Informatik
Hauptcampus

H O C H
S C H U L E
T R I E R

Personal and Secure Voice Assistant

Ein datenschutkonformer Helfer

Denis Wagner

Bachelor-Abschlussarbeit

Betreuer: Prof. Georg Rock, Christoph Zinnen

Trier, 22.08.2019

Kurzfassung

In dieser Dokumentation wird die Entwicklung eines datenschutzkonformen Sprachassistenten beschrieben. Die angenehmen APIs für Spracherkennung und Sprachsynthetisierung von Google und co. können daher aufgrund ihrer datensammelnden Eigenschaften für diese Arbeit nicht verwendet werden. Somit stellt die datenschutzkonforme Verarbeitung der Sprachdaten eine besondere Herausforderung dar. Jede Berechnung und besonders die Spracherkennung müssen lokal und kontrollierbar erfolgen, um diese Datenschutzkonformität zu erreichen. Die Grundlage dieses Assistenten soll *Mozilla DeepSpeech* bilden, einem künstlichen neuronalen Netzwerk, welches zur Spracherkennung lokal verwendet wird. Zu Beginn der Arbeit wird DeepSpeech mit dem offenen, deutschen Sprachdatensatz des *Mozilla Common Voice* Projekts trainiert und ein allgemeiner Entwurf eines Sprachassistenten erstellt. Dieser beinhaltet die Aufteilung des Programms in einen Speech-, Conversation- und Skill-Manager. Der Speech-Manager ist das Sprachzentrum des Assistenten, welcher für das Aufnehmen der Sprache in Worten mittels DeepSpeech und der Sprachsynthetisierung mit Hilfe von *eSpeak*. Der Conversation-Manager ist für die Erkennung und die Verwaltung des Außerdem ist er die Schnittstelle für die programmweite Kommunikation zwischen den einzelnen Modulen. Der Natural-Language-Understanding-Manager ist wiederum für das konkrete Verständnis der aufgenommenen Sprache verantwortlich. Dieser beinhaltet zusätzlich einen Plugin-Manager, der die Fähigkeiten - hier *Skills* genannt - verwaltet und entscheidet, welcher Skill für das Gesagte in Frage kommt. Der Sprachassistent ist aufgrund des modularen Aufbaus und des Plugin-Systems stark modifizier- und erweiterbar.

Inhaltsverzeichnis

1

Einleitung und Problemstellung

Sprache ist eine für uns sehr natürliche Art der Kommunikation. Und für die meisten sicherlich auch die angenehmste. Diese Idee verfolgen auch sogenannte *Sprachassistenten* (engl. *voice assistants*), die zur Eingabe natürliche Sprache in Form von Ton anstelle von Text aufnehmen. Diese Systeme bieten neben der natürlichen Art, mit Sprache ein Computersystem zu steuern, einige Vorteile. Es können andere vernetzte Geräte angesteuert und alltägliche Dinge erledigt werden, ohne auch nur an eine Tastatur gehen zu müssen. Heutzutage sind Sprachassistenten auf nahezu jedem Gerät zu finden und ihre Popularität ist nicht zu übersehen. Laut einer Studie von Microsoft benutzen rund 72 % aller Befragten mindestens einen Sprachassistenten, wobei Apples *Siri*, Googles *Google Assistant* und Amazons *Alexa* von allen die populärsten sind [CO19]. Eine Umfrage mit etwa 280 Teilnehmern, die im Zuge dieses Projektes durchgeführt wurde, ergab jedoch deutlich negativere Ergebnisse. Nur etwas mehr als 15 % der Befragten würden ihre Nutzungsmenge mindestens mit *Häufig* bezeichnen. Die Umfrage lieferte jedoch auch Aufschluss über eine mögliche Ursache dieses – in Wirklichkeit – geringen Interesses gegenüber Sprachassistenten. Sie zeigt, dass der Datenschutz eine ausschlaggebende Rolle spielt, da über 81 % der Befragten diesen für sich mindestens als *Wichtig* einstufen. Die Einschätzung des Datenschutzes im Bezug auf konkrete Sprachassistenten wurde als äußerst schlecht empfunden, was Abbildung 1.1 zu erkennen ist. Es ist außerdem zu erkennen, dass die Bewertung des Datenschutzes mit sinkender Nutzungshäufigkeit weiter abnimmt. Es herrscht also ein großes Misstrauen in solche Technologien und Geräte. Besonders interessant ist es, dass auch regelmäßige Nutzer die datenschutzrechtlichen Probleme erkannt haben, sich dennoch für eine so aktive Nutzung entscheiden, trotz der Überzeugung, dass Privatsphäre und Datenschutz wichtig sei. Statt einer konkreten Bewertung haben über 25 % der Befragten die Ausweichoption „Weiß ich nicht“ angeben. Dies weist ebenfalls auf die niedrige Transparenz von populären Sprachassistenten hin. Es wird nicht klar kommuniziert, welche Daten wie verarbeitet werden und wofür. Der Benutzer besitzt kaum bis gar keine Kontrolle über die Abläufe.

Es wurde bestätigt, dass der Sprachassistent Alexa Sprachdaten von Nutzern an Amazon übermittelt und von Menschen auswerten lässt [MD19]. Dies stellt ein großes Problem für viele Nutzer dar, da häufig auch intime und private Momente mitgehört werden. Dass es sich hierbei um eine datenschutzrechtliche Katastro-

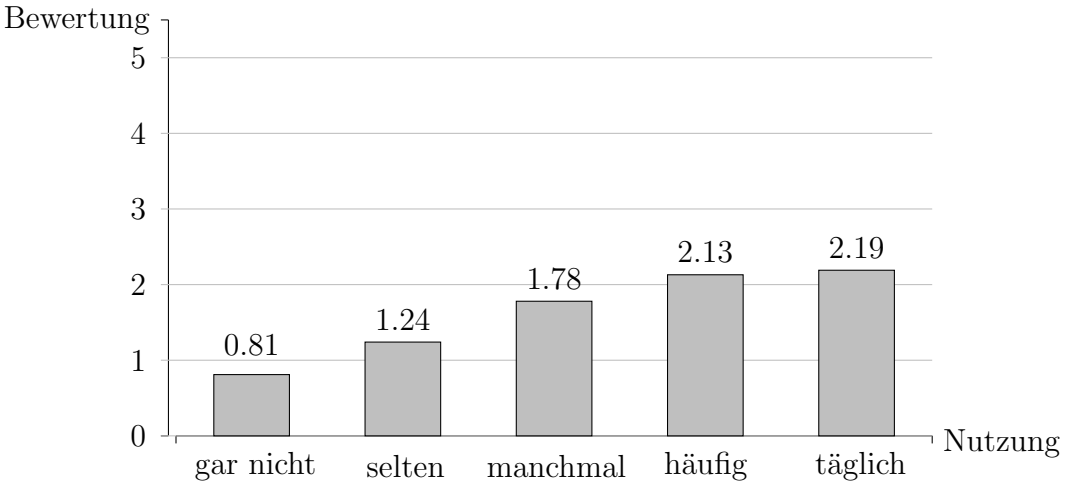

Abb. 1.1. Diagramm resultierend aus der Umfrage innerhalb dieses Projekts, das die durchschnittliche Bewertung des Datenschutzes von aktuellen Sprachassistenten in Relation zur Häufigkeit der Nutzung aller Befragten zeigt. Dabei steht 5 für einen guten Datenschutz und 0 für einen schlechten.

phe handelt, steht außer Frage. Der Grund für diese Vorgehensweise ist jedoch nachvollziehbar. Moderne Spracherkennung, wie sie auch in Sprachassistenten vorkommen, basieren in der Regel auf neuronalen Netzen. Diese müssen trainiert werden, wofür Unmengen an Sprachdaten und die dazugehörigen Transkripte vorhanden sein müssen. Diese Transkripte werden jedoch von Menschen angefertigt, um die Daten für das Training vorzubereiten. Ein Recyclen der Sprachdaten von Nutzern liegt hier aus wirtschaftlicher Sicht nahezu auf der Hand. Die Daten sind bereits vorhanden und können sogar mit Hilfe von personalisierter Werbung vermarktet werden, um Profit zu erlangen. Hier verkaufen die Nutzer im Grunde ihre persönlichen Daten, um den Sprachassistenten nutzen zu dürfen. Dies schädigt die Benutzer in ihrer Privatsphäre enorm und vor allem ihr Vertrauen in zukünftige Technologie. Dieses Geschäftsmodell hat jedoch rein gar nichts mit Sprachassistenten selbst zu tun.

In dieser Arbeit soll nun eine datenschutzkonforme Variante untersucht und entwickelt werden. Das resultierende System soll aufgenommene Sprachdaten grundsätzlich lokal verarbeiten, um die Privatsphäre des Endnutzers nicht zu gefährden. Nur extrahiertes Wissen könnte je nach Anwendung verbreitet werden, um eine gewisse Aktion durchzuführen, welche jedoch im Sinne des Nutzers liegt. Das Übermitteln von ganzen Sprachdaten ohne Einwilligung des Anwenders soll ausgeschlossen sein. Dazu ist ein System nötig, welches Sprache zu Text lokal umwandeln kann. Ein solches System stellt *Mozilla DeepSpeech* dar. Dabei handelt es sich um ein neuronales Netzwerk, welches mithilfe von Sprachdatensätzen auf eine bestimmte Sprache trainiert und zur Spracherkennung verwendet werden kann. Solche Datensätze müssen entsprechend groß sein, um etwaige Sätze von unterschiedlichen Personen (Alter, Tonlage, Akzent, Betonung etc.) korrekt verstehen zu können. Das Sammeln von Daten in dem Stil marktorientierter Sprachassisten-

ten kommt für dieses Projekt selbstverständlich nicht in Frage, somit wird ein frei zugänglicher Datensatz ohne Personenbezug benötigt. Ein solcher Sprachdatensatz wäre unter anderem *Mozilla Common Voice*, welcher zur Zeit der Bearbeitung dieses Projektes 325 Stunden verifizierte deutsche Sprachdaten und deren textuelle Transkription beinhaltete. Mit diesen Bestandteilen könnte die Verarbeitung der Sprachdaten nun vollständig lokal ablaufen und einen datenschutzkonformen, dezentralen Betrieb gewährleisten, bei dem die Transparenz und Kontrolle für den Benutzer oberste Priorität hat. Die theoretische und praktische Umsetzung sowie weitere Problematiken werden im Laufe dieser Arbeit elaboriert und die Ergebnisse vorgestellt.

2

Entwurf

Ein Sprachassistent besteht aus einigen interdependenten Teilen, welche allerdings grob in folgende Überbegriffe zusammengefasst werden können: die auditive Spracherkennung, die semantische Sprachverarbeitung und die auditive Sprachsynthetisierung. Jeder dieser Teilsysteme beinhaltet seine eigene Komplexität, jedoch konzentriert sich die vorliegende Arbeit hauptsächlich auf den Teil der auditiven Sprachaufnahme und ihrer semantischen Verarbeitung.

2.1 Software-Architektur

Sprachassistenten sind sehr ähnlich zu textuellen Chatbots, allerdings mit dem Unterschied, auf Audiosignale zu reagieren und auf gleicher Ebene zu antworten. Chatbots sind weit verbreitet und es existieren einige Frameworks, wobei die meisten eine bestimmte Architektur, wie in Abbildung 2.1 zu sehen, aufweisen [Jan17]. In diesem Diagramm sind bereits alle der oben genannten Bestandteile zu

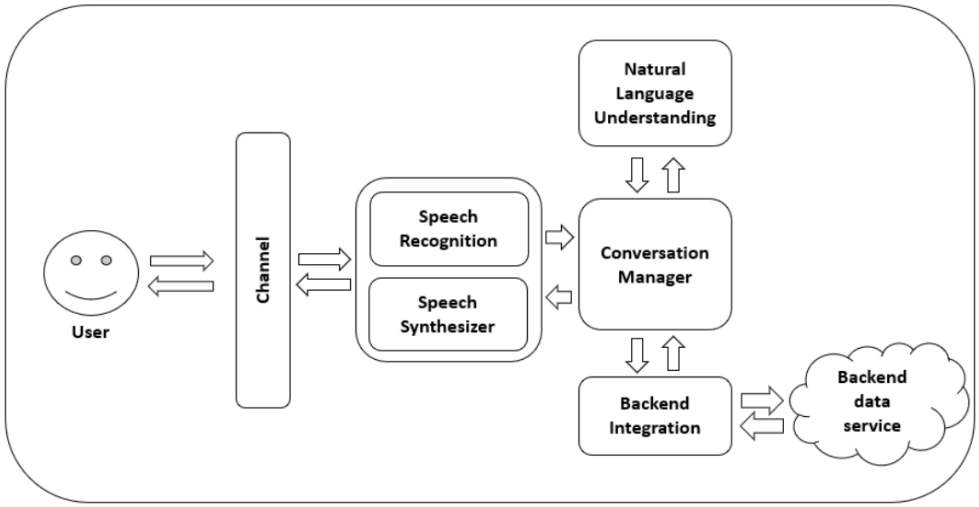

Abb. 2.1. Diagramm eines allgemeinen Chatbots [Jan17]

erkennen. Es existiert ein Sprachsystem, welches für die Spracherkennung (in der

Abbildung engl. *Speech Recognition*) und die Sprachsynthetisierung (in der Abbildung engl. *Speech Synthesizer*) zuständig ist. Die semantische Sprachverarbeitung wird hier in zwei Module unterteilt. Das erste Modul verwaltet das Gespräch selbst (in der Abbildung engl. *Conversation Manager*) und sorgt dafür, dass Sätze und Äußerungen des Benutzers oder Systems an die richtigen Stellen geleitet werden. Es stellt im Grunde die Schnittstelle zwischen allen anderen Modulen des Systems dar. Das zweite Modul ist nun das Herzstück des Sprachassistenten, da dort die eigentliche Bedeutung des Gesagten interpretiert wird (in der Abbildung engl. *Natural Language Understanding*). Der Sprachassistent in diesem Projekt soll, wie erwartet, auditiv agieren und reagieren können. Die Spracherkennung wird durch ein System namens *Mozilla DeepSpeech* durchgeführt, einem neuronalen Netzwerk auf Basis von Googles *Tensorflow*, was in Kapitel 3 näher erläutert wird. Ein auf Ton basierender Sprachassistent muss möglichst *parallel* arbeiten, da Töne aus der Umgebung nicht auf die langwierige Verarbeitung des Systems, beispielsweise die Berechnung der Ausgabe von DeepSpeech, warten. Somit müssen Audioaufnahme und Sprachverarbeitung parallel ablaufen und einige Veränderungen an dem Ursprungsmodell vorgenommen werden. Das Modell aus Abbildung 2.1 ist allerdings recht statisch und vor allem *sequentiell*. Eine etwas flexiblere Lösung zeigt Abbildung 2.2. Die Grundidee hinter diesem Aufbau ist es, dass die drei Teilmodule

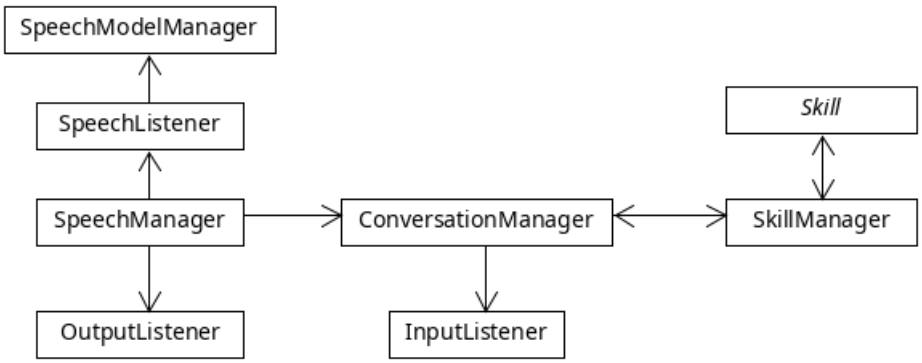

Abb. 2.2. UML-Diagramm in spezialisierter Form eines parallelisierten Sprachassistenten

Speech-Manager, *Conversation-Manager* und *Skill-Manager* entsprechend voneinander gekapselt werden können und ihre eigene Infrastruktur aufweisen und somit nicht sequentiell von anderen Modulen abhängig sind, was eine einfachere Parallelisierung ermöglicht. Ein Modul besitzt generell einen *Manager* und einen oder mehrere *Listener*. Der Manager stellt die Schnittstelle des jeweiligen Moduls dar und kann von anderen Managern angesprochen werden. Der Manager gilt auch als Verteiler innerhalb eines Moduls, um Aufgaben und Ergebnisse unter seinen Listenern zu organisieren und kann zu diesem Zweck auch alleinstehende *Unter-Manager* besitzen. Listener hingegen operieren nur modullokal und kümmern sich um die einzelnen Aufgaben. Jeder Listener repräsentiert einen Thread, damit Auf-

traggeber nicht warten müssen und ihre Aufträge in eine Warteschlange einreihen können.

In dem UML-Diagramm ist zu erkennen, dass das auditive Sprachsystem im `SpeechManager` zusammengefasst wird, dieser allerdings aus dem `SpeechListener` und dem `OutputListener` besteht. Bei Ersterem handelt es sich um das Spracherkennungsmodul, welches sich einer Helferklasse namens `SpeechModelManager` bedient. Die allgemeine Tonaufnahme und Audioverarbeitung übernimmt der `SpeechListener`. Die jedoch wichtigere Aufgabe obliegt dem `SpeechModelManager`. Dieser führt die eigentliche Spracherkennung durch, also die *Text-To-Speech*-Funktion und tut dies durch die Verwendung von Mozilla DeepSpeech. Der `OutputManager` ist nun für die Sprachsynthetisierung verantwortlich und nimmt den auszugebenden Text an, der von dem System aus gesagt werden soll. Das semantische Sprachsystem besteht auch in diesem abgewandelten Modell aus zwei Teilmodulen, dem `ConversationManager` und `SkillManager`. Ersterer ist äquivalent zu dem in Abbildung 2.1, jedoch benötigt er die Klasse `InputListener`, um parallel auf einkommende Texte reagieren zu können. Der SkillManager, zusammen mit seinen `Skills`, bildet aus Abbildung 2.1 das fehlende Teilmodul *Natural Language Understanding*. Die Klasse Skill ist ein Interface, welches für Plugins verwendet wird, die der *SkillManager* lädt und verwaltet. Skills repräsentieren die konversationellen Fähigkeiten des Sprachassistenten und sind für das eigentliche Führen der Konversation verantwortlich. Kommt eine neue Äußerung von dem `SpeechListener` bis zu dem `SkillManager`, entscheidet dieser, welcher *Skill* am besten für die semantische Verarbeitung geeignet ist. Das Plugin-System ist insofern sinnvoll, dass der Benutzer frei entscheiden können soll, welche Plugins installiert sein dürfen. Damit sollen keine Internet-gebundenen Teile aufgezwungen werden können und dem Benutzer die maximale Kontrolle über den Sprachassistenten verleihen. Dies bietet außerdem eine weitere Modularisierung des Systems, wodurch die Wartbarkeit deutlich gesteigert wird. Fähigkeiten des Sprachassistenten müssen also nicht tief in das System verflochten werden und sollen leicht änder- und austauschbar sein. Der allgemeine Aufbau dieses Modells unterscheidet sich somit in seiner Bedeutung nur leicht von dem Original, welchem es entlehnt ist.

Die einzelnen Teile des Systems sind interdependent und gleichzeitig lose miteinander gekoppelt. Durch den parallelen Entwurf kommunizieren parallel laufende Module über eine sogenannte `BlockingQueue`. Diese Datenstruktur ist wie eine normale *Queue*, jedoch Thread-sicher (engl. *thread save*). Wird versucht, ein Element von der Queue zu entfernen, blockiert der jeweilige Thread solange, bis die Queue nicht mehr leer ist. Somit können Threads ihre Ergebnisse in einer solchen Queue ablegen und andere parallel laufende Threads diese Ergebnisse abholen, sobald sie vorliegen. Ein Beispielverlauf könnte sein, dass der `SpeechListener` zu Beginn einen Ton aufnimmt. Dieser Ton stellt sich als Sprache heraus und ein relevanter Teil davon wird an den `SpeechModelManager` geleitet, der dann den entsprechenden Klartext liefern soll. Liegt dieser vor, holt der `InputListener` in einem parallel laufenden Thread dieses Ergebnis ab und wartet gegebenenfalls, bis der Benutzer zu Ende gesprochen hat. Ist dies der Fall, leitet er den gesprochenen Satz als Klartext an den `SkillManager` weiter. Dieser entscheidet nun,

welcher Skill am besten passt und beauftragt ihn mit der semantischen Analyse des vorliegenden Textes. Ist der Skill fertig, ruft er eine Funktion aus dem `SkillManager` auf, welche die Antwort – ebenfalls als Klartext – dann weiter an den `ConversationManager`, der den Text in einer `BlockingQueue` ablegt. Sobald dies der Fall ist, läuft der blockierte `OutputListener` weiter und synthetisiert den erhaltenen Text.

Der oben beschriebene Ablauf innerhalb des Sprachassistenten soll einen groben Aufschluss darüber geben, wie die einzelnen Teile dieser Softwarearchitektur miteinander interagieren. Der genaue Aufbau und die Funktionsweise einzelner Klassen und Module werden in den folgenden Kapiteln ausführlich erläutert.

2.2 Konfigurierbarkeit

Änderungen an bestehender Software durchzuführen wird in der Regel durch eine gewisse Konfigurierbarkeit erreicht. Dies bietet sowohl dem Endnutzer die Möglichkeit das System zu individualisieren als auch ein Zeitersparnis bei der Entwicklung, da ein Neukompilieren auch bei kleinen Änderungen – vor allem bei zweigeteilten Build-Verfahren, wie es bei C++ der Fall ist – einen erheblichen Zeitverlust bedeuten kann. Diese Änderungen und Konfigurierbarkeit würde sich hauptsächlich auf Konstanten innerhalb des Systems anwenden lassen.

Die Konfigurationen soll zunächst in Dateiform möglich sein. Dafür würde sich das moderne JSON aufgrund seiner einfachen Lesbarkeit anbieten. Auch besitzt dieses Format die Fähigkeit, komplexere Strukturen darzustellen, falls dies in Zukunft notwendig sein würde. Die Dateiform könnte sich zu einem späteren Zeitpunkt auch in eine GUI erweitern lassen, wobei die persistente Speicherung weiterhin in Dateiform durchgeführt wird. Es sollte so umgesetzt werden, dass das System selbst eine Standardkonfiguration besitzt und diese durch die Konfigurationsdatei überschrieben werden kann. Fehlende Einträge in der Datei würden in Standardwerten des Systems resultieren. Somit wäre nur die Angabe der für den Nutzer interessanten Einstellungen notwendig, was die Datei bei vielen möglichen Einträgen deutlich übersichtlicher gestaltet. Andersherum sollte auch das System überschriebene Einstellungen in die Datei schreiben können. Dies könnte der Fall sein, wenn beispielsweise über eine GUI die Einstellungen geändert wurden und nun persistiert werden sollen. Beispiele für die Art der Konfigurationsmöglichkeiten wären:

- Aktivierungswort des Sprachassistenten (siehe Kapitel 5)
- Sprachsynthetisierungsstimme des Systems (siehe Kapitel 4.3)
- GUI-Sprache
- GUI-Farbthema
- Audioeinstellungen

3

Mozilla DeepSpeech

Spracherkennung funktioniert heutzutage häufig mit neuronalen Netzen, wodurch eine erhebliche Menge an Sprachdaten und deren Transkripte notwendig sind. Diese werden in den gängigen Sprachassistenten von den Nutzern aufgezeichnet und transkribiert, um ihn verbessern zu können. Diese Vorgehensweise bietet sich an, da die Daten den Unternehmen kostenlos zur Verfügung stehen. Eine notwendige Konsequenz ist es nun, dass Mitarbeiter diese Sprachdaten überprüfen und damit tief in die Privatsphäre der Nutzer eindringen. Somit stellt die Wahl des Spracherkennungssystems eine der wichtigsten Lösungsansätze zur Erreichung ordnungsgemäßem Datenschutzes und akzeptabler Transparenz dar. In diesem Projekt wurde *Mozilla DeepSpech* gewählt [Moz19c]. Es ist eine Tensorflow-Implementierung des *Baidus Deep Speech Research Papers* und stellt ein frei trainierbares neuronales Netz zur Spracherkennung dar [HCC+14]. Dies erlaubt das Training durch einen beliebigen Sprachdatensatz und die Nutzung des *Mozilla-Common-Voice*-Projektes [Moz19a]. Dabei handelt es sich um eine Ansammlung an freien, freiwillig gespendeten und validierten Sprachdaten und Transkripten in mehreren Sprachen, darunter auch Deutsch und Englisch.

Im Folgenden wird kurz die Funktionalität von DeepSpeech erläutert sowie das Training des neuronalen Netzes und seine Vorbereitung elaboriert, die genutzt wird, um eine deutsche Spracherkennung zu ermöglichen.

3.1 Funktionalität

Laut [HCC+14] wird die Spracherkennung durch ein rekurrentes neuronales Netzwerk umgesetzt. Das Netz besteht aus fünf Schichten, wobei nur die letzte rekurrent ist. Bei der Architektur wurde besonders auf Parallelität geachtet, wodurch eine effiziente Berechnung mit mehreren Grafikkarten möglich ist. Das System nutzt Sprachdaten und deren Transkripte, um wichtige Eigenschaften menschlicher Sprache auf Audiodatenebene zu erlernen. Die Audiodaten werden dafür in Frequenzspektren bestimmter Zeitabschnitte unterteilt, für diese jeweils die Wahrscheinlichkeiten eines jeden Zeichens im verwendeten Alphabet ausgewertet werden [HCC+14]. Alle Zeitabschnitte einer Audiosequenz zusammen ergeben demnach eine Zeichensequenz, welche dann mit dem zugehörigen Transkript verglichen und ein Gradient erzeugt wird. Das Netz nutzt diesen nun, um mit Hilfe

von *Rückpropagieren* (engl. *Backpropagation*) die Gewichte in seinem Modell anzupassen [HCC+14]. Laut [HCC+14] wurde dieses System mit 7.380 Stunden an Sprachdaten trainiert und liefert bedeutend bessere Ergebnisse als vergleichbare Spracherkennungssysteme.

Aufgrund der zeichenweisen Berechnung kann es in seltenen Fällen dazu kommen, dass einzelne Zeichen rechtschreiblich falsch inferiert werden, an dieser Stelle jedoch phonetisch plausibel wären. Diese Fehler können durch ein *Language Model* (dt. etwa *Sprachmodell*) behoben werden. Diese Modelle geben mit Hilfe von *n-Grammen* eine Wahrscheinlichkeitsverteilung über Wortsequenzen an, wodurch aufgrund von n vielen vorgehenden Wörtern mit einer bestimmten Wahrscheinlichkeit auf das nächste Wort geschlossen werden kann. Diese Modelle müssen mit einer großen Menge an Beispielsätzen trainiert werden. Das Research Paper empfiehlt rund 220 Mio. Äußerungen, um erwartungsgemäße Schätzungen durchführen zu können.

3.2 Trainingsdaten

DeepSpeech kann beliebige Sprachdaten verarbeiten, wodurch eine Reihe von Datensätzen in Frage kommen [yno18]:

- Common Voice mit 325 Stunden
- The Spoken Wikipedia Corpus (SWC) mit 129 Stunden
- VoxForge mit 50 Stunden
- TUDA-Korpus mit 36 Stunden

Zusammen stünde eine potentielle Menge von 540 Stunden an Sprachdaten für das Training zur Verfügung.

3.2.1 Importierung der Daten

Die Sprachdatensätze besitzen unterschiedliche Formatierungen und müssen zunächst in eine vorgegebene Form gebracht werden, um für DeepSpeech nutzbar zu sein. Dabei müssen die Sprachdaten als WAV-Dateien vorliegen und durch drei CSV-Dateien namens `train.csv`, `dev.csv` und `test.csv` indiziert werden, wobei eine Audiodatei nur einmal in allen CSV-Dateien genannt werden darf. Sie enthalten kommasepariert den relativen oder absoluten Pfad zur Audiodatei, die Dateigröße in Byte und das entsprechende Transkript. Der TUDA-Korpus besitzt bereits die korrekte Formatierung und muss nicht weiter bearbeitet werden. Der Datensatz von Common Voice kann über ein von DeepSpeech bereitgestelltes Python-Skript, `import_cv2.py`, importiert und in die entsprechende Form gebracht werden. Im Kontrast dazu gibt es bei der Importierung des SWC einige Probleme. Die Ausgangsformatierung ist sehr unübersichtlich und nur schwer nachvollziehbar. Dennoch wäre eine Importierung durch ein Java-Programm von [Nat19] möglich. Da keine Binärdatei zur Verfügung steht, muss das Programm samt seiner Abhängigkeiten mit dem Build-System *Apache Maven* kompiliert werden. Das Auflösen der Abhängigkeiten ist allerdings nicht möglich, sodass das Kompilieren

immer fehlschlägt. Somit wird aus Zeitgründen auf diesen Datensatz verzichtet. Der VoxForge-Korpus ist leider zum Zeitpunkt der Bearbeitung unzugänglich und steht demnach nicht für das Training zur Verfügung. Resultierend bleiben nur noch Common Voice und TUDA als verwendbare Datensätze übrig und bilden damit eine Menge von 361 Stunden Sprachdaten. Dies ist verhältnismäßig wenig zu der genutzten Menge innerhalb Baidus Deep Speech Research Paper. Mozilla selbst empfiehlt rund 10.000 validierte Stunden an Audioaufnahmen, um ein produktiv einsetzbares Sprache-Zu-Text-System zu trainieren [Moz19b].

3.2.2 Erstellung des Language Models

Wie in Unterkapitel 3.1 erläutert wurde, benötigt DeepSpeech ein Language Model. Dieses muss zusätzlich mit einer hohen Anzahl von Äußerungen trainiert werden. Eine solche große Menge stellt beispielsweise eine Sammlung von etwa acht Millionen deutschen Äußerungen von [CB19] dar. Diese Menge ist zwar recht gering in Relation zu den verwendeten 220 Mio. Äußerungen in dem Research Paper, jedoch ausreichend für den ebenfalls kleinen Trainingsdatensatz. Die in der Sammlung enthaltenen Sätze sind jedoch in alter Rechtschreibung und somit nicht für ein modernes Spracherkennungssystem geeignet. Um dieses Problem zu lösen, bieten sich die Transkripte des Trainingsdatensatzes an. Beide Datensätze zusammen besitzen rund 33.000 Äußerungen, was für ein produktreifes Projekt deutlich zu wenig wäre, jedoch dem kleinen Testrahmen des vorliegenden Projektes genügt.

Zur Verarbeitung des Language Models wurde innerhalb der DeepSpeech-Implementierung von Mozilla auf das *KenLM-Toolkit* zurückgegriffen [Moz19c] [yno18][Hea11]. Somit wird dies auch für die Erstellung des eigenen Language Models für diesen Sprachdatensatz genutzt. Für die Generierung sieht KenLM die Angabe der n-Gramm-Länge vor. Laut [yno18] sind für einen kleinen Sprachdatensatz, wie er hier verwendet wird, n-Gramme der Länge Drei ausreichend. Längere Wortsequenzen würden zu Ungenauigkeiten führen, da es nicht genügend verwandte Beispielsätze gibt, die gewisse Wahrscheinlichkeiten untermauern. Viele n-Gramme könnten aufgrund eines einzigen Satzes entstanden sein und somit keinerlei tragfähige Aussagekraft über die tatsächliche Wahrscheinlichkeit des Folgewortes liefern. Die Generierung führt zu einer Datei namens `lm.binary`. Mozillas Implementierung sieht eine spezielle Weiterverarbeitung in ein *Trie* (auch *Präfixbaum* genannt) vor. Für dessen Erzeugung steht eine vorkompilierte Binärdatei von DeepSpeech zur Verfügung, welche die zuvor erstellte `lm.binary`-Datei benötigt. Beide erstellten Dateien, `lm.binary` und `trie`, müssen nun in einen speziellen Ordner unter `data/lm` abgelegt werden, sodass DeepSpeech sie finden kann. Damit wären die Vorbereitungen der Sprachdaten abgeschlossen und könnten nun für ein Training benutzt werden.

3.2.3 Robustheit

Die Sprachdaten von Common Voice wurden von unabhängigen Menschen in unterschiedlichsten Aufnahmesituationen erhoben. Dies ergibt eine recht hohe Ro-

bustheit gegenüber Hintergrundrauschen und generelle Varietät von Aufnahmequalitäten bei der resultierenden Spracherkennung. Dies gilt jedoch nicht für alle Datensätze, da viele von ihnen, wie der TUDA-Korpus auch, in einer kontrollierten Umgebung entstanden sind und somit eine gewisse Einheitlichkeit unter den Sprachdaten herrscht. Um dennoch eine robuste Spracherkennung mit ihnen zu erreichen, bietet sich das *Voice Corpus Tool* von Mozilla an [Moz19g]. Damit lässt sich ein Sprachdatenkorpus erstellen und verwalten, aber vor allem auch mit Effekten belegen, um beispielsweise Hintergrundrauschen nachzuahmen und die Tonqualität künstlich zu reduzieren. Dies könnte auch dazu genutzt werden, generell den Datenpool zu vergrößern und somit eine bessere Spracherkennung zu gewährleisten und sie nicht zuletzt auch resistenter gegen schlechte Aufnahmebedingungen zu machen.

3.3 Training

DeepSpeech ist stark auf das Training mit mehreren Grafikkarten ausgelegt, um eine geringere Trainingszeit zu erreichen. Um diesen Umstand nun auszunutzen, wurde dem Projekt für die Durchführung ein Computer mit zwei Grafikkarten des Typs *Nvidia GeForce RTX 2080 Ti* zur Verfügung gestellt. Dies ermöglichte das schnelle Justieren der Hyperparameter.

3.3.1 Einrichtung der Trainingsumgebung

Vor Beginn des Trainings muss DeepSpeech selbst zunächst eingerichtet werden. Dafür sind einige Dateien aus der entsprechenden GitHub-Repository notwendig [Moz19c]. Die erste Anweisung aus [Moz19c] für das Klonen der Repository auf den eigenen Rechner sieht die Installation von `git-lfs` vor, was für das Herunterladen besonders großer Dateien über Git genutzt wird. Diese kann jedoch ignoriert werden, da es sich bei den besagten großen Dateien um `trie` und `lm.binary` handelt. Sie wurden jedoch projektspezifisch für die deutsche Sprache erstellt und liegen bereits vor. Als nächstes müssen die Python-Abhängigkeiten von DeepSpeech über `python3 install -r requirements.txt` installiert werden. Es ist allerdings wichtig darauf zu achten, dass in der Datei `requirements.txt` eine Tensorflow-Version für die reine CPU-Berechnung angegeben ist. Um dies zu ändern, muss nach der Installation über `pip3 uninstall tensorflow` deinstalliert und über `pip3 install 'tensorflow-gpu==1.13.1'` installiert werden. Alternativ kann auch die Tensorflow-Version manuell in `requirements.txt` entsprechend geändert werden. Für die Berechnung auf einer Grafikkarte stehen nur diese von Nvidia zur Verfügung, da Tensorflow stark mit der CUDA-Schnittstelle von modernen Nvidia-Grafikkarten arbeitet, um eine besonders effiziente Berechnung ihrer neuronalen Netze durchzuführen. Dafür werden CUDA 10.0 und CuDNN v7.5 benötigt, welche durch den Paketmanager des Linux-Systems installiert oder über die Nvidia-Internetseite herunterladbar sind. Aus diesem Grund sind weitere CUDA-spezifische Abhängigkeiten notwendig. Alle verbleibenden Lauf-

zeitabhängigkeiten können auf der GitHub-Seite nachgelesen werden [Moz19c]. Zuletzt benötigt das Python-Skript `DeepSpeech.py`, welches das Training durchführt, das Python-Paket `ds_ctcdecoder` zur Dekodierung der Textausgaben des akustischen Modells von DeepSpeech [Moz19c]. Dies geschieht durch das Kommando `pip3 install $(python3 util/taskcluster.py --decoder)`. Dadurch wird das benötigte Python-Paket mit Hilfe von Mozillas hauseigenem Framework für ihre kontinuierliche Integration namens *Taskcluster* installiert [Moz19f]. Somit kann `DeepSpeech.py` auf alle benötigten Programme und Bibliotheken zugreifen, wodurch alle Vorbereitungen hinsichtlich der Trainingsumgebung abgeschlossen wären.

3.3.2 Durchführung des Trainings

Für das Training ist die präzise Einstellung der Hyperparameter notwendig. Diese können dem Python-Skript `DeepSpeech.py` als Parameter für das Training mitgegeben werden. Dabei handelt es sich vor allem um die Folgenden:

- `train_batch_size`
  Batch-Größe der Trainingsmenge. Diese beschreibt, wie viele Trainingsdaten aus dem gesamten Datensatz auf einmal von dem neuronalen Netz berechnet werden. Danach erst wird das Netz hinsichtlich ihrer Gewichte aktualisiert und eine Rückpropagierung durchgeführt. Größere Batches bedeuten durch die erhöhte Datenmenge eine akkuratere Bestimmung des Gradienten, wodurch eine genauere Aktualisierung der Gewichte vorgenommen werden kann. Jedoch wird auch eine höhere Speichernutzung verursacht [itd15].
- `dev_batch_size`
  Batch-Größe der Validierungsmenge.
- `test_batch_size`
  Batch-Größe der Testmenge.
- `learning_rate`
  Lernrate, die beschreibt, wie stark Neuronen abhängig vom Ergebnis der Verlustfunktion bestraft werden sollen. Ein zu hoher Wert würde starke Fluktuationen verursachen, die das Minimum des Trainingsverlustes nie erreichen lassen würden. Ein zu niedriger Wert verlangsamt den Trainingsprozess unnötig [Zul18].
- `dropout_rate`
  Ausfallrate der Neuronen. Es wird zufällig ein prozentualer Anteil der *aktivierten* verdeckten Neuronen ignoriert, um so Überanpassung zu verhindern [HCC+14].
- `epochs`
  Anzahl von Epochen, die das Training durchlaufen wird. Eine Epoche nutzt alle Trainingsdaten.
- `n_hidden`
  Anzahl von Neuronen zur Initialisierung der verdeckten Schichten.

Zusätzlich sind noch einige organisatorische Parameter notwendig, die beispielsweise Pfade zu den entsprechenden Trainingsdaten angeben oder das Ausgabeverzeichnis für etwaige Ergebnisse. Damit das Austesten der oben genannten Hyperparameter leichter fällt, kann ein Bash-Skript erstellt werden, welches `DeepSpeech.py` mit den gewünschten Parametern startet (in diesem Projekt genannt `run_training.sh`). Um nicht mit zufälligen Werten zu beginnen, bietet es sich an, die vorgeschlagenen Parameterwerte von Mozilla zu verwenden [Moz19d]:

- `train_batch_size`: 24
- `dev_batch_size`: 48
- `test_batch_size`: 48
- `learning_rate`: 0.0001
- `dropout_rate`: 0.15
- `epochs`: 75
- `n_hidden`: 2048

Es stellt sich heraus, dass diese Parameter bereits gut funktionieren. Einige Tests führten im Nachhinein zu einer Änderung von `dropout_rate` auf 0.2, welche in den meisten Fällen ein etwas besseres Ergebnis erreichte. Es zeigt sich allerdings auch, dass die Anzahl der verdeckten Neuronen recht hoch angesetzt ist und für `n_hidden` bereits ein Wert von 1024 ausreicht, um die gleichen Ergebnisse zu erhalten. Dies deckt sich auch mit den Erfahrungen von [yno18], wobei dort eine ältere DeepSpeech-Version genutzt wurde. Dies reduziert nicht nur die Trainingszeit auf Grund der geringeren Rechenauslastung, sondern führt auch zu besseren Ausgabezeiten bei der Sprache-Zu-Text-Umwandlung. Dies bietet einen besonderen Vorteil bei der Nutzung innerhalb des zu entwickelnden Sprachassistenten. Die Anzahl der Epochen ist ebenso sehr hoch angesetzt, da nach etwa 15 Epochen keine Verbesserung der Verlustwerte mehr festgestellt werden konnte. DeepSpeech besitzt auch die Funktion zu erkennen, ob sich die Verlustwerte in einem lokalen Minimum aufhalten und somit eventuell das Ziel erreicht wurde. Dies funktioniert, indem der Parameter `es_steps` auf einen Wert größer Null gesetzt wird. Dabei werden die Verlustwerte des Validierungsschrittes auf Grund ihrer Standardabweichung und Varianz voneinander analysiert. Die Validierung berechnet, wie viel sich das Training wirklich verbessert hat, indem, ähnlich wie bei der Testmenge, die Performanz mit einer abgegrenzten Validierungsmenge gegengeprüft wird. Bei einer einstellbaren Wertüberschreitung, also zu hoher Abweichung, wird der Trainingsvorgang unterbrochen. Wird diese Funktion verwendet, liegt die Anzahl der gerechneten Epochen in allen Fällen ebenfalls im Bereich von 15. Ein Trainingsdurchlauf mit 15 Epochen, bei `n_hidden` auf 2048, hat etwa acht Stunden angedauert. Es wurde anschließend versucht, das Training mit dem englischen Sprachdatensatz von Common Voice, der mehr als die doppelte Menge an Daten besitzt, durchzuführen. Die Anzahl der Äußerungen war allerdings immer noch nicht genug, um signifikante Verbesserungen in der Spracherkennung festzustellen. Jedoch zeigte sich ein durchschnittlicher minimaler Verlust von 53 - 49, und somit eine kleine Verbesserung, was zeigt, dass mit mehr Sprachdaten durchaus bessere Ergebnisse erzielt werden können. Die Trainingsdauer, bei gleichen Hyperparametern nach ähnlicher

Epochenanzahl etwa 15 Stunden. Bei einer angemessenen, also deutlich größeren Menge an Trainingsdaten könnte die Dauer bei mehreren Tagen liegen, wodurch eine präzise Anpassung der Hyperparameter bei gleichbleibender Hardware sehr aufwendig und langwierig wäre. Somit wurde aus Zeitgründen das Training mit *mehreren* großen, englischen Datensätzen nicht durchgeführt.

Die oben genannten Probleme und Erkenntnisse können allerdings auch daher stammen, dass ein zu geringer Datensatz im Vergleich zu Mozillas Trainingsumgebung vorliegen. Nach dem Training wird die *Testmenge* genutzt, um ohne Anpassung der Gewichte im neuronalen Netz die Performanz mit unbekannten Daten zu prüfen. Die Ergebnisse fallen in allen Tests sehr schlecht aus und es kann kaum eine verwendbare Inferierung festgestellt werden. Dies ist der geringen, deutschen Trainingsmenge zu verschulden. Der resultierende Verlust beträgt in den besten Fällen rund 55 - 60, wobei davon ausgegangen wird, dass höchstens ein Verlust von Zehn bis Fünf eine ausreichende Spracherkennung liefert.

Die unzureichende Performanz des resultierenden akustischen Modells aus den verwendeten, deutschen Sprachdaten schließt dessen Nutzung für die Entwicklung des Sprachassistenten in diesem Projekt aus. Allerdings stellt Mozilla ein vortrainiertes, *englisches* Modell von DeepSpeech bereit [Moz19c]. Es ist also notwendig dieses zu nutzen und auf eine Anhäufung von ausreichend vielen deutschen Sprachdaten zu warten.

4

Speech-Manager

Der Speech-Manager ist das sprachliche Zentrum und verantwortlich für das Hören und Sprechen des Assistenten. In diesem Kapitel wird auf die Echtzeit-Audio-Verarbeitung eingegangen, die dafür sorgt, dass Sprache aufgenommen, erkannt und entsprechend verarbeitet wird, um sie an den nächsten Teil, die Sprache-Zu-Text-Umwandlung, zu leiten. Bei erfolgreicher Durchführung und anschließender Generierung einer Antwort kommt der letzte Teil, die Sprachsynthetisierung, zum Einsatz, auf die ebenfalls kurz eingegangen wird. Im Folgenden fasst der Begriff *Speech-Manager* alle Bestandteile des Sprachmoduls zusammen, wobei mit der Bezeichnung `SpeechManager` (ohne Bindestrich) die konkrete Klasse gemeint ist. Dies wird auch für andere Module und deren gleichnamigen Klassen innerhalb dieser Arbeit so verwendet.

Im allgemeinen Entwurf des Sprachassistenten, zu sehen in Abbildung 2.2 aus Kapitel 2, wird ein verallgemeinerter Entwurf des Speech-Managers gezeigt, um die übergeordnete Idee der Architektur zu erläutern. Nun muss jedoch der konkrete Aufbau der einzelnen Bestandteile und deren Klassen konzipiert werden. So wie der `ConversationManager` eine Hauptschnittstelle für das gesamte System darstellt, dient die Klasse `SpeechManager` als Hauptschnittstelle für das Speech-Manager-Modul und verbindet die Klassen `SpeechListener` und `OutputListener`. Der `SpeechListener` ist, wie bereits in Kapitel 2 beschrieben, für die Tonaufnahme und -verarbeitung zuständig. Dies bedeutet allerdings nicht nur den Ton ordnungsgemäß aufzunehmen, sondern auch aufzubereiten und für die Benutzung mit DeepSpeech verwendbar zu machen. Es beinhaltet die Implementierung einer sogenannten *Sprechpausenerkennung* (engl. *voice activity detection*, kurz *VAD*) [JRS07], die in einem beliebigen Strom von Audiodaten erkennt, wann menschliche Sprache vorkommt und wann nicht. Dies ist unerlässlich, da DeepSpeech nur endliche Mengen von Audiodaten verarbeiten kann. Die effiziente und vor allem effektive Durchführung des VAD führt zu einer Vermeidung unnötiger Inferenzen, was notwendig zur Reduktion der benötigten Rechenleistung ist, da die Berechnung eines neuronalen Netzwerks höchst aufwendig sein und bei entsprechender Größe des Netzes einen erheblichen Leistungseinbruch bedeuten kann. Da eine solche Inferenz die Tonaufnahme für die Zeit der Verarbeitung blockiert, können keine weiteren Audiodaten erhoben und eventuelle Satzanfänge verpasst werden. Dieses Problem könnte durch eine Parallelisierung der Tonaufnahme und der Inferenz

durch DeepSpeech gelöst werden. Dafür wird nun die Klasse `SpeechModelManager` eingeführt, welche eine Kapselung der Sprache-Zu-Text-Funktion von DeepSpeech darstellt. Für diese parallele Zusammenarbeit werden *BlockingQueues* (beschrieben in Kapitel 2) benötigt, welche als Produzent-Konsument-Schnittstellen fungieren. Diese sind nötig, da alle Sprache-Zu-Text-Aufträge asynchron ablaufen sollen, wodurch die Tonaufnahme nicht unterbrochen wird. Die Idee ist, dass der `SpeechListener` die aufbereiteten Audiodaten als Produzent in eine dieser Queues ablegt und nicht weiter auf Anderes warten muss. Der `SpeechModelManager` wartet als Konsument auf eingehende Audiodaten an derselben Queue. Kommen nun welche an, beginnt er mit der Inferenz durch DeepSpeech. Nach Beendigung dieses Schrittes schreibt er sein Ergebnis – hier als Produzent – in eine andere BlockingQueue. An eben dieser Queue würde nun der `ConversationManager` warten, um mit der semantischen Verarbeitung zu beginnen. Dies wird jedoch erst in Kapitel 5 näher elaboriert. Das Einzige, was den `SpeechManager` an dieser Stelle betrifft, ist es, dass er die Schnittstelle für das Sprachmodul darstellt und somit Funktionen implementieren sollte, wodurch genau diese von DeepSpeech erzeugten Klartexte, aus der vom `SpeechModelManager` verwendeten BlockingQueue, abgeholt werden können. Zuletzt fehlt noch der `OutputListener`, welcher jedoch exakt nach dem gleichen Prinzip arbeitet. Diese Klasse ist für die Sprachausgabe verantwortlich. Da der Fokus dieser Arbeit allerdings nicht auf der Synthetisierung von menschlicher Sprache liegt, wird auf ein bereits vorhandenes System zurückgegriffen, worauf in Unterkapitel 4.3 näher eingegangen wird. Wie auch das Aufnehmen eines Satzes nicht unterbrochen werden sollte, darf dies auch bei der *Ausgabe* von Sprache nicht passieren. Somit sollte diese Klasse ebenfalls parallelisiert sein und nur asynchrone Aufträge über eine BlockingQueue annehmen. Die Äußerungen, welche auditiv ausgegeben werden sollen, sind die des Systems. Also muss die genannte BlockingQueue ebenfalls durch den `SpeechManager` zugreifbar gemacht werden, damit letztlich der `ConversationManager` darauf zugreifen und „auszusprechende“ Äußerungen des Systems in Auftrag geben kann. Diese und weitere Klassen des Speech-Manager-Moduls werden in dem UML-Diagramm aus Abbildung 4.1 als Entwurf veranschaulicht. Die bisher nicht genannten Teile des UML-Diagramms werden in den nächsten Unterkapiteln dieses Kapitels näher erläutert. Zusätzlich wird die Funktionsweise noch genauer beschrieben und eine mögliche Implementierung zu den oben genannten Problemen und deren Lösungen aufgezeigt. Alle folgenden Erklärungen zu erwähnten Klassen werden auf Basis dieses Entwurfsmodells durchgeführt.

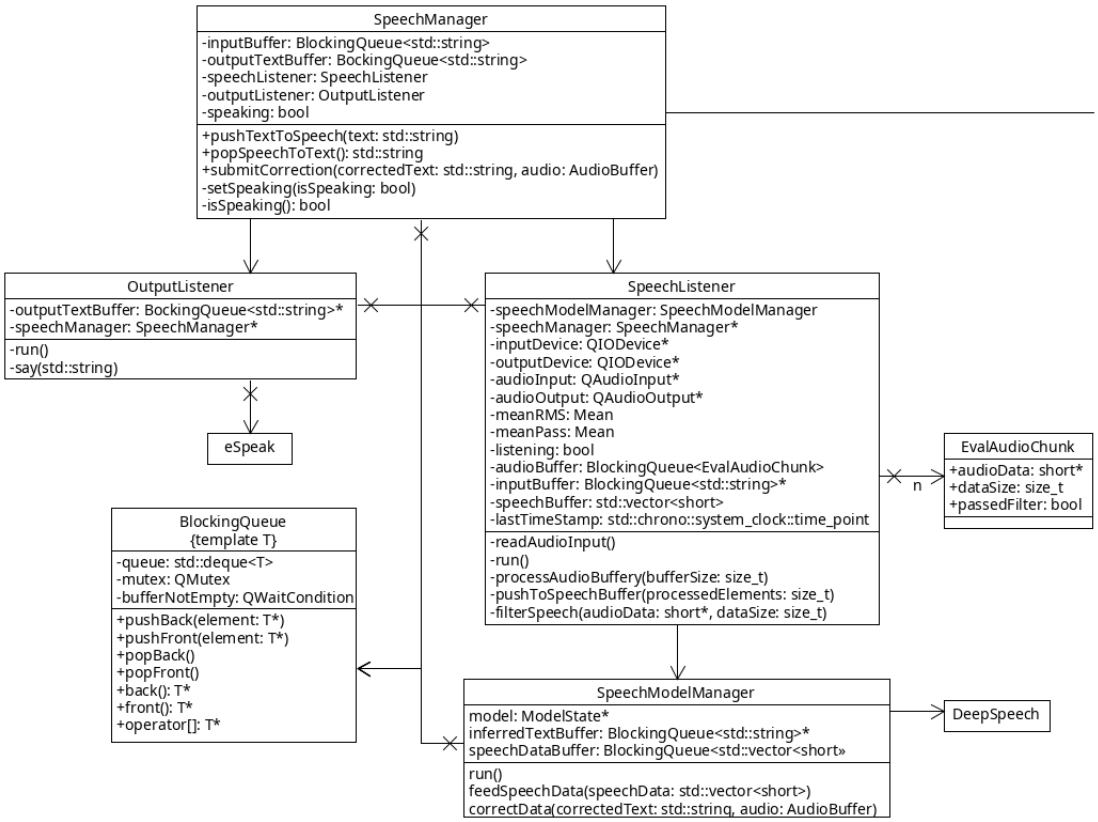

Abb. 4.1. UML-Klassendiagramm des Speech-Manager-Moduls und dessen Bestandteile in detaillierterer Form. Der Pfeil (oben rechts) signalisiert, dass der `SpeechManager` die Schnittstelle für das restliche System darstellt, um mit dem Speech-Manager-Modul zu interagieren.

4.1 Echtzeit-Audio-Verarbeitung

In dem `SpeechListener` sollen nun Audiodaten aus der Umgebung aufgenommen und durch eine VAD-Implementierung für DeepSpeech passend verarbeitet werden. Zuallererst sollte allgemein beschrieben sein, wie Audiodaten aufgenommen werden und vor allem in welcher Form sie vorliegen. Als nächstes wird erklärt, wie die Schnittstelle zu DeepSpeech aussieht und anzusprechen ist, wodurch auch erklärt wird, welche Idee und Technik hinter einem VAD steckt und wie es implementiert werden könnte. Es wurde sich in diesem Projekt für eine eigene Implementierung eines VADs entschieden, da viele bestehende Systeme entweder nicht frei zugänglich sind oder nicht das gewollte Verhalten mitbringen, wie es beispielsweise bei Sprachcodes der Fall ist – Codecs, die Audiodaten enkodieren und dabei Sprache besonders hervorheben und Hintergrundgeräusche herausfiltern, jedoch einen gleichmäßigen Ton beibehalten.

4.1.1 Audioaufnahme

Eine Tonaufnahme innerhalb eines Computerprogramms ist ein diskreter, digitaler Strom von Bytes eines bestimmten Formats. Dieses wird durch die *Abtastrate* (engl. *sample rate*) in Hertz, die *Abtastgröße* (engl. *sample size*) in Bit, der *Abtasttyp/-Datentyp* (engl. *sample type*) und die *Byte-Ordnung* (engl. *byte order*) definiert. Die Abtastrate gibt an, wie viele Datenpakete pro Zeiteinheit aufgezeichnet werden sollen, somit steht eine höhere Zahl für eine höhere Tonqualität. Die Abtastgröße beschreibt die Genauigkeit, in der festgelegt wird, welchen Wert ein Ton zu einem gewissen Zeitpunkt hat. Eine höhere Genauigkeit bedeutet ebenfalls eine erhöhte Tonqualität, aber auch einen erhöhten Speicherbedarf, da diese Zahl auch die verwendete Speichergröße in Bit pro Wert darstellt. Die Abtastgröße spielt auch eine Rolle bei dem Abtasttyp. Zusammen wird mit diesem der zu verwendende Datentyp bestimmt, wenn zum Beispiel konkrete Daten extrahiert werden sollen. Der Abtasttyp ist eine ganze Zahl, entweder vorzeichenbehaftet (engl. *signed int*) oder vorzeichenunbehaftet (engl. *unsigned int*) oder auch gänzlich eine Gleitkommazahl (engl. *floating point*). Ist die Abtastgröße nun 16 Bit und der Abtasttyp Signed-Int, würde der zu verwendende Datentyp – auf den meisten Systemen – ein `short` sein. Da es sich allerdings um einen *Byte*-Strom handelt, müssen sich größere Datentypen in mehrere Bytes aufteilen. Soll nun ein Datum wieder zusammengesetzt werden, muss es eine eindeutige Reihenfolge dieser Bytes geben. Diese wird durch die Byte-Reihenfolge definiert. Sie kann entweder *big-endian* (dt. etwa *großendig*) oder *little-endian* (dt. etwa *kleinendig*) sein, also mit dem hoch- oder niederwertigsten Byte enden. Um nun mit diesen Werten tatsächlich arbeiten zu können, wird in diesem Projekt auf die Bibliothek *Qt* zurückgegriffen [Com19d]. Sie beinhaltet neben Werkzeugen zur Erstellung graphischer Benutzeroberflächen einige andere Multimedia-Tools, also bietet die Einbindung dieser Bibliothek eine gewisse Vorbereitung, falls eine graphische Benutzeroberfläche eingebaut werden sollte. Mit der Klasse `QAudioDeviceInfo` kann ein digitales Audioformat mit den oben beschriebenen Daten definiert und mit `QAudioInput` eine konkrete Schnittstelle zu einem Aufnahmegerät erzeugt werden. Diese ist von dem Typ `QIODevice`, welcher in der Qt-Infrastruktur für jegliche Art von Ein-/Ausgabegeräten steht und demnach entsprechende `write`- und `read`-Funktionen besitzt. Nach einer Pufferungszeit können die neuen Daten des Audiodatenstroms in Form eines `QByteArrays` gelesen werden, also einer Reihe von Bytes in der vorher definierten Form.

DeepSpeech fordert Audiodaten mit einer Abtastgröße von 16 Bit des Abtasttyps Signed-Int als *Rohdaten*. Die digitalisierten Daten werden in der Regel mit einem bestimmten *Codec* codiert, wodurch eine effizientere Speicherung der Daten erfolgen kann. Rohdaten sind allerdings eine Art der digitalen Speicherung, in der Daten durch keinen oder nur einen sehr simplen Codec codiert wurden. Die Klasse `QAudioDeviceInfo` lässt einen Codec namens *PCM* (*Pulse-Code-Modulation*) zu, welcher eine reine Pulsmodulation durchführt, um aus einem zeit- und wertkontinuierlichem Signal ein zeit- und wertdiskretes zu gewinnen [For10], was die benötigten Audiorohdaten liefert. Die Abtastrate ist von DeepSpeech vorgegeben und muss 16 kHz betragen. Die fehlende, einzustellende Byte-Ordnung ist leider

nicht definiert, somit wurde big-endian bevorzugt, da dies eine etwas natürlichere Verarbeitung ermöglicht (der Grund dafür wird in dem folgenden Kapitel beschrieben).

4.1.2 Audiovorbereitung

Die aufgenommenen Audiodaten liegen nun als Array (dt. *Feld*) von Bytes vor. Diese müssen nun wieder zu den oben definierten 16 Bit zusammengefügt werden, um ein Array aus 16 Bit Daten zu erlangen, also `shorts`. Dafür muss ein Algorithmus implementiert werden, der dies unter Berücksichtigung der Endianness des verwendeten Computersystems durchführt. Moderne, auf Intel basierende Prozessoren arbeiten häufig mit little-endian, somit müsste für diese das hochwertigste Byte auch die höchste Speicheradresse sein und vice versa [Gee19]. Aufgrund der Einstellung des Formats auf big-endian muss also das erste von zwei Bytes – das niederwertigere – auf einen leeren `short`-Wert addiert und um ein Byte, also 8 Stellen, nach links verschoben werden, damit es auch die niedrigste Speicheradresse erhält. Das zweite Byte – das hochwertigere – muss nun, bevor es auf den `short`-Wert addiert wird, mit einem Byte voller Einsen verundet werden. Der Grund dafür liegt in der Promotions-Mechanik von C++. Wird eine Variable mit einer anderen verrechnet, wird der komplexeste Datentyp von beiden verwendet und die andere wird konvertiert. Da in diesem Fall ein Byte, oder für C++ konkret ein `char`, mit einem `short` verrechnet wird, gibt es einige Probleme, wenn das zweite Byte eine Eins als hochwertigstes Bit besitzt. Dies würde als Vorzeichen interpretiert und entsprechend des Zweierkomplements der neue Platz vorne in der Variable mit Einsen aufgefüllt werden. Wird diese Zahl nun auf die ursprüngliche Zahl addiert, überdecken die aufgefüllten Einsen das erste, bereits verrechnete Byte und das Ergebnis würde verfälscht werden. Somit ist dieser Schritt notwendig für das ordnungsgemäße Zusammenfügen von Bytes zu einem größeren Datentyp. Für einen big-endian-Prozessor müssten die Bytes natürlich genau anders herum eingefügt werden, jedoch nach dem gleichen Prinzip. Um nun entscheiden zu können, welche Endianness das aktuelle Computersystem nutzt, bietet sich laut [Gee19] ein einfacher Trick an. Eine Variable eines Typs, welcher größer ist als ein Byte, erhält einen Wert, welcher maximal die Größe eines Bytes hat, jedoch nicht Null ist. Wird nun diese Variable in einen `char`-Zeiger umgewandelt, zeigt er auf die erste/kleinste, Speicheradresse davon. Wird also der `char`-Wert dieser Adresse betrachtet und ist es keine Null, handelt es sich um ein little-endian-System, da nach dieser Art der Speicherung die niedrigste Adresse der Variable den hinzugefügten Wert beinhalten muss, bei welchem es sich um das niederwertigste Byte handelt. Ist der Wert jedoch eine Null, ist es ein big-endian-System, da der Wert in der Variable nicht groß genug ist, um auch die höherwertigen Bytes zu besetzen und die anderen Bytes demnach nur Nullen enthalten.

Die zusammengefügten Werte wieder in einen Byte-Strom umzuwandeln, funktioniert nach dem gleichen Prinzip und müsste bei einer eventuellen Ausgabe der aufgenommenen Audiodaten vorher durchgeführt werden, beispielsweise zu Test- oder Debugging-Zwecken.

4.1.3 Sprachvorfilterung

Es wurde beschrieben, dass ein VAD benötigt wird, um Sprache aus einem endlosen Audiodatenstrom zu filtern und in endliche, verarbeitbare Datenabschnitte zu unterteilen. Um dies zu erreichen, muss zuerst definiert werden, was genau Sprache von anderen Geräuschen unterscheidet und welche Techniken dafür zur Verfügung stehen. Sprache befindet sich samt Obertönen im Frequenzspektrum etwa zwischen 85 Hz und 12 kHz [RJB87]. Einfach wäre es nun diesen Frequenzbereich zu filtern und alle Audiodaten, welche außerhalb diesem liegen, zu ignorieren. Dieser ist jedoch sehr groß und viele alltägliche Geräusche fallen in genau diesen Frequenzbereich. Eine bekannte Technologie, welche sich dem gleichen Problem stellen musste, ist die Telefonie. Üblicherweise wird dort ein Frequenzspektrum von etwa 300 bis 3.400 Hz übertragen und andere Frequenzen abgeschnitten [ITU03]. Jedoch klingt eine so radikale Einengung sehr unklar, und lässt Buchstaben wie „P“ oder „T“ und „F“ oder „S“, aufgrund der fehlenden hohen Töne, sehr ähnlich klingen. Nach einigem Testen zeigt sich, dass ein Frequenzbereich von 100 bis 8.000 Hz die besten Ergebnisse erzielt. Ein klares Klangbild ist wichtig, da DeepSpeech anfällig für unklare oder rauschende Sprachaufnahmen ist – zumindest ohne spezielles Training in einer rauschenden Umgebung. Diese Frequenzen legen nun den Filterbereich fest. Um eine Analyse auf unterschiedliche Frequenzen durchzuführen, wird ein Audiosignal im *Frequenzraum* (engl. *frequency domain*) benötigt. Die vorliegenden Audiodaten befinden sich jedoch im *Zeitbereich* (engl. *time domain*), es handelt sich also noch um ein Zeitsignal. Damit es in den Frequenzraum umgewandelt werden kann, bietet sich eine Technik namens *Fourier Transformation* an [But11], auf dessen komplexe Vorgehensweise hier nicht weiter eingegangen wird. Ist das Signal nun im Frequenzbereich, können die Frequenzen außerhalb des oben definierten Bereiches einfach abgeschnitten und anschließend wieder in ein Zeitsignal transformiert werden.

Für dieses Projekt wird die DSP-Bibliothek (*Digital Signal Processing*, dt. *digitale Signalverarbeitung*) *Aquila* verwendet [Sic19]. Sie bietet für die hier wichtige Frequenzfilterung die Möglichkeit einen sogenannten *High-Pass-* (dt. *Hochpass*) und *Low-Pass-Filter* (dt. *Tiefpass*) zu erzeugen, indem sie die *Schnelle Fourier Transformation* (engl. *Fast Fourier Transformation* im Folgenden *FFT* genannt) implementiert [Lan18]. Aquila arbeitet mit Gleitkommazahlen wie `double`. Die aufgezeichneten Audiodaten liegen aber als `short` vor und müssen zunächst in entsprechende `double`-Werte umgewandelt werden. Im Grunde stellen ganze Zahlen natürlich auch Gleitkommazahlen dar, jedoch ohne Nachkommastellen. Es kann allerdings nicht garantiert werden, dass die ganzzahligen Gleitkommawerte nach der FFT immer noch ganzzahlig sind. Die spätere Umwandlung zurück in den ausschließlich ganzzahligen Bereich auf die gleiche Weise wäre demnach verlustbehaftet und könnte zu einer Qualitätsreduktion, oder sogar zur gänzlichen Korruption der Audiodaten führen. Um dies zu vermeiden, soll ein fest definierter Wertebereich der Gleitkommazahlen verwendet werden, der die ganzzahligen Werte in ihrem maximalen Wertebereich abbilden. Vorzeichenbehaftete 16-Bit-Zahlen, oder `shorts`, besitzen Werte im Bereich von -2^{15} bis $2^{15} - 1$. Die Idee ist nun, dass die neuen

`double`-Werte das Verhältnis der `short`-Werte widerspiegeln, den sie in ihrem Wertebereich hatten. Dies lässt sich gut in dem `double`-Wertebereich von $1,0$ und $-1,0$ durchführen (es wäre allerdings auch jeder andere Wertebereich möglich, jedoch nicht so leicht überprüfbar). Eine negative Zahl müsste nun durch den kleinsten negativen Wert des Wertebereichs, und eine positive Zahl durch den höchsten positiven geteilt werden. Der vorzeichenbehaftete 16-Bit-Wert $2^{14}-1$ würde daher dem Gleitkommawert $0,5$ entsprechen, da es genau die Hälfte des maximalen Wertebereichs repräsentiert. In gleicher Weise können nun Gleitkommawerte wieder zurück in den ganzzahligen Bereich umgerechnet werden, ohne oder nur in geringem Maße an Genauigkeit zu verlieren. Da nun die Audiodaten in entsprechende `double`-Werte umgewandelt wurden, kann ein *Aquila-Signal* daraus gemacht werden, eine Kapselung von Audiodaten in der Klasse `Aquila::SignalSource`. Es sollte an dieser Stelle erwähnt werden, dass Aquila ausschließlich Datenlängen verarbeiten kann, welche ein Exponential von Zwei sind, was ein Auffüllen (engl. *padding*) mit Nullen auf das nächsthöhere Zweierexponential erfordert. Dieses Padding muss nach der Frequenzfilterung wieder entfernt werden, um Rauschen zu vermeiden, welche durch die Null-Bytes verursacht werden. Das entsprechend formatierte Signal muss nun mit Hilfe der FFT in ein Audiospektrum umgewandelt werden, welches das Signal im Frequenzbereich darstellt. Dafür wird die Klasse `Aquila::fft` verwendet, der als Parameter die mit Nullen aufgefüllte Signallänge erhält und zusammen mit den Audiodaten als Zeitsignal das Frequenzspektrum als Objekt der Klasse `Aquila::SpectrumType` erstellt. Um nun eine Filterung vorzunehmen, wird ein Frequenzband benötigt, was aussagt, welche Frequenzen durchgelassen werden sollen und welche nicht. Dafür bietet sich ein weiteres `Aquila::SpectrumType` Objekt an, in dem wir jeweils eine `1` für jede Frequenz eintragen, die durchgelassen werden soll und eine `0` für jede andere. In diesem Fall wären es die Frequenzen im Bereich von 100 bis 8.000 Hz. Diese beiden Frequenzbänder, also das mit den aufgezeichneten Daten und das Filterband, können so nun miteinander multipliziert werden, was Frequenzen eliminiert, die an korrespondierender Stelle im Filterband mit einer `0` multipliziert werden. Da die `1` das neutrale Element der Multiplikation ist, bleiben Frequenzen, welche im Filterband mit einer `1` gekennzeichnet wurden, unverändert. Das resultierende, gefilterte Frequenzspektrum muss nun wieder zurück in ein Zeitsignal transformiert werden, was durch die *inverse FFT* funktioniert. Nach der Rücktransformation muss, wie oben angesprochen, das Padding am Ende der Audiodaten entfernt werden.

Dieses Vorgehen hat jedoch nur grob einen Bereich geschaffen, in dem Sprache auf jeden Fall auftauchen würde, jedoch nicht ausschließlich. Außerdem wird weiterhin auf einem endlosen Audiodatenstrom gearbeitet, was keinen Aufschluss darüber liefert, wann ein Satz anfängt und wieder endet.

4.1.4 Sprechpausenerkennung

Die menschliche Sprache wurde im vorherigen Abschnitt dahingehend definiert, dass sie in einem gewissen Frequenzspektrum zu finden ist. Dieses Spektrum teilt sie sich jedoch auch mit anderen Geräuschen. Ein weiteres Merkmal, welches Ton

unterscheidbar macht, ist die Lautstärke. Auch wenn sie keinen direkten Aufschluss darüber gibt, ob es sich nun wirklich um Sprache handelt, da Sprache abhängig von den Umständen natürlich unterschiedlich laut sein kann, gibt es jedoch einen Unterschied zwischen Tönen, welche man hören *soll* und die, die ignoriert werden *sollen*. Menschen reagieren auch nicht auf alle Geräusche, sei es nun Sprache oder nicht: Es hängt auch von der Lautstärke ab. Können Menschen gut und deutlich verstanden werden, spricht die Person vermutlich in die Richtung des Angeredeten und es wird in der Regel zugehört, da davon ausgegangen wird, dass sie gemeint ist (sofern keine weiteren Faktoren hinzugezogen werden). Der Sprachassistent besitzt keine Kamera zur Lokalisierung von Personen, um zu erkennen, ob eine Person in die Richtung des Geräts spricht oder nicht. Demnach ist die Lautstärke, mit welcher gesprochen wird, der ausschlaggebendste Faktor, ob etwas nun gehört werden soll oder nicht. Nun kann natürlich nicht nur Sprache eine gewisse Lautstärke in dem hier definierten Frequenzbereich erreichen. Jedes beliebige Geräusch könnte unter Umständen laut sein. Jedoch besitzt Sprache auch gewisse Charakteristika innerhalb ihres *Klangbildes*. Während die meisten Geräusche eher kurz sind, oder wenn sie länger andauern, ihren lautstarken Höhepunkt häufig nur recht kurz halten, hat die menschliche Sprache ein kontinuierliches Lautstärkepegel. Natürlich gibt es auch in dieser Klasse von Geräuschen wieder einige, welche keine Sprache sind, jedoch ist die Wahrscheinlichkeit nun deutlich gesunken. Die Kontinuität der menschlichen Sprache ist allerdings nicht perfekt. Es existieren Stottern, Füllwörter oder Stimmeinbrüche, bei denen für kurze Zeit leiser gesprochen wird als normal. Also benötigt ein System, welches diese Lautstärkefilterung vornimmt, dahingehend eine gewisse Toleranz. Wenn all diese Faktoren innerhalb einer Analyse auf das zurzeit noch endlose Audiosignal berücksichtigt werden, würde eine deutlich verbesserte Unterscheidung zwischen Sprache und Nicht-Sprache möglich sein, um somit den Anfang oder das Ende dieser Abschnitte finden zu können. Ein solches System stellt nun das in dem Oberkapitel beschriebene VAD, also die Sprechpausenerkennung, dar.

Als nächstes gilt es nun herauszufinden, wie ein VAD programmiertechnisch umgesetzt werden kann. Es bietet sich an, den Analyseprozess in zwei Teile zu trennen, um dieses komplexe Verfahren in simplere Teilsysteme aufzuteilen. Der erste Teil ist eine Art Vorfilterung, die aus einer kurzfristigen Analyse hinsichtlich der Lautstärke einzelner Audiodaten besteht und eine einfache Lautstärkefilterung vornimmt, um Audiodaten als „laut genug“ oder „zu leise“ zu klassifizieren. Der zweite Teil analysiert das längerfristige Klangbild über einen längeren Zeitraum, um eventuelle Toleranzen und Unreinheiten in einem Satz abzufedern oder eben zu kurze Töne als Nicht-Sprache zu ignorieren.

Audiodaten kommen in unterschiedlichen Zeitabständen hintereinander, gepuffert in verschieden großen Datenmengen an. Diese werden hier als *Audio-Chunks* (dt. etwa *Audiodatenhaufen*) bezeichnet. Dieser Umstand ist wichtig, da die Lautstärke eines Tons üblicherweise nicht an einzelnen Audiodaten gemessen, sondern über einen gewissen Zeitraum approximiert wird. Diese Approximation erfolgt in der Regel durch die Errechnung des RMS (*Root Mean Square*, dt. *Qua-*

dratisches Mittel) [Han10], welches sich durch $\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}$ definiert. Der Sinn hinter dieser Formel ist, dass hohe Werte (hier x_i) einen höheren Einfluss besitzen als beim arithmetischen Mittel. Einzelne Audiowerte können so klein betrachtet auf Grund von Rauschen oder fehlerhafter Übertragung sehr stark fluktuieren und somit keinen wirklichen Aufschluss darüber geben, ob es wirklich einem lauten Ton entstammt. Dies kann allerdings auch nicht ausgeschlossen werden, was der Grund für die Höhenaffinität der Formel ist. Eine Analyse auf einen etwas größeren Zeitraum ist demnach weniger fehleranfällig. Auf dem getesteten Computersystem liefert Qt etwa 22 Audio-Chunks pro Sekunde, somit beinhaltet ein Audio-Chunk die Daten von circa 45 Millisekunden, was rund 2.600 einzelner Audiodaten entspricht. Der zu analysierende Zeitraum darf jedoch auch nicht zu groß sein, da sich sonst Daten von tatsächlich lauten Geräuschen nicht gegen benachbarte Daten leiser Töne durchsetzen können. Somit erscheint ein Audio-Chunk als Analysegröße genau richtig. In dem Unterkapitel 4.1.3 wurden die Audio-Chunks hinsichtlich ihrer Frequenz gefiltert, was manche von ihnen gegebenenfalls leiser werden ließ, da sie weniger relevante Frequenzen beinhalteten. Nun werden von genau diesen Audio-Chunks, mit Hilfe der statischen Funktion `Aquila::rms()` nach dem oben beschriebenen Verfahren, die Lautstärken ausgerechnet. Das Ergebnis, also der RMS-Wert des Audio-Chunks, kann jetzt mit einem Grenzwert (engl. *threshold*) verglichen werden, der entscheidet, ob ein RMS-Wert „laut genug“ ist. Jedoch stellt sich nun die Frage, was genau diesen Grenzwert definiert. Es könnte ein einmalig gemessener Lautstärkewert des aktuellen Raumes sein, in dem der Sprachassistent eingesetzt wird. Eine solch statische Vorgehensweise hätte allerdings den Nachteil, dass es auf sich ändernde Umgebungen nicht reagieren könnte. Es muss also ein dynamisches, adaptives Prinzip dahinter stecken. Eine weitere Idee könnte sein, dass über einen längeren Zeitraum in gewissen Intervallen eine Durchschnittslautstärke berechnet wird, um sie als Grenzwert zu verwenden. Diese Vorgehensweise hat jedoch einen ähnlichen Nachteil. Der aktuell gesetzte Grenzwert spiegelt einen vergangenen Zeitabschnitt wider, welcher auf die aktuelle, sich potentiell schnell ändernde Umgebungslautstärke nicht reagieren kann, bis das nächste Intervall anfängt und ein neuer Grenzwert gesetzt wird, der genau diese, jedoch schon wieder vergangenen Lautstärkeschwankungen berücksichtigt. Die Lösung zu diesem Problem wäre nun, die vergangenen Grenzwerte mit der aktuellen Lautstärke zu verbinden. Anstatt also erst auf die Beendigung des Intervalls zu warten, um die Gesamtdurchschnittslautstärke als Grenzwert einzusetzen, könnte das Zeitfenster dieser Berechnung variabel verschoben werden und sozusagen mitlaufen. Somit wäre die aktuelle Durchschnittslautstärke mit den aktuellsten Audiodaten immer der aktuellste Grenzwert. Das Problem, welches noch bleibt, ist, dass das variable Zeitfenster keine Beendigung des Intervalls vorsieht. Dies führt jedoch dazu, dass sehr alte Lautstärkeinformationen, welche keinen Bezug auf die gegenwärtigen Audiodaten haben, aber genau diese beeinflussen. Optimal wäre es, sobald das Zeitintervall eine gewisse Maximallänge erreicht hat, zu jedem neu hinzugefügten Wert den ältesten zu entfernen. Der Durchschnitt ist allerdings nicht assoziativ, also kann eine solche Operation nur durchgeführt werden, wenn alle Zahlen, die den

Durchschnitt ausmachen, bekannt sind. Je nach Länge des maximalen Zeitintervalls kann das neue Ausrechnen aus allen Zahlen aufwendig sein, somit würde es sich aus Performanzgründen anbieten, das maximale Zeitfenster nach Erreichen zu halbieren, da dies die einfachste Operation darstellt. Dafür muss lediglich die Summe aller Werte und die Anzahl der Werte, die sie ausmacht, gespeichert werden, woraus sich leicht der Durchschnitt errechnen lässt. Diese beiden Werten würden durch Zwei geteilt werden, um das Zeitfenster ebenfalls zu halbieren. Dies hätte allerdings den Nachteil, dass die gemessene Lautstärke direkt nach der Halbierung des Zeitfensters einen erheblich größeren Einfluss hätte als noch die davor. Es muss also zwischen Performanz und Genauigkeit entschieden werden. Wird das Zeitfenster recht groß gehalten, schlägt die Ungenauigkeit kaum ins Gewicht, ganz anders als bei kleineren. Mit diesem System kann nun der Grenzwert dynamisch nach aktuellem Lärmpegel angepasst werden. Eine graphische Repräsentation ist als Aktivitätsdiagramm in Abbildung 4.2 zu sehen. Um kleine Ausschläge auch

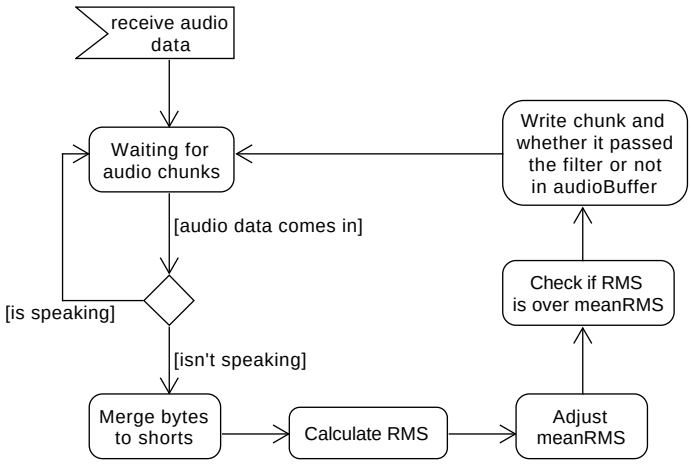

Abb. 4.2. Ein UML-2-Aktivitätsdiagramm, welches die Lautstärkefilterung anhand des RMS-Wertes eines Audio-Chunks, mit Hilfe des dynamischen Grenzwertverfahrens (hier dargestellt als `meanRMS`) innerhalb der Klasse `SpeechListener` zeigt.

in dieser kurzfristigen Analyse zu berücksichtigen, wird zu dem aktuellen Grenzwert, welcher die *genaue* Lautstärkegrenze beschreibt, ein statischer Wert hinzuaddiert, welcher noch zusätzlich von der Lautstärke des aktuellen Audio-Chunks überschritten werden muss. Es sollen eben nur signifikant lautere Geräusche in Betracht gezogen werden. Die Frage lautet nun, wie lang das maximale Zeitfenster sein darf, um kurze Geräusche abzufedern, sich dennoch an lang anhaltende Umgebungsgeräusche, wie Hintergrundmusik, anzupassen kann. Normale Kommandos für einen Sprachassistenten liegen höchstwahrscheinlich im Bereich zwischen zwei und sechs Sekunden. In diesem Zeitraum erhält das Grenzwertsystem sehr laute Audiodaten, die den Grenzwert drastisch höher werden lassen. Damit er nun

die aktuelle Gesprächslautstärke nicht erreicht, wodurch relevante Audio-Chunks als zu leise eingestuft würden, muss das Zeitfenster entsprechend groß sein, allerdings auch nicht zu groß. Nach einigen Tests stellte sich für dieses Projekt und die verwendete Testumgebung heraus, dass ein maximales Zeitintervall von 500 Audio-Chunks, was etwa 23 Sekunden entspricht, die besten Ergebnisse liefert. Überschreitet die Lautstärke also den Grenzwert, beziehungsweise nicht, wird der Audio-Chunk entsprechend markiert, jedoch nicht gänzlich eliminiert. Der Grund liegt darin, dass an dieser Stelle noch nicht entschieden werden kann, ob der analysierte Audio-Chunk wirklich nicht zu irrelevanten Geräuschen gehört oder nicht. Hier existiert noch keine Toleranz bezüglich menschlicher Sprache und kann auf so kurzfristigem Zeitraum auch gar nicht erfolgen. Also ist der erste Schritt als eine Art Voranalyse zu betrachten, die den nächsten Schritt deutlich vereinfacht.

Der zweite Teil des VAD nimmt sich nun der längerfristigen Analyse an, um mögliche Zusammenhänge zwischen einzelnen Audio-Chunks zu erkennen und eine Reihe von ihnen schlussendlich als menschliche Sprache zu klassifizieren. Wie bereits beschrieben wurde, ist Sprache eher kontinuierlich statt abgehackt. Demnach könnte ein Lösungsansatz sein, die Audio-Chunks über einen festgelegten Zeitraum auf Kontinuität in ihrer Bewertung zu analysieren. Wenn also viele Audio-Chunks in diesem Zeitraum positiv bewertet wurden, spricht dies für die Existenz von Sprache innerhalb dieses Zeitfensters. Somit könnte mit einer ausreichenden Menge von positiv bewerteten Audio-Chunks der Start einer Sprachsequenz festgelegt werden. Eine einfache Möglichkeit wäre es nun, alle Zeitfenster, in der eine ausreichende Menge positiv bewerteter Audio-Chunks vorliegt, dieser Sprachsequenz beizufügen und auf eine unzureichende Menge zu warten, die dann das Ende der Sprachsequenz markiert. Diese würde dann als ein einheitliches `short`-Array zum `SpeechModelManager` gesendet werden, um es in schriftlichen Text umwandeln zu können. Diese Vorgehensweise hätte jedoch noch zwei ungeklärte Parameter: die Länge des Zeitfensters und die Menge an benötigten, positiv bewerteten Audio-Chunks. Wenn das Zeitfenster zu kurz ist, ermöglicht es keine Analyse des längerfristigen Klangbildes und kurze Stotterer oder Stimmeinbrüche bleiben unerkannt und es wird als Sprechpause gedeutet. Wird aber ein zu großer Zeitraum gemessen, könnten besonders kurze Sätze oder gar einzelne Wörter ignoriert werden und nur besonders lange Sätze bekämen eine Chance, überhaupt beachtet zu werden. Die perfekte Länge muss sich also durch die Eigenschaften einer bestimmten Sprache/Sprachgruppe rechtfertigen lassen. Im Deutschen können einzelne kurze Wörter weniger als eine Sekunde zur Aussprache benötigen, jedoch kaum unter einer halben. Somit ließe sich, zumindest für die deutsche Sprache ein Zeitfenster von zwischen einer halben und ganzen Sekunde begründen. Es müssten also mindestens 0,5 Sekunden lang eine bestimmte Menge positiv bewerteter Audio-Chunks vorliegen, um als Teil einer Sprachsequenz erkannt zu werden. Diese Menge lässt sich nicht so einfach durch tatsächliche Sprache rechtfertigen, sondern eher durch die schiere Mehrheit. Durch einiges Testen erwies sich dieser Ansatz als am wirkungsvollsten, wodurch die benötigte Menge bei einem Grenzwert um 50 % herum, also etwa der Hälfte aller Audio-Chunks, liegen müsste. In diesem Projekt wurden 500 Millisekunden als Zeitfenster und 50 % aller Audio-Chunks

innerhalb dieses Zeitfensters als Grenzwert festgelegt. Würde diese Methode genau so Anwendung finden, hätte dies einiges an Stottern und Lücken innerhalb der Sprachsequenz zur Folge. Sie würden häufiger unterbrochen werden und DeepSpeech würde keine ordentliche Inferenz durchführen können. Durch das Anpassen des Zeitfensters könnte dieser Umstand ein wenig abgeschwächt, jedoch nicht großteils beseitigt werden. Dies steigert zusätzlich die Menge an *False-Negatives* (dt. etwa *fehlerhaftes Negativ*), da eben kurze Wörter kaum noch erkannt würden. Der Grund für die schlechte Performanz des Verfahrens liegt darin, dass eben Lautstärkeschwankungen in der Stimme, kurzes Stocken oder leise Wortenden/-anfänge häufig Audio-Chunks produzieren, welche innerhalb eines Zeitfensters zu selten positiv bewertet wurden, die Äußerung des Sprechers allerdings noch gar nicht abgeschlossen wurde. Diese „Sprechfehler“ können somit ebenfalls innerhalb dieses Zeitfensters liegen. Und genauso wie Einzelwörter damit berücksichtigt werden, lässt es diese Sprechfehler aus der Sprachsequenz fallen. Dieses Problem lässt sich also auch nicht durch die Senkung der benötigten Anzahl positiv bewerteter Audio-Chunks innerhalb eines Zeitintervalls lösen. Dieses Problem ist auf die Klangstruktur von menschlicher Sprache zurückzuführen. Die Toleranz des Verfahrens muss also erhöht werden. Eine Herangehensweise wäre nun den Analysezeitraum zu erweitern, ohne das Zeitfenster des Messintervalls zu verändern. Dies kann darüber geschehen, dass klassifizierte Zeitintervalle als Ganzes mit anderen auf Kontinuität geprüft werden. Das einzelne Zeitintervall schützt Wörter. Der erweiterte Analysezeitraum soll nun auch die Ganzheit von Äußerungen schützen. Wenn also auf ein positiv klassifiziertes Intervall ein negatives folgt, kann dies ein Ende der Sprachsequenz bedeuten. Kommt diesem jedoch erneut ein positiv klassifiziertes Intervall nach, könnte es sich hierbei um eine zusammenhängende Sprachsequenz handeln. Eine solche Sequenz hätte bis zu diesem Zeitpunkt bereits eine Länge von 1,5 Sekunden und wäre somit immer noch sehr kurz für einen richtigen Satz. Es lässt also die Annahme zu, dass ein solches Muster menschlicher Sprache zu Grunde liegt. Mit dieser Verbesserung wird die Wahrscheinlichkeit eines False-Negatives deutlich gesenkt. Im Gegenzug ist die der *False-Positives* (dt. etwa *falsches Positiv*) jedoch drastisch gestiegen. Einige Tests zeigen, dass selbst bei einem ruhigen und alltäglichen Umfeld die False-Positives-Rate immer noch extrem hoch ist und es teilweise vorkommt, dass eine Sprachsequenz sehr lang wird und kaum zum Ende kommt, unabhängig davon, ob nun Sprache vorliegt oder nicht. Dieser Ansatz stellt sich also als nicht effektiv heraus. Jedoch ist die Idee dahinter erfolgversprechend, da die False-Negative-Rate gegen Null geht. Ein ganzes negatives Zeitintervall passieren zu lassen ist also zu aggressiv und den Grenzwert für die Menge der positiv bewerteten Audio-Chunks pauschal zu senken führt auch zu schlechten Ergebnissen. Ein nächster Ansatz könnte nun sein, dass das erste Zeitintervall mit dem normalen Grenzwert von 50 % überprüft wird, um False-Positives zu vermeiden. Sobald allerdings eine ausreichende Menge positiv bewerteter Audio-Chunks vorliegt, wird der Grenzwert für alle nachfolgenden Zeitintervalle gesenkt, um einem verfrühtes Abschneiden der Sprachsequenz vorzubeugen. Dieser verbesserte Ansatz erweist sich als sehr ertragreich und führt bei einem getesteten Zweitgrenzwert von etwa 40 % zu deutlich saubereren Sprach-

sequenzen, da durch die Reduktion des Grenzwertes die Toleranz erhöht wurde, aber nicht wahllos unqualifizierte Zeitintervalle durchgelassen werden. Nur bei Hintergrundmusik und lautstarken Gesprächen, welche nicht dem Sprachassistenten gewidmet sind, können einige False-Positives verursachen, jedoch greift hier der dynamisch angepasste Regler für den Lautstärkegenzwert aus dem ersten Teil des VAD, der die False-Positive-Rate wieder drastisch senkt. Es wurde allerdings für wichtiger empfunden, vor allem legitime Sprache stets als solche zu klassifizieren und dafür einige False-Positives in Kauf zu nehmen. Nun ist der nächste Schritt, dieses Verfahren zur Bestimmung von Anfang und Ende einer Sprachsequenz programmiertechnisch zu realisieren.

Für die Implementierung werden zwei neue Klassen benötigt: `EvalAudioChunk` (*Eval* steht für *Evaluated*, dt. *bewertet*), welche die Daten eines Audio-Chunks, die Anzahl der enthaltenen Daten und dessen Klassifikation als „laut genug“, in Form eines `bools` speichert. Außerdem noch die Klasse `Mean`, welche lediglich ein Funktionsobjekt in Form eines arithmetischen Durchschnitts darstellen soll. Beide Klassen sind in dem UML-Diagramm aus Abbildung 4.1 zu sehen. Um das oben beschriebene Verfahren mit den bekannten Klassen und Operationen besser zu veranschaulichen, bietet Abbildung 4.3 eine bildliche Repräsentation. Für das VAD muss nun manuell eine gewisse Zeit gemessen werden. Es bietet sich also an, den `QTimer` aus der Qt-Bibliothek zu verwenden, der eine angegebene Funktion aufruft, sobald ein Intervall zu Ende gelaufen ist. In diesem Zeitintervall – hier konkret 500 Millisekunden – werden nun Audio-Chunks gesammelt und, wie in Unterkapitel 4.1.2 beschrieben, entsprechend ihrer Lautstärke markiert und in `EvalAudioChunk`-Objekte gekapselt, welche anschließend in einen Audiopuffer geschrieben werden. Dieser Audiopuffer wird durch ein `std::vector<EvalAudioChunk>`-Objekt repräsentiert. Ist die Zeit abgelaufen, werden die `EvalAudioChunks` des Audiopuffers darauf überprüft, ob die Hälfte von ihnen positiv bewertet wurden. Dies geschieht durch den Einsatz eines `Mean`-Objektes namens `meanPass`, dem eine `1` für jedes positive `EvalAudioChunk`-Objekt, und eine `0` bei jedem negativen zugewiesen wird. Ergibt sich zum Schluss ein Wert gleich oder größer als $0,5$, wurde der Grenzwert erreicht und die Audio-Chunks werden in einen speziellen Sprachpuffer geschrieben, der eine Sprachsequenz darstellt. Dieser Puffer ist in diesem Projekt ein Objekt vom Typ `std::vector<short>`. Der Template-Parameter ist hier ein `short`, da der Audio-Chunk bereits ausreichend überprüft wurde und Teil der finalen Sprachsequenz wird und demnach nur die Audiodaten selbst relevant sind. Das Kopieren des Audiopuffers in den Sprachpuffer markiert den Beginn der Sprachsequenz, was bedeutet, dass die weitere Verarbeitung unterschieden werden muss. Diese Unterscheidung kann anhand einer booleschen Variable namens `listening` abgelesen werden, welche jetzt mit `true` besetzt wird. Sollte der Wert von `meanPass` allerdings niedriger als $0,5$ sein, wird der Audiopuffer geleert. Die Leerung erfolgt jedoch nicht einfach so, da es sein kann, dass innerhalb des gemessenen Zeitraumes von 500 Millisekunden der Beginn einer Sprachsequenz enthalten ist, jedoch erst ganz am Ende und somit nur einen geringen Teil der Gesamtlautstärke ausmacht und demnach das Zeitintervall nicht akzeptiert wird. Es dürfen also nur die `EvalAudioChunk`-Objekte gelöscht werden, bei denen das

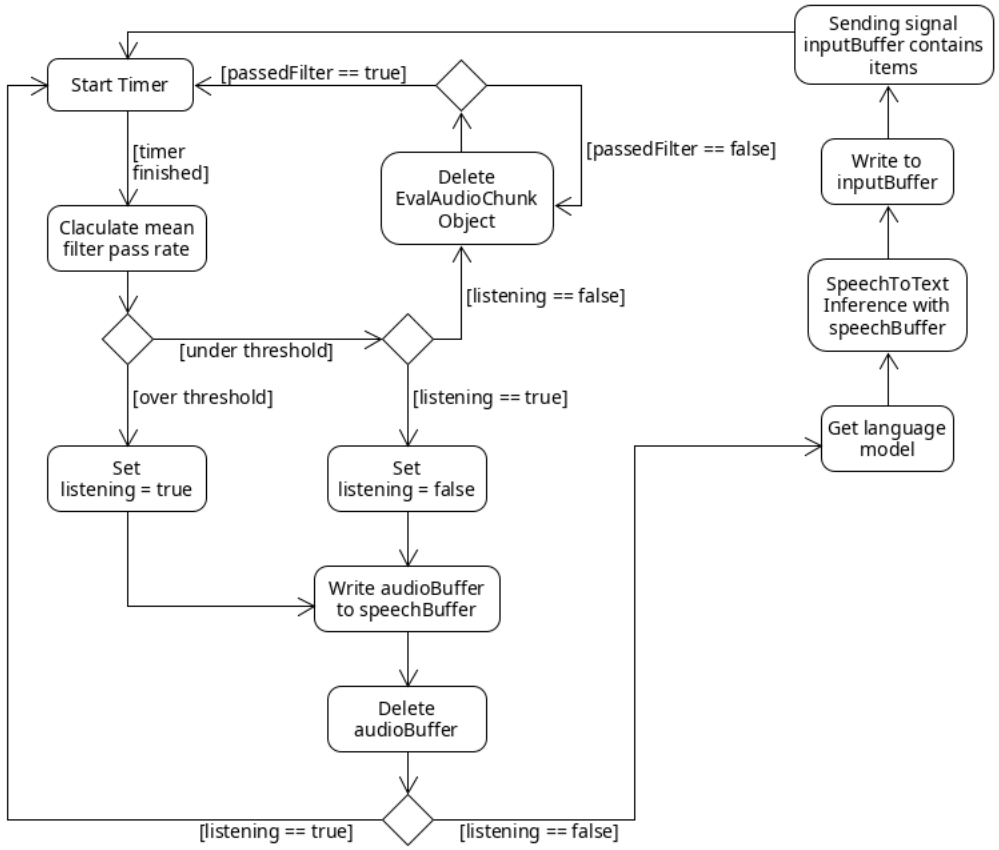

Abb. 4.3. Ein UML-2-Aktivitätsdiagramm, welches das VAD mit seinen zwei Teilen zeigt.

`passedFilter`-Flag auf `false` gesetzt wurde. Wird wieder eines gefunden, bei dem es auf `true` gesetzt ist, muss erneut ein 500-Millisekundenintervall durch `meanPass` geprüft werden. Dies kann allerdings nur erfolgen, nachdem der `QTimer` fertig gelaufen ist. Dieser hat jedoch wieder ganze 500 Millisekunden und Audiodaten gesammelt, welche wiederum auch im Audiopuffer landen und somit zusammen mit den vorherigen `EvalAudioChunk`-Objekten einen gegebenenfalls deutlich größeren Zeitraum ausmachen, wenn beispielsweise recht früh ein positiv bewertetes gefunden wurde. Der Audiopuffer würde sich also bei einer nicht vollständig ruhigen Umgebung unendlich füllen und die Audioverarbeitung somit lähmen. Die manuelle Zeitmessung darf also nicht als das pauschale Warten auf eine sich wiederholende Zeiteinheit verstanden werden. Eher sollte die Menge an vorhandenen `EvalAudioChunk`-Objekten Aufschluss über die *bereits verstrichene* Zeit geben. Dazu muss eine kleine Anpassung des ersten Teils im VAD durchgeführt werden. Die `EvalAudioChunk`-Objekte brauchen nun ein Attribut, worin ein Delta der Zeit eingetragen werden kann, was die benötigte Dauer der Verarbeitung zwischen dem letzten Audio-Chunk und dem jetzigen beschreibt. Wenn also innerhalb von zehn Millisekunden drei Audio-Chunks verarbeitet wurden, besitzt beispielsweise

das erste ein Delta von Drei, das zweite Vier und das dritte nochmal Drei. Zusammengerechnet ergibt sich die verstrichene Zeit der Verarbeitung zu Zehn, was exakt der Gesamtdauer der Verarbeitung entspricht. Wird nun diese Modifikation bei der Erstellung der `EvalAudioChunk`-Objekte angewandt, kann, nicht wie bei der Verwendung von `QTimer`, nun so lange gewartet werden, bis die Summe aller Deltas der angekommen Objekte 500 Millisekunden erreicht. Gegebenenfalls muss auch gar nicht gewartet werden, falls der Audiopuffer schon entsprechend gefüllt ist, und der Verarbeitungsprozess kann ohne Verzögerung fortgeführt werden. Dies würde den Audiopuffer entsprechend schnell abbauen, bis anschließend wieder gewartet werden muss. Es kann also nicht einfach der gesamte Audiopuffer durch `meanPass` überprüft werden, sondern nur so viele Objekte, die auch das Zeitintervall ausmachen. Kommt es nun dazu, dass `listening` auf `true` gesetzt wurde, wird der nächste Wert von `meanPass` des neuen Zeitintervalls nicht mehr gegen $0,5$ geprüft sondern gegen $0,4$, da nun die Toleranz von Sprechfehlern innerhalb einer Sprachsequenz berücksichtigt werden muss. Bei Erreichen wird wie bekannt der betrachtete Bereich des Audiopuffers in den Sprachpuffer kopiert. Bei Nicht-Erreichen jedoch wird ebenfalls der betrachtete Bereich des Audiopuffers in den Sprachpuffer geschrieben und zusätzlich die `listening`-Variable auf `false` gesetzt. Dies markiert das Ende einer Sprachsequenz und die Daten des Sprachpuffers können nun als ganzes Array von `shorts` zum `SpeechModelManager` gesendet werden, um die Sprache-Zu-Text-Funktion von DeepSpeech in Anspruch nehmen zu können. Der Sprachpuffer wird zur Vorbereitung für folgende Sprachsequenzen geleert, was einen Durchlauf des `SpeechListeners` abschließt.

4.2 Sprache-Zu-Text mit DeepSpeech

DeepSpeech wird innerhalb des `SpeechModelManager` verwendet und stellt somit die Schnittstelle der Sprache-Zu-Text-Funktion des `SpeechManagers` dar. In dem UML-Diagramm aus Abbildung 4.1 ist in dieser Klasse die Funktion `feedSpeechData()`. Diese wird innerhalb des `SpeechListeners` aufgerufen, sobald eine Sprachsequenz zum Ende gekommen ist und nun in schriftlichen Text umgewandelt werden soll. Bevor dies allerdings funktioniert, muss DeepSpeech als System erst einmal in den Sprachassistenten eingebunden werden.

DeepSpeech ist quelloffen, somit ist der Quellcode frei zugänglich und kann aufgrund der Mozilla Public Licence 2.0 frei verwendet werden. Dies lässt nun die Möglichkeit zu, dass DeepSpeech mit den Quelldateien selber kompiliert wird, was die Kompilierung von *Tensorflow* mit Hilfe von *Bazel* einschließt. Die zweite Möglichkeit ist jedoch das Herunterladen der vorkompilierten Binärdateien über Mozillas Taskcluster. Beide Varianten liefern die benötigte *Shared-Library* in Form einer `so`-Datei namens `libdeepspeech.so`. Diese muss nun im System installiert werden, damit der Compiler beim Übersetzen des Sprachassistenten die Bibliothek finden kann. Üblicherweise bietet sich unter Linux das Verzeichnis `\usr\lib` für die Installation von Shared-Libraries an, da viele Build-Systeme, wie das in diesem Projekt verwendete *CMake*, standardmäßig in dem Verzeichnis suchen. Wird die

Bibliothek CMake bekannt gemacht, kann sie über ein `#include` eingebunden und verwendet werden. Es ist also keine aufwendige Integration von DeepSpeech nötig, um es in einem eigenen Programm nutzen zu können. Dies war zu Beginn des Projektes jedoch nicht ganz klar. Da auf Grund der noch geringen Kenntnisse über CMake die Bekanntmachung der Datei `libdeepspeech.so` zunächst nicht funktionierte, wurde von einer Inkompatibilität der Binärdatei mit dem aktuellen System ausgegangen. Es wurde angenommen, dass die Binärdateien auf dem gewünschten System neu kompiliert werden mussten. Ein erhöhter Aufwand bestand auch darin, die passenden Versionen von Compiler und Build-System zu finden. Tensorflow verwendet Bazel in der Version 19.2 zum Kompilieren, was allerdings nicht die aktuellste ist. Auch die verwendeten *gcc*- und *g++*-Installationen durften die Version 5.5 nicht übersteigen. Nach der Neukompilierung konnte DeepSpeech erwartungsgemäß in das Projekt eingeführt und dort verwendet werden.

Die Verwendung von DeepSpeech bedeutet die Nutzung des in Kapitel 3.3 trainierten Modells. Dieses wird innerhalb eines `ModelState`-Objektes gespeichert – DeepSpeech ist in C geschrieben, weswegen dies eigentlich eine *Struktur* (engl. *struct*) ist, in C++ allerdings Strukturen in dieser Form nicht mehr existieren. Dieses Objekt wird durch die freie Funktion `DS_CreateModel` erzeugt, welche neben dem verwendeten Alphabet und dem Modell einige zusätzliche Parameter, wie `aNContext` enthält. Dieser gibt das Kontext-Fenster, mit dem das Modell trainiert wurde an. Dabei handelt es sich um einen Bereich am Anfang und Ende einer Audiosequenz, welche dem Modell zugeführt wird [HCC+14]. Die Bedeutung jeglicher anderer Parameter ist nicht bekannt und Standardwerte von [Moz19e] wurden verwendet. Zur Inferenz wird die Funktion `DS_SpeechToText()` verwendet, die ein `ModelState`-Objekt, ein `short`-Array mit den Audiodaten, die Größe des Arrays und die Abtastrate, in der die Audiodaten vorliegen, benötigt. Der Aufruf dieser Funktion ist synchron, also kann der Kontrollfluss erst weiter laufen, sobald die Inferenz abgeschlossen ist. Das kann bei einem großen Netz und je nach Menge der Audiodaten einige Sekunden in Anspruch nehmen, sofern es auf einer CPU gerechnet wird. Dies ist besonders kritisch, da die Anforderungen aus Kapitel **??** eine maximale Verarbeitungsdauer von drei Sekunden vorschreiben. Somit sollte das Netz so klein wie möglich bleiben, sodass dieser Schritt diese Zeitgrenze nicht überschreitet und weitere Zeit für die semantische Sprachverarbeitung übrig bleibt. Außerdem nimmt dieser Schritt die CPU/GPU bei einem großen Netz sehr in Anspruch, wodurch auch eine weitere Anforderung verletzt werden würde, da die Auslastung hierbei einen kritischen Punkt erreicht und einen Leistungseinbruch des restlichen Systems zur Folge hat. Für den `SpeechListener` selbst bedeutet dies auch, dass in dieser Zeit keine weiteren Audiodaten erhoben werden können, wodurch die Performanz des Sprachassistenten stark sinkt. Diese Funktion muss also parallel in einem separaten Thread ausgeführt werden. Um dies zu erreichen, muss der Aufruf der Funktion `feedSpeechData()`, welcher im `SpeechListener` erfolgt, von `DS_SpeechToText()` durch eine `BlockingQueue`, namens `speechDataBuffer`, entkoppelt werden. `feedSpeechData()` schreibt die erhaltenen Audiodaten in `speechDataBuffer` und kann direkt zur Aufruferstelle zurückkehren. Der Thread in `SpeechModelManager` wartet an dieser `BlockingQueue` und führt die Inferenz

durch, sobald die Queue nicht mehr leer ist und Daten vorliegen. Der Text, also das Ergebnis von `DS_SpeechToText()`, muss so gespeichert werden, dass er ebenfalls parallel zugreifbar sein kann. Denn der `SpeechManager` muss eine Schnittstelle für genau diese umgewandelten Texte liefern, damit der Rest des Systems – der Conversation-Manager – darauf zugreifen kann, auch ohne sich sicher sein zu müssen, dass wirklich ein neuer umgewandelter Text vorliegt. Diese Schnittstelle stellt nun eine weitere `BlockingQueue`, namens `inferredTextBuffer`, dar, auf die der `SpeechManager` ebenfalls eine Referenz besitzt, um sie nach außen hin zugreifbar zu machen. Wurde ein neuer Text inferiert, wird er in `inferredTextBuffer` geschrieben. Wartet ein anderer Thread, beispielsweise im Conversation-Manager, durch den `SpeechManager` an dieser `BlockingQueue`, würde sie nun benachrichtigt werden und mit dem neuen Text weiterlaufen können.

Wie schon aus Kapitel 3.3 hervorgeht, ist der umgewandelte Text von DeepSpeech nicht immer vollständig korrekt. Es kann vorkommen, dass einige Wörter nicht richtig geschrieben sind oder es gänzlich falsche sind, jedoch der Rest des Satzes vorwiegend richtig ist. Dies würde bei der semantischen Verarbeitung dieser Texte zu einige Problemen führen, sofern dort keine Toleranzen eingebaut wurden. Es sollte jedoch davon ausgegangen werden, dass möglichst saubere und wohlgeformte Sätze den semantischen Verarbeitungsteil des Sprachassistenten erreichen sollen. Eine Möglichkeit wäre es, den sogenannten `Trie`, der in Kapitel 3.3 für das Training erstellt und benutzt wurde, auch für die Inferenz innerhalb des Sprachassistenten zu verwenden. Dies würde bedeuten, dass möglichst bekannte Wörter eingesetzt und somit Fehlformungen vermieden werden. Dies bedeutet allerdings auch, dass Wörter, welche nicht in dem Trie enthalten sind, mit einer hohen Wahrscheinlichkeit falsch inferiert und durch ähnlich aussehende/klingende Wörter ausgetauscht werden. Die Verwendung eines Tries kann also ebenfalls zu einer unsauberen Inferenz führen, allerdings in viel geringerem Maße als ohne. Somit wäre dies eine gute Variante, um Modelle etwas zu verbessern, die häufig kleine Fehler in den umgewandelten Texten verursachen.

Verbesserungsvorschläge

Falls die semantische Sprachverarbeitung ausschließlich wohlgeformte Texte erwartet, können schon kleine Fehler eine fehlerhafte Verarbeitung bedeuten. Dagegen könnte eine Grammatikprüfung eingesetzt werden, die den Satz auf etwaige Fehler überprüft und sie gegebenenfalls behebt. Eine Rechtschreibprüfung wäre nicht notwendig, da die Verwendung eines Language Models, wie aus Kapitel 3.1, Rechtschreibfehler ausschließt. Bei zu großer Fehlermenge könnte der Text verworfen werden, da er nicht mehr korrigierbar wäre und demnach ohnehin nicht verstanden werden würde. Dies könnte das System performanztechnisch entlasten. Der Benutzer würde dies zwar als negativ empfinden, da der Sprachassistent dann nicht reagiert, jedoch würde es auch weitere Interaktionen vermeiden, die aufgrund von fehlerhaften Daten oder gänzlich unverstandenen Sätzen stattfinden würde.

4.3 Sprachsynthetisierung zur Sprachausgabe

Die letzte Aufgabe des Speech-Managers ist es auch Sprache ausgeben zu können, welche von der Klasse `OutputListener` übernommen wird. Diese Ausgaben stammen von der semantischen Sprachverarbeitung und sind Reaktionen auf die zuvor gehörten Sprachaufnahmen oder Aktionen des Sprachassistenten selbst, um eine Konversation zu starten. Da es also mehrere Arten und Zeitpunkte gibt, an denen eine Sprachausgabe erfolgen soll, ergibt sich natürlicherweise eine gewisse Parallelität. Es kann nicht davon ausgegangen werden, dass immer nur *eine* Quelle versucht etwas auszugeben. Reaktionen und Aktionen des Assistenten können *gleichzeitig*, also parallel, passieren (wie es dazu kommen kann wird in Kapitel 6 näher erläutert). Die Architektur dieses Teils des Speech-Managers muss ebenfalls durch den `SpeechManager` ansprechbar sein. Um eben diese Parallelität zu erreichen, bietet sich, wie für die vorherigen Klassen, eine `BlockingQueue` an, um eine parallele Schnittstelle zu repräsentieren. Es können asynchrone Aufträge, beispielsweise von dem Conversation-Manager, im `SpeechManager` abgelegt werden, welche ein separater Thread dann abarbeitet. Diese Aufgabe implementiert der `OutputListener` aus dem UML-Diagramm in Abbildung 4.1.

Die Ausgabe der Sprache, also die *Sprachsynthetisierung*, soll eine externe Bibliothek durchführen. Es wurden einige dieser Systeme verglichen und auf ihre Natürlichkeit geprüft. Ein wichtiger Faktor eines Sprachassistenten ist es, dass er sich „menschlich“ anfühlt. Dies beinhaltet nicht zuletzt auch die Art, wie sich das Ausgesprochene anhört. Am populärsten unter diesen Systemen war wohl *Festival*, was in der Universität Edinburgh entwickelt wurde [oE19]. Dieses System ist jedoch nur auf die englische Sprache ausgelegt und spricht deutsche Wörter demnach nicht korrekt aus. Abhilfe schafft hierbei das quelloffene Projekt *IMS German Festival* der Universität Stuttgart, was im Grunde eine deutsche Version von Festival ist [Stu18]. Es existieren allerdings keine Binärdateien zum Herunterladen, sodass das Projekt von Hand kompiliert werden muss. Nach erfolgreicher Anfrage per E-Mail können die Quelldateien heruntergeladen und mit Hilfe einiger komplexer Durchführungsvorschriften kompiliert werden, welche unter anderem von den Readme-Dateien des Projektes selbst und von [uBo19] stammen. Allerdings konnten die Binärdateien nicht ohne Fehlermeldungen verwendet werden, was die Nutzung innerhalb des Sprachassistenten unmöglich machte. Somit wurde sich einem Alternativsystem namens *eSpeak* gewidmet, welches standardmäßig die deutsche Sprache unterstützt [en19]. Die benötigten Binärdateien können über den Paket-Manager – in diesem Projekt *apt* – installiert werden. Wie schon in Kapitel 4.1.1 beschrieben, besitzt Qt einige zusätzliche Multimediafunktionen, die auch in diesem Fall von Nutzen sein können. Die Klasse `QTextToSpeech` ist eine Schnittstelle zur Nutzung von eSpeak innerhalb eines C++-Programms. Sie besitzt allerdings nur die Möglichkeit Standardstimmen zu verwenden, welche bereits in eSpeak vorkonfiguriert sind. Diese hören sich allerdings sehr abgehackt und nicht besonders „menschlich“ an. Um dieses Problem zu lösen, hat eSpeak selbst die Funktion andere Stimmmodule als Plugins zu nutzen und so eine deutlich erhöhte Sprachsynthetisierungsqualität zu erreichen. Diese Stimmen

stammen unter anderem von dem MBROLA-Projekt, worunter auch deutsche Stimmen existieren [Dut05]. Da `QTextToSpeech` diese Funktion aber nicht unterstützt, bietet sie sich nicht zur Nutzung an. Stattdessen wird eine *Header-Only-Bibliothek* von eSpeak selbst angeboten, die ebenfalls eine Schnittstelle zum Hauptsystem bietet. Es ist eine Datei namens `speak_lib.h` und wird üblicherweise unter `\usr\include` auf einem Linux-System zur globalen Nutzung installiert, um die Funktionalität durch ein `#include <speak_lib.h>` in das Programm einzubinden, sofern es dem Build-System bekannt gemacht wurde. Zu Beginn der Nutzung muss die Funktion `espeak_Initialize()` mit einigen Parametern aufgerufen werden, um das vorher separat installierte eSpeak-System zu initialisieren. Als nächstes kann ein alternatives Stimmmodul über `espeak_SetVoiceByName` eingesetzt werden, welche auch die MBROLA-Stimmen einschließen. Diese Stimmen sind ebenfalls über den Paket-Manager installierbar und besitzen dann Namen in dem Schema `mb-<Länderkürzel><Nummer>`. In diesem Projekt wurde sich für eine weibliche Stimme entschieden, die den Codenamen `mb-de5` trägt. Zur Sprachsynthetisierung besitzt die Bibliothek eine Funktion namens `espeak_Synth()`. Das Problem ist allerdings, dass dieser Funktionsaufruf ansynchron ist und somit nicht auf das Beenden der Sprachausgabe gewartet wird. Das kann dazu führen, dass bei vielen eingereihten Ausgabeaufträgen mehrere Texte gleichzeitig ausgegeben werden. Bei einem solchen Fall kann der Benutzer natürlich kaum etwas verstehen, weswegen dies unterbunden werden sollte. Für die Synchronisierung existiert allerdings die Funktion `espeak_Synchronize()`, welche den Thread solange blockiert, bis die Sprachausgabe zum Ende gekommen ist, was eine Überlagerung von mehreren Ausgaben vermeidet. Des Weiteren sollte beachtet werden, dass die Lautsprecher, über den die synthetische Sprache ausgegeben wird, häufig sehr nah an dem aufnehmenden Mikrofon liegt. Dieser Umstand führt dazu, dass diese Ausgaben ebenfalls von dem Mikrofon und demnach auch von dem `SpeechListener` verarbeitet werden. Da es sich dabei um menschenähnliche Sprache handelt, lässt das VAD diese häufig durch, was in der späteren semantischen Sprachverarbeitung zu massiven Problemen führen kann. Aus diesem Grund soll um die Sprachausgabe eine Sperre gesetzt werden, die durch den `SpeechManager` verwaltet wird und dieser entsprechende Funktionen zum Sperren und Lesen bereitstellt. Diese Sperre muss von dem `SpeechListener` immer abgefragt werden, bevor ein neuer Audiopuffer vom Mikrofon gelesen wird.

Zur Benutzerfreundlichkeit eines solchen Sprachassistenten gehört nicht nur das saubere Verstehen der Sprachausgabe, sondern auch der Umstand, dass nicht zu viel auf einmal in unterschiedlichen Kontexten gesagt wird. Wie vorher schon angedeutet, können sich mehrere Ausgabeaufträge ansammeln. Diese müssten alle ausgegeben werden, was je nach Anzahl sehr lange dauern kann und sie teilweise auch von unterschiedlichen Quellen mit unterschiedlichen Thematiken stammen können. Dies verspricht eine sehr unangenehme Konversation, die nicht dinglich für den Benutzer sein kann. Somit bietet es sich an nur eine maximale Anzahl von Ausgabeaufträgen zuzulassen. In diesem Projekt wurde diese Obergrenze auf fünf Aufträge begrenzt, was in der Praxis jedoch immer noch eine erhebliche Menge darstellen dürfte.

4.4 Verwaltung des Neuronalen Netzwerks

Aus den Anforderungen in Kapitel **??** geht hervor, dass der Sprachassistent seine Spracherkennungsfähigkeit an den Benutzer anpassen soll. Wenn also beispielsweise ein Wort nicht oder nur falsch erkannt werden kann, soll der Benutzer die Möglichkeit haben, dem Sprachassistenten dieses Wort korrekt beizubringen, oder ihn zu korrigieren. Diese Verbesserung würde die in Kapitel 4.2 genannte Sprache-Zu-Text-Funktion von DeepSpeech betreffen, was von der Klasse `SpeechModelManager` verwaltet wird. Die Fähigkeit zu dieser Funktion erlangt es durch das Training mit Hilfe von Sprachdaten und deren korrekte Transkription, was in Kapitel 3 erläutert wurde. Somit könnte eine Möglichkeit sein, den Sprachdatensatz zu modifizieren und das Sprachmodel, mit welchem DeepSpeech initialisiert wird, mit den neuen Sprachdaten zu trainieren. Dafür wird also eine Sprachsequenz in Form von Audiodaten und dessen Transkription in Text benötigt. Eine entsprechende Schnittstelle ist bereits in dem UML-Diagramm aus Abbildung 4.1 namens `correctData` innerhalb des `SpeechModelManagers` und eine globale Schnittstelle des Speech-Managers in dem `SpeechManager` namens `submitCorrection()`) angegeben. Die Idee ist nun die, dass, sobald ein Wort falsch erkannt wurde, der Benutzer die Möglichkeit hat es zu melden. Tut er dies, würde er dazu aufgefordert werden, eine entsprechende Sprachsequenz aufzunehmen und die schriftliche Transkription zum Gesagten anzugeben. Dieser Schritt müsste eventuell mehrmals in unterschiedlichen Stimmlagen, oder von verschiedenen Stimmen durchgeführt werden, um genügend Sprachdaten für ein erfolgreiches Training zu erhalten. Diese Daten würden nun zum `SpeechModelManager` gelangen, der sie in einem separaten Sprachdatensatz speichert. In bestimmten Intervallen könnte das vorhandene Modell mit den neuen Sprachdaten trainiert werden, sofern welche vorliegen. Diese Technik wurde in diesem Projekt aus Zeitgründen noch nicht implementiert, stellt aber eine gute Möglichkeit dar, das System weiter zu verbessern und die Benutzerfreundlichkeit zu erhöhen, da dies eines der wichtigsten Aspekte dieses Projektes ist. Das Konzept wurde allerdings in einem kleinen Rahmen getestet und dem System erfolgreich ein neues Wort beigebracht, das es vorher nicht verstehen konnte. Dabei wurde eine Trainingmenge aus 20 Äußerungen erstellt, die das Wort in zehn Sätzen an unterschiedlichen Positionen im Satz verwendeten. Jede dieser Sätze wurde zwei Mal in unterschiedlicher Sprechweise aufgenommen, um die Robustheit zu erhöhen. Das Training erfolgte nur mit den neuen Daten. Das Ergebnis war herausragend und es wurde keine Verschlechterung der allgemeinen Spracherkennung außerhalb des neu trainierten Wortes festgestellt. Die genaue Art des Trainings müsste noch weiter erforscht werden, da noch nicht ganz klar ist, ob ein Training ausschließlich mit den *neuen* Sprachdaten, oder eher eines mit allen vorhandenen Daten zusammen zu einem besseren Ergebnis führen würde. Wenn nur mit den berichtigenden Sprachdaten trainiert wird, kann es auf Dauer möglicherweise zu einer *Überanpassung* (engl. *overfitting*) des Modells für die neuen Wörter und Sätze führen und gleichzeitig zu einem Verlust der Fähigkeit altbekannte Wörter zu verstehen. Wird allerdings der gesamte Sprachdatensatz verwendet, benötigt dies sehr viel Rechenleistung und/oder Zeit, um das Training

durchzuführen. Außerdem könnte es sein, dass die berichtigenden Sprachdaten einen zu geringen Einfluss auf die Gesamtperformanz des Modells haben und die Fähigkeit neue Wörter zu verstehen kaum bis gar nicht vorhanden sein wird. Beide vorgestellten Schlimmstfälle tragen weder zur Erreichung des Ziels, noch zur allgemeinen Verbesserung der Benutzerfreundlichkeit bei. Entweder existiert für dieses Problem ein perfekter Mittelweg, sodass beides erzielt werden kann, oder es ist genau diese unlösbare Problematik, welche andere Projekte wie dieses von höherer Popularität abhält.

5

Conversation-Manager

In den vorherigen Kapiteln wurde die Audioverarbeitung und Spracherkennung des Systems erläutert. Diese Teile sind von äußerster Wichtigkeit für das, was sie tun, jedoch benötigen auch sie ein größeres Ganzes, um ihren Zweck zu erfüllen. Ein gehörter oder gesprochener Satz gehört zu einer *Konversation*. Diese muss aufrechterhalten und verwaltet werden. Sie kann einen oder mehrere Kontexte beinhalten und auch kurze Pausen einlegen. Mit diesem Umstand beschäftigt sich das Conversation-Manager-Modul und stellt für das Gesamtsystem eine Art Hauptverwaltung und -schnittstelle dar. Sie kontrolliert den Datenfluss zwischen Modulen und ermöglicht das Beginnen und Beenden einer Konversation.

5.1 Verwaltung einer Konversation

Eine Konversation findet in der Regel zwischen zwei Individuen statt. Für einen Sprachassistenten gilt dies ebenso, wobei es selbst eines der beiden Parteien darstellt. Somit gelten Selbstgespräche – weder von dem Assistenten, noch dem Benutzer – nicht als Konversation im Sinne dieses Projekts. Beginnt sich eine der Gesprächsteilnehmer in einer ansprechenden Weise zu äußern, signalisiert dies den Start einer Konversation. Es muss vorher also definiert werden, was diese „ansprechende Weise“ genau bedeutet, sodass beide Parteien eindeutig kommunizieren können. Bekannte Sprachassistenten lösen dieses Problem mit einem *Aktivierungswort*, welches vor einer Äußerung genannt werden muss, damit der Assistent weiß, dass nun eine Konversation stattfindet. Beispielsweise nutzt der *Google Assistant* die Phrase „Ok Google“ zur Aktivierung. Amazons *Alexa* sieht den eigenen Namen, also „Alexa“, zur Aktivierung vor. Es ist zu erkennen, dass es möglichst *kurze* und *prägnante* Phrasen oder Wörter sind. Sie hängen häufig auch mit dem Produktnamen zusammen, um beispielsweise Fehlaktivierungen zu vermeiden (wobei „Alexa“ wohl keine besonders gute Wahl darstellt, da dies auch der Name einer Person sein kann und unabhängig von dem Sprachassistenten erwähnt werden könnte). Es bietet sich also an, diese Mechanik auch in diesem Projekt anzuwenden und auf ein Aktivierungswort zu warten. Das Wort muss auch an einer vordefinierten Stelle vorkommen, um zu wissen, was alles an Text betrachtet werden soll. Üblicherweise kommt das Aktivierungswort vor Beginn der eigentlichen

Konversation vor, somit würde alles Aufgezeichnete davor ignoriert werden. Die Konversation kann zu diesem Zeitpunkt also erkannt und gestartet werden. Das Ende einer Äußerung kann entweder über ein zweites, also ein *Deaktivierungswort*, erfolgen, oder über eine *semantische Bestimmung*. Ein Deaktivierungswort könnte leicht vergessen werden und zu einer negativen Benutzererfahrung führen. Daher bietet sich die semantische Bestimmung des Endes am ehesten an. Dafür wird ein Verständnis natürlicher Sprache benötigt. Dem widmet sich der *Skill-Manager* und dessen *Skills* (erläutert in Kapitel 6). Diese können also genutzt werden, um zu bestimmen, ob eine Äußerung vollständig und damit zu Ende ist. Bis dahin müsste intern ein Text erfasst werden, der sich aus den Inferenzen des Speech-Managers zusammensetzt und die Gesamtäußerung darstellt. Da nicht bekannt ist, wann genau die Äußerung beendet ist, sollte nach jeder ankommenden Inferenz der interne Text aktualisiert und an den Skill-Manager gesendet werden, um dies zu prüfen. Akzeptiert er die aktuelle Gesamtäußerung, gilt sie als beendet und die semantische Verarbeitung kann begonnen werden. In dieser Zeit werden bisher jedoch weitere Audiodaten aufgezeichnet und inferiert. Es ist also möglich dem Sprachassistenten „dazwischenzureden“. Ist die Verarbeitung durch den Skill-Manager noch nicht abgeschlossen, kann das weitere Hinzufügen von Texten das Ergebnis verfälschen oder sogar die Verarbeitung gänzlich verhindern. Es besteht allerdings auch die Wahrscheinlichkeit, dass es zu einer verfrühten Akzeptanz des Textes durch den Skill-Manager gekommen ist, sodass die eigentliche Äußerung nicht oder nur zum Teil erkannt wurde und somit Informationen zur ordnungsgemäßen Funktionalität fehlen. Dies hängt jedoch stark von der Qualität des Skill-Managers ab, auf die Problematik auch in dem entsprechenden Kapitel näher eingegangen wird. Aus Sicht des Conversation-Managers bietet es sich also an, die weitere Verarbeitung der Inferenzen zu unterbinden, bis der Skill-Manager seine Arbeit beendet hat und auf die korrekte Überprüfung seinerseits zu vertrauen. Die Konversation selbst ist jedoch immer noch nicht beendet. Dies soll der Skill-Manager entscheiden, da nur dieser aufgrund des Verstehens von natürlicher Sprache das Ende erschließen kann. Bis das der Fall ist, soll der Benutzer bei begonnener Konversation auch ohne Nennung des Aktivierungswortes dem System antworten und weitere Kommandos geben können. Dies ermöglicht eine natürlichere und schnellere Art der Kommunikation. Dieses Verfahren kann durch das Aktivitätsdiagramm aus Abbildung 5.1 visualisiert werden.

Eine Implementierung des beschriebenen Verfahrens funktioniert nach dem gleichen Muster wie bei dem `SpeechListener` und `OutputListener` des Speech-Managers. Eine Hilfsklasse namens `InputListener` wartet an dem `inputBuffer` der Klasse `SpeechManager` und reagiert auf einkommende Inferenzen. Ein Klassendiagramm, welches den Conversation-Manager und Skill-Manager samt ihrer Klassen zeigt, ist in Abbildung 5.2 zu sehen. Die einzige Problematik, welche mehr Erläuterung bedarf, ist die Verhinderung des „Dazwischenredens“ während der Verarbeitung durch den Skill-Manager (repräsentativ als hellgraue Aktivität in Abbildung 5.1). Die Lösung ist auch schon zum Teil innerhalb des Diagramms enthalten. Es beschreibt nämlich einen *sequentiellen* Ablauf, selbst durch den Skill-Manager hindurch. Dies suggeriert zunächst, dass der Conversation- und

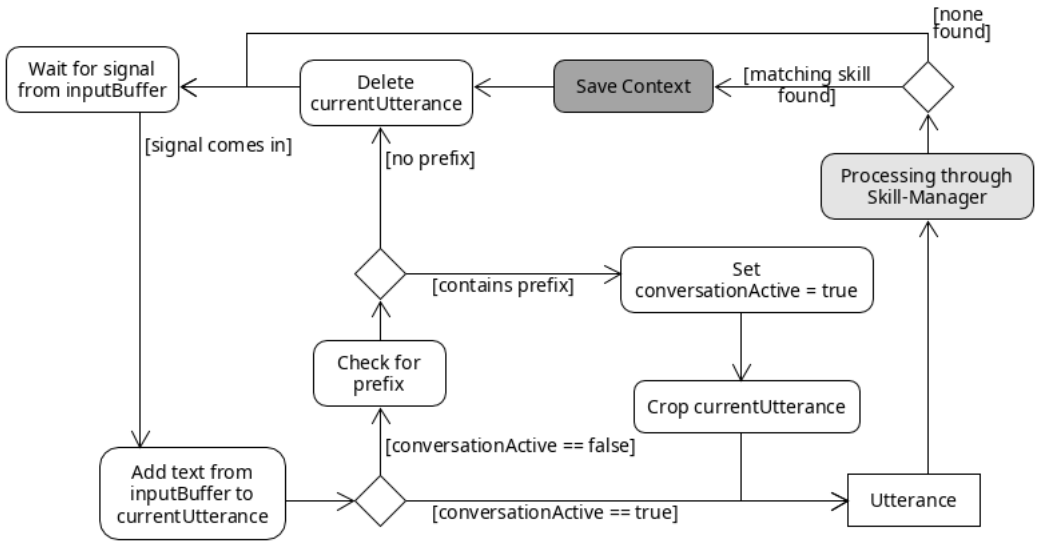

Abb. 5.1. Ein UML-2-Aktivitätsdiagramm, welches das Zusammensetzen der Äußerung und die Verwaltung der Konversation übernimmt.

Skill-Manager nicht durch mehrere Threads getrennt sind. Das bedeutet also, dass die Bearbeitung der von dem Speech-Manager stammenden Inferenzen automatisch unterbrochen wird, solange die Verarbeitung durch den Skill-Manager andauert. Erst mit der Rückkehr des Kontrollflusses zum Conversation-Manager würden eingekommene Inferenzen wieder betrachtet werden. Um das zu erreichen, wird der `InputListener` von `QThread` abgeleitet und stellt somit einen eigenen Thread dar. Dieser wartet, wie oben beschrieben, an der `BlockingQueue` `inputBuffer` und läuft weiter, sobald eine Inferenz ankommt, welche anschließend einer festgelegten Zeichenkette namens `currentUtterance` (*utterance*, dt. *Äußerung*) zugewiesen wird. Der weitere Verlauf wird durch den Zustand der Konversation entschieden. Ist sie gerade aktiv – das Aktivierungswort wurde also bereits genannt – wird der aktuelle Inhalt von `currentUtterance` an den Skill-Manager geleitet. Der Aktivitätsstatus wird anhand einer booleschen Variable namens `conversationActive` abgelesen. Hat die Konversation allerdings noch nicht begonnen, muss nach dem Aktivierungswort, in diesem Modul als `prefix` benannt, innerhalb von `currentUtterance` gesucht werden. Liegt es vor, wird die Aktivierung der Konversation durch das Setzen von `conversationActive` auf `true` signalisiert. Der Text vor und einschließlich des Aktivierungswortes werden von `currentUtterance` entfernt und ebenfalls an den Skill-Manager geleitet. Ist die Konversation nicht aktiv und liegt kein Aktivierungswort vor, wird `currentUtterance` gelöscht und anschließend auf die nächste Inferenz gewartet.

Bis zum Erreichen des Skill-Managers ist die Ausführung sequentiell auf dem `InputListener`-Thread. Innerhalb des Skill-Managers wird nun überprüft, welcher seiner Skills den Text aus `currentUtterance` am besten verarbeiten kann. Wird ein solcher Skill ausgewählt, beauftragt der Skill-Manager diesen mit der Verarbei-

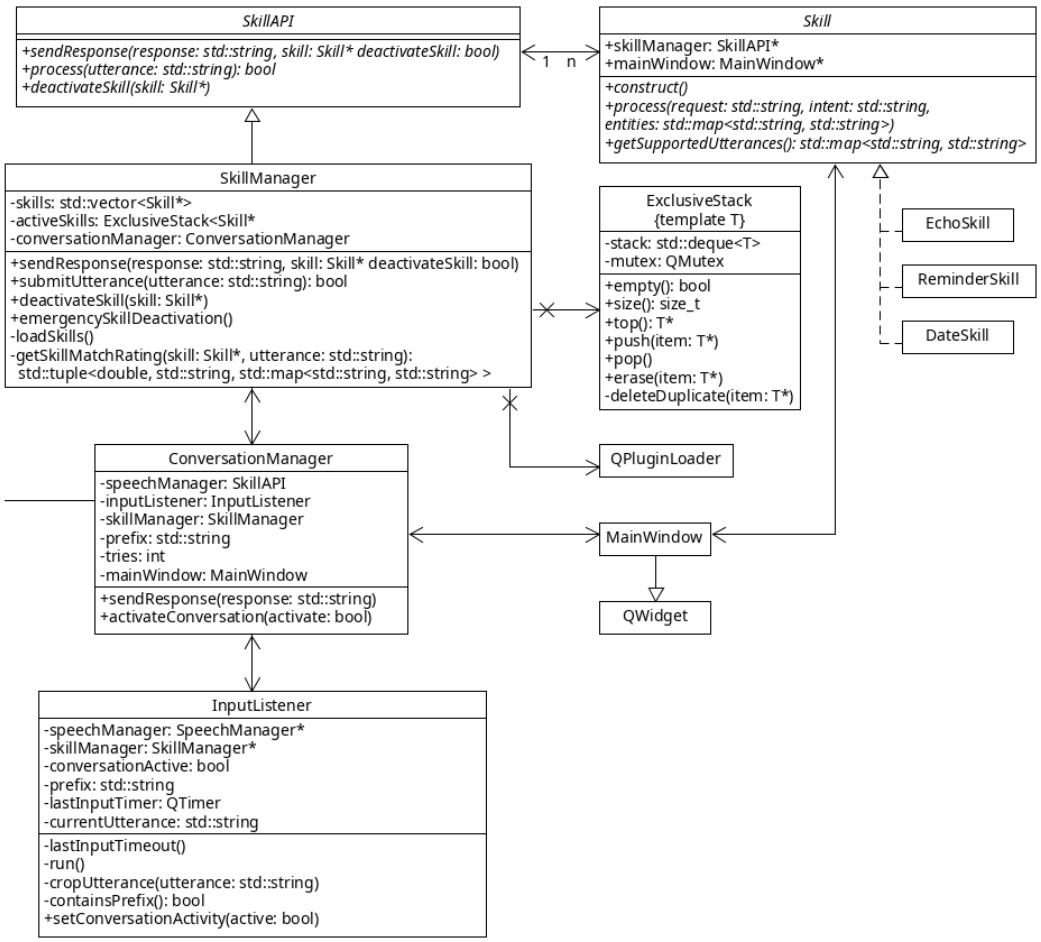

Abb. 5.2. Ein UML-Klassendiagramm, das den Conversation-Manager zusammen mit dem Skill-Manager samt seinen Klassen zeigt. Sie werden gemeinsam gezeigt, da sie eine enge Zusammenarbeit besitzen.

tung des Textes. Dies wird in Kapitel 6 näher erklärt, spielt hier allerdings auch eine entscheidende Rolle. Denn sowohl die Beauftragung als auch der Auftrag selbst wird sequentiell auf dem gleichen Thread wie vorher schon ausgeführt. Demnach kehrt der Kontrollfluss erst nach Beendigung der Verarbeitung durch einen Skill zurück zu dem Aufruf innerhalb des `InputListeners`. Ein Teil dieses Prozesses entscheidet auch, ob die Konversation noch andauert oder nicht. Für dieses Kapitel soll aber davon ausgegangen werden, dass sie nach der erfolgreichen Auswahl eines Skills weiterhin andauert. Somit kann `currentUtterance` gelöscht und auf eine neue Inferenz gewartet werden, da sie bereits verarbeitet wurde. Sollte allerdings kein passender Skill gefunden werden können, wird keine Verarbeitung des Textes durchgeführt und `currentUtterance` wird nicht gelöscht. Stattdessen wird es mit der nächsten einkommenden Inferenz konkateniert und damit die Äußerung weiter aufgebaut. Danach werden die vorherigen Schritte wiederholt, bis ein passender Skill gefunden wurde. In der Abbildung 5.1 ist die dunkelgraue Aktivität *Save*

Context (dt. *speichere Kontext*) zu sehen, auf welche im nächsten Unterkapitel eingegangen und hier erst einmal ignoriert wird. Kommt eine neue Inferenz an, ist `conversationActive` immer noch auf `true` gesetzt. Dies erlaubt dem Benutzer ohne die Nennung des Aktivierungswortes direkt mit dem Sprachassistenten zu kommunizieren, so lange wie die Konversation aktiv ist.

Bisher würde das System für alle Fälle funktionieren, in denen der Skill-Manager die zusammengesetzte Äußerung in `currentUtterance` irgendwann akzeptiert. Es ist jedoch auch möglich, und sogar sehr wahrscheinlich, dass eine Äußerung mit Aktivierungswort auf keinen installierten Skill des Sprachassistenten passt. Es würden immer mehr Teiläußerungen an `currentUtterance` angefügt werden, auch wenn diese schon lange keine Relevanz mehr darstellen. Es fehlt also auch eine Erkennung, ob eine Äußerung akzeptiert werden *kann*. Dieses Problem entspricht etwa dem *Halteproblem*, bei dem herausgefunden werden soll, ob ein beliebiges Programm terminiert, was unmöglich ist. Es kann allerdings davon ausgegangen werden, dass nach einer bestimmten Zeit und Länge von `currentUtterance` keine sinnvolle Bedeutung mehr im Sinne des Sprachassistenten vorliegt. Ansonsten könnte eine Äußerung für immer weiterlaufen und kein Ende finden. Ein Satz kann auch als beendet betrachtet werden, wenn für eine gewisse Zeit keine weiteren Inferenzen mehr ankommen, also nichts mehr gesagt wird. Die Implementierung davon sieht nun so aus, dass ein Timer namens `lastInputTimer` vom Typ `QTimer` ein festgelegtes Intervall wartet, sobald eine neue Inferenz angekommen ist. Kommt im Laufe dieses Intervalls eine Weitere an, wird `lastInputTimer` zurückgesetzt, sodass er das Intervall von Neuem warten muss. Erst wenn dieser ablaufen konnte, wird die Funktion `lastInputTimeout()` aufgerufen, welche die Konversation beendet, indem sie `conversationActive` auf `false` setzt und die Funktion `emergencySkillDeactivation()` aufruft. Letztere dient dazu, den aktuell aktiven Skill zu deaktivieren, falls es einen gibt. Um Kapitel 6 etwas vorwegzunehmen, verwaltet die Klasse `SkillManager` aktuell *aktive Skills*. Sie gelten als aktiviert, wenn sie für die semantische Verarbeitung ausgewählt wurden und sich noch nicht selber deaktiviert haben. Wie oben bereits beschrieben, ist es eigentlich die Aufgabe des Skill-Managers zu entscheiden, ob eine Konversation beendet ist. Da dies allerdings eine Frage der Aufrechterhaltung einer Konversation ist, in welcher der Skill-Manager durch die Inaktivität des Benutzers gar nicht mehr kontaktiert wird, ist die Funktion `emergencySkillDeactivation` dafür da, die *Unwissenheit* des Skill-Managers zu kompensieren. Wenn also die Konversation seitens des Conversation-Managers beendet wurde, muss dies auch für den Skill-Manager gelten. Inwieweit die Konversation nach dem Aufruf dieser Funktion als beendet gilt, wird in Kapitel 6 näher erläutert. Es kann jedoch an dieser Stelle davon ausgegangen werden, dass sie nun zu Ende ist. Jedoch wäre es sehr aggressiv, wenn die Konversation schon bei der ersten nicht verstandenen Äußerung beendet werden würde. Besser wäre es, vor dem Einleiten des oben beschriebenen Schrittes eine bestimmte Menge dieser Versuche zuzulassen. In diesem Projekt wird drei Sekunden lang gewartet, bevor eine Äußerung als nicht verstanden gilt und dies wird für drei Versuche durchgeführt, bevor die Konversation als beendet betrachtet wird. Eine erweiterte Version des Aktivitätsdiagramms aus Abbildung 5.1 könnte

also wie in Abbildung 5.3 aussehen. Um dem Benutzer nun zu signalisieren, dass die

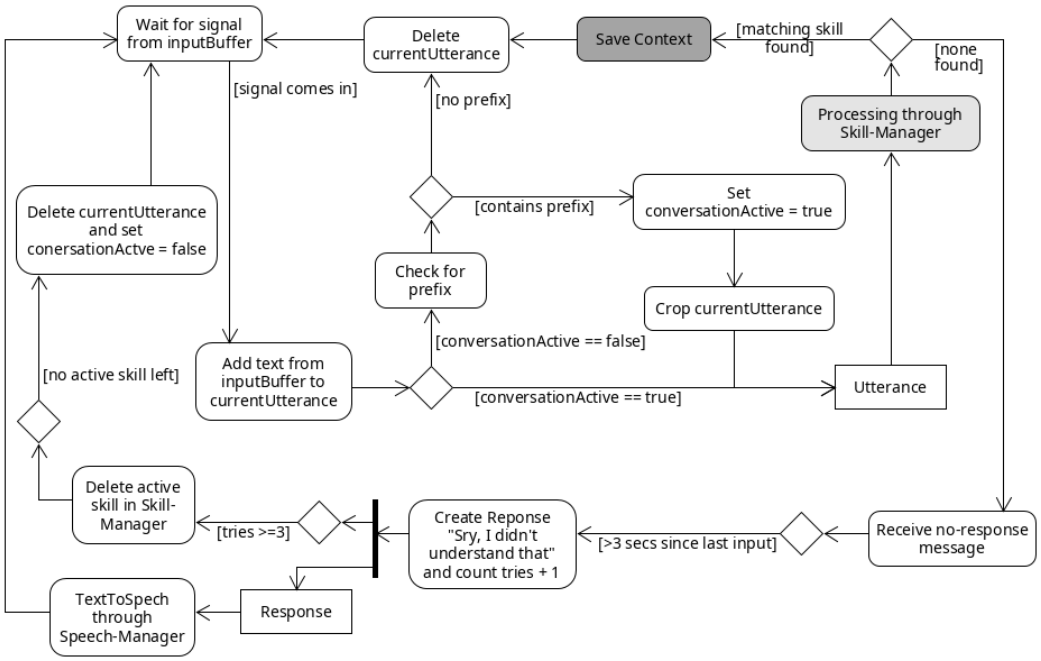

Abb. 5.3. Ein UML-2-Aktivitätsdiagramm wie in Abbildung 5.1, jedoch erweitert um eine zeitlich gesteuerte Beendigung der aktuellen Konversation nach Inaktivität des Benutzers und der Möglichkeit, durch den Skill-Manager eine verfrühte Deaktivierung der Konversation zu erzwingen.

Äußerung nicht verstanden wurde, sendet er mit der Funktion `sendResponse()` eine Antwort an den Speech-Manager zur sprachlichen Ausgabe, in welcher der Benutzer über diesen Umstand in Kenntnis gesetzt wird. `sendResponse()` gilt auch als Schnittstelle für den Skill-Manager, um Nachrichten sprachlich durch den Speech-Manager ausgeben zu lassen. Dafür besitzt der `SpeechManager` die Funktion `submitOutputText`, welche asynchrone Aufträge annimmt.

5.2 Verwaltung von Kontext

In Abbildung 5.1 und 5.3 ist die Aktivität *Save Context* abgebildet, welche bisher jedoch noch keine Erläuterung erhalten hat. Die dunkelgraue Markierung, soll darauf hindeuten, dass sie zurzeit noch nicht implementiert wurde. In diesem Kapitel wird auf die Verwaltung des Kontextes eingegangen und erläutert, wie sie in das bestehende System eingebaut werden könnte.

Kontext ist für eine natürliche Konversation sehr wichtig, damit komplexere, satzübergreifende, oder sogar konversationsübergreifende Themen besprochen werden können. In diesem Fall soll es sich um konversationsübergreifende Themen handeln, die zwischen den einzelnen Fähigkeiten bestehen bleiben sollen. Den

Gesprächsverlauf selbst steuern die Skills des Skill-Managers. Jedoch haben sie kein Verständnis für die Existenz anderer Skills und deren Themenbereiche. Dies kann allerdings durchaus nützlich sein, wenn eine Art *Meta-Skill* entwickelt werden soll, welcher sich mit der Verarbeitung durch andere Skills beschäftigt, anstatt nur seinen eigenen Themenbereich abzudecken. Ein solcher Skill könnte zum Beispiel Statistiken über die Nutzung von Skills führen oder sogar mit anderen zusammenarbeiten und eine Art Skill-Cluster darstellen. In dem jetzigen Stand des Systems ist dies allerdings nicht möglich, da zwar jeder Skill alle Äußerungen mitbekommt, jedoch keine Auskunft darüber erhält, welcher Skill wann ausgewählt wurde. Dies kann zu einigen Sicherheitslücken führen, welche dabei beachtet werden sollten. Jedoch wird für die erstmalige Überlegung eines Kontextsystems dieser Aspekt außer Acht gelassen und ist somit Gegenstand für spätere, eventuelle Erweiterungen.

Eine Implementierung eines Kontextsystems könnte so aussehen, dass zusammen mit akzeptierten Äußerungen der entsprechend ausgewählte Skill in einer limitierten Liste abgespeichert wird. Sie sollte limitiert sein, da es ab einer gewissen Größe des Kontextes keinen Sinn mehr ergibt, sich besonders alte Themen zu merken, da voraussichtlich nicht mehr darauf referenziert wird, oder kein Zusammenhang mehr mit dem aktuellen Kontext besteht. Somit würde eine *Aktualität* aufrecht erhalten werden. Dies würde auch Verwechslungen und Konfusionen vermeiden, falls ein Skill oder Thema häufiger vorkommen sollte. Für das Abspeichern der Skills wird auch sein Name benötigt, um ihn eindeutig identifizieren zu können. Dies ist allerdings nicht ganz einfach zu lösen, da eine manuelle Namensgebung zu Kollisionen bei unabhängigen Entwicklungen führen könnte, jedoch benötigt wird, da Skills sich untereinander identifizieren können sollten, um deren Daten besser zu verarbeiten. Kollidierende Namen könnten durch eine fortlaufende Nummer eindeutig separiert werden, jedoch auch die Identität ändern, wodurch andere Skills sie nicht mehr erkennen könnten. Eine Möglichkeit dieses Problem zu lösen, wäre es eine Plattform zu errichten, auf der Skills als Plugins von beliebigen Entwicklern hochgeladen werden können und die Skills einen Namen besitzen, der den eindeutigen Benutzernamen beinhaltet und somit Eindeutigkeit deutlich einfacher aufrechterhalten werden kann. Die weitere Vermeidung von Kollisionen obliegt dann demjenigen Entwickler, um für den gleichen Benutzernamen unterschiedliche Skill-Namen zu verwenden (beispielsweise `wagnerd_Skill1` und `wagnerd_Skill2` für den Benutzer `wagnerd`). Es wäre selbstverständlich nach wie vor nicht versichert, dass alle möglichen Skills unterschiedliche Namen besitzen, jedoch würde die Herkunft von dieser Plattform eine Art Gütesiegel darstellen, welches eine Kollisionsfreiheit unter Skills gleicher Herkunft sicherstellt.

Eine andere Problematik wäre, welche Äußerungen alle gespeichert werden sollen. Bisher wurden nur diese in Betracht gezogen, die auch von einem Skill akzeptiert wurden. Jedoch könnte es durchaus von Interesse sein, auch diese Äußerungen zu analysieren, um beispielsweise passende Skills empfehlen zu können. Hierbei wären jedoch nur Äußerungen interessant, die innerhalb einer aktiven Konversation stattgefunden haben – nachdem das Aktivierungswort genannt wurde.

5.3 Grafische Benutzeroberfläche

Sprachassistenten existieren in verschiedenen Ausprägungen, auf unterschiedlichen Geräten und haben jeweils andere Funktionen. Jedoch haben alle gemeinsam über einen Audiokanal zu kommunizieren. Manche, wie der Google Assistant oder Siri, besitzen aber die Fähigkeit, Dinge visuell zu präsentieren. In der Regel kommen diese Sprachassistenten auf einem Smartphone oder PC zum Einsatz, bei denen bereits ein Bildschirm vorhanden ist. Da der in diesem Projekt entwickelte Sprachassistenten vorrangig für den PC entwickelt wird, bietet es sich an auch eine grafische Komponente für eine bessere und umfassende Darstellung von Ergebnissen und Prozessen zu ermöglichen.

Eine grafische Benutzeroberfläche (engl. *graphical user interface*, im Folgenden *GUI* genannt) bietet die Möglichkeit, nach einer Frage auditiv eine knappe Erklärung geben zu können und gleichzeitig Grafiken zur Veranschaulichung oder weiterführende Artikel anzuzeigen. Dies würde die Benutzererfahrung deutlich verbessern und diesen Sprachassistenten eventuell sogar von anderen, rein auditiven, abheben. Auch ermöglicht dies die Umsetzung einer Reihe ganz neuer Fähigkeiten des Sprachassistenten, die beispielsweise eine komplexere Eingabe erfordern, für welche die Spracherkennung unzureichend wäre. Dies würde sogar die Berichtigung des Assistenten bei falsch oder gar nicht erkannten Wörtern ermöglichen, welche in Kapitel 4.4 angesprochen wurde.

In diesem Projekt kommt bereits Qt zur Umsetzung einiger Funktionen zum Einsatz, wodurch die Integration einer GUI nahezu natürlich erscheint. In Abbildung 5.2 ist innerhalb einiger Klassen eine Referenz auf ein Objekt des Typs `MainWindow` zu erkennen, welches von `QWidget` abgeleitet wurde. Dies sind die Klassen von Qt zur Erzeugung von Fenstern. Es wäre also denkbar, eine Art *Haupt*-GUI innerhalb des `ConversationManagers` zu verwalten, in dem dann alle Meta-Funktionen enthalten sind, wie beispielsweise eine grafische Variante die Konfigurationsdatei zu bearbeiten, welche in Kapitel 2 angesprochen wurde. So könnten Skills installiert, verwaltet und weitere Funktionen des Sprachassistenten eingestellt werden. Die konkrete Darstellung von Inhalten könnte dann durch die Skills selbst erfolgen, da diese den Konversationsfluss steuern und wissen, wie das aktuelle Thema am besten visuell dargestellt werden kann. Demnach müssten sie ebenfalls Zugriff auf den inneren Darstellungsteil des `MainWindow`-Objekts haben. Es würde sich anbieten, auch eine eigene Schnittstelle für GUI-Elemente anzulegen, damit Entwickler eines Skills diese nutzen können, um das gestalterische Thema des Sprachassistenten zu treffen und ihnen die Entwicklung zu erleichtern. Des Weiteren wäre es äußerst wichtig, eine sichere Trennung der Zugriffsberechtigungen zwischen den Skills zu gewährleisten. Es ist sicherlich nicht im Sinne eines hilfreichen Skills, die GUI eines anderen zu stören oder korrumpieren. Dennoch ist dies eine Gefahr, welche unterbunden werden muss. Eine eindeutige Lösung wurde im Rahmen dieses Projekt nicht erarbeitet, da dafür ein tieferes Verständnis von Qt und dessen Sicherheitskonzepte fehlt. Qt arbeitet mit einem Objektbaum zur Speicherverwaltung, bei dem alle Qt-Objekte vom Typ `QObject` (Grundtyp aller Objekte ähnlich wie in Java) Kindobjekte besitzen, die jeweils immer auf

ihre Elternobjekte zugreifen können [Com19a]. Selbst die Zuweisung eines einzelnen separaten Fensterteils (unabhängig davon, wie dieser aussehen mag) zu einem Skill würde dieses Problem nicht lösen. Da ein Teil eines Fensters ein Kind dessen ist, könnte trotzdem auf das gesamte Fenster und somit auch auf dessen andere Kinder zugegriffen werden. Somit ist eine so einfache Lösung nicht dinglich für die tatsächliche Sicherheit. Ein ambitionierteres Rechtesystem muss also genutzt werden, um dies zu erreichen. Die Vergabe von Rechten ist auch ein wichtiges Thema für die Skills selbst, was in Kapitel 6.3 weiter ausgeführt wird, jedoch auch dort als Gegenstand für weiterführende Entwicklungen betrachtet wird.

6

Skill-Manager

In den vorherigen Kapiteln wurde genau darauf eingegangen, wie Audiodaten verarbeitet, in Text umgewandelt und eine Konversation daraus erkannt und verwaltet werden kann. In diesem Kapitel wird ein Modul beschrieben, wo nun alle Anstrengungen zu einem Ende kommen und womöglich auch der wichtigste Schritt stattfindet: das Verstehen natürlicher Sprache. Dies ist schließlich auch der Kern eines Sprachassistenten, so natürlich wie möglich mit einer echten Person kommunizieren zu können. In diesem Kapitel wird der Skill-Manager und seine Skills beschrieben, wie dieser mit dem Rest des Systems, aber vor allem mit dem Conversation-Manager zusammenarbeitet und in Verbindung steht.

Die semantische Sprachverarbeitung, oder auch *Natural Language Processing* (*NLP*, dt. *Verarbeitung natürlicher Sprache*), kann sehr komplex und tiefgehend sein. Um jedoch die Problematiken dahinter zu verstehen, muss zunächst definiert werden, was natürliche Sprache überhaupt ist und wie sie sich von anderen Sprachtypen unterscheidet. Eine natürliche Sprache entsteht über historische Entwicklungen und wurde nicht „entworfen“ [Lew85]. Sie folgt somit nicht immer festen Regeln und kann teilweise sogar willkürlich sein. Sie stehen somit im Gegensatz zu *formalen Sprachen*, wie beispielsweise Programmiersprachen, welche festen syntaktischen Regeln folgen. Es ist also zu erkennen, dass die Verarbeitung natürlicher Sprache auf keiner statischen Analyse beruhen darf, bei dem von exakten Wortzahlen oder Ähnlichem ausgegangen wird. Ein System, das diese Aufgabe bewältigt, müsste also Informationen über Konversationsstatus und Kontext erhalten, um auch nur die einfachsten Sätze sachgemäß verstehen zu können, was eine tiefe Verflechtung in das Gesamtsystem darstellen würde. Eine Weiterentwicklung dieses Systems wäre äußerst aufwendig und komplex, wodurch sich eine direkte Einprogrammierung in den Sprachassistenten zum Nachteil des Projekts herausstellen dürfte. Wie schon in den vorherigen Kapiteln angedeutet, wird diese Aufgabe modular gelöst. Die Idee hinter dem Skill-Manager ist, dass er, wie der Name schon sagt, sogenannte *Skills* verwaltet, welche dann die konkreten kommunikativen Fähigkeiten des Systems darstellen. Der Grund für diese Modularisierung ist der, dass sie sehr einfach ausgetauscht und gewartet werden können. Dies ergibt die Möglichkeit, Skills nach Belieben des Benutzers zu wechseln und diese in Form von *Plugins* zu installieren. Die Konzeptionierung dieses Systems soll die einzelnen Skills so abgekapselt betrachten wie möglich, damit sie nicht tiefer in das System

integriert werden müssen als nötig. Diese Abkapselung erleichtert außerdem die Entwicklung solcher Skills, wodurch komplexere Fähigkeiten implementiert werden können, ohne die spezifischen Einzelheiten des Sprachassistenten zu kennen. Dies stellt jedoch das Äußere, also die Schnittstelle für und die Verwaltung von den Skills, vor eine große Herausforderung. Die Komplexität des NLP wurde zwar aufgeteilt, muss jedoch von einem der Module durchgeführt werden. Da vorher gesagt wurde, dass die einzelnen Skills so einfach und abgekapselt wie möglich sein sollen, muss also ein Großteil der Verarbeitung anderweitig erfolgen. Jedoch nur in generalisierter Form, da sich die Skills mit dem *Konkreten* auseinandersetzen und das Äußere mit dem *Generellen*. In Abbildung 5.2 sind die Klassen `SkillManager`, `SkillAPI` und `Skill` zu sehen. Dies sind die Hauptbestandteile des Skill-Managers, die in den folgenden Unterkapiteln näher erläutert werden.

6.1 Verwaltung von Skills

Wie bereits in Kapitel 5 beschrieben wurde, ist eine enge Zusammenarbeit mit dem Conversation-Manager unerlässlich, da die Verwaltung einer Konversation auch einen gewissen Teil der semantischen Verarbeitung abdeckt. Die erste Kommunikation zwischen diesen beiden Modulen erfolgt bei der Übergabe der aktuellen Äußerung, die innerhalb des Conversation-Managers zusammengesetzt wurde und ein Teil der derzeitigen Konversation ist. Es soll geprüft werden, ob es einen Skill gibt, der gut auf die Äußerung passt und diese verarbeiten möchte. In diesem Unterkapitel soll unter anderem dieses Problem erläutert und eine mögliche Lösung und dessen Implementierung vorgestellt werden.

Für die Entscheidung, ob ein Skill passt oder nicht, muss erst herausgefunden werden, auf was ein Skill reagieren möchte. Da es sich hier um das Äußere, also die generelle Verarbeitung handelt, dürfen keine Annahmen darüber getroffen und dieses konkrete Detail den Skills selber überlassen werden. Somit wird diese Information von ihnen benötigt. Es stellen sich also die Fragen, wie diese Information aussieht und wie Skills im Nachhinein auf ihre Kompatibilität mit der Äußerung untereinander verglichen werden können. Es reicht nämlich nicht herauszufinden, *ob* ein Skill passt, sondern auch *wie gut*. Schließlich soll der *passendste* Skill gewählt werden. Eine triviale Lösung dieses Systems könnte sein, dass ein Skill ganze zu unterstützende Sätze bereitstellt, auf die er reagieren möchte. Diese stößt allerdings an ihre Grenzen, wenn es sich um unsicheres Wissen innerhalb der zu akzeptierenden Äußerungen handelt. Soll beispielsweise ein Wetter-Skill auf alle möglichen Orte reagieren können, würde dies nur sehr umständlich durch die genannte Lösung funktionieren. Vollständig versagt das System bei beliebigen Sätzen, die aufgenommen werden sollen. Große existierende Sprachassistenten wie Google Assistant haben schon zuvor dieses Problem gelöst. Auch wenn sie im Gesamten nicht quelloffen sind, kann sich eine gewisse Art ihrer Problemlösung erahnen lassen. Konkret der Google Assistant ist ebenfalls sehr modular durch Plugins erweiterbar, wodurch neue Fähigkeiten hinzugefügt werden können – ähnlich wie in diesem Projekt angestrebt wird. Ihr System für die Erstellung dieser Plugins

heißt *Dialogflow* und arbeitet mit der Extraktion von sogenannten *Entities* (dt. *Entitäten*) und ihres dazugehörigen *Intents* (dt. *Intention*) aus aufgenommenen Äußerungen [Goo19][Jan17]. In Abbildung 6.1 ist die Benutzeroberfläche dieses Systems bei der Erstellung einer Intention zu sehen. Das Formular ist in mehrere

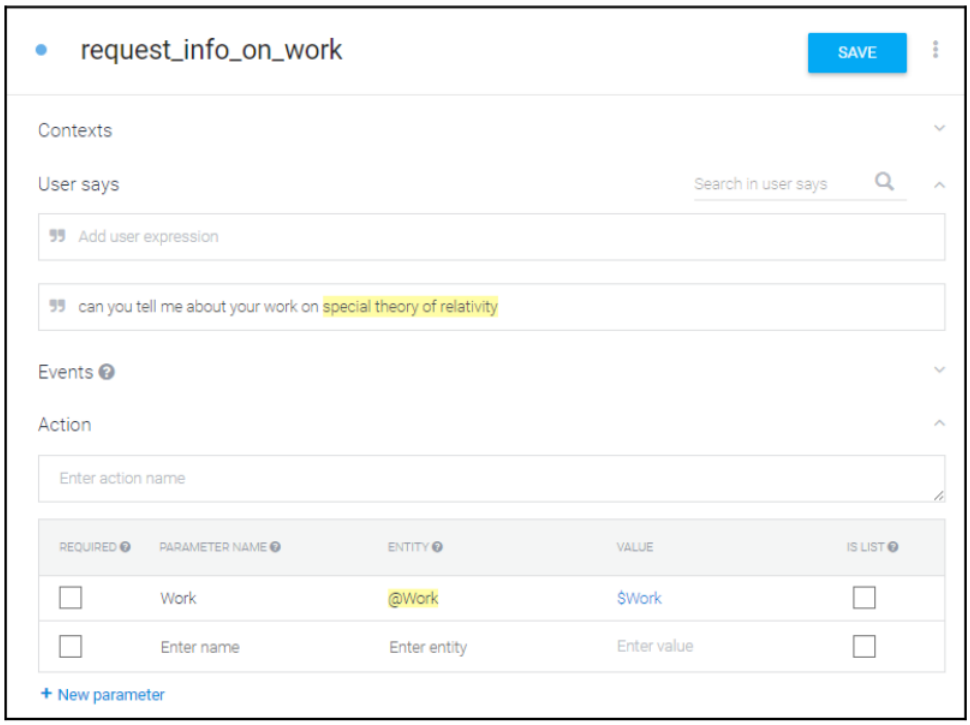

Abb. 6.1. Die Benutzeroberfläche von Dialogflow zur Erstellung eines Intents für ein Plugin [Jan17].

Abschnitte unterteilt, von denen *User says* (dt. *Benutzer sagt*), *Action* (dt. *Aktion*) und *Response* (dt. *Antwort*, jedoch im Bild nicht mehr zu sehen) hier am wichtigsten sind. Bei Ersterem handelt es sich um Äußerungen, die ein Benutzer sagen könnte und für das Plugin von Relevanz sind. Diese beschreiben die Intention des Benutzers. Zweiterer beschäftigt sich mit den oben genannten Entitäten, die eine Information, also unsicheres Wissen, innerhalb einer Äußerung markieren sollen. Dies ist in der Abbildung dargestellt, indem der Text unter *User says* und die dazu passende Entität gelb unterlegt wurden. Schlussendlich können noch Antworten angegeben, welche bei der aktuellen Intention des Benutzers ausgesprochen werden soll. Die Idee hinter diesem System ist nun, dass versucht wird, zu den angegebenen Entitäten die entsprechenden Textstellen innerhalb der Äußerung zu finden. Diese können dann extrahiert und als Informationen zur weiteren Verarbeitung genutzt werden. In Dialogflow, so geht es aus ihrer Vorstellungsseite hervor, wird diese Aufgabe durch maschinelles Lernen gelöst. Dieser Ansatz könnte auch hier Verwendung finden, ist jedoch in dem Zeitrahmen dieses Projektes nicht möglich gewesen. Stattdessen kann dieses Prinzip auf eine textuelle Analyse beschränkt werden. Da in dem ursprünglichen System von Dialogflow keine Informationen

vorliegen, welcher Teil eines Textes welche Bedeutung hat, kann eine textuelle Analyse so nicht funktionieren, da sonst weltliches Wissen notwendig wäre, um zu erkennen, dass es sich bei einer Information wirklich um die gesuchte handelt – dies ist der Grund für die Verwendung von maschinellem Lernen seitens Google. Eine Alternative wäre es anzugeben, *wo* sich eine Entität innerhalb einer Äußerung befinden *kann*, jedoch ohne Angabe, welche konkrete Information an dieser Stelle enthalten ist. Darüber kann nun entschieden werden, welche Satzstruktur eine Äußerung haben soll, um einem Skill zu genügen und gleichzeitig die Extraktion des unsicheren Wissens zu ermöglichen. Des Weiteren spielt hierbei auch die oben genannte Intention eine Rolle. Existiert zu einem Text, welcher mit Entities ausgestattet wurde, eine gewisse Intention, kann das System nicht nur die für einen Skill wichtigen Daten aus einer Äußerung herausfiltern, sondern ihm auch mitteilen, welche Intention laut dem Skill hinter der Äußerung steckt. Somit entfällt sehr viel zusätzliche Analysearbeit, die ein Skill nach der Auswahl hätte durchführen müssen. Innerhalb des Skills könnte so ein Zustandsautomat entwickelt werden, welcher anhand dieser Intentionen die Transitionen durchführt (dies wird in Unterkapitel 6.3 näher erläutert). Es fehlt an dieser Stelle also nur noch das Bewertungssystem, mit welchem entschieden wird, welcher Skill nun am Besten passt und ausgewählt werden sollte.

Implementierung

Für eine programmiertechnische Umsetzung des oben beschriebenen Verfahrens ist vor allem ein Bewertungssystem notwendig. Dies wird allerdings erst in Kapitel 6.2 näher erläutert und soll hier als gegeben betrachtet werden. In Abbildung 6.2 wird eine schematische Realisierung als Aktivitätsdiagramm visuell dargestellt, bei der die Aktivitäten für das Bewerten der Äußerungen und die Verarbeitung durch einen Skill hellgrau unterlegt wurden, da diese an späterer Stelle erläutert werden. In Abbildung 5.2 ist die Funktion `submitUtterance()` zu erkennen. Diese ist die Schnittstelle, welche der Conversation-Manager nutzt, um die Äußerungen des Benutzers auf Semantik zu prüfen. Innerhalb dieser Funktion wird nun gemäß des Aktivitätsdiagramms die Gesamtbewertung aller Skills erhoben und der beste davon ausgewählt, sofern ein passender existiert. Der nächste Schritt beinhaltet die bisher noch nicht besprochene Klasse `ExclusiveStack`, welche ebenfalls in Abbildung 5.2 zu sehen ist. Dabei handelt es sich um einen normalen `Kellerspeicher` (engl. `stack`), der einen exklusiven Bestand an Objekten beibehält. Sollte ein neues Datum hineingegeben werden, wird der Kellerbestand auf Übereinstimmungen geprüft. Existiert das selbe Datum bereits, wird das alte entfernt und das neue oben auf die Datenstruktur gelegt. Dieser Mechanismus wird benötigt, da so die aktiven Skills verwaltet werden. Sie dürfen nur einmal in der Datenstruktur existieren, um nicht fälschlicherweise mehrfach aufgerufen zu werden. Aktive Skills sind diese, die ausgewählt oder etwas von sich aus geäußert haben. Diese Art der Verwaltung ist nötig, da Skills, welche sich in einer Konversation befinden, weiterhin ansprechbar sein sollen. Dafür dient also ein `ExclusiveStack<Skill*>`-Objekt namens `activeSkills`. Die Erhebung aller Skill-Bewertungen ist also nur nötig,

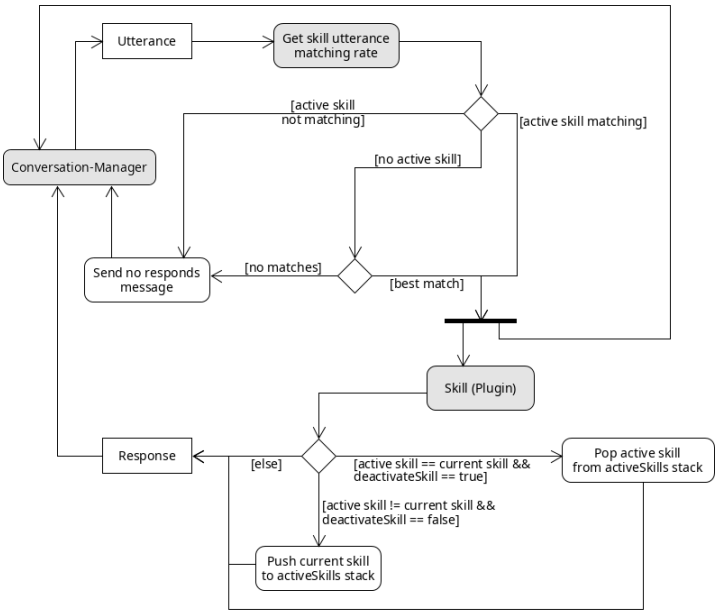

Abb. 6.2. Ein UML-2-Aktivitätsdiagramm, das die Verarbeitung und Verwaltung von Skills zeigt.

wenn es zurzeit keinen aktiven Skill in `activeSkills` gibt, da sonst einfach die Bewertung von diesem errechnet werden kann. Sollte der aktive Skill passen, oder bei nicht existentem aktiven Skill ein passendster gefunden werden, wird der jeweilige Skill mit der semantischen Verarbeitung beauftragt. Dazu wird die Funktion `process()` aus der Schnittstelle `Skill` mit der Äußerung des Benutzers aufgerufen, die jeder Skill implementiert (dies wird in Kapitel 6.3 näher erläutert). Sollte jedoch keine der beiden Fälle zutreffen, wird der Conversation-Manager über diesen Umstand benachrichtigt (der nachfolgende Verlauf innerhalb des Conversation-Managers wurde in Kapitel 5 beschrieben). Der Skill selber hat die Fähigkeit, nach Auswahl oder von sich aus, eine Antwort über die Klasse `SkillManager` über die Funktion `sendResponse()` zu senden (die genauen Umstände, wie dies möglich ist, wird in Kapitel 6.3 erläutert). Sollte ein Skill diese Funktion aufrufen, besitzt er die Möglichkeit sich selber dabei zu deaktivieren. Die Aktivierung ist hierbei der Standardfall. Dies soll den Umgang erleichtern, da bei einer gewünschten Deaktivierung ansonsten zusätzlich die Funktion `deactivateSkill()` aufgerufen werden muss. Anschließend würde eine entsprechende Nachricht über den Conversation-Manager an den Speech-Manager geleitet, um sie auditiv ausgeben zu lassen. Es sollte dabei angemerkt werden, dass dieser Aufruf im Conversation-Manager asynchron zum Ausgeben der Nachricht im Speech-Manager erfolgt. Wenn ein Skill von sich aus eine Nachricht senden möchte, müsste er dies über die Erzeugung eines Threads tun. Der Aufruf von `sendResponse()` kann also durch mehrere Skills gleichzeitig erfolgen und muss demnach synchronisiert werden. Um nicht die gesamte Verarbeitung zu blockieren – die `BlockingQueue`, welche die asynchronen

Aufrufe ermöglichen sind bereits synchronisiert – müssen nur noch Operationen der Klasse `ExclusiveStack` thread-sicher gemacht werden, um *Race Conditions* und demnach Datenverlust zu vermeiden.

6.2 Skill-Bewertung

Durch das Entitätensystem, welches im Vorkapitel angesprochen wurde, kann nun die Kompatibilität errechnet und diese mit denen der anderen Skills verglichen werden. Bei der Errechnung von Kompatibilitäten kann auch von *Bewertung* der Skills gesprochen werden, da sie sich in gewissem Maße versuchen zu qualifizieren. Dafür müssen feste Regeln gelten, was nun für eine positive Bewertung sorgt und was für eine negative. Positive sollen für einen hohen Kompatibilitätsgrad stehen. Die Frage ist nun was sie kompatibel macht. Eine naheliegende Antwort ist natürlich, dass möglichst gleiche Wörter in der Äußerung des Benutzers und in der akzeptierten Äußerung des Skills vorkommen. Denn der Satz „wie ist das wetter heute“ ist deutlich ähnlicher zu dem Satz „wie ist das wetter in trier“ als zu „wie heiß wird es heute“. Es ist schon hier zu erkennen, welchen Nachteil eine rein textuelle Analyse hat. Auch wenn der letzte Satz dem ersten *semantisch* deutlich ähnlicher ist, würde bei einem textuellen Vergleich der zweite eine höhere Bewertung erhalten, da er *textuell* ähnlicher ist. Dieser Umstand muss immer berücksichtigt werden bei der Konzeptionierung dieses Bewertungssystems. Es gibt hierfür unterschiedliche Ansätze, die im Rahmen dieses Projektes erdacht und teilweise auch implementiert wurden, jedoch liegt das Entitätensystem allen zugrunde.

Variante 1

Die erste Variante des Skill-Bewertungssystems zählt in einem Zahlenraum von $(-\infty; 1,0]$, wobei $1,0$ eine perfekte Übereinstimmung beschreibt. Wie bereits erwähnt, spielt die Gleichheit von Wörtern eine entscheidende Rolle. Deswegen wird ein übereinstimmendes Wort mit $1,0$ bewertet. Somit würde „wie heißt du“ bei einer exakt identischen Äußerung eine Bewertung von $3,0$ erhalten, da alle drei Wörter mit der Äußerung übereinstimmen und jedes einen Punkt erhält. Um nun diese Punktzahl unabhängig von der Satzlänge in den Bewertungszahlenraum zu bekommen, wird diese Zahl durch die Anzahl der Wörter in der zu akzeptierenden Äußerung geteilt, wodurch eine Endbewertung von $1,0$ herauskäme. Dieser Fall ist trivial. Es kann jedoch vorkommen, dass es ein Wort zu viel oder zu wenig auf beiden Seiten gibt. Fehlende Wörter können einen erheblichen Bedeutungswechsel bedeuten. Ob nun ein Wort in der Äußerung zu viel ist oder eines in der zu akzeptierenden fehlt. Einige dieser Wörter sind beispielsweise Verneinungen. Die beiden Sätze „ich möchte das kaufen“ und „ich möchte das *nicht* kaufen“ haben vollständig gegensätzliche Bedeutungen trotz ihrer hohen Wortübereinstimmungen. Somit darf ein fehlendes Wort (egal auf welcher Seite) nicht einfach nur *nicht bewertet* werden, sondern einen negativen Einfluss auf die Punktzahl besitzen und somit mit einer $-1,0$ bewertet werden. Auch Wortreihenfolgen spielen eine Rolle, da nicht einfach

nur die Existenz eines übereinstimmenden Wortes positiv bewertet werden darf, sondern nur bei richtiger Position innerhalb der Äußerung. Es ist allerdings schwer nachzuvollziehen, ob Wörter lediglich an der falschen Stelle oder einfach nur falsch sind. Beispielsweise ist „wie geht es dir“ vollständig in „hallo wie geht es dir“ enthalten und lediglich das „hallo“ wäre ein fehlendes Wort. Diese Bewertungsart ist bereits gegeben, jedoch müsse hier erkannt werden, dass die *Ausrichtung* der beiden Sätze um ein Wort verschoben ist, da sonst jedes Wort mit einer $-1,0$ bewertet werden würde. Besonders kritisch wird es, wenn viele übereinstimmende Wörter existieren, jedoch häufig an falschen Stellen. Es muss sich nun darauf geeinigt werden, ab wann zwei Äußerungen als ausgerichtet gelten. Die beiden folgenden Sätze und deren Markierungen verdeutlichen diese Problematik:

(ÄB) „und wie ist *heute* **das** wetter *in* trier **denn** so“
(ÄS) „wie ist **das** wetter *heute* **denn** *in* trier“

Hierbei steht (ÄB) für die Äußerung des Benutzers und (ÄS) für die vom Skill akzeptierte Äußerung. Jede anfängliche Übereinstimmung könnte theoretisch ein Satzanfang darstellen ohne weltliches Wissen anzuwenden. Um diesen Umstand zu vereinfachen, kann das Problem nach dem *Greedy*-Verfahren gelöst werden. Dabei wird immer die erste Übereinstimmung als Satzanfang gewählt, was in dem obigen Beispiel sogar tatsächlich die sinnvollste wäre. Es kann nicht garantiert werden, dass dies immer der Fall ist, es stellt jedoch eine geeignete Alternative dar, da auf Grund der generellen Verarbeitung ohnehin keine Annahme über die Bedeutung der Äußerungen getroffen werden darf, welche auch die spätere Implementierung vereinfacht. Auch wenn als Satzanfang das Wort „wie“ für das obige Beispiel gewählt wird, enthält der jeweils andere Satz viele Wörter, die zwar an anderer Stelle in der Äußerung vorkommen, davor jedoch noch einige andere Wörter existieren, die gegebenenfalls nicht in der anderen Äußerung an dieser Stelle zu finden sind, wie in dem Beispiel das „heute“ und „das“. Sie kommen an gleicher Stelle vor, stimmen nicht überein und ihre jeweiligen Übereinstimmungen kommen in den anderen Äußerungen an anderer Position vor. Also muss auch hier wieder der Teilsatz neu ausgerichtet werden. Dies funktioniert nach dem gleichen Prinzip wie das erste Mal, und somit würde „heute“ als nächstes Ausrichtungswort gewählt werden, da die Äußerung des Benutzers bevorzugt wird. Es ist zu erkennen, dass durch das Ausrichten der Satz auseinandergerissen wird und einige Wörter aus (ÄS) in den Zwischenräumen der übereinstimmenden Wörter verbleiben. Diese werden auch als fehlend angesehen und demnach mit $-1,0$ bewertet. Genauso verhält es sich bei schlichtweg *falschen* Wörtern, die an der gleichen Position wie in der anderen Äußerung vorkommen und nicht ausgerichtet werden können, da eine Übereinstimmung innerhalb der anderen Äußerung an anderer Stelle fehlt. Dies wäre in dem obigem Beispiel bei dem Wort „das“ der Fall. Es kann nicht neu ausgerichtet werden, da das einzige gleiche Wort aus (ÄS) *hinter* einem Ausrichtungswort steht. Somit würde dieses Wort ebenfalls mit einer $-1,0$ bewertet werden, wie auch jedes weitere Wort, da keines von ihnen auf andere Wörter in der anderen Äußerung passt oder ausgerichtet werden kann. Es würde sich somit die Gesamtpunktzahl durch folgende Rechnung ergeben:

(ÄB)	und	wie	ist			heute	das	wetter	in	trier	denn	so
(ÄS)		wie	ist	das	wetter	heute	denn		in	trier		
Berechnung	−1	+1	+1	−1	−1	+1	−1	−1	−1	−1	−1	−1
Punkte	−6											

Es fehlt nur noch die Division durch die Anzahl der Wörter in (ÄB), also Zehn, um somit die vergleichbare Bewertung von $-6/10 = -0,6$ zu erhalten. Das Ergebnis für diese akzeptierte Äußerung ist sehr schlecht, was vermutlich auch mit der subjektiven, menschlichen Bewertung übereinstimmen würde.

Das Entitätensystem hat allerdings noch keine Verwendung gefunden, da nur vollständig definierte akzeptierte Äußerungen angegeben wurden. Wie bereits erläutert, beschreiben Entitäten unsichere und beliebige Information und fungiert als eine Art *Platzhalter*. Dieser muss in irgendeiner Form markiert werden, damit es nicht als normales Wort bewertet wird. Für dieses System wurde das @-Zeichen verwendet. Als nächstes Beispiel werden die folgenden Äußerungen betrachtet:

(ÄB) „wie ist das wetter *am* dienstag“
(ÄS1) „wie ist das wetter *@tag*“
(ÄS2) „wie ist das wetter *@praeposition* @tag“
(ÄS3) „wie ist das wetter am @tag“

Zu beginn wird (ÄS1) betrachtet, um die allgemeine Bepunktung zu erklären. Die ersten vier Wörter passen perfekt zu der in (ÄB) und besitzen somit keinen Erläuterungsbedarf. Ab der Entität *@tag* wird es allerdings interessanter. Zu erkennen, auf welches Wort eine Entität passen soll, ist für eine textuelle Analyse unmöglich, wie in dem Kapitel vorher bereits erläutert. Stattdessen wird die Position für diese Bestimmung verwendet. Die Erkennung des Endes einer Entität ist dabei deutlich aufwendiger, für (ÄS1) jedoch trivial, da es das letzte Element in der Äußerung ist. Somit enthält @tag also den Teilsatz „am dienstag“. Damit hat @tag *zwei* Wörter. Für (ÄS2) existieren jedoch für diesen Teil der Äußerung ebenfalls *zwei* Entitäten und gleicht somit eher der gegebenen Satzstruktur von (ÄB). Es gäbe allerdings die Möglichkeit, dass @praeposition beide Wörter enthält und @tag keines. Dies würde allerdings nicht der Wortaufteilung und den Bedeutungen der Entitäten in (ÄS2) entsprechen. Somit wäre es sinnvoll, dass Wörter auf die existierenden Entitäten aufgeteilt werden. In diesem Beispiel ist dieser Vorgang recht simpel, da jede Entität entsprechend das Wort ihrer Position erhält. Komplexer wird dies allerdings, wenn nach „dienstag“ in (ÄB) mindestens ein weiteres Wort folgt. In diesem Fall wäre die Aufteilung nicht klar, es sei denn, die Wörter wären gleichmäßig auf die Anzahl der Entitäten partitionierbar. Ein solcher Ansatz wäre jedoch sehr unsicher und schlecht bei der Modellierung der akzeptierten Äußerung vorhersehbar. Hier würde sich auch das Greedy-Verfahren anwenden lassen, wodurch bei mehreren, aufeinanderfolgenden Entitäten, jede nur ein Wort erhält. Sollten danach noch Wörter übrig bleiben, werden sie der letzten Entität zugeschrieben. Andersherum, wenn es weniger Wörter als Entitäten gibt, wird das gleiche Prinzip angewandt, nur, dass am Ende welche übrig bleiben, die keine Wörter enthalten. Im Falle von (ÄS3) ist es jedoch vollständig klar, wie

vorgegangen werden muss. Eine weitere Problematik, welche sich mit dem Greedy-Verfahren lösen lässt, ist das Finden eines potentiellen Endes einer Entität. Falls eine passende Nicht-Entitäten folgt, bedeutet dies eine Beendigung der Entität. Da sie jedoch beliebige Wortketten beinhalten können, könnte somit auch der gesamte Satz innerhalb der Entität liegen. Jedoch erwies sich in einigen Tests das Greedy-Verfahren als sehr nützlich, welches in diesem Fall aussagt, dass das erste Wort der Äußerung, welches mit der ersten Nicht-Entität hinter einer Entität übereinstimmt das Ende von dieser bestimmt. Wie beispielsweise bei „*hallo* das ist ein test“ und „*@gruß* das ist ein test“

Die Gegenüberstellung der oben genannten akzeptierten Äußerungen soll zeigen, dass je nach Anzahl und Positionierung der Entitäten trotz ihrer hohen Kompatibilität die einen eindeutiger auf die Äußerung passen als andere. Dies macht die Bepunktung deutlich komplexer. Eine bessere Annäherung der Entitätenzahl an die tatsächliche Anzahl der Wörter entspricht auch einer besseren Kompatibilität, da mehr Ähnlichkeit in der Satzstruktur besteht. Somit müsste (ÄS2) gegenüber (ÄS1) eine bessere Bewertung erhalten. (ÄS3) jedoch besitzt sogar ein fest definiertes Wort, welches mit der Äußerung übereinstimmt, das „am“. (ÄS2) hingegen lässt die Definition des genauen Wortes offen und ist demnach weniger spezifisch. Daher sollte (ÄS3) gegenüber (ÄS2) eine höhere Bewertung erhalten. Aus diesen Feststellungen lassen sich die folgenden Regeln ableiten:

1. Konkrete Wörter sind mehr wert als Entitäten.
2. Mehr Wörter pro Entität wirken sich umgekehrt proportional auf die Wertigkeit aus.

Ein konkretes Wort wird bekannterweise mit einer $1,0$ bewertet, somit müssen Wörter innerhalb einer Entität eine niedrigere Punktzahl erhalten. Aufgrund der zweiten müssen die Wörter absteigend anhand ihrer Position in der Entität bewertet werden. Dies sorgt dafür, dass mehr Wörter in einer Entität eine geringere Gesamtbewertung erhalten, und Entitäten mit jeweils weniger Wörtern eine höhere. Nach diesem Prinzip könnte nun die Wertigkeit eines Entitätenwortes $\frac{1}{p}$ lauten, wobei p die Position innerhalb der Entität beschreibt. Das würde jedoch für das erste Entitätenwort eine Bewertung von $\frac{1}{1} = 1,0$ ergeben, was die erste Regel verletzt. Eine feste Bewertung von $0,75$, die zwischen der eines konkreten Wortes von $1,0$ und der des zweiten Entitätenwortes von $\frac{1}{2} = 0,5$ liegt, stellt somit die Konsistenz dieser beiden Regeln wieder her. Die Formel für deren Berechnung lautet:

$$0,75 + \sum_{i=2}^{n} \frac{1}{i}$$

wobei n für die Anzahl der Entitätswörter steht. Dies begründet folgende Gesamtbewertung:

(ÄB)	wie	ist	das	wetter		am	dienstag
(ÄS1)	wie	ist	das	wetter		@tag	
	+1	+1	+1	+1		+0,75	+0,5
(ÄS2)	wie	ist	das	wetter	@praeposition		@tag
	+1	+1	+1	+1		+0,75	+0,75
(ÄS3)	wie	ist	das	wetter		am	@tag
	+1	+1	+1	+1		+1	+0,75

Der einzige Randfall, der bereits noch nicht bepunktet wurde, ist wenn Entitäten übrig bleiben, die allerdings keine Wörter beinhalten. In diesem Fall ist auch nach einigem Testen nicht ganz klar, wie hier vorgegangen werden soll. Ein Ansatz wäre, sie negativ zu bewerten, da an dieser Stelle ein Wort erwartet wird, jedoch keines existiert und somit ein fehlendes Wort darstellt. Eine andere Argumentation wäre es jedoch, dass Entitäten beliebige Wortketten beinhalten können, also auch gar keine Wörter. Somit wäre es durchaus ein legitimer Zustand einer Entität, keine Wörter zu beinhalten, was einer Negativbewertung entgegen spricht. Ein Kompromiss wäre hier eine Punktzahl von Null zu vergeben, sodass es weder einen negativen noch einen positiven Einfluss hat.

Die Vorteile dieses Systems sind, dass es recht einfach umzusetzen ist und aus wenigen, verständlichen Regeln besteht, was die Modellierung der akzeptierten Äußerungen erleichtert. Letzteres ist zugleich auch ein Nachteil, da dadurch nicht die nötigen Werkzeuge existieren, um die akzeptierten Äußerungen präziser in der Erkennung von Unterschieden zu gestalten. Ein Randfall, der unter Umständen einen besonders großen Nachteil des Systems bedeuten kann, ist, wenn sehr viele Entitäten innerhalb der akzeptierten Äußerung vorkommen oder ein großer Teil der Äußerung zu einer Entität gehört. Dies wird in dem folgenden Beispiel deutlich:

(ÄB)	wiederhole	spiele	musik	ab	
(ÄS1)	wiederhole	@satz			
	+1	+0,75	+0,5	+0,33	2,58
(ÄS2)		spiele	musik	ab	
	−1	+1	+1	+1	2

Obwohl hier offensichtlich (ÄS1) gemeint sein dürfte, stimmt deutlich mehr aus (ÄS2) exakt mit der Äußerung überein. Würde (ÄS2) nun ein passendes Wort mehr besitzen, hätte es mehr Punkte als (ÄS1) erhalten und wäre fälschlicherweise als passendster Skill (zumindest unter diesen beiden) gewählt worden. Es gibt keine Möglichkeiten, die Gewichtung auf konkrete Wörter zu verändern, sodass in diesem Beispiel das Wort „wiederhole“ eine viel höhere Wichtigkeit für die akzeptierte Äußerung darstellt. Dieses Problem wird unter anderem in der nächsten vorgestellten Variante adressiert.

Bei besonders passenden akzeptierten Äußerungen muss natürlich die höchste Kompatibilität gefunden werden, jedoch ist es auch wichtig in Betracht zu ziehen, *ab wann* die Kompatibilität hoch genug ist. Denn nicht alle Äußerungen werden immer gut auf irgendeine akzeptierte passen. In diesen Fällen muss eine Untergrenze festgelegt werden, die eine akzeptierte Äußerung und dessen Bewertung überschreiten muss, um überhaupt als *passend* zu gelten. Dafür existieren viele

Ansätze und es muss auch festgelegt werden, welcher Grad an Passgenauigkeit überhaupt erwartet wird. Ein Ansatz könnte von der gleichmäßigen Aufteilung von Positiv- und Negativbewertungen ausgehen. Dabei würde der Mittelwert von einer solchen gleichmäßigen Aufteilung als Grenzwert verwendet werden, der in diesem Fall $-1 + 1 = 0$ wäre, also einer gleichmäßigen Aufteilung von maximal positiven und negativen Bewertungen. Somit würden akzeptierte Äußerungen als passend gelten, wenn sie mindestens mit der Hälfte der Wörter aus der Äußerung des Benutzers übereinstimmen. Dieser Grenzwert könnte sich auch durch die geringste positive Bewertung begründen lassen: Entitätswörter. Konkrete Wörter bekommen einen vollen Punkt und demnach das Maximum, wobei Entitätswörter mit zunehmender Wortmenge immer mehr an Punkten verlieren, sofern sie sich innerhalb der gleichen Entität aufhalten. Dafür muss ein Extrem hergestellt werden, bei dem eine akzeptierte Äußerung aus nur *einer Entität* besteht und die Äußerung *unendlich viele Wörter* besitzt. Wird nun die obige Formel zur Bewertung von Entitäten auf dieses Szenario angewandt und das Dividieren durch die Anzahl der Wörter innerhalb der Äußerung hinzugefügt, um die Gesamtbewertung zu erhalten, ergibt sich folgende Formel:

$$\lim_{n\to\infty} \frac{0,75 + \sum_{i=2}^{n} \frac{1}{i}}{n} = 0$$

Hier zeigt sich ein Grenzwert von Null, da die Additionen immer kleiner aber der äußere Dividend immer größer wird (in diesem Fall gegen Unendlich strebt). So eine Äußerung müsste auf jeden Fall akzeptiert werden, da alle Wörter innerhalb einer Entität per Definition als passend gelten. An dieser Stelle bleibt jedoch die Frage offen, ob solche eine akzeptierte Äußerung überhaupt geben können sollte. Es wurde davon ausgegangen, dass ein Skill die größtmögliche Freiheit haben sollte und es durchaus auch nützliche Anwendungsbeispiele für eine solche akzeptierte Äußerung geben kann. Dies könnte beispielsweise ein Rückfall-Skill sein, der auf alles, was nicht besser bewertet wurde, eingeht. Aus diesem Grund wird die obige Begründung als valide empfunden und ein Grenzwert von Null zur Mindestkompatibilität verwendet.

Variante 2

Die erste Variante war zwar einfach aber auch gleichzeitig ungeeignet, um sehr feine Abgrenzungen zwischen akzeptierten Äußerungen herzustellen. In dieser Variante wird vorrangig dieses Problem behandelt. Das Entitätensystem und seine Bewertung kann bleiben, wie sie in Variante 1 vorgestellt wurde. Das Problem liegt in der geringen Anzahl an Werkzeugen. Beispielsweise ist eine Entität auch ein Werkzeug und besitzt die zusätzliche Annotation eines @-Symbols. Es gilt also neue Annotationen einzuführen, die weitere Bedeutungen haben, die bei der Spezifikation von akzeptierten Äußerungen und dessen Wörtern helfen.

In dem letzten Beispiel aus Variante 1 wurde festgestellt, dass trotz (aus menschlicher Sicht) offensichtlicher Passgenauigkeit, die Bewertung sehr knapp ausfiel und ausschließlich wegen der geringen Anzahl an Wörtern für die „richtige“ akzeptierte

Äußerung positiv ausfiel. Dies kann bei sensiblen Daten oder komplexer semantischer Verarbeitung sehr kritisch sein. Das Verfahren sollte in allen Situationen vorhersehbar sein. In dem angesprochenen Beispiel ist das Wort „wiederhole“ das Ausschlaggebende, was in der Äußerung die meiste Aufmerksamkeit erhalten sollte. Diese Möglichkeit liefert Variante 1 allerdings nicht. In dieser Variante ist es nun denkbar, eine weitere Annotation für diesen Umstand hinzuzufügen. Als Markierung wurde das *Ausrufungszeichen* gewählt, da es auch semantisch Sinn ergibt, da dieses Wort eine besondere Wichtigkeit darstellt. Es müssen aber auch für diese Annotation Regeln gelten, um die korrekte Verwendung und Bewertung festzulegen. Ein wichtiges Wort innerhalb einer Äußerung sagt aus, dass sie ohne jenes keinen Sinn mehr ergibt und es auch alleine vorkommen könnte, ohne dass der Sinn verloren ginge. Der Satz „spiele !musik ab“ könnte auch ohne Zutun der anderen Wörter „spiele“ und „ab“ verstanden werden. Die Bepunktung ist hierbei ein großes Problem, da eine besondere Wichtigkeit mit dem Bewertungssystem aus Variante 1 nicht möglich ist. Eine leichte Anhebung der Punktzahl für ein *wichtiges Wort* könnte sich zu gering auswirken und eine gebrochene Maximalpunktzahl von über Eins erzeugen. Die Bepunktung für diese wichtigen Wörter muss also im Einklang mit der Maximalpunktzahl liegen und von ihr abgeleitet sein. Es liegt also nahe, den Wertebereich der Gesamtbewertung zu verändern. Die Idee ist, dass ein wichtiges Wort alleine die gesamte Bedeutung der Äußerung trägt und somit auch als solches bewertet werden sollte. Der Wertebereich von Gesamtbewertungen spaltet sich also in diese auf, die *semantische* Kompatibilität besitzen und welche, die nur *textuelle* Kompatibilität erreichen. Ersteres wird durch ein passendes wichtiges Wort erreicht, und Zweiteres durch ein System ähnlich zu dem in Variante 1, bei dem ausschließlich *nicht wichtige Wörter* gezählt werden. Dies impliziert eine besonders hohe Bewertung von Eins bis Zwei und eine niedrige bis normale von Null bis Eins (unpassende akzeptierte Äußerungen im Wertebereich $(-\infty, 0]$ werden hier nicht betrachtet, da sie genauso wie in Variante 1 behandelt werden). Der Sinn dahinter ist, dass eine akzeptierte Äußerung, die ein passendes, wichtiges Wort beinhaltet, bereits eine höhere Kompatibilität erreicht hat als akzeptierte Äußerungen ohne !-Annotationen. Diese erhöhte Kompatibilität ist semantischer Natur und wird auch höher gewertet. Da also ein wichtiges Wort alleine schon in der zweiten Kategorie landen muss, erhält es eine Punktzahl von $n + 1$, wobei n nach wie vor die Anzahl der Wörter in der Äußerung ist. Mit dieser Bewertung würde bereits eine Gesamtbewertung von $\frac{n+1}{n} > 1$ erreicht werden, um sich von einer perfekten, rein *textuellen* Bewertung abzuheben, welche maximal eine $1,0$ erhalten könnte. Es ist jedoch möglich, mehrere !-Annotationen innerhalb einer akzeptierten Äußerung anzugeben. Eine Mehrfachbewertung in der gleichen Weise würde eine maximal mögliche Gesamtbewertung von über Zwei bedeuten, was nach dem neuen Wertebereich nicht zulässig ist. Somit müsste jede weitere passende !-Annotation eine Maximalbewertung von Eins erhalten. Die restlichen Regeln würden allerdings weiterhin auch für dieses Wort gelten. Da dieses Wort allerdings so wichtig für die Bedeutung der gesamten Äußerung ist, würde bei Fehlen eine Bewertung < 0 ausgegeben werden, was einer nicht passenden Äußerung entspricht. Die gleiche Bewertungsmechanik wird auch bei einer Gesamtbewertung von ≤ 1

angewandt, sofern ein passendes, wichtiges Wort enthalten war. Der Grund dafür liegt darin, dass dies einer Bewertung von ≤ 0 bei einer nicht !-annotierten akzeptierten Äußerung entspräche und somit nicht mehr als passend gelte. Diese Regeln stellen nun einen klaren Vorteil gegenüber *nicht wichtigen* Wörtern dar und könnten ausgenutzt werden, um häufiger als passendster Skill zu gelten. Dies ist zwar korrekt, aber nicht im Sinne des Skills, da ein fehlerhaftes Auswählen nicht wirklich genutzt werden kann und sogar die Verarbeitung von weiteren Äußerungen erschwert und den Sprachassistenten unnötig lähmt. Somit funktioniert dieses System nur, wenn jeder Skill die akzeptierten Äußerungen korrekt annotiert, da dies allerdings auch im Sinne eines jeden Skills ist, würde dies automatisch der Fall sein. Um jedoch bei dieser Annotation zu bleiben, soll noch erläutert werden, warum es nicht ausreicht, wichtige Wörter mit dieser einen Annotation zu markieren. Es existieren einige wichtige Wörter, welche *nicht* die gesamte Bedeutung der Äußerung in sich tragen, also nicht als einzelne Wörter verstanden werden können, jedoch ohne sie die Äußerung definitiv nicht passen würde. In dem Satz „wie spät ist es“ könnte kein Wort alleine die gesamte Bedeutung implizieren, jedoch hätte eine Äußerung ohne das Wort „spät“ wohl kaum hohe Ähnlichkeit mit der Ursprungsbedeutung (zumindest nicht aus textueller Sicht). Die nächste Annotation, die diesen abgeänderten Regeln entspricht wird durch ein das &-Zeichen markiert. Die Bepunktung ist eine Mischung aus der !-Annotation und einem normalen Wort wie in Variante 1. Wenn ein solches Wort passt, wird es mit einer Eins bewertet und wenn nicht, dann entspricht die Gesamtbewertung einer nicht passenden Äußerung also < 0. Dies scheint möglicherweise zunächst sinnfrei, da es die eigene akzeptierte Äußerung mehr bestraft als belohnt. Dies ist allerdings auch notwendig, da es nicht nur darauf ankommt, möglichst oft als passend zu gelten, sondern auch bei entsprechender Stelle als *unpassend* zu gelten. Um eine präzise Abgrenzung zwischen akzeptierten Äußerungen herstellen zu können, müssen die einen passen und die anderen explizit nicht. Dies erlaubt eine höhere Kontrolle, da in diesen Fällen die Bewertung direkt abgebrochen werden kann, ohne auf eine besonders niedrige Bewertung *hoffen* zu müssen.

Ein weiteres großes Problem von Variante 1 war die harte Bestrafung von fehlenden Wörtern. Dies konnte vor allem am Satzanfang kritische Folgen haben, da semantisch *unnötige* Füllwörter, welche in der natürlichen Sprache häufig vorkommen, negativ bewertet werden. Abhilfe könnte die neue Regel schaffen, dass genau diese Wörter am Satzanfang der Äußerung nicht zu Minuspunkten führen, sondern nur falsche Wörter bei erfolgreicher Ausrichtung der akzeptierten Äußerung. In dieser Regel steckt allerdings noch ein weiterer Gedanke. Nicht nur am Satzanfang kann es semantisch unnötige Wörter geben, sondern auch mitten in der Äußerung und am Ende. In Variante 1 mussten viele Versionen der gleichen akzeptierten Äußerung angegeben werden, um jegliche Variationen davon zu unterstützen. Auch semantisch unwichtige Wörter konnten negativ bewertet werden, falls sie nicht, an falscher Stelle oder zu viel vorkamen. Mit einer weiteren Annotation sollten Wörter als *optional* angegeben werden. Diese könnten ausschließlich positive oder gar keine Bewertungen erhalten. Entsprechende Wörter können durch eine runde Klammerung als solche markiert werden: „wie (wird) (das) wetter“. Dieses kleine Beispiel

zeigt, dass auch ohne die beiden annotierten Wörter „wird“ und „das“ der Satz verstanden werden kann, diese Wörter allerdings zu einer höheren Kompatibilität beitragen sollen (in Abgrenzung zu einer alleinigen Verwendung von „!wetter“, die einen ähnlichen Effekt erzielen würde). Es können auch mehrere Alternativen innerhalb einer akzeptierten Äußerung angegeben werden: „(wie) (womit) kann ich dir helfen“. Es spielt keine Rolle, ob nun „wie“ oder „womit“ in der Äußerung vorkommt, was eine Speicherplatz sparende Möglichkeit darstellt, anstatt diese beiden akzeptierten Äußerungen separat anzugeben.

Ein Gesamtbeispiel lautet wie folgt:

(ÄB)	wie	ist	der	wikipedia	eintrag	zu	wetter	
(ÄS1)	wie	(wird)	(das)	!wetter	@tag			
	+1	0	0	$-\infty$	+0.75	+0.5	+0.33	< 0
(ÄS2)	wie	(ist)	(der)	!wikipedia	(eintrag)	(zu)	@thema	
	+1	+1	+1	+8	+1	+1	+0.75	$1,96$
(ÄS3)	wie	ist	der	!wikipedia	eintrag	zu	@thema	
	+1	+1	+1	+8	+1	+1	+0.75	$1,96$
(ÄS4)	wie	ist	der	wikipedia	eintrag	zu	@thema	
	+1	+1	+1	+8	+1	+1	+0.75	$0,96$
(ÄS5)	zeige	(mir)	(den)	!wikipedia	(artikel)	(zu)	@thema	
	0	0	0	+8	0	+1	+0.75	$1,25$
(ÄS6)	zeige	mir	den	!wikipedia	artikel	zu	@thema	
	0	0	0	+8	-1	+1	+0.75	$0,82$

Hier wird gezeigt, welchen Einfluss der Einsatz von einzelnen Annotationen auf die Gesamtbewertung haben kann. In (ÄS1) ist eine abgebrochene Bewertung durch ein fehlendes Wort, welches durch eine !-Annotation markiert wurde, zu sehen. Den Unterschied zwischen der Nutzung von optionalen Wörtern zeigen (ÄS2) und (ÄS3). In (ÄS4) wird eine akzeptierte Äußerung ähnlich zu Variante 1 und dessen Bewertung im Vergleich zu denen aus Variante 2 verdeutlicht. In (ÄS5) und (ÄS6) wurden leicht veränderte akzeptierte Äußerungen angegeben, welche die gleichen !- und @-Annotationen besitzen wie (ÄS2) und (ÄS3) und deren unterschiedliche Bewertung aufgezeigt. Dieses Beispiel soll außerdem die Wichtigkeit einer korrekten Annotation aufzeigen, welches eine Schwäche des Bewertungssystem darstellt. Falls es akzeptierte Äußerungen gibt, welche schlichtweg schlecht annotiert sind, leidet das gesamte System darunter, da sonst richtig annotierte akzeptierte Äußerungen nicht die Chance bekommen, für die sie entworfen wurden. Es kann also nicht eindeutig gesagt werden, welche Variante nun per se besser ist. Da Variante 2 mehr Werkzeuge zur Verfügung stellt und bei korrekter Verwendung deutlich bessere Ergebnisse erzielt, wird davon ausgegangen, dass zumindest dieser Ansatz weiter ausgebaut werden sollte.

Implementierung

Die Realisierung in C++ wurde bisher nur von Variante 1 vorgenommen und dient ausschließlich als eine Art *Proof of concept*, da sie noch nicht den gesetzten

Standard an Performanz erreicht. Jedoch soll hier die Grundidee erläutert werden, die auch für eine Implementierung von Variante 2 gelten.

Es ist wohl nicht vermeidbar, für eine Äußerung des Benutzers über jede akzeptierte Äußerung eines Skills zu iterieren. Um diese zu bekommen, wird die Funktion `getSupportedUtterances()` der Schnittstelle `Skill` aufgerufen, welche eine `std::map<std::string, std::string>` zurückgibt. Diese enthält die akzeptierte Äußerung als Schlüssel und die dazu passende Intention als Wert, welche bei der Entwicklung eines Skills wichtig ist. Bei der Analyse der akzeptierten Äußerungen geht es vor allem darum, die Annotationen zu finden. Diese können mit Hilfe eines *regulären Ausdrucks* sehr leicht und effizient gefunden werden. In C++ existiert die Klasse `std::regex`, die einen solchen regulären Ausdruck kapselt. Beispielsweise für das Finden von @-Annotationen könnte diese so aussehen: `std::regex findEntities("@\\w+")`. Dieser Ausdruck sagt aus, dass alle Zeichenketten, die mit einem @-Symbol und mit einem Leerzeichen beginnen, passen. So können die Entitäten beliebige Namen haben. Bei der Iteration über jedes Wort der Äußerung und der akzeptierten Äußerung – was auch annotierte Wörter einschließt – kann durch `std::regex_match` geprüft werden, ob eine gewisse Zeichenkette auf den gegebenen regulären Ausdruck passt. Wenn dies beispielsweise der Fall ist können alle folgenden Wörter zu dieser Entität gezählt werden, bis das nächste Wort aus der akzeptierten Äußerung innerhalb der Benutzeräußerung vorkommt. Um die Inhalte der Entitäten nicht zu verlieren, bietet es sich an, diese samt ihrer Entitäten innerhalb einer `std::map` zu speichern, um sie eindeutig zuordenbar zu machen, um dies dem Skill später übergeben zu können.

Die nächste große Herausforderung ist das Ausrichten der beiden Äußerungen aufeinander. In der Lösung dieses Projektes wurde eine geschachtelte Schleife verwendet, die für jedes Wort in der Benutzeräußerung über die gesamte akzeptierte Äußerung iteriert, bis ein passendes Wort gefunden wurde. Diese Lösung liegt in $\mathcal{O}(n^2)$ und stellt eventuell keine optimale Performanz dar. Für alle weiteren Implementierungsdetails soll an dieser Stelle auf den Quellcode dieses Projekts im Anhang verwiesen werden.

Während der textuellen Analyse können die Punkte direkt vergeben werden. Je nach Implementierung könnte es allerdings von Vorteil sein, die Daten zuerst zu sammeln und hinterher die angegebenen Formeln zur Berechnung der Gesamtpunktzahl anzuwenden.

6.3 Skills

Bei Skills handelt es sich, wie bereits häufig angesprochen, um Plugins, welche dynamisch in den Sprachassistenten installiert und deinstalliert werden sollen. In diesem Kapitel geht es um die Realisierung eines Plugin-Systems innerhalb des Skill-Managers und die Interaktion dessen mit den Skills. Anschließend wird noch erläutert, wie deren Entwicklung aussieht und was dabei beachtet werden soll.

6.3.1 Plugin-System

Plugins sind kompilierte Programme, die kein eigenständiges Programm darstellen und nur von einem bestehenden System genutzt werden um dieses zu erweitern. Genau das sollen auch die Skills in diesem Sprachassistent sein. Ein solches Plugin-System kann je nach Programmiersprache sehr aufwendig sein, da es Metainformationen über den im Plugin enthaltenen Code benötigt. Es muss wissen können, welche Funktionen das Programm implementiert und ob dieses Plugin überhaupt kompatibel mit dem System ist. Die Problematik besteht, da das Plugin nicht in das Hauptprogramm einkompiliert wird. In Java existiert ein Konzept namens *Reflections*, welches genau diese Metainformationen über ein kompiliertes Programm liefern kann. C++ implementiert dieses Konzept allerdings nicht, somit müsste es nachgebaut werden. Die Bibliothek Qt implementiert allerdings in etwa dieses Konzept und stellt auch ein solches Plugin-System zur Verfügung. Die zu verwendende Klasse heißt `QPluginLoader` [Com19c]. Die Nutzung sieht eine Schnittstelle vor, die jedes Plugin implementieren muss. In diesem Projekt ist diese die Schnittstelle `Skill` und ist in Abbildung 5.2 zu sehen. Mit dieser kann der Skill nun angesprochen werden, da diese Funktionen bekannt sind. Mit Hilfe von `QDir` können diese Plugins in bestimmten Verzeichnissen gesucht werden [Com19b]. Diese liegen in Form von *Shared-Libraries* vor, also `.so`-Dateien. Diese können dann durch `QObject* pluginObj = pluginLoader.instance()` geladen werden. Sie liegen allerdings noch als `QObject`-Zeiger vor. Nach einer Prüfung, ob dieses Objekt auch existiert, kann es mithilfe von `qobject_cast<Skill*>` in ein `Skill`-Zeiger gecastet werden. An dieser Stelle soll nochmal erwähnt sein, dass es sich hierbei leider um Heap-Objekte handelt und die Speicherverwaltung nun an den `SkillManager` übergeben wurde. Eine Möglichkeit dieses Problem zu lösen, wäre der Einsatz von `shared_pointer`, die das Heap-Objekt kapselt und die Speicherverwaltung übernimmt. Da die `Skill`-Zeiger jedoch nie ihren Besitzer ändern, ist es kein Problem die Speicherverwaltung selbst zu übernehmen. Abgesehen davon existiert ein weiteres Problem mit den `Skill`-Objekten. Um mit dem `SkillManager` kommunizieren zu können, muss er diesen auch kennen. Da diese Objekte allerdings von dem `QPluginLoader` erzeugt werden, gibt es keine Möglichkeit den Konstruktor aufzurufen und es mit einer `SkillManager`-Referenz zu initialisieren. Wenn also innerhalb des Konstruktors eines `Skill`-Objekts auf die noch uninitialisierte `SkillManager`-Referenz zugegriffen wird, kann es zu Speicherzugriffsfehlern (engl. *segmentation fault*) führen. Aus diesem Grund muss in der `Skill`-Schnittstelle eine Funktion `construct()` existieren, die anstelle des Konstruktors zur Initialisierung verwendet werden soll. Also bevor diese Funktion aufgerufen wird, muss die `SkillManager`-Referenz von außerhalb gesetzt werden. Dafür wurde das Attribut `skillManager` in `Skill` auf `public` gesetzt. Dies spart die Notwendigkeit einer *Setter*-Funktion. Dies ist unkritisch, da der SkillManager die einzige Klasse ist, die auf Skills zugreifen kann und diese Situation auch die einzige ist, in der dieser Zugriff notwendig ist. Danach kann `construct()` aufgerufen werden, um eine sichere Initialisierung anzustoßen. Dieses Laden der Plugins wird durch den `SkillManager` in der Funktion `loadSkills()` durchgeführt. Nach erfolgrei-

chem Laden eines Skills wird es in einem `std::vector<Skill*>` namens `skills` gespeichert.

Plugins können, wie vorher beschrieben, dynamisch geladen oder auch direkt in den Sprachassistenten einkompiliert werden. Dabei wird von eingebetteten Skills gesprochen. Diese Art von Plugins implementieren auch die Schnittstelle `Skill`, werden aber von dem `SkillManager` selbst erzeugt und können sich somit über den Konstruktor initialisieren. Dies könnte beispielsweise für Standard-Skills verwendet werden, damit sie nicht als Shared-Libraries vorliegen müssen. Aber generell wird bei einem Skill von einer Shared-Library, die dynamisch geladen wird, ausgegangen.

6.3.2 Entwicklung eines Skills

Vorbereitung für das Plugin-System

Im vorherigen Kapitel wurde schon vorweggenommen, dass ein Skill eine bestimmte Schnittstelle implementieren muss, um dem ladenden System die Möglichkeit zu geben bekannte Funktionen aufzurufen. Dies ist allerdings nicht die einzige Voraussetzung für die Nutzung des `QPluginLoader`. Das Qt-Plugin-System benötigt einige Meta-Informationen für das Laden des Plugins. Über das Kompiler-Makro `Q_DECLARE_INTERFACE()` müssen innerhalb der zu verwendenden Schnittstelle – hier `Skill` – die Schnittstelle angegeben werden, für die das Plugin gelten soll – in diesem Fall auch `Skill` – und eine beliebige Identifikation: beispielsweise `Q_DECLARE_INTERFACE(Skill, "skill/0.1")`. Diese Identifikation stellt eine Versionskennung für die derzeitige Schnittstelle dar, mit der ein zu entwickelnder Skill kompiliert wird. Beim Laden dieses Skills durch den `QPluginLoader` wird diese Identifikation mit der aktuellen Kennung der angegebenen Schnittstelle geprüft. Stimmen sie nicht überein, wird das Plugin nicht geladen. Dies stellt die Kompatibilität der Plugins mit dem ladenden System sicher. Wenn sich also die Schnittstelle maßgebend verändert, sollte auch diese Identifikation aktualisiert werden, um alte und unpassende Skills zu verweigern.

Der eigentliche Ort, in dem die Meta-Daten gespeichert werden, ist eine *JSON*-Datei, die in das Plugin einkompiliert wird und dabei mit Meta-Informationen beschrieben wird, die der `QPluginLoader` anschließend ausliest. Dies muss allerdings nicht manuell gemacht werden, sondern geschieht bei der Kompilierung automatisch. Um dies allerdings zu erreichen, müssen einige Meta-Informationen innerhalb des konkreten Skill-Codes (nicht die Schnittstelle) angegeben werden. Dazu dienen die Makros `Q_INTERFACES()` und `Q_PLUGIN_METADATA()`. Ersteres gibt die Schnittstelle an, welche das Plugin implementieren wird, und Zweiteres enthält Informationen über die Identifikation der verwendeten Schnittstelle und den Namen der JSON-Datei, in welcher die Meta-Daten gespeichert werden sollen. Es muss nur sichergestellt sein, dass eine JSON-Datei mit dem angegebenen Namen im Projektverzeichnis existiert. Sollte die angegebene Identifikation mit der Schnittstellen-Identifikation nicht übereinstimmen, schlägt eine Kompilierung fehl.

Es steht allerdings noch ein Problem aus, welches bisher noch nicht vorgestellt wurde. Die Schnittstelle `Skill` sieht eine Referenz auf ein `SkillManager`-Objekt vor. Jedoch sollte die konkrete Implementierung und dessen Header nicht

durch einen Skill eingesehen werden dürfen. Dies könnte eine Reihe von Sicherheitslücken hervorrufen und unbefugten Zugriff auf andere Skills ermöglichen. In Abbildung 5.2 existiert die Schnittstelle `SkillAPI`, von dessen Typ auch das `Skill`-Attribut `skillManager` ist. Diese Schnittstelle umfasst alle für ein `Skill`-Objekt relevanten Funktionen aus der Klasse `SkillManager`. Damit die Zuweisung des `SkillManager`-Objekts auf eine `SkillAPI`-Referenz funktioniert, muss der `SkillManager` diese Schnittstelle implementieren. Somit ist sichergestellt, dass ein Skill ausschließlich die vorgesehenen Funktionen aus dem `SkillManager` aufrufen können. Dies vereinfacht auch die Kompilierung, da in dieser Schnittstelle keine konkreten Daten oder weitere Referenzen definiert sind wie in dem `SkillManager` selber. Dies würde bei der Kompilierung des Plugins zu Fehlern führen, da die Referenzen des `SkillManagers` nicht aufgelöst werden können aufgrund des fehlenden restlichen Systems.

Implementierung der Skill-Funktionalität

In den vorherigen Kapiteln wurde der Umgang mit `Skill`-Objekten beschrieben und die einzelnen Funktionen, die ein `Skill`-Objekt implementieren soll. Nun wird erläutert, wie diese Dinge im Sinne einer konkreten Skill-Funktionalität programmiertechnisch umgesetzt werden können.

Die zentrale Funktion eines `Skill`-Objekts ist `process()`. Diese wird aufgerufen, wenn es ausgewählt wurde und eine neue Äußerung zur Verarbeitung erhält. Darüber hinaus werden noch die zugehörigen Entitäten, deren entsprechende Satzteile und die Intention der passendsten akzeptierten Äußerung mitgegeben. Dies ermöglicht eine sehr komfortable Entwicklung, da keine Textanalyse mehr durchgeführt werden muss. Darüber hinaus wird sogar die Intention angegeben, dadurch kann sehr leicht entschieden werden was nun passieren soll. Von hier aus kann der gesamte Gesprächsverlauf von Beginn bis Ende gesteuert werden.

Ein Erinnerungs-Skill könnte durch die komfortable Vorverarbeitung durch den `SkillManager` leicht umgesetzt werden. Zu Beginn wird die Funktion `construct()` implementiert, in der die anfänglichen akzeptierten Äußerungen und deren Intentionen initialisiert werden. Dies könnte beispielsweise durch das Befüllen einer `std::map<std::string, std::string>` passieren. Für dieses Beispiel würden die akzeptierten Äußerungen „erinnere mich in @zeit @zeit_einheit daran @erinnerung“ mit der Intention `zeit_erinnerung` und „sag mir in @zeit @zeit_einheit bescheid“ mit der Intention `zeit` ausreichen (nach Variante 1 des Bewertungssystems). Die Funktion `getSupportedUtterances()` gibt lediglich die initialisierten akzeptierten Äußerungen zurück. Wird dieser Skill aufgrund der Äußerung „erinnere mich in fünf minuten daran loszugehen“ ausgewählt, wird die Funktion `process()` mit der genannten Äußerung, der Intention `zeit_erinnerung` und der Entitäten-Map {`zeit: fünf; zeit_einheit: minuten; erinnerung: los zu gehen`} aufgerufen. Es muss nun geprüft werden, um welche Intention es sich handelt, um das weitere Vorgehen bestimmen zu können. Da diese Information allerdings bereits gegeben ist, ist dieser Schritt trivial. Auf Grund der Intention `zeit_einheit`, muss nach der Zeit, Zeiteinheit und Erinnerung in der Entitäten-

Map gesucht werden. Durch die Information „fünf“ als Zeit, und „minuten“ als Zeiteinheit, kann nun durch simple `if`-Verzweigungen darauf geschlossen werden, dass es sich um eine Wartezeit von fünf Minuten handelt. Dazu wird ein Thread gestartet, der dann die entsprechende Zeit warten soll, bevor er `sendResponse()` mit der entsprechenden Erinnerung als Text aufruft und sich auch direkt deaktiviert, da die Arbeit des Skills an dieser Stelle abgeschlossen ist. In der Zwischenzeit des Wartens gilt der Skill allerdings als aktiviert und würde für die nächsten Äußerungen exklusiv auf Kompatibilität geprüft werden. Da er diese Funktion allerdings in der Wartezeit nicht in Anspruch nehmen soll, deaktiviert er sich durch `deactivateSkill()` selbst. Damit die Wartezeit jedoch abgebrochen werden kann, falls der Benutzer dies wünscht, gibt es die Möglichkeit, die `std::map` der akzeptierten Äußerungen so zu ändern, sodass auf entsprechende Äußerungen des Nutzers reagiert werden kann. Beispielsweise könnten akzeptierte Äußerungen wie „lösche die letzte erinnerung“ mit der Intention `löschen` hinzugefügt werden. Das hat den Effekt, dass nach einer entsprechenden Äußerung wieder dieser Skill gewählt werden würde, nur dieses Mal mit der Intention `löschen`. Dies könnte dann die Beendigung des zuletzt gestarteten Threads nach sich ziehen, was die Erinnerung löschen würde. Eventuell kann eine entsprechende Antwort mit `sendResponse()` ausgegeben werden. Dieses Beispiel sollte zeigen, dass ein nicht trivialer Skill jedoch nahezu trivial zu entwickeln ist.

Ein Zustandsautomat bietet sich für die `Skill`-Entwicklung an. Wird eine solche Struktur beispielsweise objektorientiert implementiert, kann das Übergeben von kontextuellen Informationen an andere Zustandsobjekte notwendig sein. Dieses Problem kann durch ein Kontext-Objekt gelöst werden, indem es eine allen Zustandsobjekten bekannte `std::map` definiert. In ihr können nun Entitäten aus vorherigen Zuständen oder andere kontextuelle Informationen abgelegt werden, die eine mehrstufige Konversation ermöglichen. Für ein detailliertes Beispiel soll an den zugehörigen Programm-Code dieses Projektes verwiesen werden.

7

Zusammenfassung und Ausblick

Aus Kapitel 2 ging hervor, was einen Sprachassistenten ausmacht und welche Teile darin enthalten sind. In Kapitel 2.1 wurde die Idee der verwendeten Software-Architektur vorgestellt und anhand von Beispielen erläutert. Die Architektur ähnelt sehr stark der eines Chatbots, da sich diese beiden Arten von Programmen, bis auf den verwendeten Kanal, in ihrer Funktionsweise sehr ähneln. In Kapitel 2.2 wurde auf die Konfigurierbarkeit des Systems eingegangen. Ein solcher Sprachassistent sollte auch nach Kompilieren individualisierbar sein, sodass weiterhin Einstellungen durch den Endnutzer vorgenommen werden können. Dies erhöht auch die Kontrolle durch den Benutzer und die Transparenz des Systems, da manuell entschieden werden kann, wie der Sprachassistent operiert.

In Kapitel 3 wurde erläutert, was DeepSpeech ist und aus welchem Grund es für dieses Projekt ausgewählt wurde. Weiterhin ging aus Kapitel 3.1 die Funktionalität von Mozillas DeepSpeech hervor, die auf Baidus Deep Speech Research Paper [HCC+14] basiert. In Kapitel 3.2 wurde näher auf die verwendeten Trainingsdaten und deren Vorbereitung zum Training eingegangen. Anschließend wurde in Kapitel 3.3 der konkrete Trainingsvorgang beschrieben und dessen Ergebnisse vorgestellt.

Kapitel 4 stellte erstmalig ein konkretes Implementierungsdetail, den *Speech-Manager*, vor. Das Oberkapitel erläuterte die allgemeine Idee im Zusammenhang des Entwurfs dieses Moduls aus Kapitel 2.1. Darüber hinaus wurde die Bedeutung des Speech-Managers innerhalb der internen Infrastruktur des Sprachassistenten verdeutlicht. Kapitel 4.1 erläuterte die konkrete Implementierung der Echtzeit-Audio-Verarbeitung, den ersten Teil des Speech-Manager-Moduls. Dies gliederte sich auf in die Audioaufnahme und der Vorbereitung aller Audiodaten aus Kapitel 4.1.1 und 4.1.2, in denen aufgezeigt wurde, welche Form die Audiodaten für DeepSpeech und der weiteren Audioverarbeitung aufweisen müssen, und der zweistufigen Implementierung einer *Sprechpausenerkennung* (engl. *Voice Activity Detection* oder kurz *VAD*) aus Kapitel 4.1.3 und 4.1.4, die aus dem ankommenden Audio-Byte-Strom Sprache von Nicht-Sprache unterscheiden muss. Es stellte sich heraus, dass das resultierende Filtersystem in einer ruhigen Umgebung mit leichten Hintergrundgeräuschen gut funktioniert, jedoch bei höherer Störgeräuschdichte die Sprechpausenerkennung eine deutlich schlechtere Performanz aufweist. Dabei existiert noch großes Verbesserungspotential, indem auch hier eventuell neuronale

Netze für diese Klassifizierung eingesetzt werden, um eine einfachere und gleichzeitig bessere Sprechpausenerkennung zu erreichen.

Als, wortwörtlich, mittleres Teilmodul wurde der *Conversation-Manager* in Kapitel 5 vorgestellt und seine Position sowie Hauptaufgaben verdeutlicht und die Idee im Zusammenhang mit dem groben Entwurf aus Kapitel 2.1 konkretisiert. Vorrangig wurde die Funktion als Konversationsverwalter in Kapitel 5.1 vertieft und dessen Funktionalität verknüpfend mit den anderen beiden Teilmodulen erklärt. Zusätzlich wurde der kontextuelle Aspekt eines Sprachassistenten in Kapitel 5.2 aufgegriffen, wobei festgestellt wurde, dass noch keine einfache Lösung für assistentweiten Kontext mit dieser Software-Architektur möglich ist, sich jedoch einige Möglichkeiten zeigten, diese Probleme in Zukunft zu beheben. Als weiteren Ausblick für zukünftige Verbesserungen hat Kapitel 5.3 Aufschluss zur Erweiterung des Assistenten durch eine graphische Benutzeroberfläche gegeben und dessen positiven Effekt auf die Benutzerfreundlichkeit. Dabei würde sich das Projekt an existierenden Sprachassistenten orientieren, die eine ähnliche erweiterte Gesprächsrepräsentation bieten.

In Kapitel 6 wurde die letzte Instanz der Infrastruktur des Sprachassistenten, der *Skill-Manager*, vorgestellt und den Entwurf aus Kapitel 2.1 konkretisiert. In seiner Funktion kümmert er sich um die Verwaltung der Skills, wie in Kapitel 6.1 beschrieben, und deren Bewertung, was aus Kapitel 6.2 hervorgeht. Ein Skill stellt eine konversationelle Fähigkeit des Sprachassistenten in Form eines Plugins dar. Dabei ist es wichtig, den passendsten Skill auszuwählen, der auf die aktuelle Äußerung des Benutzers passt. Das Bewertungssystem stellt dabei den Kern dieser Prozedur dar und wurde in zwei Varianten mit ihren Vor- und Nachteilen vorgestellt. In diesem Projekt ist bisher nur Variante 1 implementiert und detaillierter beschrieben worden, jedoch könnte Variante 2 in Zukunft nach ähnlichem Prinzip in den Sprachassistenten eingebaut werden und somit die Qualität des Systems deutlich steigern. In Kapitel 6.3 wurde die Funktionalität und die konkrete Entwicklung der Skills elaboriert, sowie das zugehörige Plugin-System eingeführt. Es wurde aufgezeigt, dass durch die vorgestellte Architektur Skills leicht zu entwickeln sind und nur oberflächlich mit dem System interagieren, um sie dadurch so einfach wie möglich zu gestalten.

Der in diesem Projekt entstandene Sprachassistent nutzt zur automatischen Transkription menschlicher Sprache innerhalb des Programms ein englisches akustisches Modell zusammen mit einem ebenfalls englischen Language Model, welches durch Mozilla im Rahmen des DeepSpeech-Projektes trainiert wurde. Der Assistent selbst besteht aus drei zusammenarbeitenden Modulen. Die oben genannten Teile ergeben zusammen einen voll funktionsfähigen und durch Plugins erweiterbaren Sprachassistenten. Das Ziel der dezentralen und datenschutzkonformen Verarbeitung der Sprachdaten des Benutzers wurde erreicht. Datenschutz und Sprachassistenz ist also durchaus vereinbar. Probleme existieren jedoch hauptsächlich in der Spracherkennung außerhalb des englischen Bereiches. Dort fehlen viele freie Sprachdaten, die für das Training genutzt werden können, um auch andere Sprachen für diesen Sprachassistenten anbieten zu können.

Die Anforderungen aus dem anhängenden Lastenheft wurden teilweise erfüllt und auch gegebenenfalls praktische Änderungen vorgenommen. Die, die nicht erfüllt wurden, werden hier kurz erläutert und der Grund für die Nichtumsetzung beschrieben.

Muss-Anforderungen

- **/F1.2/: Berichtigung bei Fehlern**
  Diese Anforderung wurde nicht umgesetzt, jedoch ein Umsetzungsvorschlag in Kapitel 4.4 vorgestellt.
- **/F1.3/: Reaktion ohne Präfix**
  Die Reaktion ohne Präfix wurde umgesetzt und in Kapitel 5 vorgestellt, jedoch nicht in der im Lastenheft beschriebenen Form. Es erwies sich als unpraktisch, die Konversation bis zu zehn Sekunden aufrechtzuerhalten, da fälschlicherweise inferierte Wörter – beispielsweise aus Hintergrundgeräuschen – zu einem ungewollten Konversationsfluss führen können. Somit wurde diese Funktion nur auf den Zeitraum beschränkt, in dem ein Skill aktiviert ist.

Soll-Anforderungen

Diese Anforderungen wurden gänzlich vernachlässigt und stattdessen durch die *Kann-Anforderungen* ersetzt, da diese einfacher für die Entwicklung und dennoch ausreichend für Leistungstests und Demonstrationen des Sprachassistenten waren.

Kann-Anforderungen

- **/F3.2/: Timer**
  Ein Timer kann keine Aktion eines anderen Skills durchführen lassen wie im Lastenheft beschrieben. Diese Funktion ist mit der Softwarearchitektur des Sprachassistenten nicht möglich, da der Timer-Skill nichts über die Existenz anderer Skills weiß.

Nicht-funktionale Anforderungen

- **/L1/: Reaktionszeit**
  Diese Erfüllung dieser Anforderung hängt von der Leistung der ausführenden Maschine ab und der Anzahl der installierten Skills. Das Inferieren durch DeepSpeech kann auf einer reinen CPU-Maschine bei langen Eingabesätzen bis zu fünf Sekunden dauern. Viele installierte Skills würden die Laufzeit linear zu der Gesamtzahl der zu akzeptierten Äußerungen aller Skills erhöhen.
- **/L2/: Korrektheit der Spracherkennung**
  Die Performanz von DeepSpeech hängt von der Menge an zur Verfügung stehenden Sprachdaten ab, was in Zukunft deutlich verbessert werden könnte.
- **/L3/: Verbesserung der Spracherkennung**
  Die zugehörige funktionale Anforderung wurde aus Zeitgründen noch nicht erfüllt und kann demnach nicht getestet werden.
- **/L4/: Robustheit der Spracherkennung**
  Siehe /L2/.

Dieses Projekt soll neben der Nutzung als eigenständigen Sprachassistent in Zukunft auch als eine *dezentrale* und lokal ausführbare Alternative zu bekannten *zentral* gesteuerten Sprachverarbeitungs-Backends zu betrachten sein. Dabei bietet sich dies für die Integration von sprachlichen Fähigkeiten in bestehende Projekte, wie beispielsweise bei Robotern, an. Durch die hohe Transparenz, Kontrolle und Erweiterbarkeit durch eigenständig und frei programmierbare Plugins lässt sich dieses Projekt einfach mit anderen Systemen verknüpfen und kann sogar durch sein Betriebssystem-ähnliches Wesen als zentrale Plattform für sprachgesteuerte Systeme fungieren, die auch weit über einen Sprachassistenten hinausgehen. Skills ließen sich beispielsweise durch ein aufgesetztes Rahmenwerk verbinden, um somit eine weitere Abstraktionsebene zu schaffen, die eine interdependente Arbeit zwischen den Plugins leichter ermöglicht und alles als einheitliche, ungetrennte Plattform zu präsentiert. Durch die offene und freie Architektur des Projektes kann fast alles Vorstellbare umgesetzt werden, was eine gute Grundlage für zukünftige Entwicklungen darstellt.

Literaturverzeichnis

But11. Butz, Tilman: *Fouriertransformation für Fußgänger*. Vieweg + Teubner Verlag, Wiesbaden, 2011.

CB19. Chris Biemann, Dirk Schnelle-Walka, Stephan Radeck-Arneth Benjamin Milde: *Open source acoustic models for german distant speech recognition*. http://www.inf.uni-hamburg.de/en/inst/ab/lt/resources/data/acoustic-models.html, 2019.

CO19. Christi Olson, Kelly Kemery: *Voice Report*. Technischer Bericht, Microsoft, https://advertiseonbing-blob.azureedge.net/blob/bingads/media/insight/whitepapers/2019/042019.

Com19a. Company, The Qt: *Object Trees & Ownership*. https://doc.qt.io/archives/qt-4.8/objecttrees.html, 2019.

Com19b. Company, The Qt: *QDir Class*. https://doc.qt.io/qt-5/qdir.html, 2019.

Com19c. Company, The Qt: *QPluginLoader Class*. https://doc.qt.io/qt-5/qpluginloader.html, 2019.

Com19d. Company, The Qt: *Qt*. https://www.qt.io, 2019.

Dut05. Dutoit, Thierry: *The MBROLA Project*. http://tcts.fpms.ac.be/synthesis/mbrola.html, 2005.

en19. ng espeak: *eSpeak NG Text-to-Speech*. https://github.com/espeak-ng/espeak-ng/, 2019.

For10. Formats, Digital: *PCM, Pulse Code Modulation Audio*. https://www.loc.gov/preservation/digital/formats/fdd/fdd000016.shtml, 2010.

Gee19. GeeksForGeeks: *Little and Big Endian Mystery*. https://www.geeksforgeeks.org/little-and-big-endian-mystery/, 2019.

Goo19. Google: *Build natural and rich conversational experiences*. https://dialogflow.com, 2019.

Han10. Han: *Volume from byte array*. https://stackoverflow.com/questions/4362887/volume-from-byte-array, 2010.

HCC+14. Hannun, Awni, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh,

Shubho Sengupta, Adam Coates und Andrew Y. Ng: *Deep Speech: Scaling up end-to-end speech recognition*. 2014.

Hea11. Heafield, Kenneth: *KenLM Language Model Toolkit*. https://kheafield.com/code/kenlm, 2011.

itd15. itdxer: *What is batch size in neural network?* https://stats.stackexchange.com/questions/153531/what-is-batch-size-in-neural-network, 2015.

ITU03. ITU: *Transmission characteristics for telephone band (300-3400 Hz) digital telephones*. http://handle.itu.int/11.1002/1000/6255, 2003.

Jan17. Janarthanam, Srini: *Hands-On Chatbots and Conversational UI Development*. Packt Publishing, 2017.

JRS07. J. Ramírez, J. M. Fórriz und J. C. Segura: *Voice Activity Detection. Fundamentals and Speech Recognition System Robustness*. i-Tech, Vienna, Austria, 2007.

Lan18. Lang, Hans Werner: *Schnelle Fouriertransformation (FFT)*. http://www.inf.fh-flensburg.de/lang/algorithmen/fft/fft.htm, 2018.

Lew85. Lewandowski, Theodor: *Linguistisches Wörterbuch*. Heidelberg; Wiesbaden: Quelle und Meyer, 1985.

MD19. Matt Day, Giles Turner, Natalia Drozdiak: *Amazon Workers Are Listening to What You Tell Alexa*. Bloomberg, 2019. https://www.bloomberg.com/news/articles/2019-04-10/is-anyone-listening-to-you-on-alexa-a-global-team-reviews-audio.

Moz19a. Mozilla: *Common Voice*. https://voice.mozilla.org/de, 2019.

Moz19b. Mozilla: *Common Voice*. https://voice.mozilla.org/de/faq#sentence-collection, 2019.

Moz19c. Mozilla: *DeepSpeech*. https://github.com/mozilla/DeepSpeech, 2019.

Moz19d. Mozilla: *DeepSpeech*. https://github.com/mozilla/DeepSpeech/releases/tag/v0.5.1, 2019.

Moz19e. Mozilla: *DeepSpeech*. https://github.com/mozilla/DeepSpeech/blob/master/native_clie 2019.

Moz19f. Mozilla: *Documentation*. https://docs.taskcluster.net/docs, 2019.

Moz19g. Mozilla: *Tool for creation, manipulation and maintanance of voice corpora*. https://github.com/mozilla/voice-corpus-tool, 2019.

Nat19. NatS: *The Spoken Wikipedia Corpora*. https://nats.gitlab.io/swc/, 2019.

oE19. Edinburgh, The University of: *The Festival Speech Synthesis System*. http://www.cstr.ed.ac.uk/projects/festival/, 2019.

RJB87. Ronald J. Baken, Robert F. Orlikoff: *Clinical Measurements of Speech and Voice*. Taylor and Francis Ltd., 1987.

Sic19. Siciarz, Zbigniew: *Welcome to Aquila!* https://aquila-dsp.org, 2019.

Stu18. Stuttgart, Universität: *Download IMS German Festival*. https://www.ims.uni-stuttgart.de/institut/arbeitsgruppen/phonetik/synthesis/festival_opensource.html, 2018.

uBo19. uBot: *Kompilieren*. https://wiki.ubuntuusers.de/Festival/Kompilieren/, 2019.

yno18. ynop: *Scripts for training Mozilla's DeepSpeech using german speech data*. https://github.com/ynop/deepspeech-german, 2018.

Zul18. Zulkifli, Hafidz: *Understanding Learning Rates and How It Improves Performance in Deep Learning*. https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10, 2018.

A

Erklärung des Kandidaten

☐ Die Arbeit habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen- und Hilfsmittel verwendet.

☐ Die Arbeit wurde als Gruppenarbeit angefertigt. Meine eigene Leistung ist ...

Diesen Teil habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Namen der Mitverfasser: ...

Datum

Unterschrift der Kandidatin / des Kandidaten