelles Sprechen manifestiert sich im Überspringen eines Zustandes (z. B. kann das Wort „haben“ als „habn“ ausgesprochen werden), während langsames Sprechen das Wiederholen eines Zustandes bedeutet (z. B. bei Betonung des Buchstabens „a“). Durch die Emissionswahrscheinlichkeit wird beschrieben, wie wahrscheinlich die verschiedenen Ausgabemöglichkeiten eines Zustands sind. Dadurch können Unterschiedliche Ausspracheweisen modelliert werden (im Beispiel kann „haben“ auch als „habm“ ausgesprochen werden).

---

[2] Benannt nach dem russischen Mathematiker Andrei Andrejewitsch Markow.

[3] Ein Phonem stellt die kleinste bedeutungsunterscheidende lautsprachliche Einheiten der menschlichen Sprache dar.

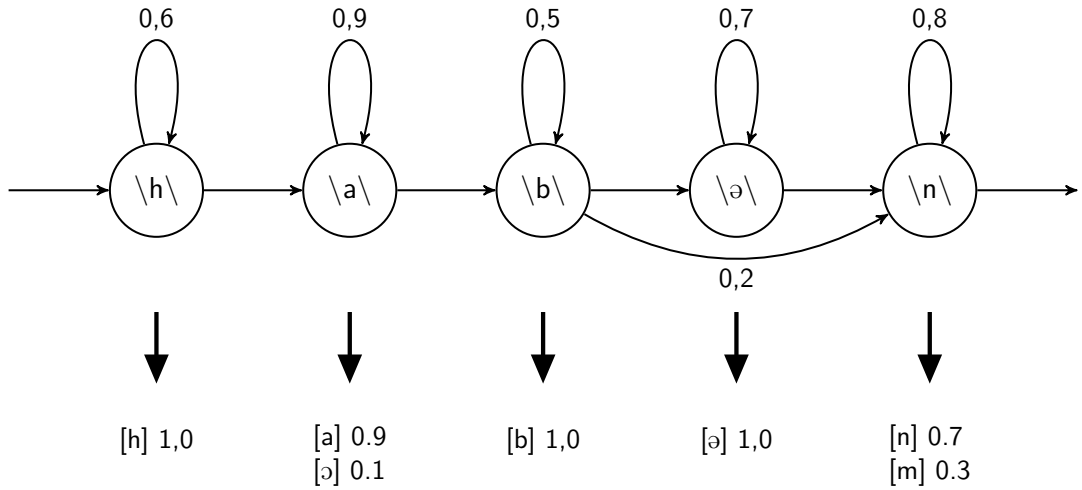


Abbildung 2.3: Vereinfachtes Beispiel für ein Hidden-Markov-Modell mit fiktiven Zustandsübergangs- und Emissionswahrscheinlichkeiten (nach [1]).

In der Praxis werden Hidden-Markov-Modelle nicht für einzelne Wörter gebildet, da bei größerem Vokabular die Anzahl der Hidden-Markov-Modelle zu groß werden würde. Anstelle dessen werden Wörter als Ketten von Grundeinheiten – typischerweise Phonemen, seltener Silben oder Halbsilben – dargestellt. Anschließend wird der Zusammenhang zwischen den Grundeinheiten und den Merkmalsvektoren mittels Hidden-Markov-Modellen statistisch beschrieben. Die einzelnen Modelle lassen sich dann zu größeren Modellen verketten. Die Parameter der Modelle – also die Zustandsübergangswahrscheinlichkeit und Emissionsverteilung – werden anhand großer Sprachkorpora geschätzt (der Vorgang wird auch als Trainieren bezeichnet). Diese Sprachkorpora enthalten idealerweise Sprachmuster von sehr vielen verschiedenen Personen in unterschiedlichen Sprachsituationen. Zur Modellierung der Emissionsverteilung werden traditionell Gaussian Mixture Models (GMM) eingesetzt. In den letzten Jahren wurden aber hybride Ansätze der akustischen Modellierung, welche künstliche neuronale Netzwerke zur Modellierung der Emissionsverteilung verwenden, immer populärer, da mit diesen bessere Erkennungsleistungen erzielt werden können [5, 6]. Es ist daher anzunehmen, dass solche Modelle sich als Standard etablieren werden.

Sprachmodell

Die natürliche Sprache stellt keine beliebige Kombination von Wörtern dar, sondern folgt den Regeln der Grammatik. Sowohl auf Basis dieser grammatikalischen Regeln als auch durch den inhaltlichen Kontext (z. B. ergibt „wir fahren zum *mehr*“ keinen Sinn), lassen sich Rückschlüsse auf die Wortwahrscheinlichkeit treffen. Diese Tatsache wird bei der Erkennung kontinuierlich gesprochener Sprache in Form des Sprachmodells berücksichtigt, um den Suchraum einzuschränken und falsche oder unwahrscheinliche Hypothesen auszuschließen.

Nach [3] ist das Sprachmodell eine Sammlung von A-priori-Kenntnissen über eine Sprache. Sprache wird in diesem Zusammenhang im abstrakten Sinne, also als eine Menge von Wortfolgen ohne bestimmte akustische Realisierung, verstanden. Das Sprachmodell stellt somit ein Regelsystem dar, welches die Struktur der Sprache modelliert, indem es beschreibt, welche Kombinationen von Wörtern möglich sind. Bei der konkreten Umsetzung des Sprachmodells werden zwei Ansätze unterschieden:

Statistisches Sprachmodell: Bei statistischen Sprachmodellen werden Auftrittswahrscheinlichkeiten von Wortgruppen mittels N-Gramm-Modellen beschrieben. Üblicherweise werden dafür Trigramm- und Bigramm-Modelle verwendet, also Wortgruppen von zwei- bis drei Wörtern betrachtet. Die Auftrittswahrscheinlichkeiten dieser Wortgruppen werden durch das Analysieren von – typischerweise domänenspezifischen – Sammlungen sprachlicher Daten ermittelt. Dieser Vorgang wird auch als Trainieren des Sprachmodells bezeichnet. Die sprachlichen Daten werden dementsprechend Trainingskorpus genannt.

Verwendung von Grammatiken: Ein Sprachmodell kann durch Grammatiken beschrieben werden. Eine Grammatik definiert dabei ein Regelwerk, nach dem Elemente der Sprache miteinander kombiniert werden können, sodass die resultierenden Ergebnisse syntaktisch korrekt sind. Grammatiken lassen sich in ein Erkennungsnetzwerk überführen, welches anschließend weiterverarbeitet werden kann. Grundsätzlich sind Grammatiken gut geeignet, um Einzelwörter oder kurze Wortfolgen zu beschreiben. Die vollständige Beschreibung natürlicher Sprache ist damit aber nicht möglich, da die Zusammenhänge zu komplex sind. Für die Realisierung von Diktieranwendungen mit großem Vokabular sind Grammatiken daher ungeeignet. Ein sinnvolles Anwendungsgebiet stellt hingegen die Sprachsteuerung dar.

Suchverfahren

Die Suche hat zum Ziel, jene Wortfolge zu ermitteln, für welche das Produkt aus akustischer und linguistischer Wahrscheinlichkeit am größten ist. Es muss also das Wissen aus dem akustischen Modell und dem Sprachmodell berücksichtigt werden.

Das am weitesten verbreitete Suchverfahren stellt die Viterbi-Suche dar. Diese kann beispielsweise wie folgt realisiert sein: Zunächst wird aus den sich im akustischen Modell befindlichen Markov-Ketten ein Netzwerk von Worthypothesen erzeugt. Anschließend wird zur Ermittlung der optimalen Wortfolge eine Pfadsuche mittels Viterbi-Algorithmus durchgeführt. Es gilt diejenigen Pfade zu finden, deren kombinierte Bewertung am besten ist, also auf denen das Produkt der Emissions- und Zustandsübergangswahrscheinlichkeiten maximiert wird. Die Liste der N besten Hypothesen wird dann in einer Nachverarbeitung mittels des Sprachmodells bewertet, um die beste Wortfolge zu ermitteln.

Die effiziente Realisierung des Suchalgorithmus ist von großer Bedeutung. Es gilt, den kürzesten Weg durch das Netzwerk zu finden, gleichzeitig aber einen möglichst kleinen Bereich des Suchraums auszuwerten, um schnell – möglichst in Echtzeit – zum Ergebnis zu kommen. Es haben sich verschiedene Ansätze und Strategien für effiziente Suchverfahren etabliert, die hier aber nicht weiter behandelt werden sollen. Als vertiefende Literatur sei dazu auf [15] verwiesen.

2.1.6 Evaluationskriterien

Als Gütemaße für die Beurteilung der Erkennungsleistung eines Spracherkennungssystems werden in der Regel die Wortfehlerrate (engl.: Word Error Rate, WER) und die Wortakkuratheit (engl.: Word Accuracy, WA) verwendet [2]. Beide Maße beruhen auf einem Vergleich der Referenzwortfolge – also jener Wortfolge, die tatsächlich gesprochen wurde – mit der erkannten Wortfolge.

Für die Berechnung der Wortfehlerrate werden Erkennungsfehler in drei Kategorien eingeteilt: Wird ein Wort falsch erkannt, liegt ein *Ersetzungsfehler* vor. Ein *Auslassungsfehler* tritt dann auf, wenn ein Wort weggelassen wurde – zum Beispiel weil es als Störgeräusch interpretiert wurde. Bei einem *Einfügefehler* wurde hingegen in die erkannte Wortfolge ein zusätzliches Wort eingefügt – beispielsweise weil ein Störgeräusch als ein Wort interpretiert wurde. Die Anzahl der aufgetretenen Fehler jeder Kategorie wird ermittelt. Für die Wortfehlerrate gilt dann:

$$WER = \frac{\#Ersetzungsfehler + \#Einfügefehler + \#Auslassungsfehler}{\#Referenzwörter} \cdot 100 \quad (2.4)$$

Die Berechnung der Wortakkuratheit erfolgt nach:

$$WA = 100 - WER \quad (2.5)$$

2.1.7 Überblick Spracherkennungssysteme und -technologien

Mittlerweile gibt es eine beachtliche Anzahl von Spracherkennungssystemen und -technologien, mit denen Spracherkennungsanwendungen umgesetzt werden können. Neben proprietären Software Development Kits (SDK) für verschiedene Plattformen, existieren auch mehrere quelloffene Spracherkennungssysteme, welche großteils an Universitäten entwickelt wurden und im wissenschaftlichen Umfeld oft Verwendung finden. Außerdem stellen viele Unternehmen Webservices für Spracherkennung sowie APIs zur

Verfügung, um diese Services einfach in eigene Anwendungen integrieren zu können. Im Folgenden werden einige populäre Beispiele kurz vorgestellt:

Dragon NaturallySpeaking SDKs: Nuance Communications gilt als einer der Marktführer im Bereich der Sprachtechnologie [30]. Bekannt ist das Unternehmen vor allem für sein Programmpaket Dragon NaturallySpeaking (Diktierfunktion und Sprachsteuerung unter Windows) und die Diktiersoftware Dragon Dictate für Mac OS und iOS. Auch die Technologie, welche hinter Apple's Siri steckt, stammt von Nuance [31]. Die Firma stellt ebenfalls zahlreiche SDKs [32, 33] zur Verfügung, welche es Entwicklern erlauben, ihre Spracherkennungstechnologie in eigene Anwendungen zu integrieren. Es existiert ein Client SDK für Windowsanwendungen, ein Server SDK und jeweils ein Mobile SDK für iOS, Android und Windows Phone. Unterstützt werden die Sprachen Deutsch, Englisch, Französisch, Italienisch, Spanisch und Niederländisch.

Microsoft Speech API: Die Microsoft Speech API (SAPI) [34] erlaubt es, auf die in Windows eingebaute Spracherkennung und Sprachsynthese zuzugreifen, um diese für eigene Anwendungen zu nutzen. Die aktuelle Version 5.4 unterstützt die Sprachen Englisch, Spanisch, Französisch, Deutsch und Chinesisch. Die SAPI wird vorrangig dazu verwendet, Windows-Anwendungen mittels vordefinierter Grammatiken zu steuern. Sie unterstützt aber auch die Erkennung kontinuierlich gesprochener Sprache. Mittels SAPI kann auch eine Server-basierte Spracherkennung umgesetzt werden. Für die Umsetzung der Spracherkennungsengine am Server stellt Microsoft das Speech Platform SDK [35] bereit, welches 26 Sprachen unterstützt.

CMU Sphinx: CMU Sphinx [36] bezeichnet eine Gruppe von quelloffenen Spracherkennungssystemen, die an der Carnegie Mellon University (CMU) entwickelt wurden. Das aktuellste System der Gruppe stellt das in der Sprache Java geschriebene Sphinx 4 dar. Sphinx 4 zeichnet sich durch seinen modularen Aufbau und die damit verbundene Flexibilität aus. Praktisch jeder Teil des Systems ist austauschbar, weshalb es vor allem für Forscher interessant ist. Die Sphinx-Entwickler stellen akustische Modelle für Deutsch, Englisch, Französisch, Chinesisch, Spanisch, Niederländisch und Russisch bereit. Mit dem Werkzeug SphinxTrain können außerdem eigene akustische Modelle für andere Sprachen trainiert werden. Des Weiteren steht mit PocketSphinx auch eine leichtgewichtige Version des Sphinx-Systems zur Verfügung (vgl. Abschnitt 2.3.2).

Julius: Julius [37] ist eine quelloffene Spracherkennungs-Engine, die für die Erkennung von kontinuierlich gesprochener Sprache mit großem Vokabular entwickelt wurde. Die am Nagoya Institute of Technology entstandene Software war ursprünglich für Linux ausgelegt, kann aber auch unter Windows verwendet werden. Um Julius

verwenden zu können, wird ein Sprachmodell und ein akustisches Modell benötigt. Diese Modelle können beispielsweise mit dem Hidden Markov Model Toolkit (HTK) erstellt werden.

HTK: Das HTK [38] ist eine quelloffene Softwaresammlung zur automatischen Spracherkennung. Das ursprünglich an der Cambridge University entwickelte Toolkit ist in C geschrieben und wird oft in der Forschung eingesetzt. Es umfasst vor allem Module für das Trainieren von N-Gramm-Modellen und HMMs, enthält aber auch Programme für die Dekodierung (HDecode). Durch die Kombination der Module kann ein vollständiges Spracherkennungssystem umgesetzt werden.

Google Web Speech API: Die Web Speech API ist eine JavaScript-Programmierschnittstelle, die es ermöglicht, Spracherkennung und Sprachsynthese in Webanwendungen zu verwenden. Die Spracherkennungsfunktionalität ist derzeit nur im Chrome Browser implementiert, wobei die eigentliche Spracherkennung aber serverseitig erfolgt. Es wird dazu derselbe Webservice aufgerufen, der auch von Google Voice Search verwendet wird. Der aktuelle Stand der Web Speech API wird im Detail in Abschnitt 2.3.3 beschrieben.

2.2 Die Webtechnologien HTML, CSS, JavaScript und XML

Technologien, welche genutzt werden, um Inhalte im World Wide Web (WWW) zu erzeugen oder auszutauschen, werden unter dem Sammelbegriff Webtechnologien zusammengefasst. Im Zuge der Umsetzung des Prototyps werden vor allem die Webtechnologien HTML, CSS, JavaScript und XML Anwendung finden. Diese Technologien werden deshalb folgend kurz vorgestellt.

2.2.1 HTML

Die Hypertext Markup Language (HTML) ist eine textbasierte Auszeichnungssprache, welche das Standardformat für die textliche Strukturierung und semantische Auszeichnung von Webseiten darstellt. HTML-Dokumente bilden die Grundlage des World Wide Web und werden durch einen Webbrowser gerendert und dargestellt. Die Sprache selbst wurde ursprünglich 1989 von Tim Berners-Lee festgelegt, seit ihrer Einführung ständig weiterentwickelt und 1997 schließlich als HTML 4 durch das W3C standardisiert. Die aktuelle Version des HTML-Standards stellt HTML 4.01 dar und wurde Ende 1999 veröffentlicht. Mit HTML 5 befindet sich derzeit die fünfte Version des Standards in der Entwicklung [16].

Die Strukturierung eines HTML-Dokumentes erfolgt durch die Auszeichnung bestimmter Textteile mittels sogenannter Tags. Die auf diese Weise ausgezeichneten Textteile, werden als HTML-Element bezeichnet und können beispielsweise Tabellen, Grafiken, Überschriften, Textabsätze oder Listen sein.[4] Die HTML-Tags selbst werden in spitzen Klammern notiert und treten in der Regel paarweise, also in Form eines einleitenden und schließenden Tags, auf, können aber auch sogenannte leere Elemente bilden. HTML-Elemente können ineinander verschachtelt werden, wodurch eine hierarchische Struktur entsteht. Zur Veranschaulichung sei folgend ein einfaches dem HTML 5-Standard entsprechendes Dokument dargestellt:

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Titel</title>
5    </head>
6    <body>
7      <p>Hallo <strong>Welt!</strong>
8      <br>Zeilenumbruch
9      </p>
10   </body>
11 </html>
```

Listing 2.1: Einfaches Beispiel für ein HTML-Dokument.

War HTML ursprünglich nur als Auszeichnungssprache für statische Dokumente gedacht, ist gerade die Umsetzung von dynamischen Webanwendungen in den vergangenen Jahren immer wichtiger geworden. Diesem Trend Rechnung tragend, ist eine zentrale Zielsetzung der Entwicklung von HTML 5, die Schaffung neuer Möglichkeiten zur Umsetzung von Webanwendungen. Neben der Erweiterung der eigentlichen Auszeichnungssprache – zum Beispiel durch neue Elemente für Audio- und Videoinhalte – werden für diesen Zweck zusätzliche JavaScript-Programmierschnittstellen entwickelt, die beispielsweise den Zugriff auf das Dateisystem erlauben (File System API), erweiterte Mechanismen zur Hintergrundverarbeitung einführen (Web Worker API) oder neue Möglichkeiten für die clientseitige Datenhaltung (Indexed Database API, Web Storage API) und die Client-Server-Kommunikation (Web Sockets API, Server-Sent Events) schaffen [16]. Im allgemeinen Sprachgebrauch wird unter dem Begriff HTML 5 daher nicht nur die eigentliche HTML-Spezifikation verstanden, vielmehr werden auch alle diese APIs mit eingeschlossen.[5] Obwohl HTML 5 sich derzeit noch in der Entwicklung befindet, wird ein großer Teil der Neuerungen bereits von den meisten modernen Webbrowsern – wie zum Beispiel

[4] Einen guten Überblick bietet die Seite `http://www.w3schools.com/tags`.
[5] Siehe dazu auch `https://github.com/SirPepe/SpecGraph`.

aktuellen Versionen von Chrome, Firefox, Safari oder Internet Explorer – unterstützt.[6]

2.2.2 CSS

Bei Cascading Style Sheets (CSS) handelt es sich um eine deklarative Sprache, welche der Formatierung und Gestaltung des Inhalts von HTML-Dokumenten dient. Vor der Einführung von CSS war eine konsequente Trennung des Aussehens und Inhalts eines HTML-Dokuments nicht möglich, da HTML schlicht nicht dafür ausgelegt war. Um Webseiten zu gestalten, wurde deshalb häufig auf zahlreiche unsaubere Tricks zurückgegriffen – ein klassisches Beispiel ist die Zweckentfremdung von Tabellen zur Steuerung des Layouts. Erstmals 1996 durch das W3C standardisiert, wurden die Möglichkeiten von CCS in den vergangenen Jahren ständig erweitert. Die in den meisten modernen Webbrowsern gut unterstützte Version 2.1, stellt heute ein mächtiges Instrument zum Gestalten von HTML-Dokumenten dar. [16]

Die Gestaltung von HTML-Dokumenten mittels CSS erfolgt über das Festlegen sogenannter CSS-Regeln, deren Aufbau gemäß der in [39] definierten Syntax erfolgt. Über Selektoren wird festgelegt, auf welche HTML-Elemente die entsprechende Regel anzuwenden ist. Durch Eigenschaftsdeklarationen lassen sich die Darstellungsattribute der entsprechenden HTML-Elemente – zum Beispiel Höhe, Breite, Hintergrundfarbe oder Anzeigeposition – anpassen. Das folgende Beispiel zeigt eine einfache CSS-Regel, welche bewirkt, dass der Text in Textabsätzen blau und fett dargestellt wird:

```
1 p {
2   color: blue;
3   font-weight: bold;
4 }
```

Listing 2.2: Einfaches Beispiel für eine CSS-Regel.

Zur Einbindung von CSS-Regeln in HTML-Dokumenten existieren verschiedene Möglichkeiten. CSS-Regeln können direkt im Kopfbereich des HTML-Dokuments definiert oder eben dort, durch einen Verweis auf eine separate CSS-Datei, eingebunden werden. Zudem können Eigenschaftsdeklarationen auch als Attribut direkt in ein HTML-Tag geschrieben werden. Des Weiteren verfügt CSS über ein Vererbungsmodell für Auszeichnungsattribute. Dadurch können Auszeichnungen von Elementen an untergeordnete Elemente weitervererbt werden, wodurch weniger Anweisungen benötigt werden [39].

2.2.3 JavaScript

JavaScript ist eine Programmiersprache, die mit der Absicht entwickelt wurde, die eingeschränkten Möglichkeiten von HTML zu erweitern und ein hohes Maß an Dynamik und

[6] Das lässt sich zum Beispiel mittels `http://html5test.com` testen.

Interaktion zu ermöglichen. Die Entwickelung von JavaScript durch Netscape begann 1995 zunächst unter dem Namen LiveScript; 1998 erfolgte durch die European Computer Manufacturers Association (ECMA) die Normung des Sprachkerns als Standard ECMA-262 (ECMAScript). Heute unterstützen alle modernen Webbrowser JavaScript. Der Browser-übergreifende Einsatz von JavaScript ist trotz Standardisierung aber nach wie vor nicht unproblematisch, da der Standard nicht von allen Webbrowser-Herstellern exakt umgesetzt wurde [16].

Bei JavaScript handelt es sich um eine Skriptsprache, die dynamisch während der Laufzeit durch den Webbrowser – also clientseitig – interpretiert wird. Sie verfügt über eine C-ähnliche Syntax, ist schwach typisiert und unterstützt sowohl objektorientierte als auch prozedurale und funktionale Programmierung. Mittels JavaScript und dem Document Object Model (DOM), einem Schnittstellen-Standard für den Zugriff auf HTML-Dokumente, lässt sich jedes Element innerhalb der HTML-Struktur manipulieren. Außerdem erlaubt JavaScript Ereignisse, welche durch Benutzerinteraktionen ausgelöst wurden (z. B. das Klicken auf eine Schaltfläche), abzufangen und zu verarbeiten. Der JavaScript-Programmcode kann entweder direkt in HTML-Dokumenten eingebettet oder mittels eines Verweises auf eine separate JavaScript-Datei eingebunden werden.

Die Entwicklung von Webanwendungen mittels JavaScript wird durch zahlreiche Bibliotheken erleichtert. Die populärste dieser Bibliotheken ist jQuery [40]. Sie stellt unter anderem komfortable Funktionen für die DOM-Manipulation und das Event Handling bereit, adressiert durch die Implementierung zahlreicher konsistenter und Browserunabhängiger Funktionen für allgemeine Programmieraufgaben, aber auch das Problem der Browser-Inkompatibilität [41].

2.2.4 XML

Bei der Extensible Markup Language – kurz XML – handelt es sich, wie bei HTML, um eine textbasierte Auszeichnunssprache. Im Unterschied zu HTML ist XML aber eine sogenannte Metasprache. Während HTML eine fest definierte Grammatik zur Beschreibung von Webseiten definiert, beschreibt XML ein allgemeines Regelwerk, auf dessen Basis anwendungsspezifische Sprachen definiert werden können.

XML ist ein offener Standard, der erstmals Anfang 1988 durch das W3C definiert wurde. Seitdem ständig weiterentwickelt, liegt er heute in der fünften Auflage vor [42]. Der Standard ist weit verbreitet und findet häufig beim Datenaustausch zwischen Computersystemen, insbesondere im Internet, Anwendung. Da textbasiert, können XML-Dokumente sowohl durch Mensch als auch Maschine interpretiert werden.

Der logische Aufbau eines XML-Dokuments ist hierarchisch und entspricht einer Baumstruktur. Die Beschreibung der Daten erfolgt durch Tags und Attribute. Entspricht ein XML-Dokument den im Standard definierten Syntax-Regeln, wird es als wohlgeformt

bezeichnet. Entscheidend dafür ist unter anderem, dass ein Dokument genau ein Wurzelelement hat, jedes geöffnete Tag ein dazugehöriges schließendes Tag besitzt und die Verschachtelung der Elemente korrekt durchgeführt wurde [42]. Das Listing 2.3 zeigt ein einfaches, wohlgeformtes XML-Dokument.

```
<?xml version="1.0" encoding="UTF-8"?>
<personen>
  <person>
    <name>Peter Silie</name>
    <alter>40</alter>
  </person>
  <person>
    <name>Karl Toffel</name>
    <alter>50</alter>
  </person>
</personen>
```

Listing 2.3: Einfaches Beispiel für ein XML-Dokument

2.3 Spracherkennung im Browser

Im folgenden Abschnitt werden Möglichkeiten zur Spracherkennung im Browser vorgestellt. Dazu soll zunächst die Thematik der Audioaufnahme und -wiedergabe im Browser kurz beleuchtet werden.

2.3.1 Audioaufnahme und -wiedergabe im Browser

Da HTML ursprünglich lediglich zur Anzeige statischer Dokumente gedacht war, gab es in den Anfangszeiten des WWW keine Möglichkeit Audioinhalte in Webseiten einzubinden. Im Zuge der weiteren Entwicklung des WWW wurde es schließlich aber als sinnvoll erachtet, Webseiten mit Audioinhalten zu verbinden.

Microsoft schuf für den Internet Explorer mit dem HTML-Element `<bgsound>` erstmals eine technische Möglichkeit, eine Audiodatei im Hintergrund abzuspielen. Da `<bgsound>` weder in den HTML-Standard aufgenommen noch von anderen Browsern aufgegriffen wurde, konnte es sich aber niemals durchsetzen. Als erfolgreicher stellte sich hingegen das `<embed>`-Element heraus. Mitte der 90er Jahre erstmals im Netscape Navigator integriert, wurde das Element auch von anderen Browsern umgesetzt und schließlich in die HTML 5-Spezifikation aufgenommen. Das `<embed>`-Element fungiert als Container für Nicht-HTML-Inhalte, deren Darstellung in der Regel ein Browser-Plug-in voraussetzt. Durch die Einbindung von Adobe Flash-Inhalten mittels `<embed>`-Element, bestand erstmals die Möglichkeit, Audiowiedergabe browserübergreifend zu

realisieren. Obwohl Flash die Installation eines Browser-Plug-ins voraussetzte, entwickelte es sich in weiterer Folge als De-facto-Standard sowohl für die Wiedergabe von Audio- als auch Videoinhalten im WWW [16, 17].

Die Aufnahme von Audioinhalten konnte bis vor kurzem ausschließlich mittels Plug-in-Technologien – dazu zählen neben dem bereits erwähnten Flash zum Beispiel auch Microsoft Silverlight, Apple QuickTime oder Java Applets – bewerkstelligt werden, da nur diese den Zugriff auf das dafür erforderliche Mikrofon ermöglichten. Obwohl diese Technologien ihren Zweck immer noch erfüllen, ist deren Einsatz mit bedeutenden Nachteilen verbunden – u. a. die Erfordernis der manuellen Installation des Plug-ins durch die benutzende Person, die oftmals existierenden Einschränkungen hinsichtlich unterstützter Plattformen (z. B. Probleme mit Flash auf Apple-Geräten) oder immer wieder auftretende Sicherheitslücken [43, 44]. Im Übrigen handelt es sich bei Plug-in-Technologien meist auch um proprietäre Technologien und nicht um standardisierte Webtechnologien.

Mit HTML 5 sollen nun neue Elemente und APIs eingeführt werden, welche ähnliche Möglichkeiten zur Verarbeitung von Audio- und Videoinhalten wie vorhandene Plug-in-Technologien bieten, nicht aber deren Nachteile besitzen. Bereits jetzt werden die meisten dieser Erweiterungen durch moderne Webbrowser unterstützt. Für die Audioaufnahme und -wiedergabe sind vor allem die folgenden Erweiterungen von Bedeutung:

`<audio>`-Element: Das `<audio>`-Element ist ein Container für Audioinhalte. Mit ihm können Audiodateien in ein HTML-Dokument eingebettet werden, sodass diese direkt im Browser wiedergegeben werden können. Probleme ergeben sich derzeit noch bezüglich der Unterstützung von Audioformaten. Die HTML 5-Spezifikation beschreibt zwar die Funktionalität des Elements, gibt aber nicht vor, welche Audioformate unterstützt werden müssen. Da die unterschiedlichen Browserhersteller diesbezüglich noch keine Einigung erzielen konnten, werden aktuell unterschiedliche Audioformate in den verschiedenen Browsern unterstützt [16].

Web Audio API: Im Gegensatz zum eher einfachen `<audio>`-Element bietet die Web Audio API deutlich mehr Funktionalität. Die JavaScript API stellt unter anderem sehr umfangreiche Möglichkeiten für Audioanalyse, -synthese und -filterung bereit. Die eigentliche Audioverarbeitung ist dabei nicht in JavaScript implementiert, sondern wird innerhalb des Browsers mit Sprachen wie C oder C++, die wesentlich schneller sind, durchgeführt. Die Web Audio API hat sich im Standardisierungsprozess gegenüber der von Mozilla entwickelten vergleichbaren Audio Data API durchgesetzt [17, 45].

WebRTC: Mit WebRTC (Web Real Time Communications) wird die Absicht verfolgt, Echtzeitkommunikation zwischen Browsern zu ermöglichen (z. B. zur Realisierung von Use Cases wie Chats oder Videotelefonie) [46]. WebRTC umfasst mehrere

JavaScript APIs zur Aufnahme, Kodierung und Übertragung von Daten, darunter auch die API getUserMedia, welche den Zugriff auf Multimedia Streams von lokalen Geräten wie Mikrofon oder Webcam ermöglicht [47]. Die getUserMedia API und die Web Audio API können relativ einfach integriert werden, sodass sich die Audioaufnahme im Browser ohne Plug-in realisieren lässt [17].

Nachdem nun die technischen Möglichkeiten zu Audioaufnahme und -wiedergabe im Browser vorgestellt wurden, gilt es zu untersuchen, wie automatische Spracherkennung im Browser realisiert werden kann. Im Folgenden werden mögliche Lösungsansätze vorgestellt. Ein besonderer Fokus wird dabei auf das Spracherkennungssystem Pocketsphinx.js sowie die Web Speech API gelegt.

2.3.2 Pocketsphinx.js

Das leichtgewichtige Spracherkennungssystem PocketSphinx ist ein Abkömmling der CMU Sphinx-Familie (vgl. Abschnitt 2.1.7). Das in den Programmiersprachen C und C++ geschriebene System wurde speziell für den Einsatz in den Bereichen Mobilgeräte und Embedded Systems entwickelt und gilt als relativ schnell [18]. Für die Spracherkennung im Browser ist seit Mitte 2013 mit PocketSphinx.js [48] eine JavaScript-Portierung des PocketSphinx-Systems verfügbar. Die Entwickler von PocketSphinx.js nutzen den LLVM-zu-JavaScript-Compiler (Low Level Virtual Machine) Emscripten und das Compiler Frontend Clang, um den C- bzw. C++-Quellcode in JavaScript zu überführen.

Mit PocketSphinx.js ist eine rein clientseitige und somit auch offlinefähige Spracherkennung realisierbar. Das System ist weder abhängig von Serverkomponenten noch nutzt es Plug-in-Technologien. PocketSphinx.js ermöglicht sowohl die Erkennung kontinuierlich gesprochener Sprache mittels statistischer Sprachmodelle und Grammatiken als auch das Erkennen von Schlüsselwörtern.

Komponenten von PocketSphinx.js

Die Programmierschnittstelle `pocketsphinx.js` enthält den nach JavaScript portierten PocketSphinx-Quellcode und implementiert die eigentliche Spracherkennung. Um die Handhabung von `pocketsphinx.js` zu vereinfachen und die der Spracherkennung vorgelagerte Audioaufnahme zu realisieren, verfügt das System über weitere Komponenten:

- Mit der Wrapper API `recognizer.js` existiert eine einfache Schnittstelle, welche die Funktionen der API `pocketsphinx.js` kapselt und diese innerhalb eines Web Workers ausführt. Durch die Verwendung dieses Wrappers wird erreicht, dass der Download des typischerweise mehrere MB großen JavaScript-Codes und die relativ rechenintensive Spracherkennung im Hintergrund ausgeführt werden, was zum

Vorteil hat, dass der UI Thread dadurch nicht blockiert und die Darstellung der Benutzeroberfläche nicht beeinträchtigt wird.

- Die Schnittstelle `callbackManager.js` stellt Hilfsfunktionen bereit, welche die Kommunikation zwischen den einzelnen Systemkomponenten mittels Callback-Mechanismus umsetzen.

- Die Recorder API `audioRecorder.js` nutzt die Web Audio API in Kombination mit der getUserMedia API (vgl. Abschnitt 2.3.1), um die von der Benutzerin oder dem Benutzer gesprochene Sprache aufzunehmen und in Form eines `AudioBuffer`-Objektes an den Spracherkenner weiterzugeben. Da die Web Audio API und getUserMedia API derzeit noch nicht von allen Browsern vollständig unterstützt werden, ist der Einsatz von `audioRecorder.js` im Moment auf Google Chrome, Mozilla Firefox und den Opera Browser beschränkt [49, 50].

Als Ergebnis der Spracherkennung wird die wahrscheinlichste Hypothese zur aufgenommenen Wortfolge – sowohl in Form einer Zeichenkette als auch in Form einer Liste, welche die einzelnen Wortsegmente inklusiver Angaben über Start- und Endbereich im Zeitverlauf enthält – geliefert.

Sprachmodelle und akustische Modelle

Statistische Sprachmodelle und akustische Modelle werden beim Portieren in den JavaScript-Code integriert. Soll die Sprache währen der Laufzeit geändert werden können, müssen daher alle relevanten Modelle bereits beim Kompilieren hinzugefügt werden. Beim Initialisieren des Spracherkenners kann dann bestimmt werden, welche Modelle verwendet werden sollen. Die Modelle können sowohl in die Schnittstelle `pocketsphinx.js` als auch in separate JavaScript-Dateien, die zusätzlich in das HTML-Dokument eingebunden werden, kodiert werden. Die resultierenden JavaScript-Dateien erreichen jedenfalls bei umfangreicheren Modellen recht schnell Größen, welche sich im zweistelligen Megabyte-Bereich bewegen[7]. Das kann unter Umständen problematisch werden (z. B. wenn die Anwendung über eine langsame Internetverbindung geladen wird).

Als akustische Modelle können dieselben Modelle wie für andere CMU Sphinx-Systeme verwendet werden. Die Sphinx-Entwickler stellen mehrere Modelle für verschiedene Sprachen bereit (vgl. Abschnitt 2.1.7). Alternativ können solche Modelle aber auch über VoxForge [51] – einem Open-Source-Projekt, welches zum Ziel, hat Sprachdaten verschiedener Sprachen zu sammeln und frei zur Verfügung zu stellen – bezogen oder mittels SphinxTrain selbst trainiert werden.

[7] Beispielsweise sind es bereits 11,8 MB für die sehr einfache Demoseite auf `http://syl22-00.github.io/pocketsphinx.js/live-demo-chinese.html`.

Die Verwendung eines statistischen Sprachmodells ist optional. Analog zu akustischen Modellen können bestehende Sprachmodelle anderer CMU Sphinx-Systeme übernommen werden. Um eigene Modelle zu trainieren, kann das Werkzeug CMUCLMTK (CMU-Cambridge Language Modeling Toolkit) oder ein vergleichbares Werkzeug – wie beispielsweise MITLM (Massachusetts Institute of Technology Language Modeling) – verwendet werden [52]. Das VoxForge-Projekt stellt ebenfalls statistische Sprachmodelle inklusive der dazu passenden Aussprachewörterbücher bereit.

Grammatiken und Schlüsselwörter können während der Laufzeit hinzugefügt werden. Die Schnittstelle `recognizer.js` bietet dafür die Methoden `addGrammar` und `addKeyword` an. Alle Schlüsselwörter und Wörter, welche in den Grammatiken oder im statistischen Sprachmodell verwendet werden, müssen mittels des Aussprachewörterbuchs dem System bekannt gemacht werden. Das Wörterbuch kann, wie auch das statistische Sprachmodell, bereits beim Kompilieren in `pocketsphinx.js` integriert werden. Alternativ können Einträge aber auch zur Laufzeit geladen werden (`addWords`-Methode der Schnittstelle `recognizer.js`).

2.3.3 Die Web Speech API

Im Gegensatz zu PocketSphinx.js handelt es sich bei der Web Speech API nicht um ein Spracherkennungssystem, sondern um eine im Browser integrierte JavaScript-Programmierschnittstelle, die Spracherkennungs- und Sprachsynthese-Funktionalitäten bereitstellt, gleichzeitig aber unabhängig von der konkreten Implementierung dieser Funktionalitäten ist.

Entwicklung

Die HTML Speech Incubator Group des W3C untersuchte ab 2010 Möglichkeiten zur Erweiterung von HTML 5, um Use Cases, wie beispielsweise sprachbasierte Websuche, Sprachsteuerung von Webanwendungen, Sprachsynthese (engl.: Text-to-Speech, TTS) und Sprachübersetzung zu adressieren. Der Abschlussbericht der Gruppe enthält einen Lösungsvorschlag, dessen zentrale Bestandteile eine JavaScript API (Speech Web API) sowie ein auf das WebSocket-Protokoll[8] basierendes Kommunikationsprotokoll (Speech Protocol) darstellen [55].

Die vorgeschlagene JavaScript API soll Webentwicklerinnen und Webentwicklern die programmatische Steuerung und Parametrierung von Spracherkennungs- bzw. Sprachsynthese-Funktionalitäten ermöglichen. Im Hintergrund greift sie dazu über andere APIs auf das Mikrofon bzw. die Lautsprecher zu und wickelt die Kommunikation mit den Speech Services ab. Wie die Kommunikation mit den Speech Services (ASR- oder TTS-

[8] Das WebSocket-Protokoll ist ein auf TCP basierendes Protokoll für die bidirektionale Kommunikation zwischen einer Webanwendung und einem WebSocket-Server.

System) konkret zu erfolgen hat, wird dabei durch das Speech Protocol definiert. Der Browser stellt über eine Schnittstelle Standard-Speech-Services zur Verfügung. Der Entwicklerin bzw. dem Entwickler sollte es aber ebenso frei stehen, einen eigenen Service zu verwenden. Bei einem Speech Services kann es sich dabei sowohl um einen lokalen als auch um einen entfernten Dienst handeln. Abbildung 2.4 stellt die beschriebenen Zusammenhänge zum besseren Verständnis grafisch dar.

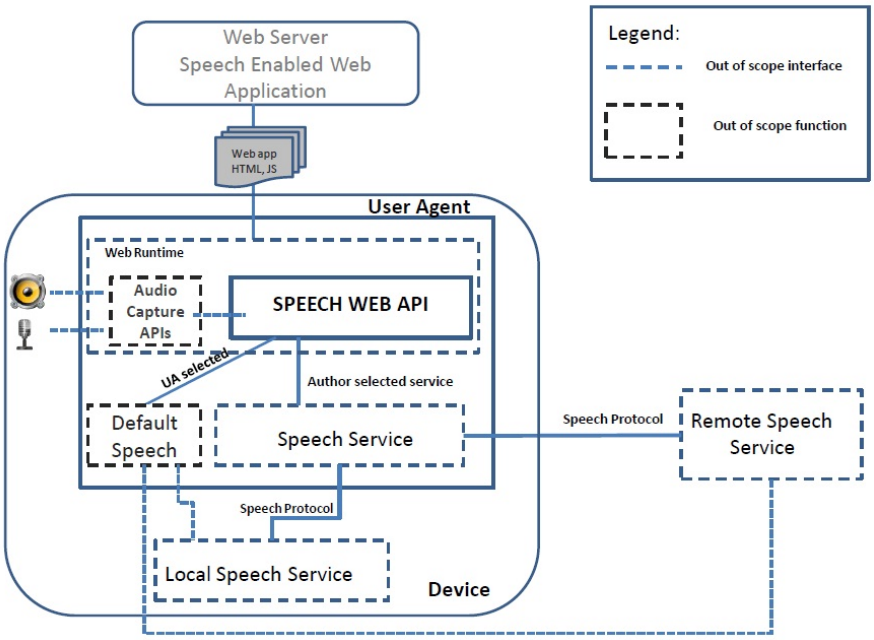


Abbildung 2.4: Übersicht zur Speech Web API [55].

Auf Basis des Abschlussberichtes der HTML Speech Incubator Group wurde in weiterer Folge durch die Speech API Community Group die Spezifikation für die JavaScript API – die zwischenzeitig in Web Speech API umbenannt wurde – erarbeitet [56]. Eine erste Implementierung der Web Speech API erfolgte durch Google und ist – wenn auch nicht ganz vollständig (vgl. Abschnitt 2.3.3) – im Google Chrome Browser ab Version 25 inkludiert. Mittlerweile werden auch Teile der API (Sprachsynthese) von Apples Safari Browser unterstützt. Auch Mozilla hat bekannt gegeben, an einer Implementierung für den Firefox Browser zu arbeiten [57, 58]. Diese Entwicklungen machen es sehr wahrscheinlich, dass sich die API in Zukunft auch wirklich als W3C-Standard durchsetzen wird.

API Spezifikation

Die Funktionalität für die Spracherkennung wird in der API durch die Schnittstelle `SpeechRecognition` abgebildet. Neben Methoden für das Starten, Beenden und Abbrechen der Spracherkennung, stellt die Schnittstelle elf Methoden für das Event Handling bereit. Damit lassen sich diverse Ereignisse, wie beispielsweise das Auftreten verschie-

dener Fehler oder den Erhalt eines Resultats vom Spracherkennungsservice, behandeln. Ebenfalls definiert sie diverse Attribute zur Konfiguration der Spracherkennung. So kann zum Beispiel die benutzte Sprache oder die maximale Anzahl der alternativen Hypothesen, die vom Spracherkennungsservice geliefert werden sollen, eingestellt werden. Ebenfalls kann definiert werden, ob die Spracherkennung kontinuierlich („continuous mode“) oder einmalig erfolgen soll. Wird die kontinuierliche Spracherkennung verwendet, hört die API ununterbrochen mit und liefert regelmäßig Resultate zurück. Andernfalls muss die Spracherkennung nach jedem erhaltenen Resultat neu gestartet werden.

Damit die Web Speech API auf das Mikrofon zugreifen kann, muss die benutzende Person ihre Erlaubnis erteilen. Sobald die Spracherkennung programmatisch gestartet wird, wird dazu im Browser ein Pop-up-Fenster eingeblendet. Erst nachdem der Zugriff akzeptiert wurde, beginnt die API mit dem Zuhören. Mit dieser Maßnahme wird die missbräuchliche Verwendung der API, um unbemerkt im Hintergrund zuzuhören und den Benutzer bzw. die Benutzerin auszuspionieren, verhindert.

Wird durch die API Sprache wahrgenommen, wird diese aufgenommen und im Hintergrund asynchron an den Spracherkennungsservice gesendet. Im Fall, dass die gesprochene Wortfolge erkannt wurde, wird ein entsprechendes Event ausgelöst und das Ergebnis als Parameter in Form eines `SpeechRecognitionResultList`-Objekts übergeben. Das `SpeechRecognitionResultList`-Objekt enthält im Falle kontinuierlicher Erkennung alle Resultate der aktuellen Sitzung. Ist hingegen die kontinuierliche Erkennung nicht aktiviert, enthält die Liste lediglich ein einzelnes Resultat. Resultate werden jedenfalls als `SpeechRecognitionResult`-Objekt geliefert, wobei ein solches Objekt wiederum mehrere Objekte vom Typ `SpeechAlternative` enthalten kann. Ein `SpeechAlternative`-Objekt repräsentiert eine vom Spracherkenner ermittelte Hypothese, bestehend aus der erkannten Wortfolge und einer Bewertung für die Wahrscheinlichkeit der Hypothese (Attribut `confidence` mit einem Wert zwischen null und eins). Die Alternativen im `SpeechRecognitionResult` sind nach der Wahrscheinlichkeit geordnet, sodass das wahrscheinlichste Ergebnis an erster Stelle steht. Abbildung 2.5 stellt die Zusammenhänge dar.

Die API Spezifikation sieht vor, dass mittels der Schnittstelle `SpeechGrammar` eigene Grammatiken zur Verbesserung der Spracherkennung verwendet werden können. Diese Schnittstelle stellt zwar Methoden für das Laden von Grammatikdateien bereit, in der zum Zeitpunkt der Erstellung dieser Arbeit aktuellen Version der Spezifikation ist aber nicht definiert, welches Format zur Beschreibung der Grammatiken verwendet werden soll. Hier ist die Spezifikation somit noch unvollständig.

Neben der soeben beschriebenen Spracherkennung stellt die API mit der Schnittstelle `SpeechSynthesis` auch Funktionen für die Sprachsynthese bereit. Diese sollen hier aber nicht weiter betrachtet werden, da sie in der vorliegenden Arbeit keine Verwendung finden werden.

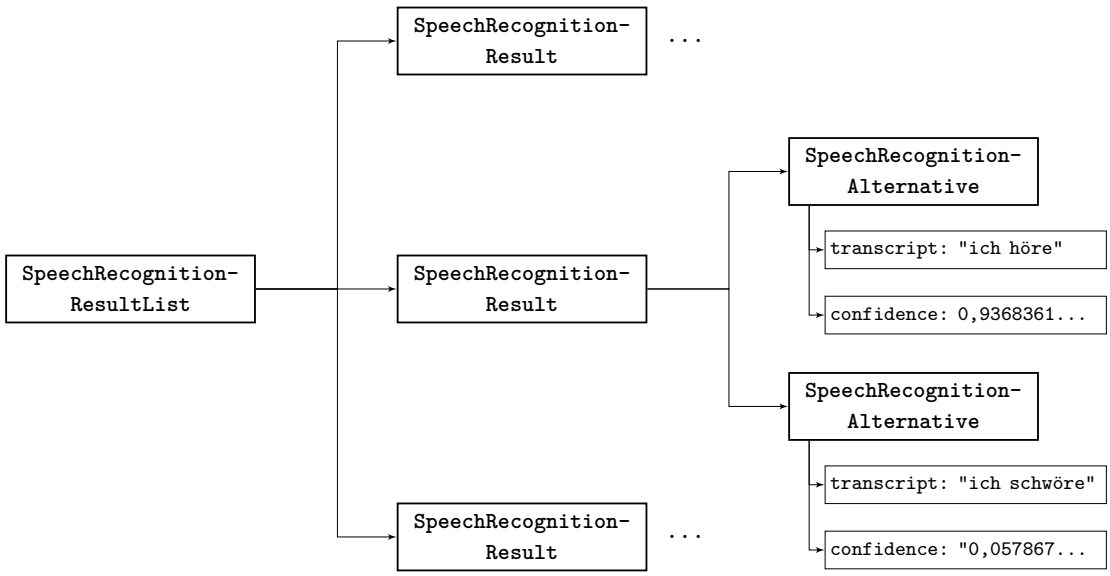


Abbildung 2.5: Resultat der Spracherkennung.

Umsetzung der API in Google Chrome

Die bisher am weitesten fortgeschrittenste Implementierung der Web Speech API ist jene des Google Chrome Browsers. Ab Version 25 war diese erstmals inkludiert. In der aktuellen Version 37 funktionieren die Sprachsynthese und Spracherkennung bereits, andere Teile der Spezifikation sind aber noch nicht umgesetzt. So können beispielsweise noch keine Grammatiken definiert werden. Ebenfalls kann noch keinen Einfluss darauf genommen werden, welche Speech Services verwendet werden (Attribut `serviceURI` aus der Spezifikation wird ignoriert).

Derzeit greift Chrome immer auf den Google Spracherkennungsservice, welcher beispielsweise auch von der Google Voice Search verwendet wird, zu. Die Spracherkennung funktioniert somit ausschließlich mit einer aufrechten Internetverbindung. Unterstützt werden vom Service insgesamt 32 Sprachen. Die Audio-Daten werden FLAC (Free Lossless Audio Codec) kodiert mittels POST-Anfrage über eine sichere HTTPS-Verbindung an den Service übertragen. Dieser führt die Spracherkennung durch und liefert das Ergebnis als JSON-Objekt (JavaScript Object Notation) zurück.

Zur genauen Funktion des Google Spracherkennungsservices sind keine öffentlichen Informationen vorhanden. Es ist bekannt, dass Google intensiv im Bereich der Spracherkennung forscht. Diverse Publikationen von Google Research lassen vermuten, dass das Unternehmen derzeit mit akustischen Modellen des Typs CD-DNN-HMM arbeitet und für das Sprachmodell sehr große N-Gramm-Modelle verwendet [6, 28]. Für das Trainieren der Modelle nutzt Google enorme Mengen an Daten, welche zum Teil über die eigenen Dienste (z. B. Google Voice Search, Google Mail) gesammelt werden [7, 28]. Zusätzlich beschäftigt Google aber auch eigenes Personal, das rund um die Welt reist,

um Sprachbeispiele in verschiedensten Sprachsituationen aufzunehmen [59].

2.3.4 Weitere Möglichkeiten

Neben PocketSphinx.js und der Web Speech API gibt es noch weitere Möglichkeiten zur Spracherkennung im Webbrowser:

WAMI Toolkit: Das WAMI Toolkit [53] stellt ein Framework für die Entwicklung multimodaler Benutzerschnittstellen für Webanwendungen dar. Multimodal bedeutet in diesem Zusammenhang, dass verschiedene Eingabemethoden – darunter auch Sprache – unterstützt werden. Das Toolkit besteht aus einer Client- und Server-Komponente. Audiodaten werden am Client mittels einer Flash-Komponente aufgenommen und an den Server übertragen, wo dann die Spracherkennung erfolgt. Im WAMI Toolkit selbst ist kein Spracherkennungssystem enthalten. Für diese Aufgabe kann z. B. auf CMU Sphinx 4 zurückgegriffen werden.

Online Speech Recognition API: Die Online Speech Recognition API [54] ist eine Programmierschnittstelle, die Spracherkennung und Sprachsynthese für Webanwendungen ermöglicht. Wie das WAMI Toolkit, besteht es aus einer Client- und Server-Komponente. Die Audioaufnahme erfolgt mittels Flash und zur Spracherkennung am Server wird CMU Sphinx 4 eingesetzt.

Kapitel 3

Analyse

Das folgende Kapitel beschäftigt sich mit der Analyse der Anforderungen und der Auswahl des technischen Lösungsansatzes.

In Abschnitt 3.1 werden im Zuge einer Anforderungsanalyse die Anwendungsfälle identifiziert und die funktionalen und nicht-funktionalen Anforderungen, welche an den Prototypen gestellt werden, ermittelt. Anschließend wird in Abschnitt 3.2 auf Basis der zuvor definierten Anforderungen der geeignetste technische Lösungsansatz für die Umsetzung des Prototyps ausgewählt und die getroffene Wahl in einer Diskussion begründet.

3.1 Anforderungsanalyse

Die Anforderungsanalyse ist eine Teilaufgabe der Softwareentwicklung und sollte am Beginn jedes Entwicklungsprozesses durchgeführt werden. Das Ziel einer Anforderungsanalyse ist es, die Anforderungen, welche die verschiedenen Stakeholder – wie beispielsweise Benutzerinnen und Benutzer, Auftraggeberinnen und Auftraggeber, Sicherheitsbeauftragte oder auch der Gesetzgeber – an ein Softwaresystem stellen, zu ermitteln und entsprechend zu strukturieren [19]. Mit der Anforderungsanalyse werden die notwendigen Grundlagen für den Softwareentwurf geschaffen.

Eine Strukturierung von Anforderungen kann nach verschiedensten Kriterien erfolgen. In der Literatur erfolgt häufig eine Unterteilung in funktionale und nicht-funktionale Anforderungen, wobei funktionale Anforderungen die eigentliche Funktionalität des Systems abbilden und nicht-funktionale Anforderungen meist Qualitätsmerkmale – wie zum Beispiel Performance, Usability oder Sicherheit – oder äußere Rahmenbedingungen (z. B. gesetzliche Bestimmungen) beschreiben [19, 20]. Auch in dieser Arbeit wird eine Unterteilung in funktionale und nicht-funktionale Anforderungen vorgenommen.

3.1.1 Anwendungsfall

Anwendungsfälle (engl.: Use Cases) stellen ein Hilfsmittel zur Anforderungsanalyse dar. Durch Anwendungsfälle wird beschrieben, wie ein System mit den verschiedenen Akteuren interagiert. Ein Anwendungsfall dokumentiert dabei das extern wahrnehmbare Verhalten des Systems aus Sicht der jeweiligen Akteure, ohne auf Details der Realisierung oder der internen Struktur des Systems einzugehen. Ein Akteur befindet sich stets außerhalb des betrachteten Systems und kann eine reale Person – welche eine bestimmte Rolle einnimmt (z. B. User oder Administrator) –, ein anderes externes System oder auch eine Organisationseinheit sein [19].

Die Unified Modeling Language (UML) ist eine standardisierte Modellierungssprache, die in der Softwareentwicklung sehr weite Verbreitung findet. Sie bietet für die Beschreibung von Anwendungsfällen Use Case-Diagramme an. Ein Use Case-Diagramm gibt auf einem hohem Abstraktionsniveau einen Überblick über die identifizierten Anwendungsfälle und zeigt die Beziehungen zwischen den einzelnen Anwendungsfällen und ihren Akteuren [20]. Abbildung 3.1 stellt das Use Case-Diagramm für das Sprachsteuerungsmodul dar.

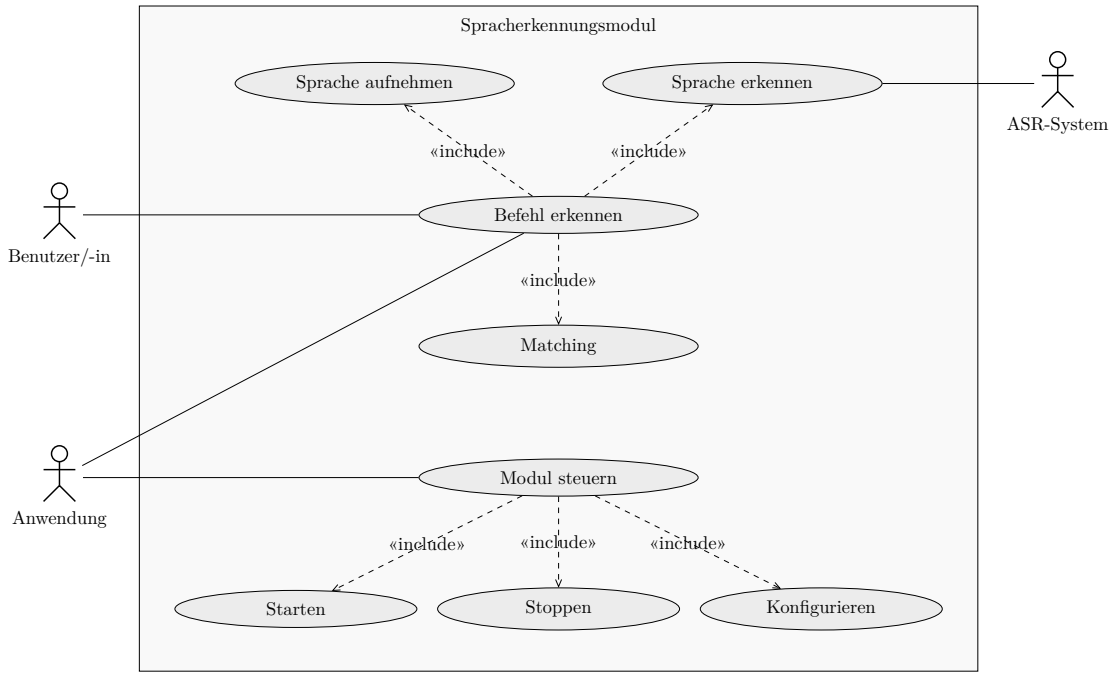


Abbildung 3.1: Use Case-Diagramm für das Sprachsteuerungsmodul.

Es lassen sich drei Akteure identifizieren, die mit dem Sprachsteuerungsmodul interagieren. Die Anwendung, welche das Sprachsteuerungsmodul integriert – also im konkreten Fall in erster Linie die LUI-Portierung –, die Benutzerin oder der Benutzer, welche oder welcher einen Befehl spricht und ein ASR-System, das die Umwandlung des gesprochenen Befehls in eine entsprechende schriftliche Repräsentation vornimmt.

Die zwei Hauptanwendungsfälle für das Sprachsteuerungsmodul sind *Befehl erkennen* und *Modul steuern*. Um einen Befehl zu erkennen, muss die Sprache aufgenommen, in ihre schriftliche Repräsentation überführt und anschließend einem definierten Systemkommando zugeordnet werden, sodass die Anwendung darauf reagieren kann (dieser Schritt wird im folgenden als *Matching* bezeichnet). Das Erkennen eines Befehls besteht somit insgesamt aus drei Unteranwendungsfällen, was im Use Case-Diagramm durch den Stereotypen `«include»` abgebildet wird. Ebenso besteht der zweite Hauptanwendungsfall aus drei Unteranwendungsfällen. Die Steuerung des Moduls umfasst das Starten, Stoppen und das Setzen von Konfigurationsparametern.

Anhand der Anwendungsfälle, der generellen Vorgaben (vgl. Abschnitt 1.1) und weiterer Überlegungen, die im Zuge eines Brainstormings angestellt wurden, konnten für das Sprachsteuerungsmodul die folgenden funktionalen und nicht-funktionalen Anforderungen identifiziert werden.

3.1.2 Funktionale Anforderungen

Die wesentlichen funktionalen Anforderungen (FA) an das Sprachsteuerungsmodul sind:

FA1: Die von der Benutzerin oder dem Benutzer gesprochenen Befehle müssen erkannt und in entsprechende Anwendungskommandos transformiert werden. Die Spracherkennung muss zumindest die Sprachen Deutsch und Englisch unterstützen.

FA2: Befehle mit einem Erkennungsmuster und einer Methode zur Ereignisbehandlung (engl.: Event Handler), welche aufgerufen wird, wenn der entsprechende Befehl erkannt wurde, sollen in Form eines geeigneten Dateiformats definiert werden können.

FA3: Das Sprachsteuerungsmodul muss Methoden für die Steuerung bereitstellen, sodass die Spracherkennung durch die benutzende Anwendung gestartet und gestoppt werden kann.

FA4: Ebenso müssen Methoden zur Manipulation aller wesentlicher Parameter bereitgestellt werden, damit das Sprachsteuerungsmodul nach den Erfordernissen der benutzenden Anwendung konfiguriert werden kann.

FA5: Um in der benutzenden Anwendung relevante Ereignisse – wie beispielsweise Fehler – entsprechend behandeln zu können, muss das Sprachsteuerungsmodul eine Möglichkeit bieten, weitere Event Handler zu registrieren. Diese Event Handler müssen beim Auftreten des ihnen zugeordneten Ereignisses ausgeführt werden.

3.1.3 Nichtfunktionale Anforderungen

Die wesentlichen nicht-funktionalen Anforderungen (NFA) an das Sprachsteuerungsmodul sind:

NFA1: Die Umsetzung soll ausschließlich auf den „Bordmitteln“ des Browsers beruhen. Das bedeutet insbesondere, dass keine Installation zusätzlicher Browser-Plug-ins erforderlich sein darf. Ebenfalls sollen keine Abhängigkeiten zu Serverkomponenten, die eigens zu implementieren sind, bestehen. Das Sprachsteuerungsmodul muss zumindest mit einem modernen Webbrowser verwendet werden können.

NFA2: Von der Verwendung proprietärer Technologien ist abzusehen. Es sollen ausschließlich offene Standardtechnologien bzw. aufkommende Technologien, welche sich höchstwahrscheinlich als Standardtechnologien etablieren werden, für die Umsetzung verwendet werden. Für Software dürfen keine Kosten entstehen.

NFA3: Das Sprachsteuerungsmodul soll eine einfache Einbindung in bestehende Applikationen erlauben. Es darf die benutzende Applikation hinsichtlich deren Performance nicht wesentlich beeinflussen (z.B. die Ladezeit negativ beeinträchtigen).

NFA4: Die Spracherkennung soll eine möglichst hohe Erkennungsgenauigkeit aufweisen. Antwortzeiten sollen im Durchschnitt unter zwei Sekunden liegen.

NFA5: Sofern erforderlich, hat die Übertragung von Daten zwischen Server und Client aus Gründen der Sicherheit verschlüsselt mittels TLS (Transport Layer Security) zu erfolgen.

NFA6: Damit das Sprachsteuerungsmodul einfach an zukünftige Erfordernisse angepasst werden kann, soll es leicht erweiterbar sein. Insbesondere soll der Quellcode gut dokumentiert werden, sodass andere Entwickler oder Entwicklerinnen sich einfach darin zurecht finden können.

NFA7: Das Sprachsteuerungsmodul soll möglichst robust gestaltet werden. Das bedeutet vor allem, dass es trotz auftretender Fehler in einem konsistenten Zustand bleiben soll. Ereignisse – insbesondere Fehler – müssen aufgezeichnet werden, sodass deren Ursache einfach und schnell bestimmt werden kann.

NFA8: Die Korrektheit der Funktionalität soll durch eine Reihe von automatisierten Tests ständig überprüft werden können. Zusätzlich soll zur Überprüfung der Funktionalität eine einfache Demoanwendung umgesetzt werden.

3.2 Auswahl des technischen Lösungsansatzes

Eine wesentliche Fragestellung bei der Umsetzung des Sprachsteuerungsmoduls ist die Realisierung der Spracherkennung. In Abschnitt 2.3 wurden bereits mögliche Lösungsansätze vorgestellt. Einer dieser Ansätze soll im Folgenden für die Implementierung des Prototypen ausgewählt werden.

3.2.1 Eingrenzung

Sowohl das WAMI Toolkit als auch die Online Speech Recognition API nutzen eine Flash-Komponente für die Sprachaufnahme (vgl. Abschnitt 2.3.4). Da Flash eine proprietäre Technologie ist und ein Plug-in voraussetzt, können mit dem WAMI Toolkit und der Online Speech Recognition API die Anforderungen NFA1 und NFA2 nicht umgesetzt werden. Diese Lösungsansätze werden daher nicht weiter berücksichtigt.

Ebenfalls werden durch NFA1 Lösungsansätze, welche auf einer serverbasierten Spracherkennung beruhen (z. B. Sprachaufnahme mittels JavaScript und Anbindung eines Spracherkennungssystems, wie Sphinx 4, über einen Webservice), ausgeschlossen. Des Weiteren soll grundsätzlich von der Implementierung eines eigenen JavaScript-Spracherkenners abgesehen werden, da der dafür erforderliche Aufwand und das zu erwartende Ergebnis, in keinem sinnvollen Verhältnis zueinander stehen.

Für die Realisierung der Spracherkennung kommen somit nur PocketSphinx.js und die Web Speech API in Frage. Im Folgenden werden daher nur diese beiden Ansätze weiter berücksichtigt.

3.2.2 Diskussion und Auswahl

Ein wesentlicher Vorteil der Web Speech API ist, dass es sich bei diesem Ansatz um eine Out-of-the-box-Lösung mit sehr breiter Sprachunterstützung handelt. Das aufwendige Trainieren von Sprach- und Akustikmodellen ist nicht erforderlich, wodurch ein wesentlich geringerer Implementierungsaufwand zu erwarten ist. Bei einer Umsetzung mittels PocketSphinx.js wird hingegen viel Zeit in die Suche und Evaluierung bzw. in das Trainieren von Sprach- und Akustikmodellen investiert werden müssen. Außerdem ist dieses Vorgehen jedes Mal zu wiederholen, wenn das Sprachsteuerungsmodul um eine zusätzliche Sprache erweitert werden soll. Bei einer Lösung basierend auf der Web Speech API ist – insofern die gewünschte Sprache von der API unterstützt wird – eine Erweiterung hingegen mit sehr geringem Aufwand möglich. Problematisch wird es erst, wenn die gewünschte Sprache nicht unterstützt wird. In diesem Fall kann nur darauf gewartet werden, dass Google den Service entsprechend erweitert. Da die Sprachunterstützung mit 32 unterstützten Sprachen – darunter fast alle europäischen Sprachen –

sehr breit ist, soll dieser Umstand bei der Wahl des Lösungsansatzes aber nicht negativ ins Gewicht fallen.

Die Erkennungsrate des Google Spracherkennungsservice kann allgemein als sehr hoch bezeichnet werden [21]. Allerdings gilt es zu berücksichtigen, dass der Service domänenunabhängige Sprach- und Akustikmodelle verwendet. Versuche haben gezeigt, dass mit einem PocketSphinx-System, welches für eine domänenspezifische Verwendung trainiert wurde, bei der Verwendung zur Erkennung kontinuierlich gesprochener Sprache in der Zieldomäne, niedrigere Wortfehlerraten erzielt werden können [60]. Im vorliegenden Anwendungsfall wird es aber als Vorteil angesehen, wenn das Sprachsteuerungsmodul für verschiedene Anwendungen verwendet werden kann, ohne neue Modelle trainieren zu müssen. Daher wird eine allgemein hohe Erkennungsleistung einer hohen domänenspezifischen Erkennungsleistung vorgezogen.

Aus der Verwendung der Web Speech API resultieren derzeit noch zahlreiche Nachteile. Im Gegensatz zu PocketSphinx.js kann mittels der Web Speech API nicht ohne zusätzliche Adaptierungen eine Schlüsselworterkennung realisiert werden. Ebenfalls werden derzeit noch keine Grammatiken unterstützt. Die Web Speech API ist daher im Moment eher für den Einsatz in einer Diktieranwendung, als für die Verwendung in einer Sprachsteuerung geeignet. Außerdem ist die Spracherkennung mit der Web Speech API derzeit nur mit dem Google Chrome Browser möglich. Sie wird zudem ausschließlich serverseitig durchgeführt, was bedeutet, dass einerseits keine Offlinefähigkeit besteht und andererseits die Antwortzeiten tendenziell höher als bei einer clientseitigen Spracherkennung sein werden. Im Sinne des Datenschutzes gilt es ebenfalls zu bedenken, dass alles, was durch die benutzenden Personen gesprochen wird, an Google übermittelt wird.

Obwohl diese Nachteile bei einer Lösung mit PocketSphinx.js nicht oder – im Fall der Browserunterstützung – nicht im selben Ausmaß zum Tragen kommen, soll die Implementierung auf Basis der Web Speech API erfolgen. Die erwähnten Nachteile werden bei dieser Entscheidung bewusst in Kauf genommen, da sie mit fortschreitender Standardisierung und Entwicklung der Web Speech API höchstwahrscheinlich verschwinden werden. Insbesondere die in der API-Spezifikation vorgesehenen Grammatiken und der lokale Sprachservice (vgl. Abschnitt 2.3.3) sollten eine Verbesserung bringen. Im Fall der derzeit nicht unterstützten Grammatiken, soll durch entsprechende technische Adaptierungen eine Lösung gefunden werden.

Kapitel 4

Umsetzung des Prototyps

Der Inhalt dieses Kapitels beschreibt die prototypische Realisierung des Sprachsteuerungsmoduls entsprechend der in Kapitel 3 definierten Anforderungen und unter Verwendung des dort ausgewählten Lösungsansatzes.

Der Abschnitt 4.1 befasst sich zunächst mit dem Softwareentwurf. Anschließend wird in Abschnitt 4.2 die auf Basis des Softwareentwurfs durchgeführte Implementierung vorgestellt. Außerdem wird auf die Durchführung des Unit Tests sowie auf die Demonstrationsanwendung eingegangen.

4.1 Entwurf

Das Ziel des Softwareentwurfs ist es, festzulegen, wie die konkrete Umsetzung der in Abschnitt 3.1 definierten funktionalen und nicht-funktionalen Anforderungen erfolgen soll. Es gilt, alle notwendigen Details des Sprachsteuerungsmoduls – dazu zählen beispielsweise die Komponenten und Schnittstellen, das Verhalten, die internen Strukturen und das Datenmodell, die verwendeten Algorithmen aber auch Testanforderungen – zu definieren, sodass in weiterer Folge die Implementierung des Moduls erfolgen kann.

4.1.1 Grundlegende Überlegungen

Die bestehende Spracherkennung des LUI-Systems benutzt Grammatiken, die in der XML-Form der Speech Recognition Grammar Specification (SRGS) [62] erstellt wurden. Mittels dieser SRGS-XML-Grammatiken wird definiert, welche Wortkombinationen gültige Befehle darstellen. Wird während des Sprechens eine Wortkombinationen erkannt, wird ein Ereignis ausgelöst. Dieses Ereignis wird durch LUI abgefangen und der entsprechende Befehl ausgeführt. Durch das Fehlen von Grammatiken bietet die Web Speech API in ihrem aktuellen Entwicklungsstand keine vergleichbare Unterstützung für eine Befehlserkennung. Derzeit ist die API eher für den Einsatz in einer Diktieranwendung geeignet. Spricht eine Benutzerin oder ein Benutzer, liefert die API ständig Hypothesen

in Form von Zwischenergebnissen. Nachdem fertig gesprochen wurde, wird nach einer kurzen Verzögerung ein finales Ergebnis geliefert. Tabelle 4.1 zeigt ein Beispiel für die Hypothesen, die beim Sprechen der Wortfolge „bitte Datei speichern und die Anwendung schließen“ empfangen werden.

Tabelle 4.1: Beispiel für empfangene Hypothesen.

Nr.	Hypothese	Final	Zeit
1	bitte	nein	t
2	bitte das	nein	$t + 360\,\text{ms}$
3	bitte dabei	nein	$t + 496\,\text{ms}$
4	bitte Datei	nein	$t + 546\,\text{ms}$
5	bitte Details	nein	$t + 732\,\text{ms}$
6	bitte Datei speichern	nein	$t + 1956\,\text{ms}$
7	bitte Datei speichern und	nein	$t + 2038\,\text{ms}$
8	bitte Datei speichern unter	nein	$t + 2152\,\text{ms}$
9	bitte Datei speichern und die	nein	$t + 2801\,\text{ms}$
10	bitte Datei speichern und die Anwendung	nein	$t + 3482\,\text{ms}$
11	bitte Datei speichern und die Anwendung schließen	nein	$t + 3703\,\text{ms}$
12	bitte Datei speichern und die Anwendung schließen	ja	$t + 4036\,\text{ms}$

Ziel bei der Entwicklung des neuen Sprachsteuerungsmoduls ist es, dieselbe Funktionalität wie bei den bestehenden Komponenten zu gewährleisten. Das bedeutet:

- Es muss eine mit den Grammatiken vergleichbare Möglichkeit geschaffen werden, Wortkombinationen, die einen Befehl repräsentieren, zu definieren.
- Der kontinuierliche Fluss eingehender temporärer und finaler Hypothesen muss nach den mit einem Befehl assoziierten Wortkombinationen durchsucht werden. Es muss also ein Musterabgleich (engl. pattern matching) implementiert werden, wobei die Wortkombinationen als Erkennungsmuster verwendet werden.

Werden beispielsweise die Wortkombinationen „Datei speichern“ und „Anwendung schließen“ als Befehle definiert, muss in den Hypothesen aus Tabelle 4.1 ersterer Befehl bei der sechsten und letzterer Befehl bei der elften Hypothese erkannt werden, also beim jeweils erstmaligen Auftreten der definierten Wortkombinationen. Wichtig ist es außerdem sicherzustellen, dass der bei der sechsten Hypothese erkannte Speichern-Befehl bei den folgenden Hypothesen nicht nochmals ausgelöst wird.

Auf die Erkennungsleistung der eigentlichen Spracherkennung kann kein Einfluss genommen werden. Es können weder Grammatiken verwendet, noch die Sprach- oder Akustikmodelle geändert werden. Die erhaltenen Hypothesen müssen somit als gegeben betrachtet werden. Um die Erkennungsleistung des Gesamtsystems zu optimieren, gilt es daher, die erhaltenen Hypothesen mit einer möglichst hohen Genauigkeit einem Befehl zuzuordnen. Dabei muss eine gewisse Fehlertoleranz erlaubt sein, da die Hypothesen

mehr oder weniger stark vom gesprochenen Befehl abweichen können. Es soll zu diesem Zweck ein konfigurierbares fehlertolerantes Matching-Verfahren, welches die phonetische und quantitative Ähnlichkeit zwischen Hypothesen und Erkennungsmustern berücksichtigt, implementiert werden.

4.1.2 Komponenten und Schnittstellen

Das Sprachsteuerungsmodul wird in Form eines JavaScript-Moduls umgesetzt. Das Modul – als Name wird *WebASR* verwendet – kann in die es nutzende Anwendung (z. B. LUI) integriert werden, wobei die Schnittstelle zwischen Anwendung und Modul eine öffentliche API bildet, welche durch das Modul bereitgestellt wird. Die zu erkennenden Befehle werden in einem speziellen Format (vgl. Abschnitt 4.1.4) in einer Datei definiert und asynchron mittels `XMLHttpRequest`-Objekt geladen. Der Google Spracherkennungsservice wird über die Web Speech API angesteuert. Um die Sprachaufnahme muss sich im Zuge der Umsetzung nicht gekümmert werden, da diese innerhalb der Web Speech API durchgeführt wird. Damit die Befehlsdatei nicht bei jedem Refresh erneut heruntergeladen und geparst werden muss, kann das interne Datenmodell mittels der Web Storage API im Local Storage abgelegt und von dort wiederhergestellt werden. Dieser Mechanismus soll dafür sorgen, dass das Spracherkennungsmodul die Performance der Anwendung möglichst wenig beeinflusst. Abbildung 4.1 stellt die beschriebenen Zusammenhänge schematisch dar. Die in grau gehaltenen Komponenten gilt es in den folgenden Abschnitten zu spezifizieren.

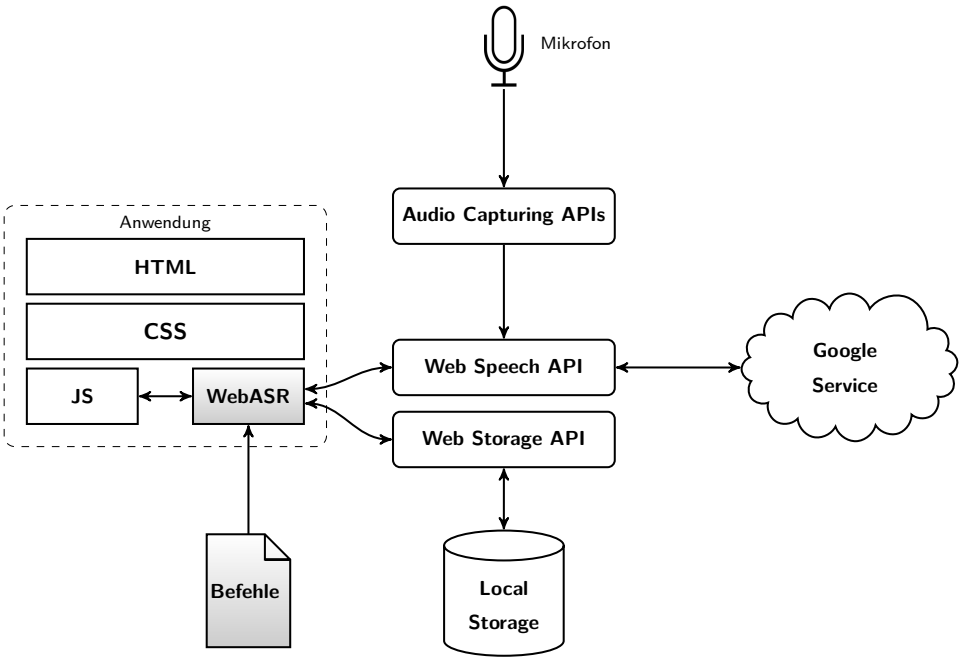


Abbildung 4.1: Schematische Darstellung beteiligter Komponenten und Schnittstellen.

4.1.3 Verhalten des Sprachsteuerungsmoduls

Im Folgenden werden die dynamischen Aspekte des Sprachsteuerungsmoduls beschrieben. Dazu werden zunächst die verschiedenen Systemzustände definiert.

Systemzustände

Das Sprachsteuerungsmodul kennt drei Systemzustände. Wird das Modul durch die Anwendung geladen, ist es im Zustand *Uninitialisiert*. Nachdem das Modul initialsiert wurde – also dann wenn die Konfiguration durchgeführt, die Callback Handler registriert und die Befehle geladen sind –, befindet es sich im Zustand *Initialisiert*. Durch das Starten wird das Modul aus dem Status Initialisiert in den Status *Zuhören* versetzt. Die Spracherkennung ist nun aktiviert und die Benutzerin oder der Benutzer können Anweisungen sprechen. Liefert die Spracherkennung ein Ergebnis, wird das Matching durchgeführt. Durch das Stoppen des Moduls wird die Spracherkennung beendet, sodass das Modul wieder in den Zustand Initialisiert übergeht. Die beschriebenen Zustände und deren Übergänge sind in Abbildung 4.2 in Form eines UML-Zustandsdiagramms dargestellt.

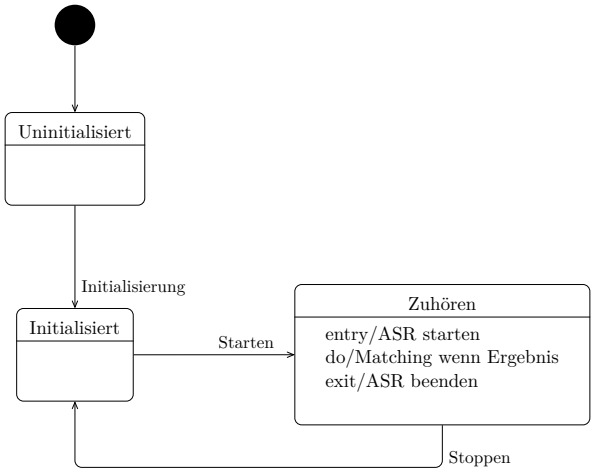


Abbildung 4.2: Zustandsdiagramm zum Sprachsteuerungsmodul.

Zeitliches Verhalten

Um das zeitliche Verhalten eines Systems zu modellieren, stellt die UML vier verschiedene Typen von Interaktionsdiagrammen bereit. Das Sequenzdiagramm ist eines davon und dient dazu, die zeitliche Abfolge des Nachrichtenaustausches zwischen den beteiligten Kommunikationspartnern (z. B. Objekte, Komponenten, Systeme etc.) zu beschreiben. Im Folgenden wird das Verhalten des Sprachsteuerungsmoduls unter Verwendung solcher Sequenzdiagramme beschrieben.

Abbildung 4.3 stellt den typischen Nachrichtenaustausch zwischen Anwendung, Sprachsteuerungsmodul und der Spracherkennungsschnittstelle der Web Speech API (`SpeechRecognition`-Objekt), von der Initialisierung bis zum Start des Moduls, anhand eines Sequenzdiagramms dar. Die Initialisierungsphase beginnt, nachdem die Anwendung durch die Benutzerin oder den Benutzer aufgerufen wurde. Im ersten Schritt werden Konfigurationsparameter – hier beispielhaft die Sprache – gesetzt und die Event Handler – in diesem Fall für die Events `onResult` und `onNoMatch` – registriert. Anschließend werden die Befehle geladen und durch das Parsen in die interne Datenstruktur überführt. Durch den Start des Moduls seitens der Anwendung, wird innerhalb des Moduls die Spracherkennungsschnittstelle initialisiert und diese ebenfalls gestartet. Damit hört das Modul nun zu und wartet darauf, dass Befehle gesprochen werden.

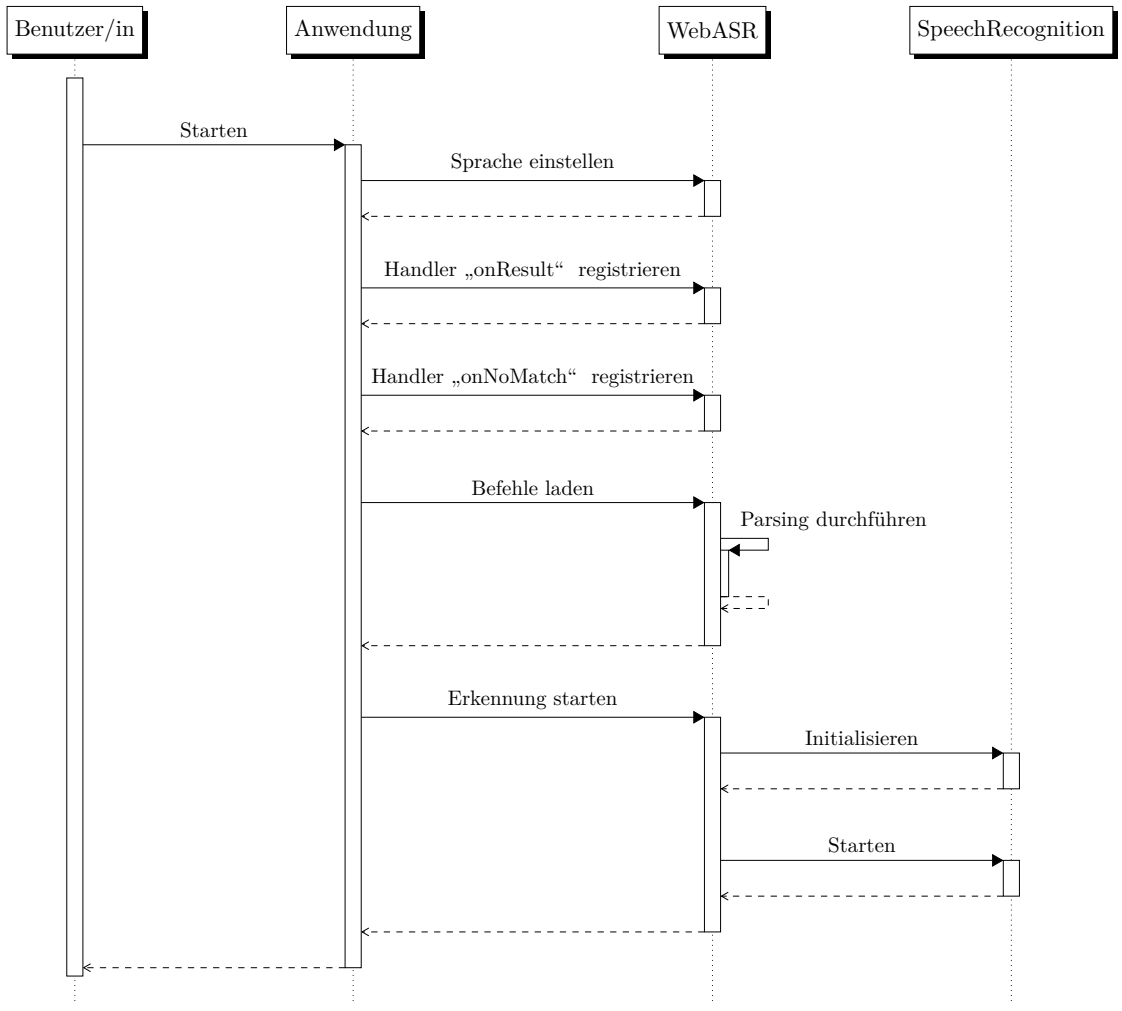


Abbildung 4.3: Sequenzdiagramm zur Initialisierung.

Abbildung 4.4 stellt den Nachrichtenaustausch bei der Befehlserkennung dar. Die Verarbeitung wird durch das Sprechen eines Wortes oder einer Wortfolge durch die Benutzerin bzw. den Benutzer angestoßen. Die Web Speech API zeichnet die Sprache auf und führt

die Spracherkennung durch. Ein Ergebnis wird zurückgegeben, indem durch die Schnittstelle das Event `onresult` gefeuert wird, wobei die Hypothesen in Form eines Objekts des Typs `SpeechRecognitionResultList` als Parameter mitgegeben werden (vgl. Abbildung 2.5). Bei den erhaltenen Hypothesen kann es sich sowohl um Zwischenergebnisse als auch um finale Ergebnisse handeln.

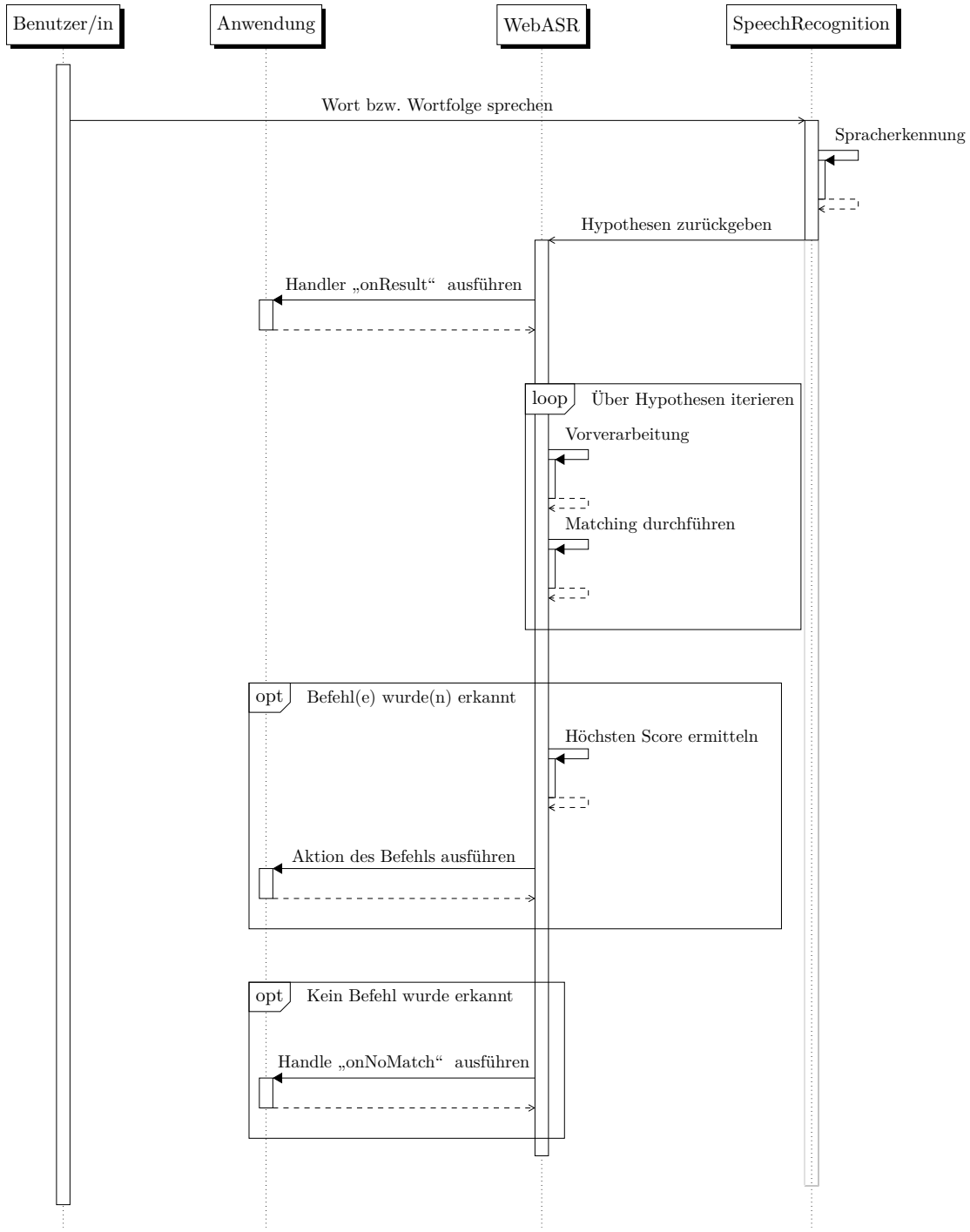


Abbildung 4.4: Sequenzdiagramm zur Befehlserkennung.

Das Event `onresult` wird durch das WebASR-Modul abgefangen. Da während der Initialisierung ein Event Handler für das `onResult`-Ereignis registriert wurde, wird dieser nun ausgelöst, sodass innerhalb der Anwendung eine entsprechende Ereignisbehandlung erfolgen kann. Anschließend wird über alle erhaltenen Hypothesen iteriert und versucht, diese einem definierten Befehl zuzuordnen.

Vor dem eigentlichen Matching werden in einer Vorverarbeitung die Hypothesen und Erkennungsmuster mittels phonetischer Algorithmen in ihre phonetischen Repräsentationen überführt. Das Matching wird dann auf Basis dieser phonetischen Repräsentationen durchgeführt, wobei – neben der qualitativen Bewertung in Form der phonetischen Ähnlichkeit – zusätzlich die Ähnlichkeit auch quantitativ bewertet wird. Liegt die Ähnlichkeitsbewertung einer Hypothese und eines Erkennungsmusters unterhalb einer definierbaren Toleranzschwelle, werden Hypothese und Erkennungsmuster als übereinstimmend betrachtet. Gibt es mehrere Übereinstimmungen, wird jenes Erkennungsmuster bevorzugt, welches die höchste Bewertung (engl. score) aufweist. Der mit diesem Erkennungsmuster assoziierte Befehl gilt somit als erkannt. In weiterer Folge wird die dem Befehl zugeordnete Aktion aufgerufen, sodass die von der Benutzerin bzw. dem Benutzer gesprochene Anweisung umgesetzt werden kann. Kann hingegen keine der Hypothesen einem der vorhandenen Erkennungsmuster zugeordnet werden, gilt der Befehl als nicht erkannt. In diesem Fall wird das `onNoMatch`-Ereignis ausgelöst, sodass von der Anwendung darauf reagiert werden kann – beispielsweise indem ein entsprechender Hinweis zur Wiederholung des Befehls ausgegeben wird.

4.1.4 Dateiformat für die Befehlsdefinition

Für die Definition von Befehlen wird ein eigenes, auf XML (vgl. Abschnitt 2.2.4) basierendes Dateiformat verwendet. Der Aufbau dieses Dateiformats wird im Folgenden spezifiziert.

Grundlegender Aufbau eines Befehls

Befehle werden mittels des Tags `<command>` definiert. Ein Befehl besteht dabei aus drei grundlegenden Bestandteilen:

1. Ein *Identifikator* (ID), der dazu dient, den Befehl innerhalb des Systems eindeutig zu identifizieren.

2. Zumindest ein *Erkennungsmuster*, das den Aufbau – also alle gültigen Wortkombinationen – des gesprochenen Befehls für eine bestimmte Sprache beschreibt.

3. Eine *Aktion*, welche ausgeführt werden soll, wenn eines der zu diesem Befehl definierten Erkennungsmuster in einer Hypothese gefunden wurde.

Das Listing 4.1 stellt den Aufbau einer Befehlsdatei anhand eines Beispiels dar.

```
<?xml version="1.0" encoding="UTF-8"?>
<commands>
  <!-- 1. Befehl -->
  <command>
    <id>cmd1</id>
    <pattern>
        <de-DE matchinterim="false"> ... </de-DE>
        <en-US matchinterim="true"> ... </en-US>
    </pattern>
    <action>
      console.log(param.id + " erkannt!");
      cmd1HandlerFunction(param);
    </action>
  </command>

  <!-- 2. Befehl -->
  <command>
    ...
  </command>
</commands>
```

Listing 4.1: Einfaches Beispiel für eine Befehlsdatei.

Zur Festlegung der ID wird das Tag `<id>` verwendet. Aktionen werden in Form von JavaScript- Programmcode, welcher innerhalb des Elements `<action>...</action>` eingefügt wird, definiert. Innerhalb des Programmcodes kann dabei über das Objekt `param`, das eine Instanz der Klasse `RecognitionResult` (vgl. Abschnitt 4.1.5) ist, auf die Ergebnisparameter zugegriffen werden.

Die Beschreibung eines Erkennungsmusters erfolgt innerhalb des Elements `<pattern>...</pattern>`. Ein Erkennungsmuster wird dabei mittels eines Sprach-Tags, welches dem Code des jeweiligen Sprach- und Gebietsschemas nach dem Request for Comments (RFC) 5646 [61] entsprechen muss, definiert (z. B. `<de-DE>` für Sprache Deutsch in der Region Deutschland). Soll der Befehl für mehrere Sprachen erkannt werden, müssen entsprechende Elemente für jede dieser Sprachen angelegt werden. Die Verwendung der Kodierung nach RFC 5646 liegt nahe, da diese auch für die Spracheinstellung der Web Speech API verwendet wird. Durch die Verwendung derselben Kodierung ist kein zusätzliches Mapping erforderlich.

Eine Besonderheit ist, dass das Sprach-Tag das Attribut `matchinterim` mit den Ausprägungen `true` und `false` enthalten kann. Ist der Wert des Attributs `false`, werden für den Mustervergleich nur finale Hypothesen herangezogen. Hat das Attribut hingegen den Wert `true` oder ist es nicht vorhanden, werden alle Hypothesen – also auch Zwischen-

ergebnisse – für den Mustervergleich verwendet. Die Absicht dahinter ist, zu vermeiden, dass kurze Erkennungsmuster den Zwischenergebnissen längerer Erkennungsmuster zugeordnet werden. Sind beispielsweise Erkennungsmuster für „Ja“ und „Jahrmarkt“ definiert, ist es möglich, dass beim Sprechen vom „Jahrmarkt“ durch die Web Speech API das Zwischenergebniss „Ja“ geliefert wird. Um zu Vermeiden, dass das Zwischenergebnis dem Erkennungsmuster „Ja“ zugeordnet und somit die falsche Aktion ausgeführt wird, kann für das Erkennungsmuster „Ja“ das Attribut `matchinterim` auf `false` gesetzt werden.

Elemente zur Beschreibung der Erkennungsmuster

Damit die Wortkombinationen, welche durch die in LUI verwendeten SRGS-XML-Grammatiken beschrieben werden, im neuen Format nachgebildet werden können, müssen durch dieses dieselben Grundkonstrukte wie in SRGS-XML unterstützt werden. Konkret bedeutet das, dass neben normalen Zeichenketten folgenden Konstrukte zur Verfügung gestellt werden müssen:

- Optionale Befehlsteile
- Wildcards
- Optionen
- Wiederholungen
- Variablen

Das grundlegende Element zur Definition eines Erkennungsmusters stellt das Element `<token>...</token>` dar. Ein solches Element kann ein oder mehrere Wörter enthalten. Werden mehrere Wörter in einem `<token>...</token>`-Element zusammengefasst, sind diese durch ein Leerzeichen zu trennen. Optionale Befehlsteile können definiert werden, indem das Element durch das Attribut `optional` mit der Ausprägung `true` ergänzt wird. Das Listing 4.2 zeigt dazu ein Beispiel. Das hier definierte Erkennungsmuster würde sowohl bei der Hypothese „Anwendung starten“ als auch bei der Hypothese „starten“ erkannt werden.

```
1 ...
2 <pattern>
3   <de-DE>
4     <token optional="true">Anwendung</token>
5     <token>starten</token>
6   </de-DE>
7 <pattern>
8 ...
```

Listing 4.2: Beispiel für optionale Befehlselemente.

Für die Erzeugung von Wildcards kann das Element `<token>...</token>` mit dem Attribut `garbage` und der Ausprägung `true` verwendet werden. An einer auf diese Weise gekennzeichneten Position des Erkennungsmusters, kann jede beliebige Wortfolge stehen. Ein Beispiel dafür zeigt das Listing 4.3. In diesem Fall würden neben „Datei speichern“ auch die Hypothesen „jetzt Datei speichern“ oder „Datei speichern jetzt“ zu einem positiven Ergebnis führen.

```
...
<pattern>
  <de-DE>
    <token garbage="true" />
    <token>Datei speichern</token>
    <token garbage="true" />
  </de-DE>
<pattern>
...
```

Listing 4.3: Beispiel für Wildcards.

Für die Definition von Optionen kann das Element `<one-of>...</one-of>` verwendet werden. Innerhalb dieses Elements sind die einzelnen Optionen durch `<option>...</option>`-Elemente festzulegen. Optional können die Elemente mit den Attributen `name` bzw. `value` ergänzt werden, wodurch die getroffene Auswahl im Ergebnis als Variable verfügbar wird. Das Listing 4.4 zeigt ein Beispiel zur Verwendung des `<one-of>...</one-of>`-Elements. Sowohl die Hypothese „Lautstärke erhöhen“ als auch die Hypothese „Lautstärke verringern“ würden hier zu einem positiven Ergebnis führen. Über `param.variables.volume` kann in der Aktion der entsprechende Wert des Attributes `value` ermittelt werden.

```
...
<pattern>
  <de-DE>
    <token>Lautstärke</token>
    <one-of name="volume">
      <option value="inc">erhöhen</option>
      <option value="red">verringern</option>
    </one-of>
  </de-DE>
<pattern>
<action>console.log(param.variables.volume);</action>
...
```

Listing 4.4: Beispiel für die Verwendung von Optionen.

Sich wiederholende Befehlsteile können mittels des Elements `<repeat>...</repeat>` definiert werden. Das obligatorische Attribut `occurence` dient dazu, durch Angabe eines Bereiches, dem System mitzuteilen, wie oft sich ein Befehlsteil wiederholen kann. Analog zum Element `<one-of>...</one-of>`, steht auch bei Wiederholungen das Attribut `name` zur Verfügung. Das Listing 4.5 zeigt ein Beispiel für die Anwendung des `<repeat>...</repeat>`-Elements. Die Hypothesen „viel lauter“ und „viel viel lauter“ würden dem Erkennungsmuster entsprechen. Der Wert von `param.variables.number` würde im ersten Fall 1 und im zweiten Fall 2 sein.

```
...
<pattern>
  <de-DE>
    <repeat name="number" occurence="1-2">viel</repeat>
    <token>lauter</token>
  </de-DE>
<pattern>
<action>console.log(param.variables.number);</action>
...
```

Listing 4.5: Beispiel für eine Wiederholung.

Das Ziel von Variablen ist es, Teile der gesprochenen Anweisung zu extrahieren, um diese in der das Modul verwendenden Anwendung für eine weitere Verarbeitung verfügbar zu machen. Variablen werden durch das `<variable>`-Tag definiert. Das Attribute `name` ist obligatorisch und legt den Namen der Variablen fest. Listing 4.6 zeigt dazu ein Beispiel. Für die Hypothese „erhöhe Lautstärke auf 22“ würde der Wert von `param.variables.value` gleich 22 sein.

```
...
<pattern>
  <de-DE>
    <token>erhöhe Lautstärke auf</token>
    <variable name="value" />
  </de-DE>
<pattern>
<action>console.log(param.variables.value);</action>
...
```

Listing 4.6: Beispiel für eine Variable.

Bei der Erstellung der Erkennungsmuster können grundsätzlich alle der beschriebenen Elemente beliebig miteinander kombiniert werden. Es gibt lediglich folgende Ausnahme: Vor und hinter einer Variable darf sich keine Wildcard und keine weitere Variable befinden. In diesen Fällen wäre es nämlich nicht möglich, zu unterscheiden, welcher Befehlsteil

als Variable oder Wildcard zu interpretieren wäre bzw. welcher Befehlsteil welcher Variable zuzuordnen werden müsste.

4.1.5 Interne Struktur

Bei der Entwicklung von größeren Anwendungen spielt die Strukturierung der Software eine bedeutende Rolle. In der Praxis werden häufig bewährte generische Lösungsmuster für diese Aufgabe eingesetzt. Für die Beschreibung der grundsätzlichen Struktur und Organisation einer Anwendung existieren verschiedene Architekturmuster (z. B. Schichtenarchitektur, serviceorientierte Architektur etc.) und zur Strukturierung des Quellcodes bietet sich die Verwendung von Entwurfsmustern (engl. Design Patterns) an.

Auch bei der Umsetzung des Prototyps soll auf eine saubere Strukturierung Wert gelegt werden. Für diesen Zweck werden verschiedene JavaScript-Entwurfsmuster verwendet, die im folgenden Abschnitt kurz vorgestellt werden.

JavaScript Entwurfsmuster

Ein gängiges Konzept aus der Praxis der Softwareentwicklung ist die Strukturierung einer Anwendung durch die Bildung von Modulen. Diese Modularisierung verfolgt das Ziel, die Komplexität durch die Unterteilung in kleine und überschaubare Komponenten beherrschbarer zu machen, um damit die Entwicklung und Wartung der Software zu erleichtern [20]. Module werden in der Regel so gestaltet, dass sie einfach austauschbar und wiederverwendbar sind.

Im Unterschied zu klassischen objektorientierten Sprachen, wie Java oder C++, kennt JavaScript manche Sprachkonzepte, die für die Modularisierung wichtig sind – wie beispielsweise Namensräume und Zugriffsattribute –, grundsätzlich nicht. Die hohe Flexibilität der Sprache ermöglicht es aber, diese Konzepte mit relativ geringem Programmieraufwand zu imitierten. Dafür haben sich diverse Entwurfsmuster etabliert.

Ein weit verbreitetes Entwurfsmuster für die Erstellung eines Moduls ist das *Revealing Module Pattern*. Dieses Entwurfsmuster wird auch für die Erstellung des Spracherkennungsmoduls sowie dessen Untermodule verwendet. Das Listing 4.7 stellt ein einfaches Beispiel für die Verwendung des Revealing Module Patterns dar. Mittels einer IIFE (Immediately Invoked Function Expression) – einer anonymen Funktion, welche bei ihrer Definition sofort ausgeführt wird – werden dabei Funktionen und Variablen in einem Objekt gekapselt, das als Namensraum dient (Zeile 1). Die öffentlichen Funktionen und Eigenschaften bilden die Schnittstelle nach außen und werden innerhalb des `return`-Blocks in Form eines sogenannten Object Literal zurückgegeben (Zeile 15-18). Alle anderen Variablen und Funktionen sind hingegen vor äußerem Zugriff geschützt und können lediglich innerhalb des Moduls verwendet werden (Zeile 11 und 25).

```
var MyModule = (function () {
    var privateVar = "private",
        publicVar = "public";

    function privateFunction() {
      console.log(privateVar);
    }

    function publicFunction() {
      console.log("public");
      privateFunction();
    }

    // Öffentliche Eigenschaften und Methoden zurückgeben
    return {
      "publicFunction": publicFunction,
      "publicVar": publicVar
    };
})();

// Erzeugt die Ausgaben "public" und "private" in der Konsole
MyModule.publicFunction();

// Fehler "is not a function"
MyModule.privateFunction();
```

Listing 4.7: Einfaches Beispiel für das Revealing Module Pattern.

Neben den erwähnten Namensräumen und Zugriffsattributen, kennt JavaScript auch das Konzept der Klassen nicht. Die Objektorientierung in JavaScript ist klassenlos und basiert auf Prototypen. Um Daten – so wie es aus anderen objektorientierten Sprachen bekannt ist – sauber zu kapseln, können aber klassenähnliche Strukturen[1] erzeugt werden, welche sich instanziieren lassen und auch Vererbung ermöglichen. Ein verbreitetes Entwurfsmuster zur Erzeugung von Klassen in JavaScript, ist das sogenannte *Constructor/Prototype Pattern*. Dieses Entwurfsmuster wird für die Erstellung aller Klassen des Sprachsteuerungsmoduls verwendet. Das Listing 4.8 zeigt dazu ein Beispiel. Eine Funktion definiert dabei den Konstruktor (Zeile 2), welcher die Eigenschaften des Objekts mittels des Schlüsselworts `this` festlegt. Methoden werden erstellt, indem sie außerhalb des Konstruktors dessen Prototyp hinzugefügt werden (Zeile 9). Um ein Objekt zu erzeugen, wird der Konstruktor mit dem Schlüsselwort `new` aufgerufen (Zeile 16).

[1] Der Einfachheit halber sollen diese Strukturen in weiterer Folge trotzdem als Klassen bezeichnet werden.

```
// Constructor
function Person(forename, surname, age) {
  // Eigenschaften
  this.forename = forename;
  this.surname = surname;
  this.age = age;
}

// Eine Methode
Person.prototype.toString = function () {
  return this.forename + " "  this.surname + " is "
    + this.age + " years old";
};

// Instanziierung
var p1 = new Person("Karl", "Toffel", 30);

// Erzeugt "Karl Toffel is 30 years old"
console.log(p1.toString());
```

Listing 4.8: Einfaches Beispiel für Klassenerzeugung mittels Constructor/Prototype Pattern.

Obwohl es noch zahlreiche andere JavaScript-Entwurfmuster gibt, die sich für die Strukturierung des Sprachsteuerungsmodul anbieten würden, soll mit dem Revealing Module Pattern und dem Constructor/Prototype Pattern das Auslangen gefunden werden. Auf die Verwendung und Erklärung weiterer JavaScript-Entwurfsmuster soll daher an dieser Stelle verzichtet werden. Als vertiefende Literatur zu dieser Thematik sei auf [22] verwiesen.

Module

Das Sprachsteuerungsmodul gliedert sich in die in Abbildung 4.5 dargestellten Module. Das Hauptmodul trägt den Namen `WebASR` und kapselt den gesamten JavaScript-Quellcode mittels einer anonymen Funktion als ein einziges Objekt. Dadurch wird einerseits der Quellcode vor unerwünschtem Zugriff geschützt, andererseits aber auch die Kompatibilität zum Quellcode anderer Entwickler gewährleistet, indem Namenskollisionen vermieden werden. Öffentliche Methoden existieren nur im Hauptmodul und können von außen über das `WebASR`-Objekt aufgerufen werden. Alle anderen Module stellen Untermodule dar, deren Deklaration im Namensraum des Hauptmoduls erfolgt. Die Methoden der Untermodule sind somit lediglich innerhalb des vom Hauptmodul gebildeten Namensraumes aufrufbar.

Neben der Implementierung der öffentlichen Schnittstelle für die Konfiguration und Steuerung der Sprachsteuerung, regelt das Hauptmodul die Initialisierung und Verknüp-

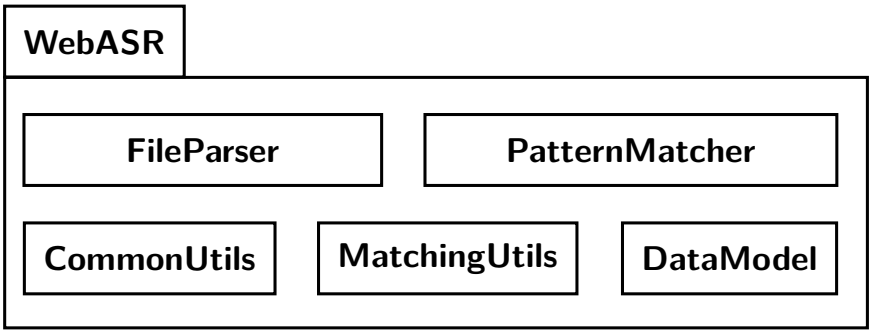


Abbildung 4.5: Überblick über die Module.

fung aller anderen Module und steuert den Datenfluss. Die Aufgaben des Ladens und Parsens der Befehlsdatei werden durch das Untermodul `FileParser` umgesetzt. Der Parser wird dabei fehlertolerant implementiert. Das bedeutet, dass bei einem fehlerhaften Element das Parsen nicht komplett abgebrochen, sondern stattdessen das fehlerhafte Element ignoriert und – nach der Ausgabe einer Warnung – mit dem nächsten Element fortgefahren wird. Das Untermodul `PatternMatcher` enthält die Logik, die für das Matching der Erkennungsmuster und Hypothesen benötigt wird. Die Implementierung der in Abschnitt 4.1.6 beschriebenen Algorithmen wird jedoch in das Untermodul `MatchingUtils` ausgelagert. Alle Klassen des Datenmodells (siehe nächster Abschnitt) sind im Untermodul `DataModel` gekapselt. Allgemeine Funktionen, welche in mehreren Modulen verwendet werden (z. B. die Funktionen für das Logging) werden im Untermodul `CommonUtils` zusammengefasst.

Datenmodell

Abbildung 4.6 stellt das interne Datenmodell mittels eines UML-Klassendiagramms dar. Die Klasse `RecognitionResult` repräsentiert das Ergebnis der Befehlserkennung. Eine Instanz dieser Klasse wird beim Aufruf der zu einem Befehl definierten Aktion als Parameter übergeben. Sie enthält die ID des erkannten Befehls (Eigenschaft `id`), die zum Treffer führende Ausprägung des Erkennungsmusters (Eigenschaft `pattern`), die entsprechende Hypothese (Eigenschaft `result`) und die Namen und Werte der Variablen und Parameter in Form des Objekts `variables` (vgl. dazu die Beispiele aus Abschnitt 4.1.4). Weitere Eigenschaften der Klasse sind die Ähnlichkeitsbewertung (Eigenschaft `score`) und der vom Google Service stammende Konfidenzwert (Eigenschaft `confidence`), der angibt, wie wahrscheinlich es ist, dass die Hypothese der ursprünglich gesprochenen Wortfolge entspricht.

Ein registrierter Callback Handler wird durch ein Objekt des Typs `CallbackHandler` repräsentiert. Die einzelnen `CallbackHandler`-Objekte werden in Form des Arrays `callbackHandlerArr`, welches eine nicht öffentliche Eigenschaft des Hauptmoduls darstellt, gespeichert. Die Klasse `CallbackHandler` enthält den Namen des Events, für

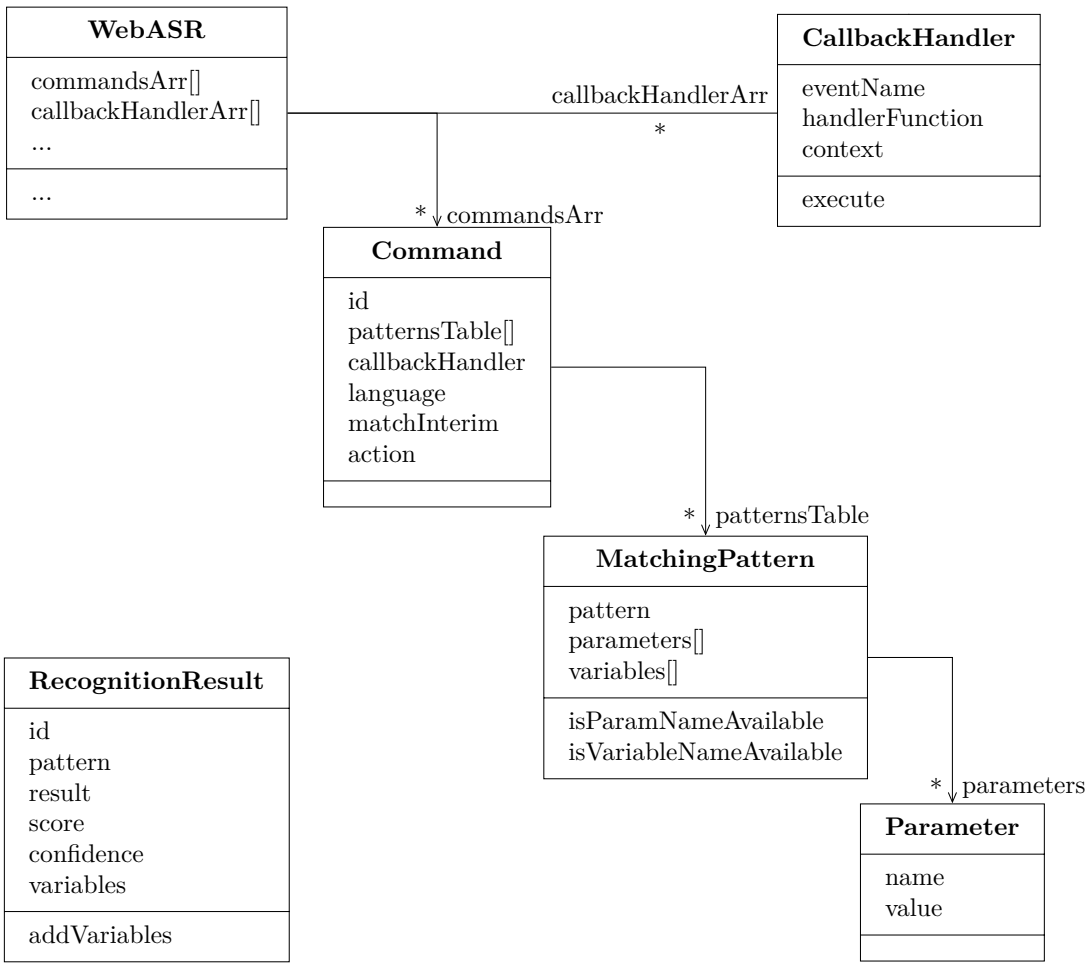


Abbildung 4.6: Datenmodell des Spracherkennungsmoduls.

welches der Callback Handler registriert wurde (Eigenschaft `eventName`), die aufzurufende Funktion (Eigenschaft `handlerFunction`) sowie einen optionalen Kontext für diese Funktion (Eigenschaft `context`). Durch die Methode `execute` wird der Callback Handler ausgeführt.

Das Modul `FileParser` überführt die in der Befehlsdatei definierten Befehle in das interne Datenmodell. In diesem werden die Befehle in Form des Arrays `commandsArr` – einer nicht öffentlichen Eigenschaft des Hauptmoduls – gespeichert. Für jeden korrekt definierten Befehl wird ein Objekt vom Typ `Command` erzeugt und diesem Array hinzugefügt. In jedem `Command`-Objekt sind die möglichen Ausprägungen des Erkennungsmusters in Form eines Arrays von `MatchingPattern`-Objekten gespeichert. Neben dem Erkennungsmuster (Eigenschaft `pattern`) enthält jedes `MatchingPattern`-Objekt auch die Namen der im Erkennungsmuster vorhandenen Variablen in der Reihenfolge deren Auftretens (Eigenschaft `variables`). Eine weitere Eigenschaft bilden die Parameter des Erkennungsmusters, die in Form eines Arrays aus `Parameter`-Objekten gespeichert werden. Ein `Parameter`-Objekt besteht dabei lediglich aus den Attributen `name` und

`value`. Abbildung 4.7 zeigt ein Beispiel für die Überführung eines Befehls in das interne Datenmodell.

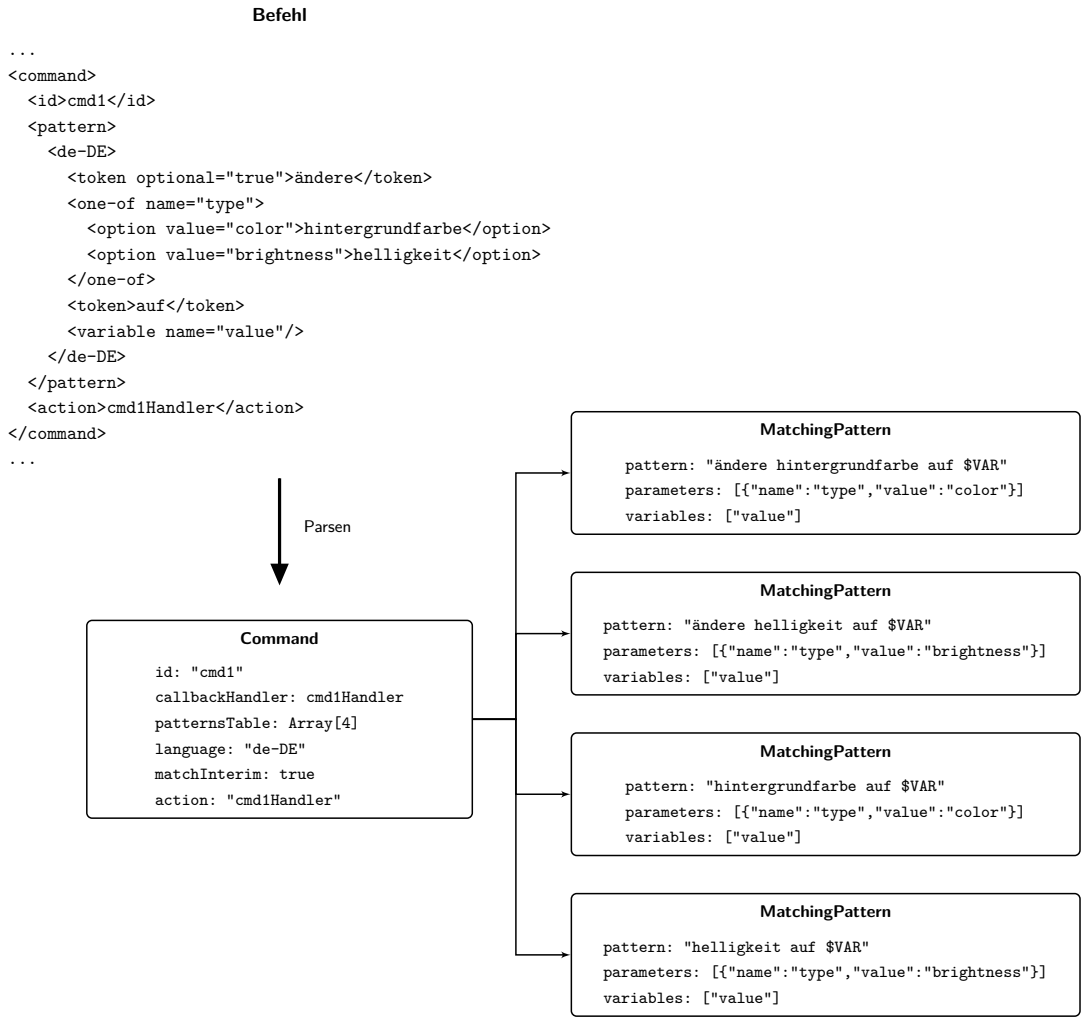


Abbildung 4.7: Beispiel für einen Befehl im internen Datenmodell.

Um die Daten im Local Storage zu speichern, werden mittels der `JSON.stringify`-Funktion die `Command`-Objekte in das textbasierte JSON-Format überführt. Da der Funktionszeiger der Aktions-Funktion (Eigenschaft `callbackHandler`) sich auf diese Weise aber nicht umwandeln lässt, muss beim Parsen zusätzlich auch der ursprüngliche Inhalt des Elements `<action>...</action>` in jedem `Command`-Objekt als Zeichenkette gespeichert werden (Eigenschaft `action`).

4.1.6 Algorithmen für das Matching

Um eine höhere Erkennungsleistung zu erreichen, wird beim Matching kein exakter Vergleich der erhaltenen Hypothesen mit den Ausprägungen der definierten Erkennungsmuster durchgeführt. Anstelle dessen wird unter der Verwendung verschiedener Algorithmen

ein unscharfer Vergleich, der hinsichtlich Abweichungen eine gewisse Toleranz erlaubt, umgesetzt. Die dafür verwendeten Algorithmen werden im Folgenden vorgestellt.

Phonetische Algorithmen

Phonetische Algorithmen basieren auf der Analyse der phonetischen Eigenschaften der menschlichen Sprache. Sie haben zum Ziel, gleich oder ähnlich ausgesprochene Wörter in eine identische Zeichenkette zu kodieren. In der Regel handelt es sich bei phonetischen Algorithmen um Reduktionsverfahren, bei welchen die für die Aussprache wichtigen Buchstaben eines Wortes nach definierten Regeln einer bestimmten Klasse zugeordnet werden. Die Buchstaben, die für die Aussprache des Wortes unbedeutend sind, werden hingegen eliminiert. Sind die so kodierten Zeichenfolgen gleich, können die ursprünglichen Zeichenfolgen als phonetisch ähnlich betrachtet werden.

Der *Metaphone*-Algorithmus ist ein solcher phonetischer Algorithmus. Das Verfahren wurde 1990 von Lawrence Philips in [23] veröffentlicht. Es beruht auf einer Reduktion der Ursprungszeichenkette auf ein Alphabet bestehend aus den Zeichen B, X, S, K, J, T, F, H, L, M, N, P, R, 0, W und Y, wobei das Zeichen 0 – als Annäherung für θ verstanden – einem TH entspricht. Ebenfalls können die Vokale A, E, I, O und U im Code vorkommen. Diese werden aber nur berücksichtigt, wenn sie am Anfang der ursprünglichen Zeichenkette stehen. Bei der Reduktion werden folgende kontextsensitiven Transformationsregeln angewendet:

- Entfernen aller doppelten Zeichen, außer bei C.
- Beginnt das Wort mit KN, GN, PN, AE oder WR, so wird das erste Zeichen entfernt.
- Beginnt das Wort mit einem X, so wird dieses durch ein S ersetzt.
- Beginnt das Wort mit mit einem WH, so wird dieses durch ein W ersetzt.
- Alle Zeichen werden sukzessive gemäß der in Tabelle 4.2 formulierten Ersetzungsregeln ersetzt.
- Alle Vokale werden entfernt, außer es handelt sich um das erste Zeichen des Wortes.

Die Anwendung dieser Regeln würde beispielsweise für die Zeichenketten „train“ und „terrain“ jeweils zum Code TRN führen.

Metaphone ist in seiner ursprünglichen Version für die englische Sprache ausgelegt. Bei der Kodierung deutschsprachiger Zeichenketten ergeben sich dadurch Probleme – beispielsweise entspricht das TH im Deutschen nicht einem [θ] sondern einem [t]. Ein phonetischer Algorithmus, welcher speziell auf die deutsche Sprache abgestimmt ist, ist die Kölner Phonetik.

Tabelle 4.2: Transformationsregeln des Metaphone-Algorithmus nach [23].

Zeichen	**Code**	**Kontext**
B	B	außer am Ende eines Wortes hinter einem M
C	X	wenn in CIA oder CH
	S	wenn in CI, CE oder CY
	K	ansonsten (inklusive in SCH)
D	J	wenn in DGE, DGY oder DGU
	T	ansonsten
F	F	
G	stumm	wenn in GH und am Ende oder bevor einem Konsonant
	stumm	wenn in GN oder GNED
	J	wenn vor I, E oder Y und nicht in GG
	K	ansonsten
H	stumm	wenn nach einem Vokal und kein Vokal folgt
	H	ansonsten
J	J	
K	stumm	wenn nach einem C
	K	ansonsten
L	L	
M	M	
N	N	
P	F	wenn vor einem H
	P	ansonsten
Q	K	
R	R	
S	X	wenn vor einem H oder in SIO oder SIA
	S	ansonsten
T	X	wenn in TIA oder TIO
	0	wenn vor einem H
	stumm	wenn in TCH
	T	ansonsten
V	F	
W	stumm	wenn nicht vor einem Vokal
	W	wenn vor einem Vokal
X	KS	
Y	stumm	wenn nicht vor einem Vokal
	Y	wenn vor einem Vokal
Z	S	

Die Kölner Phonetik ist ein von Hans Joachim Postel entwickeltes Verfahren, das in [24] vorgestellt wird. Wie beim Metaphone-Algorithmus wird auch hier anhand des Klangbildes des Wortes ein Code gebildet. Entsprechend der in Tabelle 4.3 angeführten Ersetzungsregeln wird Buchstabe für Buchstabe der Ursprungszeichenkette durch Ziffern zwischen 0 und 8 ersetzt. Dabei müssen gegebenenfalls benachbarte Buchstaben beachtet werden. Die Umlaute Ä, Ö und Ü werden durch den Code 0 ersetzt und das ß wird analog zum Buchstaben S behandelt.

Tabelle 4.3: Transformationsregeln der Kölner Phonetik nach [24].

Zeichen	**Kontext**	**Code**
A, E, I, J, O, U, Y		0
H		stumm
B		1
P	nicht vor H	
D, T	nicht vor C, S und Z	2
F, V, W		3
P	vor H	
G, K, Q		4
C	im Anlaut vor A, H, K, L, O, Q, R, U, X	
	vor A, H, K, O, Q, U, X außer nach S oder Z	
X	nicht nach C, K, Q	48
L		5
M, N		6
R		7
S, Z		8
C	nach S, Z	
	im Anlaut außer vor A, H, K, L, O, Q, R, U, X	
	nicht vor A, H, K, O, Q, U, X	
D, T	vor C, S, Z	
X	nach C, K, Q	

Nachdem im ersten Schritt alle Buchstaben umgewandelt wurden, werden alle doppelt auftretenden Codes sowie alle Nullen entfernt, wodurch sich der endgültige Code für die ursprüngliche Zeichenfolge ergibt. Im Folgenden wird das Vorgehen anhand des Beispielwortes „Wasserschloß“ demonstriert:

1. Aus der buchstabenweisen Kodierung ergibt sich die Ziffernfolge 308807885088.
2. Nach der Entfernung aller doppelten Codes bleibt 308078508 übrig.
3. Das Entfernen aller Nullen ergibt dann 387858.

Um die Hypothesen und die Erkennungsmuster in ihre phonetische Repräsentation zu überführen wird für die englische Sprache der Metaphone-Algorithmus und für die deutsche Sprache die Kölner Phonetik verwendet. Für alle anderen Sprachen wird im Prototyp kein phonetisches Matching umgesetzt. Neben der Verwendung der vorgestellten phonetischen Algorithmen, soll zusätzlich die Ähnlichkeit der Hypothesen und der Erkennungsmuster quantitativ bewertet werden. Für diese Aufgabe bietet sich die Levenshtein-Distanz an.

Levenshtein-Distanz

Die Levenshtein-Distanz ist ein vom russischen Mathematiker Vladimir Levenshtein vorgeschlagenes metrisches Verfahren zur Bestimmung der sogenannten Editierdistanz [25]. Unter Editierdistanz wird dabei die minimale Anzahl an Einfüge-, Ersetz- und Löschoperationen, welche benötigt werden, um eine bestimmte Zeichenkette in eine andere Zeichenkette zu überführen, verstanden. Das folgende Beispiel beschreibt die nötigen Schritte zur Bestimmung der Levenshtein-Distanz zwischen den Wörtern „Rebell“ und „Quelle“, welche genau 4 beträgt:

1. Rebell → Qebell (Ersetzung von „R“ durch „Q“)
2. Qebell → Qubell (Ersetzung von „e“ durch „u“)
3. Qubell → Quell (Löschen von „b“)
4. Quell → Quelle (Einfügen von „e“)

Ein populärer Algorithmus zur Berechnung der Levenshtein-Distanz arbeitet nach dem Prinzip der dynamischen Programmierung. Gegeben seien die Zeichenketten a und b mit den Längen $m = |a|$ und $n = |b|$. Die Berechnung der Editierdistanz erfolgt in einer Matrix D mit der Dimension $(m+1) \times (n+1)$, deren Elemente $D_{i,j}$ die minimale Anzahl an Operationen enthalten, die zur Überführung von $a_{1..i}$ nach $b_{1..j}$ erforderlich sind. Beginnend mit $D_{0,0}$ werden die Matrixzellen gemäß folgender Berechnungsvorschrift belegt:

$$D_{i,0} = i \quad 1 \leq i \leq m \tag{4.1}$$

$$D_{0,j} = j \quad 1 \leq j \leq n \tag{4.2}$$

$$D_{0,0} = 0 \tag{4.3}$$

$$D_{i,j} = min \begin{cases} D_{i-1,j-1} + 0 & \text{wenn } a_i = b_j \\ D_{i-1,j-1} + 1 & \text{bei Ersetzung} \\ D_{i,j-1} + 1 & \text{bei Einfügung} \\ D_{i-1,j} + 1 & \text{bei Löschung} \end{cases} \tag{4.4}$$

Nachdem die Matrix D gemäß dieser Vorschrift vollständig gefüllt wurde, steht das Ergebnis für die Levenshtein-Distanz in der unteren rechten Ecke, also an der Stelle $D_{m,n}$. Tabelle 4.4 zeigt die Levenshtein-Matrix für das Beispiel mit den Wörtern „Rebell“ und „Quelle“.

Tabelle 4.4: Beispiel für eine Levenshtein-Matrix.

		R	e	b	e	l	l
	0	1	2	3	4	5	6
Q	1	1	2	3	4	5	6
u	2	2	2	2	3	4	5
e	3	3	3	3	3	4	5
l	4	4	4	3	4	4	4
l	5	5	5	4	3	4	5
e	6	6	6	5	4	3	**4**

Der maximale Wert für die Levenshtein-Distanz zwischen zwei beliebigen Zeichenketten a und b lässt sich formulieren als $max(|a|, |b|)$. Dieser Wert entspricht somit der Länge der längeren der beiden Zeichenketten und wird nur dann erreicht, wenn die beiden Zeichenketten kein gemeinsames Zeichen haben. Der minimale Wert für die Levenshtein-Distanz ist Null. Er wird erreicht, wenn die Zeichenketten identisch sind und zusätzlich $|a| = |b|$ gilt.

Beim Matching soll eine konfigurierbare absolute Toleranzschwelle für die Levenshtein-Distanz berücksichtigt werden. Ist die Levenshtein-Distanz zwischen einer Hypothese und einem Erkennungsmuster kleiner als diese Toleranzschwelle, werden Hypothese und Erkennungsmuster als identisch betrachtet. Für den Fall, dass mehrere Hypothesen und Erkennungsmuster identisch sind, gilt jenes Erkennungsmuster als erkannt, das die größte Ähnlichkeit mit der Hypothese besitzt. Dazu wird die Wortähnlichkeit unter Verwendung der Levenshtein-Distanz (LD) bewertet. Für die Berechnung der Ähnlichkeit zweier Zeichenketten a und b wird die Gleichung 4.5 verwendet. Beim Beispiel mit den Wörtern „Rebell“ und „Quelle“ ergibt sich so eine Ähnlichkeit von $33{,}\dot{3}\,\%$.

$$sim(a, b) = 1 - \frac{LD(a, b)}{max(|a|, |b|)} \tag{4.5}$$

Zusätzlich zu der erwähnten absoluten Toleranzschwelle, ist es weiters erforderlich, eine Schwelle für die relative Ähnlichkeitsbewertung zu berücksichtigen. Diese Schwelle gibt an, wie groß die Ähnlichkeit sein muss, damit die Hypothese und das Erkennungsmuster als identisch betrachtet werden. Die Notwendigkeit dafür soll anhand der Beispiele aus Tabelle 4.5 demonstriert werden. Angenommen, es sei für die Ähnlichkeit eine Schwelle

von 80 % definiert und die Toleranz für die Levenshtein-Distanz betrage 1. Des Weiteren wurden die Befehle „alles entfernen“ und „ein“ definiert.

Tabelle 4.5: Beispiel für Ähnlichkeitsbewertung.

Klartext	**PR**	**Ähnlichkeit**	**LD**
alles entfernen nichts entfernen	058 0623766 064 0623766	81,8̇i%	2
ein ja	06 0	50%	1

Würde nun die Wortfolge „nichts entfernen“ gesprochen werden und wäre keine Toleranzschwelle für die Levenshtein-Distanz vorhanden, würde der Befehl „alles entfernen“ ausgelöst werden, da die Ähnlichkeit der phonetischen Repräsentationen über der definierten Ähnlichkeitsschwelle liegen würde. Andererseits würde bei nicht vorhandener Ähnlichkeitsschwelle beim Sprechen von „ja“ der Befehl „ein“ erkannt werden, da die Levenshtein-Distanz sich innerhalb der definierten Toleranz bewegen würde. Für die Vermeidung solcher falsch-positiven Ergebnisse ist eine kombinierte Bewertung daher unumgänglich.

4.1.7 Testanforderungen

Bei der Implementierung wird ein testgetriebener Ansatz verfolgt. Um eine möglichst hohe Qualität der entwickelten Software zu gewährleisten, werden bereits während der Entwicklung Komponententests (Unit Tests) durchgeführt. Die Erstellung von Unit Tests wird, wenn möglich, vor oder parallel zur Umsetzung der eigentlichen Funktionalität erfolgen. Die Zielsetzung ist es, eine Sammlung automatischer Unit Tests aufzubauen, welche eine weitgehend vollständige Testabdeckung (engl. Code Coverage) erreicht, also die Anzahl bei der Testausführung durchlaufener Anweisungen maximiert. Durch die automatischen Unit Tests kann während der Entwicklung und bei zukünftigen Erweiterungen stets sichergestellt werden, dass durchgeführte Änderungen keine Seiteneffekte haben.

4.2 Prototyp

Der Prototyp wurde gemäß des in Abschnitt 4.1 erstellten Entwurfs implementiert. Das Ergebnis wird im Folgenden vorgestellt.

4.2.1 Integration in die Anwendung

Der Prototyp wurde in nativem JavaScript implementiert. Er weist somit keine Abhängigkeiten zu anderen Bibliotheken – wie zum Beispiel jQuery – auf. Der gesamte Programmcode ist in einer einzigen Datei gespeichert. Für ein Projekt dieser Größe ist dies vertretbar. Bei größeren Projekten würde bei diesem Vorgehen der Quellcode aber schnell sehr unübersichtlich werden. In so einem Fall sollte dieser auf mehrere Dateien aufgeteilt und der Einsatz eines Script Loaders (z. B. *RequireJS* [63] oder *$script.js* [64]), welcher das Laden der Dateien übernimmt und die Abhängigkeiten verwaltet, angedacht werden.

Für den Prototypen wurde auf die Verwendung eines Script Loaders verzichtet. Die Integration in eine HTML-Seite erfolgt somit durch das Einbinden einer einzigen JavaScript-Datei mittels `<script>`-Tag, wie es in Listing 4.9 dargestellt ist.

```
1 ...
2 <head>
3   <script src="webasr.min.js" type="text/javascript"></script>
4   ...
5 </head>
6 ...
```

Listing 4.9: Einbinden des Sprachsteuerungsmoduls.

Der fertige Prototyp steht in zwei Versionen zur Verfügung. Neben der kommentierten Vollversion `webasr.js`, existiert auch eine komprimierte Version `webasr.min.js`, bei welcher alle für die Ausführung unwichtigen Zeichen – wie beispielsweise Kommentare, Leerzeichen, Zeilenumbrüche etc. – aus dem Quellcode entfernt und die Variablennamen gekürzt wurden. Das Ziel dieses üblicherweise als *Minification*[2] bezeichneten Vorgehens ist es, die Ladezeit des Quellcodes zu reduzieren, indem dessen Größe verringert wird [26]. Für den produktiven Einsatz empfiehlt es sich somit, die Datei `webasr.min.js` zu verwenden. Für die Minification von JavaScript-Quellcode existieren verschiedene Werkzeuge. Im Fall des Prototyps wurde das Tool *YUI Compressor* [65] verwendet.

[2] Seltener auch Minimisation.

4.2.2 Schnittstelle zur Anwendung

Wird das Modul, wie im vorhergehenden Abschnitt beschrieben, integriert, steht das globale Objekt `WebASR` zur Verfügung. Die Methoden dieses Objekts bilden die Schnittstelle zwischen Anwendung und Sprachsteuerungsmodul. Abbildung 4.8 zeigt einen Überblick der bereitgestellten Methoden.

Abbildung 4.8: Programmierschnittstelle des Spracherkennungsmoduls.

Das Listing 4.10 zeigt ein Beispiel für die Initialisierung des Sprachsteuerungsmoduls. Es ist zumindest erforderlich, eine Befehlsdatei zu laden. Dazu können die Methoden `loadCommandsFromXML` und `addCommandsFromXML` verwendet werden, wobei erstere Methode die bereits geladenen Befehle zuerst löscht. Sind Befehle im Local Storage gespeichert, können diese auch mittels der Methode `loadCommandsFromLocalStorage` wiederhergestellt werden. Das Setzen von Konfigurationsparametern (siehe Abschnitt 4.2.3) ist optional. Erfolgt dies nicht, wird die Default-Konfiguration verwendet. Das Hinzufügen von Event Handlern (siehe Abschnitt 4.2.4) ist ebenfalls optional. Durch den Aufruf der Methode `start` beginnt das Modul zuzuhören. Wird dabei nicht `false` für den optionalen Parameter `continuous` übergeben, erfolgt immer eine kontinuierliche Spracherkennung, was bedeutet, dass solange mitgehört wird, bis die Methode `stop` aufgerufen wird.

```
<script type="text/javascript">
  window.onload = function(){
    // Sprache einstellen
    WebASR.setLanguage("de-DE");

    // Laden der Kommandos
    WebASR.loadCommandsFromXML("./commands/commands.xml");

    // Handler für den Fehlerfall registrieren
    WebASR.registerCallbackHandler("onError", handleOnError);

    // Modul starten
    WebASR.start();
  }

  function handleOnError() { ... }
</script>
```

Listing 4.10: Initialisierung des Sprachsteuerungsmoduls.

4.2.3 Konfigurationsparameter

Zur Konfiguration des Sprachsteuerungsmoduls existieren verschiedene Parameter, für welche die Schnittstelle entsprechende Getter- und Setter-Methoden bereitstellt. Folgende Einstellungen können vorgenommen werden:

language: Durch diesen Parameter wird die zu erkennende Sprache festgelegt. Zur Konfiguration der Sprache wird die Kodierung nach RFC 5646 verwendet.

maxAlternatives: Dieser Parameter bestimmt die maximale Anzahl der vom Spracherkennungsservice gelieferten Hypothesen. Zu beachten ist, dass ein zu niedriger Wert die Erkennungsrate des Gesamtsystems negativ beeinflussen kann, da unter Umständen Hypothesen, die zu einem Treffer führen könnten, verworfen werden. Während der Umsetzung hat sich gezeigt, dass Werte ab 3 gute Ergebnisse liefern. Es wurde außerdem beobachtet, dass die Anzahl erhaltener Hypothesen bei normal langen Anweisungen (1 bis 5 Wörter) selten über 5 liegt, auch wenn einen höherer Wert konfiguriert ist.

levenshteinThreshold: Diese Eigenschaft erlaubt das Definieren eines Schwellenwerts für die Levenshtein-Distanz. Ist die Levenshtein-Distanz zwischen einer Hypothese und einem Erkennungsmuster kleiner als diese Schwelle, werden Hypothese und Erkennungsmuster als identisch angesehen. Um falsch-positive Ergebnisse zu vermeiden, sollte dieser Wert nicht zu hoch gewählt werden. Das gilt insbesondere bei

aktiviertem phonetischen Matching, da die phonetischen Repräsentationen in der Regel deutlich kürzer als die Ursprungszeichenketten sind. Während der Umsetzung hat sich in diesem Fall der Wert 1 bewährt.

scoreThreshold: Mit dieser Eigenschaft kann eine Schwelle für die Ähnlichkeitsbewertung gesetzt werden. Alle Treffer unter dieser Schwelle werden verworfen. Dieser Parameter bietet somit eine zusätzliche Möglichkeit, falsch-positive Ergebnisse zu reduzieren. Wird der Wert allerdings zu hoch gewählt, können bereits sehr kleine Unterschiede zwischen Hypothese und Erkennungsmuster dazu führen, dass ein Befehl nicht erkannt wird. Während der Umsetzung hat sich der Wert 0,65 als ein guter Kompromiss erwiesen.

confidenceThreshold: Durch das Setzen einer Konfidenzschwelle kann erreicht werden, dass nur Hypothesen, welche einen Konfidenzwert größer der eingestellten Schwelle besitzen, beim Matching berücksichtigt werden.

phoneticMatching: Mittels dieser Einstellung kann gesteuert werden, ob das Matching auf Basis der phonetischen Repräsentationen erfolgen soll.

acceptInterimResults: Durch diesen Parameter wird auf Systemebene festgelegt, ob Zwischenergebnisse für das Matching berücksichtigt werden sollen. Die vorgenommene Einstellung kann auf Befehlsebene mittels des Attributs `matchInterim` überlagert werden (vgl. Abschnitt 4.1.4).

debugMode: Mit dieser Eigenschaft lässt sich der Debug-Modus aktivieren, sodass Log-Ausgaben in der Konsole angezeigt werden. Der Debug-Modus ist nützlich für Test und Fehlersuche, sollte aber im produktiven Einsatz nicht aktiviert werden.

Die Default-Einstellungen sind durch das im Hauptmodul deklarierte Objekt `config` festgelegt. Listing 4.13 zeigt die Einstellungen, die in der finalen Version des Prototypen implementiert sind.

```
1  var config = {
2    'defaultLanguage': 'de-DE',     // German
3    'maxAlternatives': 5,           // Should not be lower than 3
4    'continuousMode': true,         // activated
5    'debugMode': false,             // deactivated
6    'levenshteinThreshold': 1,      // Distance of 1 is allowed
7    'scoreThreshold': 0.65,         // All results below 0.65 are skipped
8    'confidenceThreshold': 0,       // everything is accepted
9    'usePhoneticMatching': true,    // activated
10   'acceptInterimResults': true    // accepted
11 };
```

Listing 4.11: Default-Einstellungen.

4.2.4 Ereignisverarbeitung

Um Event Handler zu registrieren, kann die Methode `registerCallbackHandler` verwendet werden. Als Parameter müssen der Methode zumindest der Name des Events und der Funktionszeiger der Handler-Funktion übergeben werden. Die Angabe eines Kontexts ist optional. Durch das Aufrufen der Methode `clearCallbackHandlers` werden alle registrierten Event Handler gelöscht. Die Namen der möglichen Events sowie eine Beschreibung dieser können der Tabelle 4.6 entnommen werden.

Tabelle 4.6: Übersicht der Ereignisse.

Name	Beschreibung
onStart	Start der Spracherkennung (hört zu)
onEnd	Beenden der Spracherkennung (hört nicht mehr zu)
onResult	Hypothesen vom ASR Service erhalten (sowohl bei finalen Resultaten als auch bei Zwischenergebnissen)
onNoMatch	Gesprochene Wortfolge konnte durch ASR Service nicht erkannt werden
onCommandMatch	Matching war erfolgreich und es wurde ein Befehl erkannt
onNoCommandMatch	Matching war nicht erfolgreich und es wurde kein Befehl erkannt
onError	Es ist ein Fehler aufgetreten (allgemeines Fehlerereignis)
onNoPermission	Fehlerereignis: Zugriff auf das Mikrofon wurde durch Benutzer abgelehnt
onServiceNotAllowed	Fehlerereignis: Zugriff auf den ASR Service wurde durch den Browser abgelehnt
onNetworkError	Fehlerereignis: Bei der Netzwerkkommunikation ist ein Fehler aufgetreten
onAudioCaptureError	Fehlerereignis: Bei der Audioaufnahme ist ein Fehler aufgetreten
onNoSpeechDetected	Fehlerereignis: Es konnte kein Sprechen erkannt werden
onAudioStart	Audioaufnahme wurde gestartet
onAudioEnd	Audioaufnahme wurde beendet
onSoundStart	Ein Geräusch wurde wahrgenommen
onSoundEnd	Kein Geräusch kann mehr wahrgenommen werden
onSpeechStart	Sprechen wurde wahrgenommen
onSpeechEnd	Kein Sprechen wird mehr wahrgenommen

Hinsichtlich der Ereignisse seien noch die folgenden Punkte angemerkt: Im Fehlerfall wird immer das Ereignis `onError` zuerst gefeuert. Anschließend wird entsprechend des Fehlertyps ein weiteres Fehlerereignis ausgelöst (z. B. `onNetworkError` im Falle einer fehlerhaften Netzwerkkommunikation zwischen dem Client und dem Spracherkennungsdienst). Das Ereignis `onNoMatch` konnte während der gesamten Implementierung nicht

beobachtet werden. Selbst beim Sprechen willkürlicher Laute, wurde immer ein Resultat durch den Spracherkennungsservice geliefert. Ebenfalls waren alle Versuche, die Ereignisse `onAudioCaptureError` und `onServiceNotAllowed` zu provozieren, nicht erfolgreich. Diese Ereignisse werden direkt von der Web Speech API gefeuert und durch das Sprachsteuerungsmodul lediglich weitergereicht. Es ist möglich, dass die Ereignisse derzeit noch nicht gemäß der Spezifikation implementiert sind.

4.2.5 Demonstrationsanwendung

Die Demoanwendung besteht aus einer einfachen HTML-Seite (`demopage.html`), die das Sprachsteuerungsmodul integriert. Die Anwendung kennt acht verschiedene Befehle, welche in der Datei `commands.xml` mit jeweils einem Erkennungsmuster für Deutsch und Englisch definiert sind. Die Befehle sind so gestaltet, dass alle durch das XML-Format unterstützten Konstrukte (z. B. Wiederholungen, Variabeln etc.) Verwendung finden. Auf der HTML-Seite sind die Befehle – mit dem zu sprechenden Wortlaut und einer Beschreibung – tabellarisch dargestellt. Durch das Sprechen eines Befehls wird in der jeweiligen Resultat-Spalte die Verarbeitung visualisiert. Auf der Seite werden außerdem alle Konfigurationsparameter des Moduls durch entsprechende Bedienelemente (z. B. Checkboxen, Slider etc.) veränderbar gemacht, sodass sehr einfach verschiedene Konfigurationen getestet werden können. Um die Benutzeroberfläche optisch ansprechend zu gestalten, wurden CSS (`demopage.css`) und die JavaScript-Bibliothek *jQuery UI* [66] verwendet. jQuery UI ist eine Erweiterung der jQuery-Bibliothek und bietet umfangreiche Lösungen zur Umsetzung von Benutzeroberflächen. Die API ist gut dokumentiert und einfach zu verwenden. Ein Screenshot der mittels jQuery UI gestalteten Demoanwendung ist in Abbildung 4.9 zu sehen.

Damit die Web Speech API korrekt funktioniert, ist es erforderlich, dass die sie benutzende Anwendung von einem Webserver bereitgestellt wird. Um bei der Umsetzung der Demoanwendung den Aufwand für Installation und Konfiguration eines Webservers so gering wie möglich zu halten, wurde ein leichtgewichtiger und einfach zu verwendender Webserver gesucht, wobei die Wahl schlussendlich auf den Webserver `Mongoose` [67] fiel. Mongoose ist unter den Bedingungen der GNU General Public License (GPL) Version 2 als freie Software lizenziert. In seiner Minimalkonfiguration besteht der Webserver aus einer einzigen Programmdatei. Wird diese Datei innerhalb eines Ordners ausgeführt, wird der Inhalt des Ordners umgehend auf Port 8080 bereitgestellt. Beim erstmaligen Verwenden der Demoanwendung wurde festgestellt, dass die Zugriffserlaubnis auf das Mikrofon nicht gespeichert werden konnte. Der Zugriff musste daher bei jedem Anwendungsaufruf erneut bestätigt werden. Eine Recherche ergab, dass die Zugriffserlaubnis nur dann durch den Browser gespeichert werden kann, wenn die Anwendung über HTTPS geladen wird. In weiterer Folge wurde daher mittels `OpenSSL` [68] ein selbst-signiertes Server-

Zertifikat erstellt, welches der Webserver verwendet, um die Anwendung über HTTPS bereitzustellen. Die entsprechende Konfiguration dafür (u. a. Port 443 und Pfad zum Zertifikat) wurde in der Datei `mongoose.conf` vorgenommen.

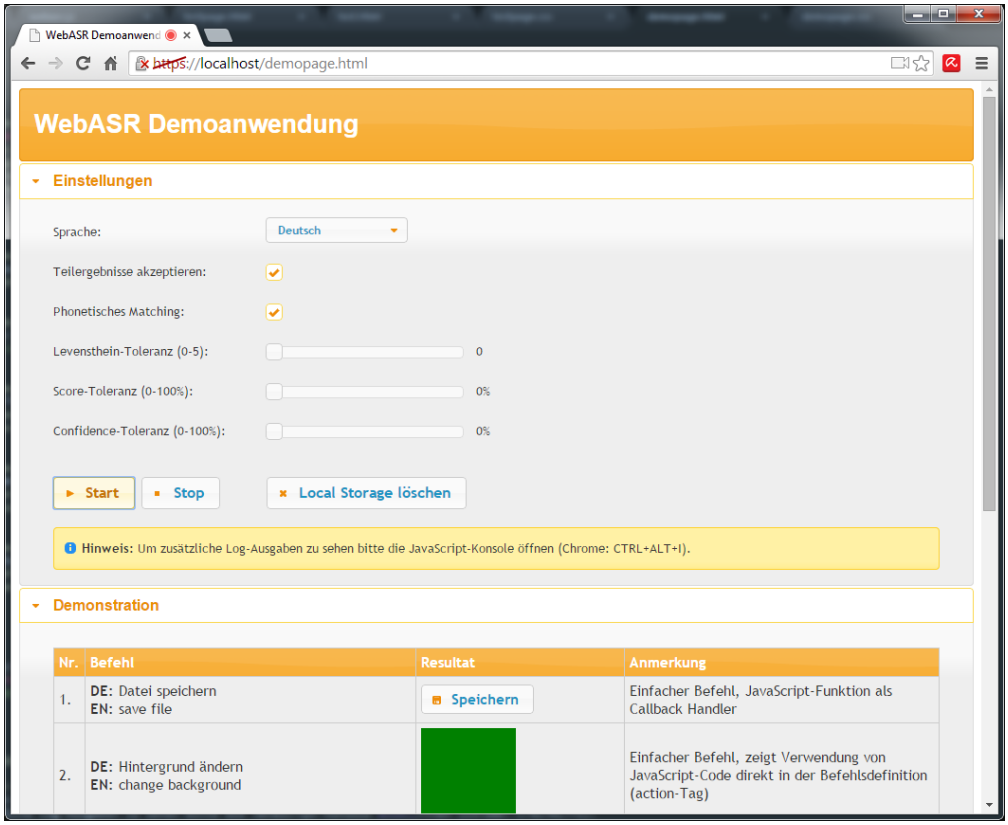


Abbildung 4.9: Screenshot der Demoanwendung.

4.2.6 Unit Tests

Für die Implementierung der Unit Tests wurde das Unit Test Framework *QUnit* [69] verwendet. Ursprünglich zum Testen von jQuery entwickelt, stellt QUnit heute ein eigenständiges Test Framework dar, welches sehr weit verbreitet ist [27]. QUnit ist quelloffen und stellt eine einfache API, welche unter anderem auch asynchrone Tests unterstützt, sowie einen Test Runner zur Verfügung. Zur Ermittlung der Testabdeckung wird das Werkzeug *Blanket.js* [70] verwendet. Blanket.js ist eine kleine und einfach zu verwendete JavaScript-Bibliothek, die sich gut mit QUnit integrieren lässt. Für die Testausführung wurde eine Unit Test Page implementiert, die QUnit, Blanket.js, das Sprachsteuerungsmodul und die in eine eigene JavaScript-Datei ausgelagerten Testfälle integriert. Ein Screenshot der implementierten Unit Test Page ist in Abbildung 4.10 dargestellt.

Insgesamt wurden 237 Testfälle implementiert. Darunter fallen sowohl einfache Testfälle (z. B. zur Überprüfung von Getter- und Setter-Methoden) als auch komplexere Testfälle, welche den gesamten Prozess der Befehlserkennung überprüfen. Das Listing 4.12

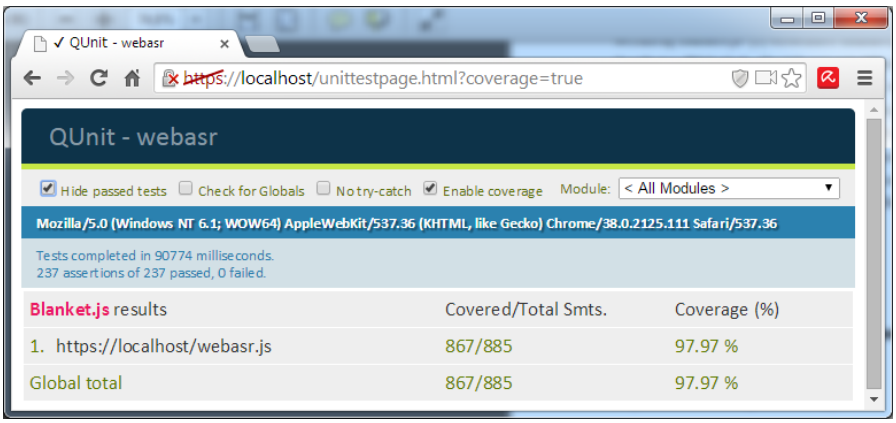


Abbildung 4.10: Ergebnis des Unit Tests.

stellt ein Beispiel für einen komplexen Testfall dar. Ziel des Testfalls ist es, zu testen, ob das `onResult`-Ereignis korrekt ausgelöst wird. Dafür wird ein asynchroner Test erstellt und dem Framework mittels `expect`-Methode mitgeteilt, dass der Test aus zwei Assertionen besteht (Zeile 1 und 2). Die erste Assertion erfolgt nach Initialisierung und Start des Spracherkennungsmoduls (Zeile 12), die zweite Assertion erfolgt innerhalb der für das `onResult`-Ereignis registrierten Handler-Funktion (Zeile 8). Mittels der eigens implementierten Funktion `playRecording`, wird unter Verwendung der Web Audi API eine Audiodatei mit dem aufgenommenen Befehl abgespielt (Zeile 13). Bei korrekter Funktion registiert das Spracherkennungsmodul den Befehl und löst in weiterer Folge das `onResult`-Ereignis aus. In der Handler-Funktion wird die Assertion durchgeführt und mittels Aufruf der Methode `start` dem Framework mitgeteilt, dass der Test beendet ist und fortgefahren werden kann.

```
1  QUnit.asyncTest("callbackhandler_10", function(assert) {
2    expect(2);
3    WebASR.clearCommands();
4    WebASR.clearCallbackHandlers();
5    WebASR.loadCommandsFromXML("./resources/commands/commands.xml");
6    WebASR.registerCallbackHandler("onResult", function() {
7      WebASR.stop();
8      assert.ok(true, "onResult handler must be called");
9      QUnit.start();
10   });
11   WebASR.start();
12   assert.ok(true, "WebASR started");
13   setTimeout(function() {
14     playRecording("./resources/recordings/DATEI_SPEICHERN.mp3");
15   }, 1500);
16 });
```

Listing 4.12: Beispiel für einen Unit Test.

Mittels der implementierten Unit Tests werden 867 der insgesamt 885 Anweisungen des Sprachsteuerungsmoduls abgedeckt, was einer Code Coverage von 97,97% entspricht. Bei den Anweisungen, die derzeit nicht abgedeckt werden, handelt es sich fast ausschließlich um Anweisungen, welche der Fehlerbehandlung dienen. Da die behandelten Fehler (z. B. Network Error) nicht kontrolliert provoziert werden können, kann dieser Quellcode auch nicht automatisch getestet werden.

4.2.7 Dokumentation

Im Sinne der Wart- und Erweiterbarkeit wurde der gesamte Quellcode des Prototyps mit Kommentaren in englischer Sprache versehen. Für die Erstellung der Kommentare wurde die Aufzeichnungssprache des *JSDoc 3*-Projekts verwendet [71]. JSDoc 3 ist ein mit dem bekannten JavaDoc vergleichbares Dokumentationswerkzeug. Es dient zur automatischen Generierung von Software-Dokumentationen, indem die JavaScript-Dateien geparst und die enthaltenen Kommentare in ein strukturiertes HTML-Dokument umgewandelt werden.

Bei der Verwendung von JSDoc 3 hat sich gezeigt, dass das Werkzeug Probleme mit Quellcode, der in Modulen strukturiert ist, hat. Module und die darin enthaltenen Klassen, Funktionen und Methoden wurden teilweise nicht automatisch erkannt. Sie mussten explizit mit Annotationen versehen werden, damit sie in die generierte Dokumentation aufgenommen wurden. Das Listing 4.13 zeigt ein Beispiel für eine mittels der JSDoc 3-Aufzeichnungssprache kommentierten Methode.

```
1  /**
2   *  Calculation of the Levenshtein distance between two given strings
3   *  using the dynamic programming approach.
4   *
5   *  @function calcLevDist
6   *  @public
7   *  @param {string}   str1 - 1st string
8   *  @param {string}   str2 - 2nd string
9   *  @returns {number}   Levenshtein distance between 1st and 2nd string
10  *  @example
11  *  //returns 2
12  *  calculateLevDist('cow','cat');
13  */
14 function calcLevDist(str1, str2) {
15   ...
16 }
```

Listing 4.13: Beispiel für einen JSDoc 3-Kommentar.

Ein Beispiel für ein mit JSDoc 3 generiertes HTML-Dokument ist in Abbildung 4.11 dargestellt. Der Screenshot zeigt die Dokumentation des Moduls `LanguageUtils`, das unter anderem die in Listing 4.13 kommentierte Methode beinhaltet.

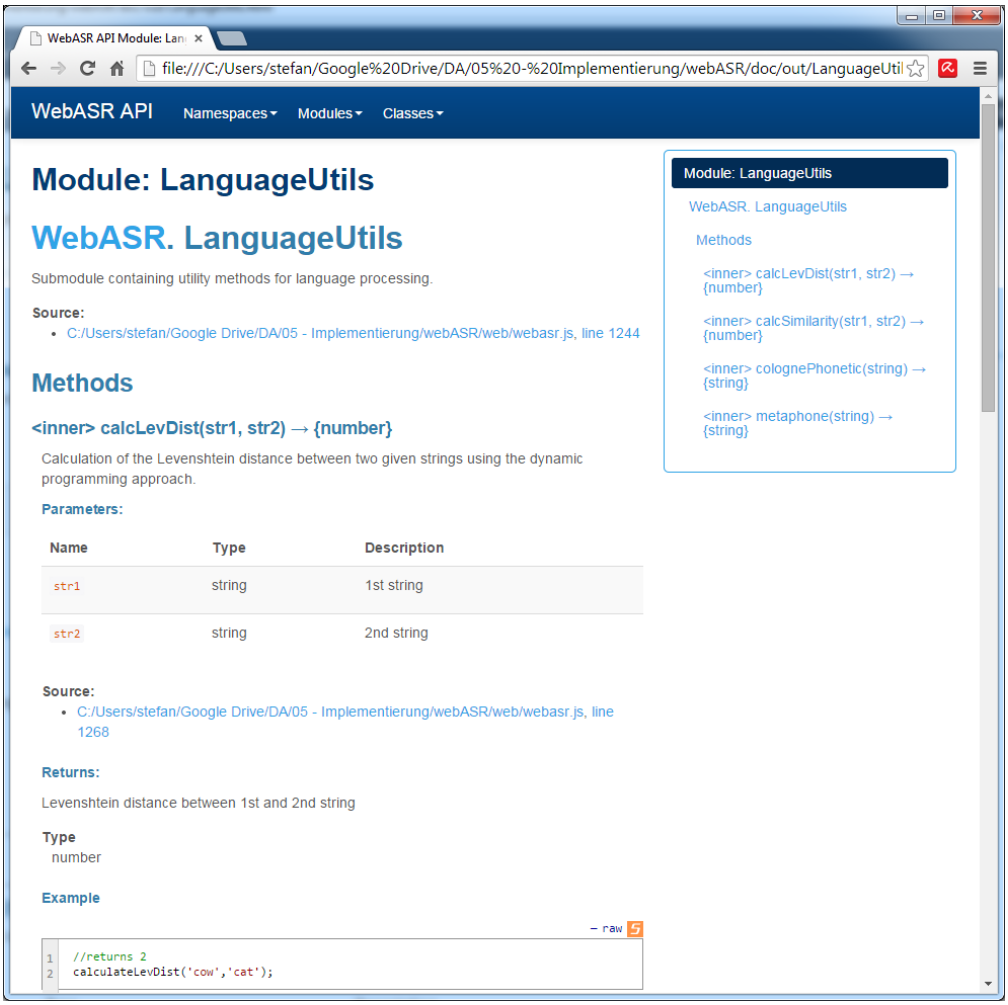


Abbildung 4.11: Beispiel für JSDoc 3-Dokument.

Kapitel 5

Evaluierung des Prototyps

Dieses Kapitel beschäftigt sich mit der Evaluierung des implementierten Prototyps. In Abschnitt 5.1 erfolgt zunächst die Definition der bei der Evaluierung verfolgten Zielsetzung. Anschließend werden in Abschnitt 5.2 das Test-Setup und die Einzelheiten der Durchführung beschrieben. Im abschließenden Abschnitt 5.3 erfolgt dann die Präsentation und Diskussion der Ergebnisse.

5.1 Zielsetzung

Das Ziel der Evaluierung des Prototyps ist es, zu untersuchen, welche Erkennungsleistungen mit diesem unter verschiedenen Bedingungen erzielt werden können. Die Ergebnisse sollen dabei helfen, eine Vorstellung davon zu bekommen, für welche Anwendungen der Einsatz des Prototyps zukünftig sinnvoll ist. Von Interesse sind die Befehlserkennungsrate und die Verzögerung zwischen dem Sprechen eines Befehls und dessen Ausführung. Im Speziellen soll untersucht werden, wie sich verschiedene Faktoren auf diese Variablen auswirken. Konkret sind dabei die Einflüsse folgender Faktoren von Interesse:

- Unterschiedliche *Mikrofon-Sprecher-Distanzen*
- Additive Störungen in Form von *Hintergrundgeräuschen*
- Unterschiedliche *Mikrofone*
- Unterschiedliche *Konfigurationen* des Spracherkennungsmoduls (u.a. phonetisches Matching, Zwischenergebnisse)

Wie in Abschnitt 2.1.3 beschrieben, verschlechtert sich mit zunehmendem Abstand zwischen Signalquelle und Mikrofon das Signal-Rausch-Verhältnis, wodurch eine deutlich reduzierte Erkennungsleistung zu erwarten ist. Hintergrundgeräusche stellen eine additive Störung dar. Sie verschlechtern das Signal-Rausch-Verhältnis zusätzlich, was die Erkennungsleistung weiter sinken lässt. Es ist anzunehmen, dass durch die Verwendung eines hochwertigen Mikrofon-Arrays diese Effekte teilweise abgefedert werden können,

wodurch bessere Erkennungsleistungen erreicht werden sollten. Des Weiteren ist anzunehmen, dass durch die Berücksichtigung von Zwischenergebnissen die Verzögerungszeit reduziert werden kann. Auch sollte durch das fehlertolerante phonetische Matching eine signifikant bessere Erkennungsrate erzielt werden können, als bei Verwendung der unverarbeiteten Hypothesen.

5.2 Setup und Durchführung

5.2.1 Testanwendung

Für die Durchführung der Messungen wurde eine eigene Testanwendung implementiert, die eine automatische Messung ermöglicht. Die Anwendung integriert das Spracherkennungsmodul und unterstützt einen Befehlssatz aus insgesamt 20 Befehlen:

1. ja
2. nein
3. abbrechen
4. annehmen
5. hilfe
6. zurück
7. weiter
8. rückgängig
9. wiederholen
10. start
11. stop
12. pause
13. herunterfahren
14. kopieren
15. einfügen
16. Datei öffnen
17. Datei speichern
18. Datei löschen
19. Programm starten
20. Programm beenden

Allen Befehlen wurde derselbe Event Handler zugeordnet. Um bei den Messungen eine Verfälschung der Messergebnisse durch unterschiedliche Sprechweisen der Befehle zu vermeiden, werden bei der Durchführung die Befehle nicht von einer Person gesprochen. Anstelle dessen werden im Vorfeld erzeugte Aufnahmen, welche mittels *Audacity* [72], einer quelloffenen Software für die Aufnahme und Bearbeitung von Audio erstellt wurden, verwendet. Die Testanwendung gibt unter Verwendung der Web Audio API die Aufnahmen der Befehle nacheinander in konstanter Lautstärke wieder. Nach jeder Wiedergabe einer Aufnahme wartet die Anwendung darauf, dass der den Befehlen zugeordnete Event Handler ausgelöst wird. Geschieht dies, wird zunächst die Verzögerung berechnet. Anschließend wird überprüft, ob der erkannte Befehl auch dem gesprochenen Befehl entspricht. Trifft dies zu, gilt der Befehl als erkannt. Wird hingegen kein Event Handler ausgelöst oder stimmen Aufnahme und erkannter Befehl nicht überein (falsch-positiven Erkennung), gilt der Befehl als nicht erkannt.

Die Verzögerung ergibt sich aus der Differenz zwischen dem Ende des Sprechens des Befehls und dem Auslösen des dem Befehl zugeordneten Event Handlers. Das Ende des Sprechens lässt sich aus dem Zeitstempel des `onSpeechStart`-Events, das durch die Web Speech API ausgelöst wird, nachdem Sprache wahrgenommen wurde, und der mittels

Audacity ermittelten Sprechdauer des Befehls errechnen. Es gilt für die Verzögerung somit:

$$\Delta t \;=\; t_{Event} - (\; t_{onSpeechStart} + t_{Sprechdauer} \;) \tag{5.1}$$

Das Ergebnis der Messung wird auf der HTML-Seite der Testanwendung in tabellarischer Form angezeigt. Abbildung 5.1 zeigt ein Beispiel einer vollständigen Messung.

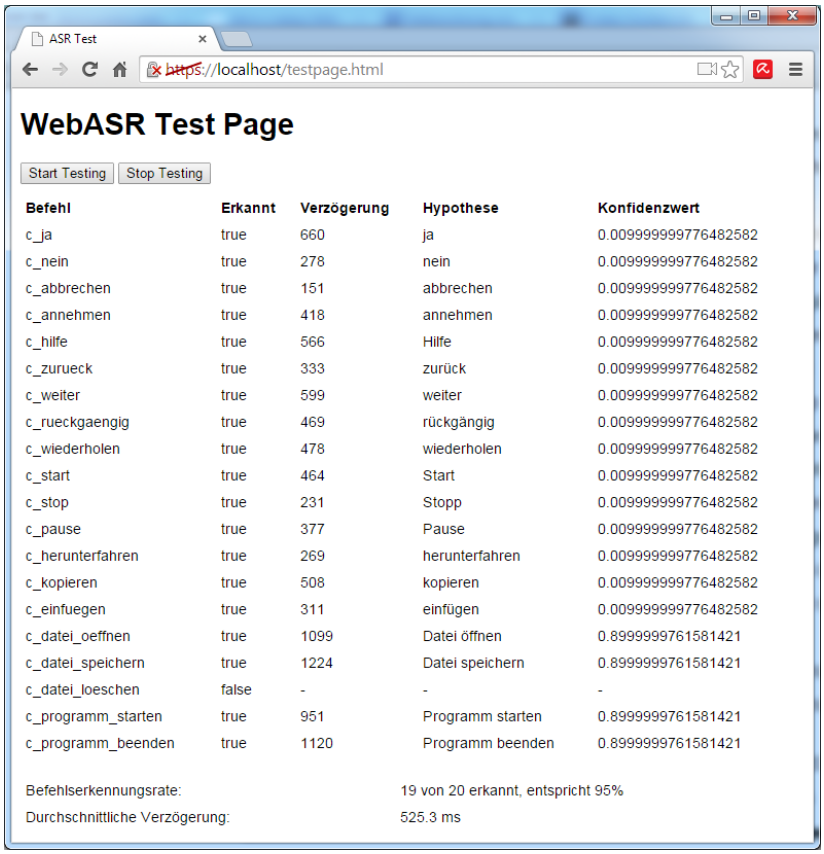


Abbildung 5.1: Ergebnis einer Messung.

5.2.2 Mikrofone

Für die Evaluierung wurden zwei verschiedene Mikrofone verwendet. Die ersten Messungen wurden mit dem *Speedlink Pure Desktop Voice Microphone SL-8702* [74] des Herstellers Jöllenbeck GmbH durchgeführt. Bei diesem Mikrofon handelt es sich um ein handelsübliches Desktop-Mikrofon, dessen idealer Sprechabstand mit 10 cm angegeben wird. Für Vergleichsmessungen wurde das Mikrofon *Superbeam Array-2S* mit der externen Soundkarte *PureAudio USB-SA* [73] verwendet. Sowohl das Mikrofon als auch die Soundkarte stammen vom Hersteller Andrea Electronics und sind für die Verwendung in geräuschvollen Umgebungen ausgelegt. Die Soundkarte wird mit einer Software

ausgeliefert, welche umfassende Konfigurationsmöglichkeiten bietet (z. B. Aktivieren der Geräuschunterdrückung, des Beamformings usw.).

5.2.3 Durchführung

Mit beiden Mikrofonen wurden Messungen für Mikrofon-Sprecher-Distanzen von 10 cm, 30 cm, 70 cm, 120 cm, 250 cm und 500 cm, jeweils mit und ohne Hintergrundgeräuschen, durchgeführt. Das Mikrofon wurde dabei in gleicher Höhe direkt gegenüber vom Lautsprecher positioniert. Zwischen Mikrofon und Lautsprecher befand sich kein Hindernis. Um statistische Fehler auszugleichen, wurde jede Messung dreimal wiederholt und eine durchschnittliche Befehlserkennungsrate bzw. eine durchschnittliche Verzögerung durch die Bildung des arithmetischen Mittelwertes errechnet. Bei Messungen mit Hintergrundgeräuschen wurde für die Erzeugung der Hintergrundgeräusche über einen zweiten Lautsprecher Musik in konstanter Lautstärke abgespielt.

5.3 Ergebnisse

5.3.1 Messungen mit dem Desktop-Mikrofon

Für das Desktop-Mikrofon wurden zwei Messreihen durchgeführt. Die erste Messreihe (MR1) wurde ohne und die zweite Messreihe (MR2) mit Hintergrundgeräusche aufgenommen. Für beide Messreihen war das phonetische Matching aktiviert und die Levenstheins-Schwelle auf den Wert 1 gesetzt. Ansonsten wurde die in Abschnitt 4.2.3 beschriebene Default-Konfiguration verwendet.

In der Abbildung 5.2 ist der Verlauf der Befehlserkennungsraten dargestellt. Die Ergebnisse zeigen deutlich, dass das Desktop-Mikrofon ausschließlich für die Nahbesprechung geeignet ist. Bei dem vom Hersteller empfohlenen Sprechabstand von 10 cm wird sowohl mit als auch ohne Hintergrundgeräuschen eine Befehlserkennungsrate von 100 % erreicht. Erwartungsgemäß nimmt die Erkennungsrate mit steigender Entfernung jedoch stark ab, wobei dieser Effekt bei Hintergrundgeräuschen stärker ausfällt. Während ohne Hintergrundgeräusche bei einer Entfernung von 70 cm noch 95 % der Befehle erkannt werden, ist die Erkennungsrate bei Hintergrundgeräuschen und gleicher Entfernung bereits kleiner als 80 %. Ab einer Entfernung von 120 cm ist auch ohne Hintergrundgeräusche nur noch eine Erkennungsrate von 76,67 % gegeben. Begründet liegt der Abfall der Erkennungsrate nur zu einem kleinen Teil darin, dass die erhaltenen Hypothesen vom tatsächlich Gesprochenen abweichen. Hauptsächlich verantwortlich ist viel mehr der Umstand, dass durch das Modul kaum noch wahrgenommen wurde, dass überhaupt etwas gesprochen wurde. Bei einer Entfernung von 250 cm wurde das `onSpeechStart`-Event bei MR2 niemals und bei MR1 lediglich in 6 % aller Fälle ausgelöst. Bei einer Entfernung von 500 cm wurde auch bei MR1 das Event kein einziges Mal ausgelöst.

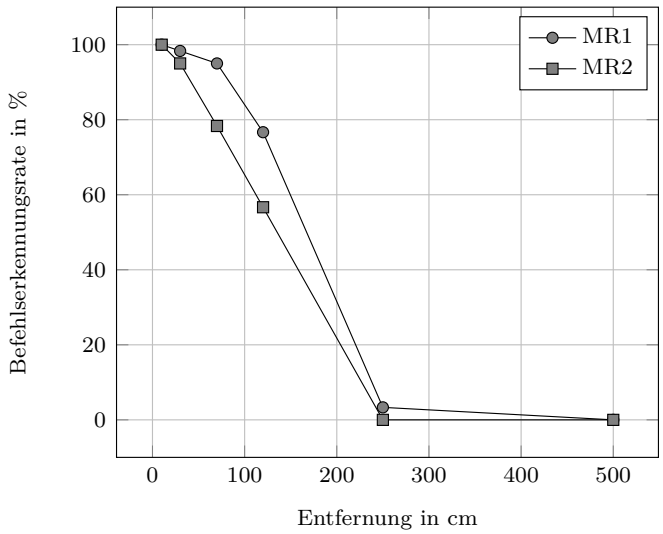


Abbildung 5.2: Befehlserkennungsraten für das Desktop-Mikrofon ohne (MR1) und mit (MR2) Hintergrundgeräuschen.

Die Abbildung 5.3 zeigt die Entwicklung der Verzögerungszeit für MR1 und MR2. Die beobachteten Verzögerungen bewegen sich alle im Bereich von 1,5 bis 1,7 Sekunden. Dieser Bereich ist zwar deutlich wahrnehmbar, erscheint aber dennoch akzeptabel. Erwartungsgemäß ist zu beobachten, dass die Verzögerungszeit mit zunehmender Entfernung zunimmt und dass bei Hintergrundgeräuschen eine größere Verzögerung zu beobachten ist, als in einer geräuscharmen Umgebung. Die Differenz zwischen MR1 und MR2 bewegt sich allerdings in einem für den Benutzer oder die Benutzerin kaum wahrnehmbaren Bereich (maximal 69 ms).

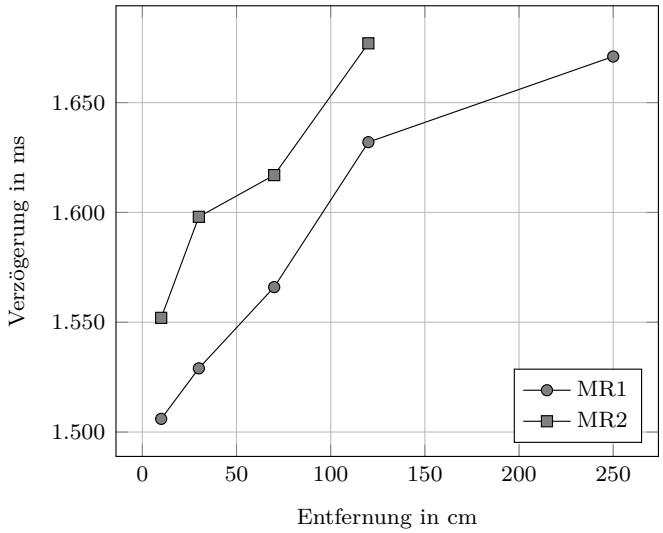


Abbildung 5.3: Verzögerungen für das Desktop-Mikrofon ohne (MR1) und mit (MR2) Hintergrundgeräuschen.

5.3.2 Messungen mit dem Mikrofon-Array

Mit dem Mikrofon-Array wurden zunächst die zwei Messreihen MR3 bzw. MR4 unter denselben Bedingungen wie MR1 bzw. MR2 durchgeführt. Anschließend wurde eine zusätzliche fünfte Messreihe (MR5) unter denselben Bedingungen wie MR2 aufgenommen, wobei für das Mikrofon-Array die Geräuschunterdrückung aktiviert wurde. Die Ergebnisse für die Befehlserkennungsrate sind in Abbildung 5.2 dargestellt. Sie zeigen erwartungsgemäß, dass durch die Verwendung des Mikrofon-Arrays deutlich bessere Resultate als mit dem Desktop-Mikrofon erreicht werden können.

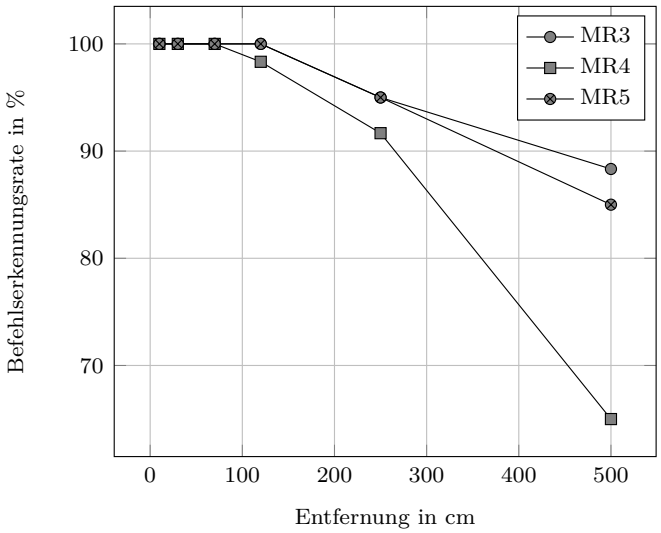


Abbildung 5.4: Befehlserkennungsraten für das Mikrofon-Array ohne (MR3) und mit (MR4) Hintergrundgeräuschen sowie mit Hintergrundgeräuschen und Geräuschunterdrückung (MR5).

Ohne Hintergrundgeräusche ist bis zu einer Sprechentfernung von 120 cm eine Erkennungsrate von 100 % gegeben. Danach fällt der Wert zwar ab, es werden aber selbst bei 500 cm noch 88,33 % aller Befehle erkannt. Mit Hintergrundgeräuschen ist bis 70 cm eine Erkennungsrate von 100 % festzustellen, bei 120 cm werden noch gute 98,33 % erreicht. Erst bei einer Entfernung von 500 cm ist ein deutlicher Abfall zu beobachten. An dieser Stelle werden nur noch 65 % der Befehle erkannt – zu wenig für einen sinnvollen Einsatz. Durch die Aktivierung der Geräuschunterdrückung werden auch mit Hintergrundgeräuschen ähnlich gute Erkennungsraten wie ohne erzielt, wie der Verlauf von MR5 zeigt.

Bei der Entfernung von 500 cm wurde bei MR3 in 100 % und bei MR4 in 88,67 % aller Fälle vom Modul wahrgenommen, dass etwas gesprochen wurde und das `onSpeechStart`-Event ausgelöst. Vor allem bei MR4 wurden dann aber durch den Spracherkennungsservice oftmals Hypothesen geliefert, die so stark vom definierten Erkennungsmuster abwichen, dass beim Matching kein Befehl erkannt werden konnten. Einige Beispiele für solche Hypothesen sind in der Tabelle 5.1 dargestellt.

Tabelle 5.1: Beispiele für abweichende Hypothesen.

Befehl	**Hypothese(n)**
annehmen	AMS, angeln
start	Gas
pause	Hauser, VfL
Datei speichern	Raiffeisen, typewriter
Datei löschen	Hainichen, das heimische

In Abbildung 5.5 sind die Verzögerungszeiten, die bei MR3 bis MR5 gemessen wurden, dargestellt. Die Verzögerungen von MR3 und MR4 liegen in einem ähnlichen Bereich wie jene von MR1 und MR2. Es ist hier kein wesentlicher Unterschied zum Desktop-Mikrofon festzustellen. Eine deutlich wahrnehmbare Steigerung der Verzögerungszeiten ist aber bei der aktivierten Geräuschunterdrückung zu beobachten. Hier bewegen sich die durchschnittlichen Verzögerungen zwischen 2 bis 4 Sekunden. Bei größeren Entfernungen treten außerdem sehr starke Schwankungen auf. Beispielsweise wurden bei 500 cm Zeiten zwischen 1,7 und 15 Sekunden gemessen. Aufgrund dieser Beobachtungen scheint die Geräuschunterdrückung für eine Sprachsteuerung ungeeignet zu sein. Die gemessenen Verzögerungen werden als unangenehm lange empfunden und die Person, welche den Befehl gesprochen hat, kann sich nicht sicher sein, ob die Erkennung immer noch nicht beendet ist oder ob der Befehl nicht wahrgenommen bzw. nicht erkannt wurde.

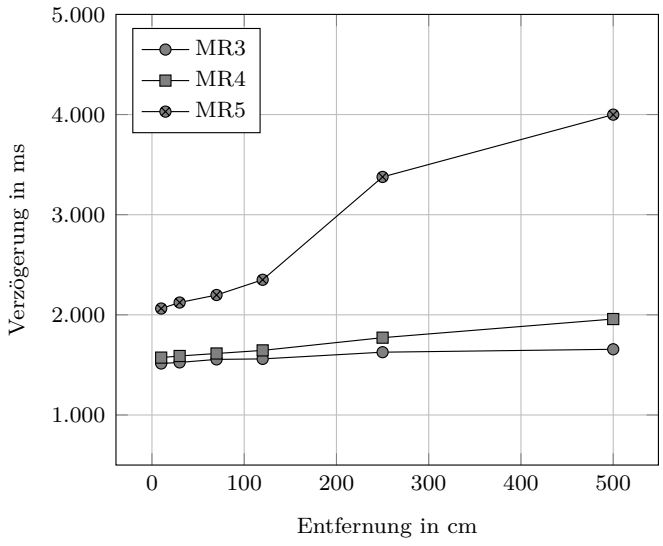


Abbildung 5.5: Verzögerungen für das Mikrofon-Array ohne (MR3) und mit (MR4) Hintergrundgeräuschen sowie mit Hintergrundgeräuschen und Geräuschunterdrückung (MR5).

Um die Verbesserung der Erkennungsrate durch das fehlertolerante Matching auf phonetischer Ebene zu quantifizieren, wurde eine sechste Messreihe (MR6) durchgeführt. Für

diese Messreihe erfolgte eine Veränderung der Konfiguration des Sprachsteuerungsmoduls. Das phonetische Matching wurde deaktiviert und die Levenshtein-Schwelle auf den Wert 0 reduziert, was soviel bedeutet, dass beim Abgleich die unverarbeiteten Hypothesen verwendet und keine Fehler toleriert werden. Die Messungen erfolgten wieder mit den genannten Mikrofon-Sprecher-Distanzen, wobei immer ohne Hintergrundgeräusche gemessen wurde. Erwartungsgemäß hatte die veränderte Konfiguration keinen Einfluss auf die Verzögerungszeit. Bei der Befehlserkennungsrate ist aber eine deutliche Verschlechterung festzustellen, wie Abbildung 5.6 zeigt. Zum Vergleich wurden die Messwerte aus der unter den selben Bedingungen durchgeführten MR3 herangezogen.

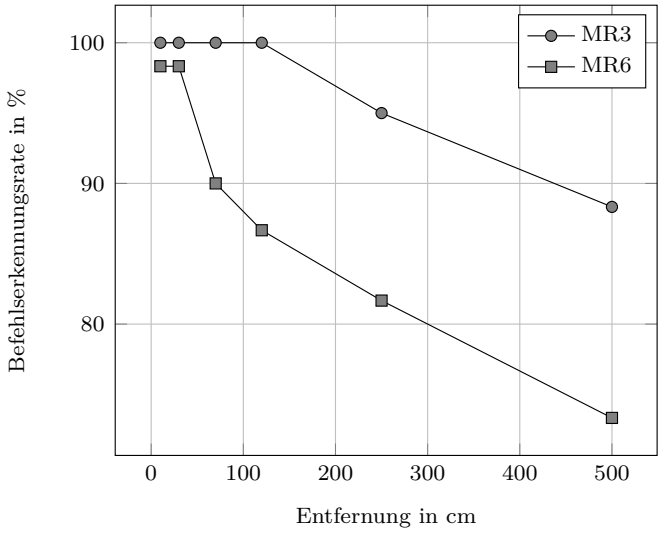


Abbildung 5.6: Befehlserkennungsraten mit (MR3) und ohne (MR6) phonetischem Matching.

Die Ergebnisse zeigen, dass durch das fehlertolerante Matching auf phonetischer Ebene signifikant bessere Befehlserkennungsraten erzielt werden können. Während beim fehlertoleranten Matching bis zu einer Entfernung von 120 cm eine Befehlserkennungsrate von 100 % gegeben ist, wird bei MR6 dieser Wert selbst bei der Nahbesprechung nicht erreicht. Bis zu einer Entfernung von 30 cm werden zwar noch 98,33 % der Befehle erkannt, bei einer Entfernung von 70 cm ist aber bereits ein deutlicher Abfall zu beobachten. Die Differenz zwischen MR3 und MR6 beträgt an dieser Stelle bereits 10 % und wird mit zunehmender Entfernung noch größer.

Der Grund für die schlechteren Befehlserkennungsraten liegt in den Abweichungen zwischen den gesprochenen Befehlen und den erhaltenen Hypothesen, die mit der verwendeten Konfiguration nicht toleriert werden. Sind diese Abweichungen im Nahbereich noch eher selten, treten sie mit zunehmender Mikrofon-Sprecher-Distanz immer häufiger auf. Beispiele für solche Abweichungen sind in Tabelle 5.2 angeführt. Für alle diese Beispiele gilt, dass bei aktiviertem fehlertoleranten Matching der Befehl erfolgreich erkannt worden wäre.

Tabelle 5.2: Beispiele für empfangene Hypothesen.

Befehl	Hypothese(n)
annehmen	arm nehmen
weiter	weiters, Weitzer
start	Stars
stop	top, Stoff
kopieren	Kopierer
Datei speichern	Dateien speichern

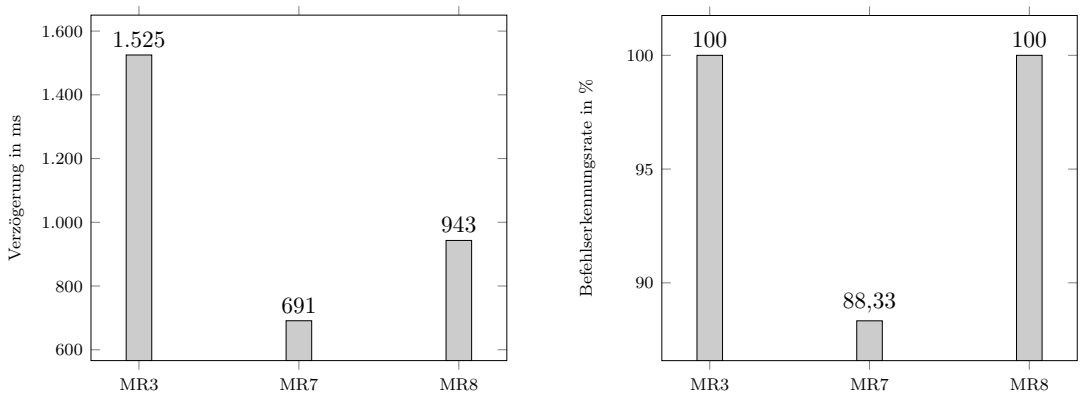
Obwohl mit dem phonetischen Matching gute Befehlserkennungsraten erzielt werden, zeigen die Ergebnisse auch eine Schwäche des Verfahrens auf. Diese macht sich besonders bei kurzen Befehlen bemerkbar. Die Tabelle 5.3 zeigt dazu zwei Beispiele.

Tabelle 5.3: Beispiele für Probleme beim phonetischen Matching.

Klartext	PR	Ähnlichkeit	LD
pause	18	50 %	1
hause	08		
nein	66	50 %	1
kein	46		

Die Hypothese „hause“, welche beim Pause-Befehl geliefert wurde, führte zu keinem Treffer, da die Ähnlichkeit zwischen der Hypothese und dem Erkennungsmuster kleiner als die Schwelle von 65 % war. Eine Reduktion der Schwelle auf 50 % kann allerdings keine Lösung für das Problem darstellen, da dann beispielsweise eine stark abweichende Hypothese wie „paar“ auch zu einem Treffer führen würde. Ein möglicher Lösungsansatz könnte aber die Berücksichtigung der Levenshtein-Distanz zwischen den Klartexten bringen, denn: Wäre das Matching auf Basis des Klartextes erfolgt, hätte die Hypothese zu einem Treffer geführt.

Während der Implementierung entstand der Eindruck, dass durch die Berücksichtigung von Zwischenergebnissen (Option `acceptInterimResults`) eine deutliche Reduktion der Verzögerungszeit erreicht werden kann. Um diese Annahme zu bestätigen, wurden weitere Messungen durchgeführt. Abbildung 5.7 zeigt eine Gegenüberstellung der Messergebnisse. Alle Messungen wurden mit einer Mikrofon-Sprecher-Distanz von 70 cm und ohne Hintergrundgeräusche durchgeführt. Als Vergleichswerte wurden wiederum die Ergebnisse aus MR3 herangezogen.



(a) Befehlserkennungsrate (b) Verzögerung

Abbildung 5.7: Optimierung der Verzögerungszeiten (MR5: keine Zwischenergebnisse, MR7: mit Zwischenergebnissen, MR8: mit Zwischenergebnissen und optimiertem Befehlssatz).

Bei der siebten Messreihe (MR7) wurde die Option `acceptInterimResults` aktiviert. Abgesehen davon blieb die Konfiguration des Sprachsteuerungsmoduls gegenüber MR3 aber unverändert. Im Vergleich zur ursprünglich verwendeten Konfiguration, zeigt sich eine Veränderung der Verzögerungszeit von 1525 ms auf 691 ms, was einer Reduktion um 55,61 % entspricht. Gleichzeitig sinkt aber auch die Befehlserkennungsrate auf 88,$\dot{3}$ %, was eine deutliche Verschlechterung bedeutet. Begründet liegt die schlechtere Befehlserkennungsrate in den vielen falsch-positiven Treffern, welche bei der Kombination von phonetischem Matching mit einer Levenshtein-Schwelle größer 0 sowie einer Ähnlichkeitsschwelle von 65 % und der Verwendung von Zwischenergebnissen entstehen. Folgend sind zwei Beispiele für beobachtete falsch-positive Treffer angeführt:

- Die Aufnahme „einfügen“ führte zum Zwischenergebnis „ei“, das den Ja-Befehl auslöste, da die phonetischen Repräsentationen (0 für „ei“ und 0 für das Erkennungsmuster „ja“) identisch sind.
- Die Aufnahme „rückgängig“ führt zum Zwischenergebnis „Brücke“, welches den Zurück-Befehl auslöst, da die Levenshtein-Distanz zwischen den phonetischen Repräsentationen (874 für „Brücke“ und 174 für das Erkennungsmuster „zurück“) kleiner gleich der definierten Schwelle von 1 ist. Ebenfalls liegt die Ähnlichkeit mit 66,$\dot{6}$ % oberhalb der definierten Schwelle von 65 %.

Während beim zweiten Beispiel durch eine Erhöhung der Ähnlichkeitsschwelle auf einen Wert größer 66,$\dot{6}$ % eine Verbesserung erzielt werden kann, ist diese Maßnahme beim ersten Beispiel wirkungslos. Eine Möglichkeit zur Vermeidung dieser Art von falsch-positiven Treffern, stellt die Optimierung des Befehlssatzes unter Verwendung des Attributes `matchinterim` dar. Für die achte Messreihe wurde für die als problematisch identifizierten Befehle (ja, zurück und stop) das Attribut mit dem Wert `true` versehen,

sodass kein Matching mit Zwischenergebnissen erfolgte. Als Resultat wurde wieder eine Befehlserkennungsrate von 100 % erreicht. Die Verzögerung stieg zwar auf 943 ms, das entspricht aber immer noch einer Reduktion von 38,16 % gegenüber dem Ursprungswert von 1525 ms.

Die Ergebnisse zeigen, dass unter der Verwendung von Zwischenergebnissen deutlich kürzere Antwortzeiten erreicht werden können. Die Verwendung von Zwischenergebnissen in Kombination mit dem fehlertoleranten phonetischen Matching ist aber nur dann sinnvoll, wenn der Befehlssatz entsprechend optimiert wurde. Nur so kann eine Häufung von falsch-positiven Treffern ausgeschlossen und eine hohe Befehlserkennungsrate erzielt werden.

Kapitel 6

Schlussbetrachtung

In diesem Kapitel werden die Erkenntnisse der Arbeit zusammengefasst und ein Fazit gezogen. Des Weiteren wird ein Ausblick auf zukünftige Entwicklungen gegeben.

6.1 Zusammenfassung und Fazit

Das Ziel dieser Arbeit war es, die Möglichkeiten für die Steuerung von browserbasierten Benutzerschnittstellen per Spracheingabe zu untersuchen und einen Prototypen für ein Sprachsteuerungsmodul umzusetzen. Hierfür wurden zunächst die nötigen Grundlagen erarbeitet, wobei insbesondere die Themenbereiche automatische Spracherkennung und Webtechnologien genauer betrachtet wurden.

Es zeigte sich, dass im Zuge der Entwicklung von HTML 5 mächtige APIs – vor allem die Web Audio API und die APIs des WebRTC Projekts – entstanden sind, die umfangreiche Möglichkeiten zur Audioaufnahme, -wiedergabe und -verarbeitung bieten. Mit PocketSphinx.js und der Web Speech API wurden dann zwei Lösungsansätze, die diese APIs nutzen und es ermöglichen eine automatische Spracherkennung – ohne die Verwendung von Browser-Plug-ins oder der Implementierung serverseitiger Komponenten – zu realisieren, genauer untersucht. Basierend auf den in einer Anforderungsanalyse erarbeiteten Anforderungen und einer sorgfältigen Abwägung der Vor- und Nachteile der einzelnen Lösungsansätze, wurde die Entscheidung getroffen, den Prototypen auf Basis der Web Speech API zu implementieren. Die Entscheidung fiel trotz mangelnder Browserunterstützung und derzeit noch unvollständiger Umsetzung der API (u. a. fehlende Grammatiken und nicht gegebene Offlinefähigkeit) auf die Web Speech API, da diese eine allgemein hohe Erkennungsleistung ohne aufwendiges Trainieren von Sprach- und Akustikmodellen versprach.

Um mittels der Web Speech API eine Sprachsteuerung zu realisieren, war es aufgrund der fehlenden Grammatiken zunächst nötig, eine Möglichkeit zu schaffen, Wortkombinationen, die einen Befehl repräsentieren, zu definieren. Zu diesem Zweck wurde

ein auf XML basierendes Dateiformat spezifiziert. Die größte Herausforderung stellte jedoch die Entwicklung eines Matchingverfahrens dar. Mit diesem sollte sichergestellt werden, dass die vom Spracherkennungsservice erhaltenen Hypothesen möglichst zuverlässig den entsprechenden Befehlen zugeordnet werden. Schließlich wurde ein konfigurierbares fehlertolerantes Verfahren umgesetzt, welches die Ähnlichkeit zwischen Hypothesen und Erkennungsmuster sowohl qualitativ als auch quantitativ bewertet, wobei dafür die phonetischen Algorithmen „Metaphone“und „Kölner Phonetik“beziehungsweise der Levenshtein-Algorithmus verwendet wurden.

Für die Umsetzung des Prototyps war eine intensive Auseinandersetzung mit der Programmiersprache JavaScript erforderlich. Da beim Entwurf der Software auf einen sauberen objektorientierten und modularen Ansatz Wert gelegt wurde, galt es sich unter anderem mit JavaScript-Entwurfsmustern für Modularisierung und objektorientierte Programmierung vertraut zu machen. Im Zuge der Implementierung war außerdem die Auseinandersetzung mit verschiedenen JavaScript-Entwicklungswerkzeugen – beispielsweise für die Durchführung automatisierter Unit Tests oder die automatische Erzeugung einer Softwaredokumentation – erforderlich. Neben dem eigentlichen Sprachsteuerungsmodul wurde zusätzlich eine Demonstrationsanwendung umgesetzt, welche die Funktionalität des Moduls demonstriert und gleichzeitig Entwicklern und Entwicklerinnen ein Beispiel für die Verwendung des Moduls bietet.

In einer abschließenden Evaluierung wurde untersucht, welche Erkennungsleistungen mit dem Prototyp unter verschiedenen Bedingungen erzielt werden können. Die Ergebnisse der Messungen zeigten, dass die Qualität des verwendeten Mikrofons von entscheidender Bedeutung ist. Das gilt insbesondere bei größer werdenden Mikrofon-Sprecher-Distanzen. Die in Abschnitt 2.1.3 erwähnten Probleme bei der DSR konnten bestätigt werden. Ebenfalls konnte gezeigt werden, dass die Erkennungsleistung des Gesamtsystems durch das fehlertolerante Matching auf phonetischer Ebene signifikant verbessert wird. Gleichzeitig konnte aber auch eine Schwäche des Verfahrens aufgezeigt werden. Insgesamt ist die Erkennungsleistung aber als zufriedenstellend zu beurteilen. Das Ziel der Arbeit konnte mit dem gewählten Lösungsansatz erfüllt werden.

6.2 Ausblick

Wie die vorliegende Arbeit zeigt, ist die Web Speech API – trotz derzeit noch vorhandener Schwächen – eine relativ robuste und verhältnismäßig einfach zu verwendende Möglichkeit, um sprachbasierte Anwendungen im Web-Kontext umzusetzen. Mit der weiteren Standardisierung der API ist es nur noch eine Frage der Zeit, bis sprachbasierte Anwendungen auch im Web Einzug halten werden. Zukünftig empfiehlt es sich, die weitere Entwicklung der API jedenfalls genau zu beobachten. Etwaige Erweiterungen oder Änderungen könnten eine Anpassung des Prototyps erfordern, die im Zuge einer

weiteren Studienarbeit durchgeführt werden könnte. Eine Erweiterung des phonetischen Matchings, sodass neben Deutsch und Englisch noch weitere Sprachen berücksichtigt werden, oder eine weitere Verfeinerung des Matching-Verfahrens bieten ebenfalls Ansatzpunkte für weitere Arbeiten.

Der nächste logische Schritt stellt die Integration des Prototypen in eine Anwendung dar. Parallel zur Arbeit wurde der Prototyp bereits in eine frühe Version der LUI-Portierung eingebaut. Ebenfalls konnte er versuchsweise in den Pflegeroboter Hobbit [75] integriert werden. Erste oberflächliche Tests zeigten, dass die erzielte Erkennungsleistung gleichwertig zu den bisher verwendeten Sprachsteuerungskomponenten zu sein scheint. Für ein aussagekräftiges Ergebnis sind aber weitere Tests erforderlich. Die vollständige Integration in eine Anwendung (softwaretechnische Umsetzung, Aufbau des Befehlssatzes, Test etc.) liefert genug Stoff für eine weitere Studienarbeit.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AAL	Ambient Assisted Living
AAT	Angewandte Assistierende Technologien
API	Application Programming Interface
ARPA	Advanced Research Projects Agency
ASR	Automated Speech Recognition
CD-DNN-HMM	Context-Dependent Deep Neural Network Hidden Markov Models
CMU	Carnegie Mellon University
CMUCLMTK	CMU-Cambridge Language Modeling Toolkit
CSS	Cascading Style Sheets
DOM	Document Object Model
DSR	Distant Speech Recognition
DTW	Dynamic Time Warping
ECMA	European Computer Manufacturers Association
FA	Funktionale Anforderung
FLAC	Free Lossless Audio Codec
GMM	Gaussian Mixture Model
GPL	General Public License
HMM	Hidden Markov Model
HTK	Hidden Markov Model Toolkit
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ID	Identifier
IIFE	Immediately Invoked Function Expression
IVR	Interactive Voice Response
JSON	JavaScript Object Notation
LD	Levenshtein-Distanz
LLVM	Low Level Virtual Machine
LUI	Local User Interface
MITLM	Massachusetts Institute of Technology Language Modeling

NFA ..... Nicht-funktionale Anforderung
OS ..... Operating System
PR ..... Phonetische Repräsentation
RFC ..... Request for Comments)
SAPI ..... Speech API
SDK ..... Software Development Kit
SUR ..... Speech Understanding Research
TCP ..... Transmission Control Protocol
TSL ..... Transport Layer Security
TTS ..... Text-to-Speech
TV ..... Television
UML ..... Unified Modeling Language
URL ..... Uniform Resource Locator
W3C ..... World Wide Web Consortium
WA ..... Word Accuracy
WAMI ..... Web Accessible Multimodal Interfaces
WebRTC ..... Web Real Time Communications
WER ..... Word Error Rate
WWW ..... World Wide Web
XML ..... Extensible Markup Language

Literaturverzeichnis

Wissenschaftliche Literatur

[1] Schukat-Talamazzini E. G.: *Automatische Spracherkennung – Grundlagen, statistische Modelle und effiziente Algorithmen.* Vieweg Verlag, Braunschweig, 1995.

[2] Renals S., Hain T.: *Speech Recognition.* In: Clark A., Fox C., Lappin S. (Hrsg.): *The Handbook of Computational Linguistics and Natural Language Processing.* 1. Auflage, Wiley-Blackwell, Chichester, UK, 2010.

[3] Pfiester B., Kaufmann T.: *Sprachverarbeitung – Grundlagen und Methoden der Sprachsynthese und Spracherkennung.* Springer Verlag, Berlin Heidelberg, 2008.

[4] Bourlard H., Morgan N.: *Continuous Speech Recognition: A Hybrid Approach.* Kluwer Academic Publishers, Norwell, MA, 1993.

[5] Dong Y., Deng L., Seide F.: *Large vocabulary speech recognition using deep tensor neural networks.* In: Proceedings of Interspeech 2012, Portland, OR, S. 6-9, 2012.

[6] Hinton G., Deng L., Yu D., Dahl G., Mohamed A., Jaitly N., Senior A., Vanhoucke V., Nguyen P., Sainath T., Kingsbury B.: *Deep neural networks for acoustic modeling in speech recognition.* In: IEEE Signal Processing Magazine, Volume 29, S. 82-97, 2012.

[7] Chelba C., Peng X., Pereira F., Richardson T.: *Large Scale Distributed Acoustic Modeling With Back-Off N-Grams.* In: IEEE Transactions on Audio, Speech and Language Processing, Vol. 21, Nr. 6, S. 1158-1169, 2013.

[8] Euler S.: *Grundkurs Spracherkennung.* 1. Auflage, Vieweg Verlag, Wiesbaden, 2006.

[9] Wölfel M., McDonough J.: *Distant Speech Recognition.* 1. Auflage, Wiley, Chichester, UK, 2009.

[10] Kumatani K., McDonough J., Bhiksha R.: *Microphone Array Processing for Distant Speech Recognition.* In: IEEE Signal Processing Magazine, Vol. 29, Nr. 6, S. 127-140, 2012.

[11] Chen J., Benesty J., Huang Y. A., Doclo S.: *New insights into the noise reduction Wiener filter.* In: IEEE Transactions on Audio, Speech and Language Processing, Vol. 14, Nr. 4, S. 1218-1234, 2006.

[12] Droppo J., Acero A.: *Environmental Robustness.* In: Benesty J., Sondhi M. M., Huang Y. (Hrsg.): *Springer Handbook of Speech Processing.* Springer Verlag, Berlin Heidelberg, 2008.

[13] Estellers V., Thiran J-P.: *Multi-pose lipreading and audio-visual speech recognition.* In: EURASIP Journal on Advances in Signal Processing, Vol. 2012, S. 1-23, 2012.

[14] Lea W. A.: *The Value of Speech Recognition Systems.* In: Waibel A., Lee K. (Hrsg.): *Readings in Speech Recognition.* Morgan Kaufmann, San Mateo, CA, 1990.

[15] Fink G.A.: *Mustererkennung mit Markov-Modellen: Theorie – Praxis – Anwendungsgebiete.* 1. Auflage, B. G. Teubner Verlag, Wiesbaden, 2003.

[16] Münz S., Gull C.: *HTML 5 Handbuch.* 9. Auflage, Franzis Verlag, Haar bei München, 2013.

[17] Smus B.: *Web Audio API – Advanced Sound for Games and Interactive Apps.* 1. Auflage, O'Reilly, Sebastopol, CA, 2013.

[18] Huggins-Daines D., Kumar M., Chan A., Black A. W., Ravishankar M., Rudnicky A. I.: *Pocketsphinx: A Free, Real-Time Continuous Speech Recognition System for Hand-Held Devices.* In: IEEE Acoustics, Speech and Signal Processing, Vol. 1, S. 185-188, 2006.

[19] Balzert H.: *Lehrbuch der Softwaretechnik – Basiskonzepte und Requirements Engineering.* 3. Auflage, Spektrum Akademischer Verlag, Heidelberg, 2009.

[20] Schatten A., Demolsky M., Winkler D., Biffl S., Gostischa-Franta E., Östreicher T.: *Best Practice Software-Engineering.* Spektrum Akademischer Verlag, Heidelberg, 2010.

[21] Schalkwyk J., Beeferman D., Beaufays F., Byrne B.: *Your word is my command: Google search by voice: A case study.* In: Neustein A. (Hrsg.): *Advances in Speech Recognition – Mobile Environments, Call Centers and Clinics.* Springer, New York, US, 2010.

[22] Osmani A.: *Learning JavaScript Design Patterns.* O'Reilly, Sebastopol, CA, 2014.

[23] Philips L.: *Hanging on the Metaphone.* In: Computer Language 7, Nr. 12, S. 39-43, 1990.

[24] Postel H. J.: *Die Kölner Phonetik. Ein Verfahren zur Identifizierung von Personennamen auf der Grundlage der Gestaltanalyse.* In: IBM Nachrichten, Nr. 19, S. 925-931, 1969.

[25] Levenshtein V. I.: *Binary codes capable of correcting deletions, insertions, and reversals.* In: Soviet Physics Doklady, Vol. 10, No. 8, S. 707-710, 1966.

[26] Souders S.: *High performance web sites : essential knowledge for frontend engineers.* O'Reilly, Sebastopol, CA, 2007.

[27] Resig J., Bibeault B.: *Secrets of the JavaScript Ninja.* Manning Publictions, Shelter Island, NY, 2013.

Online-Quellen

[28] Chelba C., Bikel D., Shugrina M., Nguyen P., Kumar S.: *Large Scale Language Modeling in Automatic Speech Recognition.* 2012. URL: `http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/40491.pdf` (besucht am 04.09.2014).

[29] Rabiner L. R., Juang B. H.: *Automatic Speech Recognition – A Brief History of the Technology Development.* 2004. URL: `http://www.ece.ucsb.edu/Faculty/Rabiner/ece259/Reprints/354_LALI-ASRHistory-final-10-8.pdf` (besucht am 25.08.2014).

[30] Heriot-Watt University: *Voice Report 2013.* 2013. URL: `http://www.macs.hw.ac.uk/texturelab/files/publications/Voice_Report_2013.pdf` (besucht am 05.09.2014).

[31] Woods P.: *Bestätigt: Siri nutzt Nuance-Technik.* In: Macwelt, 2013. URL: `http://www.macwelt.de/news/Bestaetigt-Siri-nutzt-Nuance-Technik-7934468.html` (besucht am 05.09.2014).

[32] Nuance: *Dragon Software Developer Kits.* 2014. URL: `http://www.nuance.com/for-developers/dragon/index.htm` (besucht am 05.09.2014).

[33] Nuance Mobile: *Integrate speech into your app.* 2014. URL: `http://dragonmobile.nuancemobiledeveloper.com/public/index.php?task=prodDev` (besucht am 05.09.2014).

[34] Microsoft Developer Network: *Speech API Overview (SAPI 5.4)*. 2014. URL: `http://msdn.microsoft.com/en-us/library/ee125077(v=vs.85).aspx` (besucht am 05.09.2014).

[35] Microsoft Developer Network: *Microsoft Speech Platform SDK 11 Documentation*. 2014. URL: `http://msdn.microsoft.com/en-us/library/dd266409(v=office.14).aspx` (besucht am 05.09.2014).

[36] CMU Sphinx: *Open Source Toolkit For Speech Recognition*. 2014. URL: `http://cmusphinx.sourceforge.net/` (besucht am 05.09.2014).

[37] Julius: *Open-Source Large Vocabulary CSR Engine Julius*. 2014. URL: `http://julius.sourceforge.jp/en_index.php` (besucht am 05.09.2014).

[38] htk3: *Documentation for HTK* 2014. URL: `http://htk.eng.cam.ac.uk/docs/docs.shtml` (besucht am 05.09.2014).

[39] Bos B., Çelik T., Hickson I., Wium Lie H.: *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*. 2001. URL: `http://www.w3.org/TR/CSS2` (besucht am 15.08.2014).

[40] SimilarTech: *Technologies Usage Statistics JavaScript*. 2014. URL: `https://www.similartech.com/categories/javascript` (besucht am 20.08.2014).

[41] jQuery Foundation: *jQuery – write less, do more*. 2014. URL: `http://jquery.com` (besucht am 20.08.2014).

[42] Bray T., Paoli J., Sperberg-McQueen C. M., Maler E., Yergeau F.: *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. 2008. URL: `http://www.w3.org/TR/2008/REC-xml-20081126` (besucht am 21.08.2014).

[43] Beiersmann S.: *Sicherheitslücke in Flash Player erlaubt Cookie-Diebstahl*. In: ZDNet, 2014. URL: `http://www.zdnet.de/88198295/sicherheitsluecke-flash-player-erlaubt-cookie-diebstahl` (besucht am 26.09.2014).

[44] Beiersmann S.: *Adobe schließt sechs kritische Sicherheitslücken in Flash Player*. In: ZDNet, 2014. URL: `http://www.zdnet.de/88195467/adobe-schliesst-sechs-kritische-sicherheitsluecken-flash-player` (besucht am 26.09.2014).

[45] Adenot P., Wilson C., Rogers C.: *Web Audio API – W3C Working Draft 10 October 2013*. 2013. URL: `http://www.w3.org/TR/webaudio` (besucht am 22.06.2014).

[46] Bergkvist A., Burnett D., Jennings C., Narayanan A.: *WebRTC 1.0: Real-time Communication Between Browsers – W3C Editor's Draft 01 July 2014*. 2014. URL: `http://dev.w3.org/2011/webrtc/editor/webrtc.html` (besucht am 29.09.2014).

[47] Burnett D., Bergkvist A., Jennings C., Narayanan A.: *Media Capture and Streams – W3C Working Draft 03 September 2013*. 2013. URL: `http://www.w3.org/TR/mediacapture-streams` (besucht am 29.09.2014).

[48] Chevalier S.: *PocketSphinx.js – Speech Recognition in JavaScript*. 2014. URL: `https://github.com/syl22-00/pocketsphinx.js` (besucht am 29.09.2014).

[49] Can I use: *Compatibility table for support of Web Audio API in desktop and mobile browsers*. 2014. URL: `http://caniuse.com/#feat=audio-api` (besucht am 07.10.2014).

[50] Can I use: *Compatibility table for support of getUserMedia/Stream API in desktop and mobile browsers*. 2014. URL: `http://caniuse.com/#feat=stream` (besucht am 22.06.2014).

[51] VoxForge: *VoxForge*. 2014. URL: `http://www.voxforge.org` (besucht am 30.09.2014).

[52] CMUSphinx: *Building Language Model Tutorial*. 2013. URL: `http://cmusphinx.sourceforge.net/wiki/tutoriallm` (besucht am 30.09.2014).

[53] WAMI: *WAMI – A Java-script API for speech recognition*. 2009. URL: `https://code.google.com/p/wami` (besucht am 01.10.2014).

[54] Online Speech Recognition API: *Online Speech Recognition API – Add speech recognition to your website with javascript and flash*. 2010. URL: `http://speechapi.com/apis/javascript` (besucht am 01.10.2014).

[55] Bodell M., Bringert B., Brown R., Burnett D. C., Dahl D., Druta D., Ehlen P., Hemphill C., Johnston M., Pettay O., Sampath S., Schröder M., Shires G., Tumuluri R., Young M.: *HTML Speech Incubator Group Final Report – W3C Incubator Group Report 06 December 2011*. 2011. URL: `http://www.w3.org/2005/Incubator/htmlspeech/XGR-htmlspeech-20111206/` (besucht am 10.09.2014).

[56] Shires G., Wennborg H.: *Web Speech API Specification – 19 October 2012*. 2012. URL: `https://dvcs.w3.org/hg/speech-api/raw-file/tip/speechapi.html` (besucht am 10.09.2014).

[57] Can I use: *Compatibility table for support of Web Speech API in desktop and mobile browsers*. 2014. URL: `http://caniuse.com/#feat=web-speech` (besucht am 22.06.2014).

[58] Kamat S., Nyman R. (Hrsg.): *Enabling Voice Input into the Open Web and Firefox OS*. 2014. URL: `https://hacks.mozilla.org/2014/09/enabling-voice-input-into-the-open-web-and-firefox-os/` (besucht am 11.09.2014).

[59] Postinett A.: *So will Google zum Sprachgenie werden*. In: Handelsblatt, 2011. URL: `http://www.handelsblatt.com/technologie/forschung-medizin/forschung-innovation/spracherkennung-so-will-google-zum-sprachgenie-werden/5798652.html` (besucht am 13.09.2014).

[60] Lange P., Suendermann-Oeft D.: *Tuning Sphinx to Outperform Google's Speech Recognition API*. 2014. URL: `http://suendermann.com/su/pdf/essv2014.pdf` (besucht am 07.10.2014).

[61] Phillips A., Davis M.: *RFC 5646 – Tags for Identifying Languages*. 2009. URL: `http://tools.ietf.org/html/rfc5646` (besucht am 14.10.2014).

[62] Hunt A., McGlashan S.: *Speech Recognition Grammar Specification Version 1.0 – W3C Recommendation 16 March 2004*. 2004. URL: `http://www.w3.org/TR/speech-grammar` (besucht am 14.10.2014).

[63] RequireJS: *A JavaScript Module Loader*. 2014. URL: `http://requirejs.org` (besucht am 14.10.2014).

[64] $script.js: *Asyncronous JavaScript loader and dependency manager*. 2014. URL: `https://github.com/ded/script.js` (besucht am 14.10.2014).

[65] YUI Compressor: *YUI Compressor Documentation*. 2013. URL: `http://yui.github.io/yuicompressor` (besucht am 18.10.2014).

[66] jQuery UI: *jQuery User Interface*. 2014. URL: `http://jqueryui.com` (besucht am 15.09.2014).

[67] Mongoose: *Easy to use web server*. 2014. URL: `https://code.google.com/p/mongoose` (besucht am 16.09.2014).

[68] OpenSSL: *Cryptography and SSL/TSL Toolkit*. 2014. URL: `http://qunitjs.com` (besucht am 25.10.2014).

[69] QUnit: *A JavaScript Unit Testing framework*. 2014. URL: `http://qunitjs.com` (besucht am 25.10.2014).

[70] BLANKET.JS: *Seamless javascript code coverage*. 2014. URL: `http://blanketjs.org` (besucht am 25.10.2014).

[71] JSDoc3: *@use JSDoc*. 2014. URL: `http://usejsdoc.org/about-getting-started.html` (besucht am 24.10.2014).

[72] Audacity: *Audacity ist freie, plattformunabhängige Open-Source-Software für die Aufnahme und Bearbeitung von Audio*. 2014. URL: `http://audacity.sourceforge.net/?lang=de` (besucht am 03.11.2014).

[73] Andrea Electronics: *Pure Audio® USB-SA with Free Array-2SA*. 2014. URL: `http://shop.andreaelectronics.com/pure-audio-usb-sa-with-free-array-2sa` (besucht am 03.11.2014).

[74] Speedlink: *PURE Desktop Voice Microphone*. 2014. URL: `http://www.speedlink.com/?p=2&cat=184&pid=24283&paus=1` (besucht am 03.11.2014).

[75] Hobbit: *Hobbit - The Mutual Care Robot*. 2013. URL: `http://hobbit.acin.tuwien.ac.at` (besucht am 17.11.2014).