Dies ist der Titel der Bachelorarbeit und bitte ohne Rechtschreibfehler

Bachelorarbeit

für die Prüfung zum

Bachelor of Science

des Studiengangs Informatik

an der Dualen Hochschule Baden-Württemberg Heidenheim

von

Michael Lustig

Septemper 2017

Bearbeitungszeitraum	12 Wochen
Matrikelnummer, Kurs	1234510, TINF2014MI
Ausbildungsbetrieb	Firma GmbH, Firmenort
Erstgutachter	Dipl.-Ing. (FH) Peter Pan
Zweitgutachter	Prof. Dr. Rolf Assfalg

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einführung

Wir leben in einer Zeit, in der der Einsatz von Technik im Alltag nicht mehr weg zu denken ist. Die Menschheit ist so stark vom technischen Fortschritt und den daraus resultierenden Entwicklungen geprägt wie nie zuvor. Technologien wie Autos, zu deren Betrieb eine körperliche Interaktion nicht mehr von Nöten ist, wurden vor wenigen Jahren noch als Science Fiction klassifiziert und sind trotz allem heute realisierbar. Auch In Fabriken werden selbst komplexere, erfolgsentscheidende Tätigkeiten von Robotern umgesetzt und zu Hause gehört das eigenständige Staubsaugen der Vergangenheit an, weil dies eine programmierte Maschine zuverlässig übernehmen kann.
Neue Technologien wirken sich in vielen Alltagssituationen bereichernd und erleichternd auf unser Leben aus. Ein hiervon stark betroffener Wirtschaftssektor ist das wissens- und technologiegetriebene Gesundheitswesen. Der medizintechnische Fortschritt, der sich in den letzten Jahrzehnten ereignet hat, ermöglicht weitaus zielführendere Diagnose- und Therapiemöglichkeiten als sie früher vorstellbar waren. So lassen sich beispielsweise in der diagnostischen Fachabteilung Radiologie mit dem Einsatz neuer Technologien, wie Computer- und Kernspintomographien, aussagekräftigere Befunde erzielen als sie ein Pathologe früher durch einen händischen Eingriff hätte anfertigen können [Buc13, S.3].
Die Statistiken des statistischen Bundesamtes zur durchschnittlichen Lebenserwartung der Bevölkerung Deutschlands zeigen einen stetigen Aufschwung im Verlauf der vergangenen Jahrzehnte und lassen daher auf einen erheblichen Anstieg der allgemeinen Patientenversorgungsqualität im Gesundheitswesen aufgrund der Entwicklung IT-basierter Medizintechnologien schließen.
Diese Tatsache gilt als ausschlaggebender Beweggrund dafür, die Studienarbeit im Rahmen eines medizininformatischen Entwicklungsprojektes zu absolvieren. Der Fokus ist hierbei auf Menschen gerichtet, die von Hörschädigungen betroffen sind und tagtäglich mit ihren schwerwiegenden Folgen konfrontiert werden. Soziale Integrationsschwierigkeiten und psychische Beeinträchtigungen bilden nur das Grundgerüst der gängigen Symptomatik. Das Ziel des Projekts besteht darin, den oben genannten Belastungen durch Hörstörungen mithilfe eines IT-Produktes entgegenzuwirken.
Als Entwicklungsgrundlage dient eine Technologie mit dem Namen Augmented Reality (zu Deutsch: Erweiterte Realität). Darunter lässt sich eine Variation der Ursprungstechnologie Virtual Reality verstehen, mit deren Hilfe Brillen entwickelt werden können, die dem User optisch das Gefühl vermitteln sich in einer virtuellen Welt zu befinden. Die reale Welt um ihn herum wird dabei vollständig ausgeblendet und ist für ihn somit nicht übersehbar.
Augmented Reality (kurz: AR) sah dies als Schwachstelle und versuchte dem entgegen-

zuwirken. Durch AR lassen sich digitale Daten oder computergenerierte Informationen auf Brillen visualisieren und in die reale Welt integrieren. Gegenüber einer VR-Brille ist die reale Welt Teil der AR-Technologie und somit für den User sichtbar. AR erlaubt die Erfassung und Verarbeitung von Daten, beispielsweise die Aufzeichnung und Analyse eines gesehenen Bildes, und ermöglicht die Projizierung der Ergebnisse auf die Brillengläser [KR12]. Gleichermaßen lässt sich diese Konvertierung auch in die entgegengesetzte Richtung vornehmen, wie es auch im hier behandelten Studienprojekt verwendet wird:
Der von einem Mobiltelefon aufgenommene Ton wird auf das Vorkommen von Sprachbausteinen untersucht und die erkannte Sprache in Text konvertiert. Der Text wird auf einer AR-Brille ausgegeben, die in Verbindung mit dem Mobiltelefon steht. So kann ein Mensch auch mit Hörschädigungen einem Vortrag oder einer Unterrichtsstunde akustisch und visuell problemlos folgen. Auch das kommunizieren mit Menschen, die der Gebärdensprache nicht mächtig sind, wird vereinfacht, da diese einfach in das Mikrofon des Mobiltelefons sprechen müssen. Durch die Integration des Konvertierers in ein Mobiltelefon, wäre es sogar möglich beim Telefonat die Stimme des Anrufers direkt zu verarbeiten und dem Hörgeschädigten live als Text auf die AR-Brillengläser zu projizieren.

2 Struktur

Die Applikation ist untergliedert in zwei Hauptbestandteile. Einen Recorder, welcher die Tonaufnahme durchführt und die erkannte Sprache in Text umwandelt, und ein Printer, welcher sich zu einer Augmenter Reality Brille verbindet und den vom Recorder erkannten Text forformatiert und darauf ausgibt. Das Bindeglied zwischen diesen beiden Komponenten ist das User Interface mit Klassenname MainActivity. Die MainActivity wird durch Usereingaben gesteuert. Sie erstellt bei bedarf neue Recorder oder Printer Instanzen, startet beziehungsweise stoppt die Aufnahme bzw die AR-Ausgabe und informiert den AR-Printer über neu verfügbare Ausgabetexte. Diese drei Komponenten werden im folgenden näher erleutert.

2.1 Main Activity

Wie bereits erwähnt ist die MainActivity das Zentrum der Applikation, ein UML-Klassendiagramm ist in XXX abgebildet.
Die bestandteile der MainActivity:

1. Ein Spinner zur Wahl einer verfügbaren Sprachkonvertierungsoption und ein beschreibendes Textfeld
2. Ein Spinner zur Wahl eines verfügbaren AR-Printers und ein beschreibendes Textfeld
3. Ein Button zum Starten bzw. Stoppens einer Konvertierung
4. Ein SideView Menü, welches Einstellungsmöglichkeiten enthält wie zum Beispiel das Festlegen der gesprochenen Sprache
5. Eine Recorder Instanz vom Typ IRecorder. IRecorder ist ein interface, welches von allen Recorder-Implementierungen verwendet wird, um eine einheitlich API zu gewährleisten. Dieses Interface wird im Abschnitt XXX näher erleutert.
6. Eine AR-Printer Instanz vom Typ IPrinter. Iprinter ist ebenfalls ein Interface, welches von allen AR-Printern zur Gewährleistung einer einheitlichen API implementiert wird und wird in Abschnitt XXX näher erleutert.
7. Eine boolsche Hilfsvariable, die Auskunft darüber gibt, ob aktuell ein Konvertierungsvorgang im Gange ist.

8. Zwei Integer-Variablen, die Auskunft über den aktuell gewählten Recorder bzw Printer geben.

Zu 1. Und 2.:
Die beiden Spinner, enthalten alle aktuell supporteten Printer bzw Recorder. Der Inhalt der Spinner wird dynamisch genereiert, hierbei spielt die Klasse Constants eine tragende Rolle.
!!Genaueres über die initialisierung eines spinner arrays!!
Jedem der beiden Spinner ist eine OnItemSelectListener Instanz zugeordnet. Der OnItemSelectListener, ruft im Falle des Printer Spinners die Methode setPrinterMode(int mode, String description) und im Falle des Recorder Spinners die Methode setRecorderMode(int mode, String description) auf. Diese beiden Methoden initialisieren den gewünschten AR-Printer beziehungsweise den Recorder/Converter. Die Funktionalität dieser Methoden wird XXXX näher beschrieben.
Zu 3.:
Der OnClickListener der Buttons fragt die Hilfsvariable (7) ab und ruft abhängig von deren Wert die Methoden IRecorder.startRecording() und IPrinter.startPrinting() oder die Methode notifyRecorderStop() der eigenen Klasse auf.
4. Menü: XXXX

Methoden der Klasse MainActivity:

onCreate() und initialize_components():
In der onCreate()-Methode, welche beim start der Applikation ausgeführt wird, werden zunächst all Ihre bestandteile initialisiert durch die initialize_components()-Methode.
setPrinterMode(int mode, String description)
Diese Methode prüft zuerst, ob im Moment ein Konvertierungsvorgang im Gange ist, durch das abfragen der boolschen Hilfsvariable isRecording. Im Falle einer laufenden Konvertierung wird eine Meldung ausgegeben, die den User darauf hinweißt, dass der AR-Printer während einer Konvertierung nicht gewechselt werden kann. Sonst wird nichts weiter unternommen. Ist keine Konvertierung im Gange wird überprüft, ob der aktuelle Wert in der Hilfsvariable PRINTER_MODE mit dem Wert des ausgewählten Printers in der übergebenen Variable mode übereinstimmt, denn auch dann muss keine weitere Aktion ausgeführt werden.
Wurde ein anderer Printer gewählt, als der gerade verwendete, wird die Methode generatePrinter(int mode, String description) der Klasse PrinterFactory aufgerufen. Die Methode generatePrinter prüft durch ein Switch Statement, welcher ARPrinter vom User gewünscht ist und gibt eine Instanz des gewünschten Printers and die aufrufende Methode zurück. Ist der übergebene Integer-Wert der Factory unbekannt, wird standardmäßig eine Instanz

der Klasse TextPrinter erzeugt. Dieser erzeugt ein TextFeld in der MainActivity, in dem Speech to Text Conversion Ergebnisse ausgegeben werden.
setRecorderMode(int mode, String description):
Diese Methode hat den gleichen Ablauf, wie die Methode setPrinterMode(), mit dem einzigen Unterschied, dass die Methode generateRecorder der Klasse RecorderFactory aufgerufen wird, welche eine IRecorder Instanz zurück gibt. Auch die Recorder Factory hat für unbekannte Eingaben einen Standardwert. Sie gibt einen TextFieldRecorder zurück, welcher ebenfalls ein Textfeld erzeugt und die Eingaben and den gewählten AR-Printer sendet. Dieser Recorder verfehlt zwar den Sinn der Speech to Text Conversion, ist aber eine Erleichterung um die Funktionalität eines einzelnes AR-Printer Moduls zu testen.

receiveResult()
notifyStopRecord()

2.2 Constants Klasse

In der Klasse Constants werden globale Applikationskonstanten abgelegt. So gibt es für jede IRecorder-Implementierung einen Integer Wert, welcher den Recorder repräsentiert. Außerdem gibt es für jeden Recorder eine beschreibende String-Variable. Eine Map, welche beim start der Anwendung erzeugt wird, bringt die String-Werte mit den jeweiligen Int-Werten in Verbindung. Analog dazu besteht die selbe Struktur für Instanzen des Typs IPrinter, die unterstützten AR-Printer verwalten zu können.

2.3 Interface für Augmented-Reality-Brillen-Printer

Das Interface IPrinter definiert ein Interface, welches den gesamten Kommunikationsverlauf zwischen MainActivity und dem Augmented Reality Gerätes ermöglicht. Für den AR-Connector wurde ein Interface erzeugt um dem Erweiterbarkeitsparadigma von Software gerecht zu werden:
Die Klasse MainActivity ist mit einer Instanz der Klasse IPrinter, also einer Klasse, welche das Interface IPrinter implementiert, assoziiert. Innerhalb der Klasse MainActivity werden ausschließlich Methoden des Interfaces aufgerufen, keine Methoden der Implementation selbt. Aus diesem folgt, dass um ein weiteres Augmented Realitty Gerät zu unterstützen, lediglich eine einzelne neue Klasse implementiert werden muss, welche das IPrinter-Interface implementiert und der Klasse MainActivity hinzugefügt werden muss. Durch das Interface

wird sichergestellt, dass alle Implementierungen über den selben Mindestmethodenumfang verfügen und Aufrufe innerhalb dieses Umfangs von allen Implementierungen verarbeitet werden können.

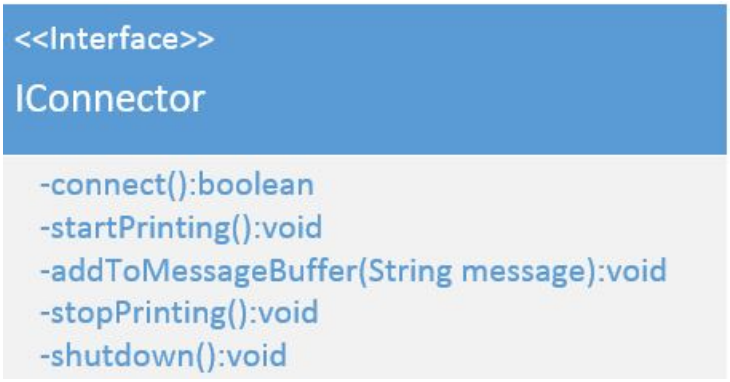

Abbildung 2.1: Interface für Augmented Reality Gerät Konnektoren

Interface-Methoden Das Interface verlangt die im Klassendiagramm **??** dargestellten Methoden von seinen Erben.
Die Methode connect() wird nach der Initialisierung einer neuen Connector Instanz von der MainActivity aufgerufen. Diese Methode, versucht eine Verbindung zu einem Augmented Reality Gerät aufzubauen. Ihr boolscher Rückgabewert gibt Auskunft über den Erfolg des Verbindungsaufbaus. Die startPrinting()-Methode versetzt den Connector in den Ausgabemodus. In diesem Zustand, wartet der Connector auf Nachrichten, die von der MainActivity Klasse durch die Methode addToMessageBuffer() in seinem Nachrichtenspeicher abgelegt werden und sendet diese an das Augmented Reality Gerät.
Die Methode stopPrinting() beendet diesen Zustand und mit der shutdown() Methode wird die Verbindung der App zum Augmented Reality Gerät beendet. Physikalisch kann die Verbindung dennoch weiterbestehen. Andere, Connectorspezifische Aktionen, welche bei Verbindungsende fällig werden, können ebenfalls in der shutdown()-Methode durchgeführt werden.

2.4 Interface für Speech-To-Text-Konvertierer

3 Graphical User Interfaces

3.1 ActionBar

3.2 NavigationView als Sidebar

3.3 Buttons / Buttondesigning

3.4 Spinner

3.5 TextViews

3.6 EventListener

3.7 SurfaceView für Animation

4 Canvas Animationen

4.1 DrawingThread / SoundAnimationThread

4.2 Die Methode „draw“

4.3 Die Methode „update“

4.4 Soundanimation durch animierte Canvas Komponenten

5 AR-Glass-Printer Implementierungen

Im folgenden werden implemntierungsspezifische Details über das Unterstützten verschiedener AR-Geräte näher erläutert. Die mit einem AR-Gerät kommunizierenden Module sind hier als AR-Printer bezeichnet. Jeder AR-Printer ist für die Dauer einer Verbindung mit dem ihm zugeteilten Gerät umfassend verantwortlich für die Kommunikation zwischen App und Gerät. Dies beinhaltet:

1. Den Verbindungsaufbau
2. Die Verarbeitung von von der Applikation gesendeten Anfragen zur Textausgabe auf dem AR-Gerät
3. Die Verarbeitung von vom AR-Gerät gesendeten Statusnachrichten
   Dies beinhaltet auch das Handling von Error-Codes.
4. Den Verbindungsabbau

Nachdem über das User Interface ein verfügbarer AR-Printer gewählt wurde, erzeugt eine Factory Klasse die jeweilige Printer-Instanz, welche im Konstruktor versucht eine Verbindung zum AR-Gerät aufzubauen. Ist der Verbindungsaufbau nicht möglich, wird der User über die fehlende Verbindung benachrichtigt und der Start-Record-Button deaktiviert, bis eine Verbindung zum Gerät hergestellt wurde oder vom User ein anderer AR-Printer gewählt wird.

5.1 Die Klasse AbstractPrinter

Die Klasse AbstractPrinter implementiert das Interface IPrinter und abstrahiert gemeinsames Verhalten aller AR-Printer. Die Implementierung eines spezifischen AR-Printers erfolgt über die Ableitung dieser Klasse und nicht über das Interface direkt. Dies dient der Vermeidung von Fehlern und der Vereinheitlichung der Arbeitsweise von unterschiedlichen Implementierungen.

Implementierte Methoden Während der Verbindungsauf- beziehungsweise -abbau geräte- und somit implementierungsspezifisch sind, kann das Verhalten der Methoden startPrinting(), addToMessageBuffer(String message), und stopPrinting() Geräteübergreifend implementiert werden.
Die Methode addToMessageBuffer(String message) erfüllt die Aufgabe, den übergebenen String zum Message Buffer hinzuzufügen. Der Message Buffer ist durch eine LinkedBlockingQueue realisiert und stellt somit eine Schlange dar, bei der Entnahmen immer am vorderen Ende geschehen und Elemente an ihrem Ende angefügt werden können. Sie arbeitet nach dem First-In-First-Out-Prinzip.
Die Methode startPrinting() erzeugt einen neuen Thread, welcher innerhalb einer eigenen Methode doPrinterJob() eine Schleife solange durchläuft, bis das boolsche Attribut isProcessing auf false gesetzt wurde, was bei Beendigung des Konvertierungsvorgangs der Fall ist. Innerhalb der Schleife wird überprüft, ob der Message Buffer neue Elemente zur Entnahme enthält. Ist dies der Fall, wird erfolgt ein Aufruf der abstrakten Methode printMessage(String message), welche die Nachricht im gerätespezifischen Protokol an das verbundene AR-Gerät sendet. Diese Methode muss für jeden spezifischen AR-Printer implementier eigens implementiert werden. Ein boolscher Rückgabewert gibt Auskunft über den Erfolg des Sendens der Nachricht. Liefert die Überprüfung der isProcessing-Variable false zurück, so deutet dies auf das Ende des Vorgangs hin. Um Ausgaben nicht apprupt zu beenden, wird neben des isProcessing-Attributes auch noch der Inhalt des Message Buffers geprüft. Is isProcessing zwar false, aber der Message Buffer enthält noch Elemente, die zur Ausgabe in Auftrag gegeben wurden, fährt der Printing-Thread fort, bis alle Ausgaben getätigt wurde.
Die Methode stopPrinting() setzt das boolsche Attribut isProcessing auf false. Dies führt zum Beenden der Schleife innerhalb des Printing-Threads und somit zur Beendigung des Threads an sich.

5.2 Support der GlassUp AR Brille

Die GlassUp Brille GlassUp ist ein italienisches Unternehmen, welches eine 65 Gramm **faq** schwere Augmented Reality Brille herstellt. Sie ist in 5.1 abgebildet. Die Brille befindet sich zwar noch in der Entwicklung, kann aber alles bieten, was zur Funktionalität unserer Applikation notwendig ist:
Nach eigener Aussage legt die Firma Wert darauf, eine unauffällige Brille zu produzieren, welche weniger einen Multimedialen Gegenstand als einen generellen Alltagshelfer darstellt [glad].
So besitzt diese Brille keine Kamera, wie beispielsweise die von Google angebotene Alternative[goo]. Eine Kamera gefährdet den Einsatz der Augmented Reality Brille insoweit, dass

Abbildung 5.1: Die GlassUp Augmented Reality Brille**home**

nach Paragraph 90 des Telekommunikationsgesetzes der Besitz von Gegenständen, "die ihrer Form nach einen anderen Gegenstand vortäuschen oder die mit Gegenständen des täglichen Gebrauchs verkleidet sind und auf Grund dieser Umstände [...] dazu bestimmt sind, das [...] Bild eines anderen von diesem unbemerkt aufzunehmen."
Auch ist die GlassUp Brille mit einem Preis von 299 Euro [glaa] noch erschwinglich. Die frei erhältliche SDK enthält außerdem einen Emulator, wodurch ein Brillenkauf für Entwicklungszwecke nicht von Anfang an zwangsläufig notwendig ist [glac].
Die Brille wird durch die Bluetooth-Technik mit einem Mobiltelephon verbunden und Projeziert Texte und einfache Grafiken in der Farbe grün in den Mittelpunkt des Sichtfelds des Anwenders. Außerdem enthält die Brille einen Beschleunigungssensor, einen Kompass und einen Helligkeitssensor[Gla15]. Sie verfügt über ein Touchpad mit drei Eingabeknöpfen, durch die Smartphone-Anwendungen alternativ gesteuert werden können[Gla15]. Der niedrige Preis, sowie der kompakte aber ausreichende Funktionsumfang führten zur Entscheidung, diese Brille als erste zu Unterstützen. Erweiterungen zum Unterstützen weiterer Brillen, sind durch den modularen Aufbau der Applikation allerdings ohne weiteres möglich.

Kommunikation mit der GlassUp AR Brille Die Kommunikation mit dem GlassUp Augmented Reality Gerät erfolgt laut der Library Dokumentation durch eine Service-App welche auf einem Smartphone installiert werden muss[Gla15]. Die Kommunikation mit der Service Applikation ist durch die Klasse GlassUpAgent, welche in der GlassUp

Support Library enthalten ist abstrahiert[Gla15]. Über ein Objekt dieser Klasse kann durch die Methode sendConfiguration() ein zu verwendendes Layout spezifiziert werden (Nur Bild, Bild und Text, Vier Bilder, Nur Text). Nachdem ein Layout definiert wurde können Nachrichten über die Methode sendContent() an die Service Applikation gesendet werden, welche die Daten an die Brille weiterleitet. Die Service Applikation verwaltet den Zugriff auf das AR-Gerät von verschiedenen Applikationen gleichzeitig und ebenfalls die Benachrichtigung über User-Interaktionen wie beispielsweise das drücken eines Buttons auf dem Touchpad der Brille.
Die Klasse GlassUpAgent bietet auch Schnittstellen für ConnectionListener, EventListener und ContentResultListener. Der erste wird bei einer Änderung des Verbindungszustands benachrichtigt, der als zweites genannte beim Vorkommen einer User-Interaktion und der ContentResultListener erhält Rückgabedaten eines Darstellungsvorgangs.

Anpassungen an der GlassUpAgent Klasse zum Support von Android 5.0+ Die Klasse GlassUpAgent der GlassUp Support Library abstrahiert die Kommunikation einer Applikation mit der GlassUp Service Applikation. Daten werden an die Service Applikation gesendet indem sogenannte Intents gestartet werden. Intents sind Objekte, die zum Datenaustausch zwischen verschiedenen Anwendungen verwendet werden. So können beispielsweise bestimmte Funktionalitäten einer anderen Applikation durch ein Intent-Objekt ausgeführt werden[anda]. Ein Intent-Objekt zum starten eines neuen Services enthält eine String zur Identifiktion des Services, welcher im Konstruktor übergeben wird. Außerdem können durch die putExtra()-Methode variable Daten über einen Key->Value Mechanismus übergeben werden, welche vom gefragtem Service ausgelesen werden können. Über Context.startService(Intent intent) wird das System aufgefordert den Service mit übergebener ID zu starten. Weil nun ein im Hintergrund laufender Service mit der angeforderten ID auf das Request antworten kann, ohne dass das System seine Echtheit überprüfen kann, stellt dieser impliziete Aufruf ein Sicherheitsrisiko dar und es wird deshalb in Android Versionen ab Android 5.0 eine Exception geworfen wenn eine Applikation versucht einen Service durch ein implizietes Intent zu starten [andb]. Innerhalb der Support Library wurden Intents ausschließlich impliziet gestartet:

```
1  //Service ID
2  Intent intent = new Intent("glassup.service.action.SEND_AGENT_CONTENT");
3  //Extra
4  intent.putExtra("appId", this.context.getPackageName());
5  //Extra
6  intent.putExtra("contentId", contentId);
7  //Extra
8  intent.putExtra("layoutId", layoutId);
9  //Extra
10 intent.putExtra("texts", texts);
```

```
//Extra
intent.putExtra("images.id", imagesId);
//Service starten
this.context.startService(intent);
```

Die Schwachstelle wurde erst beim Wechsel zu einem neueren Test-Smartphone bekannt. Weil nach der bitte um Änderung der Library keine Antwort der Firma erhalten wurde und kein Library-Update in Aussicht war, war es nötig, die Klasse GlassUpAgentVersionSupport zu implementieren.
Die Klasse enthalt den gesamten dekompilierten byte-code der GlassUpAgent Klasse und wurde um expliziete Intent Calls ergänzt. Für alle Intents wurd somit, um Appcrashes zu verhindern hinzugefügt:

```
//Service ID
Intent intent = new Intent("glassup.service.action.SEND_AGENT_CONTENT");
//BUGFIX:
//Expliziete Angabe des Packages, in dem sich die Service-Klasse
    befindet
intent.setPackage("glassup.service");
//Extra
intent.putExtra("appId", this.context.getPackageName());
//Extra
intent.putExtra("contentId", contentId);
//Extra
intent.putExtra("layoutId", layoutId);
//Extra
intent.putExtra("texts", texts);
//Extra
intent.putExtra("images.id", imagesId);
//Service Starten
this.context.startService(intent);
```

Der GlassUp-Connector Die Klasse GlassUpConnector ist die Verbindungsklasse der Applikation zur GlassUp Augmented Reality Brille. Sie verbindet sich mit Hilfe der GlassUp Support Library mit dem AR-Gerät.
Der Connector implementiert eine Ableitung der in Die Klasse AbstractPrinter beschriebenen Klasse AbstractPrinter und implementiert somit indirekt das IConnector Interface. Da ein großer Teil der Funktionalität in der Klasse AbstractPrinter bereits implementiert ist, ergänzt der GlassUpConnector gennante Klasse lediglich um die Methode printMessage(String message) und einige private Hilfsmethoden.

Defiziete der GlassUp AR Brille Das in einem voran gegangenen Paragraph bereits beschriebene einfache Interface der GlassUp Brille mit reduzierter Funktionalität besitzt verschiedene Defiziete, welche durch das ergänzen fehlender Funktionalitäten innerhalb der Connector Klasse ausgeglichen werden müssen.

1. Beim Darstellen langer Nachrichten: Als lange Nachrichten werden hier Nachrichten bezeichnet, deren Länge über die auf der AR-Brille zur selben Zeit darstellbare Zeichenzahl hinausgeht. Da die Support Library nicht über eine automatische Scrolling-Funktion für lange Nachrichten verfügt, werden lange Nachrichten abgeschnitten, nachdem der Bildschirm vollständig gefüllt ist. Der Overflow geht hierbei verloren.

2. Das Anfügen einer zweiten Nachricht an eine schon auf dem Gerät dargestellte, voran gegangene kurze Nachricht ist nicht möglich, da beim Senden einer neuen Nachricht die im Moment dargestellte Nachricht vom Display gelöscht wird um die neue Nachricht darzustellen.

Erläuterung der Defiziete am Beispiel

1. Annahme: Ein Sprache zu Text Konvertierungsmodul sendet als Teilergebnis die Zeichenkette

   > *"Dieser Testsatz ist ziemlich lange um ein möglichst großes Teilergebnis innerhalb der Sprach zu Text Umwandlung zu generieren, welches die Augmented Reality Brille der Firma GlassUp an ihre funktionalen Grenzen bringt"*

   an den Connector.
   Die GlassUp AR Brille verfügt einen Darstellungsraum von 6 Zeilen von je 17 Zeichen. Worte, welche über ein Zeilenende hinaus reichen werden in die darauf folgende Zeile gedruckt, ist ein Wort länger als eine ganze Zeile, wird es am Zeilenende ohne Berücksichtigung der Silbentrennung getrennt.
   Unter berücksichtigung dieser Regeln wäre der auf der Brille dargestellte Text folgender:

   > *Dieser Testsatz*
   > *ist ziemlich*
   > *lange um ein*
   > *möglichst großes*
   > *Teilergebnis*
   > *innerhalb der Spr*

2. Annahme: Ein Sprache zu Text Konvertierungsmodul sendet als Teilergebnis die Zeichenkette

   "Hallo"

   an den Connector.
   In einem zeitlich vernachlässigbaren Abstand empfängt der Connector eine weiter Zeichenkette

   "Wie geht es dir"

   .
   Durch Die Service Applikation ist für den User global einstellbar, wie lange Text auf dem GlassUp Augmented Reality Device angezeigt werden soll. Nach dieser Zeit wird der Text wieder gelöscht. Ein User würde von daher nun bei einer gewählten Anzigezeit folgendes erwarten:

   Sekunde 1:
   Hallo

   Sekunde 2:
   Hallo
   Wie geht es dir

   Sekunde 3:
   Hallo
   Wie geht es dir

   Sekunde 4:
   Wie geht es dir

   Sekunde 5:

   Statdessen sieht ein Benutzer aber:

   Sekunde 0,5 [Blinkt nur kurz auf]:
   Hallo

   Sekunde 1:

Wie geht es dir

...

Definition des Soll-Zustandes Durch die GlassUpConnector Klasse soll beschriebenem Verhalten entgegengewirkt werden. Die genauen Anforderungen an die Brille sind folgende:

1. Wird ein zu langer Text gesendet, so soll der bedruckbare Bereich der Brille einmal ganz gefüllt werden. Der weitere noch zu sendende Text soll dann Zeile für Zeile unten angefügt werden, während am oberen Ende immer eine Zeile verschwindet, so dass ein durchgehender "Flowëntsteht.

2. Wird nach einer kurzen Nachricht, eine weitere Nachricht gesendet, so soll die kurze vorangegangene Nachricht nicht direkt vom Bildschirm gelöscht werden. Vielmehr soll auch in diesem Fall das Display upgedated werden und der dargestellte String um die neue Nachricht ergänzt werden, soweit dies möglich ist, sind die Nachrichten in zusammengefügtem Zustand zu lang für das Display soll wieder ein "Flow"gestartet werden.

Erfüllung des Soll-Zustandes Um diesen Soll-Zustand zu erfüllen, muss die GlassUpConnector Klasse über eine eigene Logik beim Senden von Nachrichten verfügen.
Die Klasse muss ankommende Nachrichten selbst in Zeilen teilen und muss über einen Zeilenbuffer verfügen, an den Zeilen angefügt werden können, gleichzeitig aber auch am forderen Ende Elemente wieder entnommen werden können.
Durch die Vater-Klasse AbstractPrinter wird definniert, dass beim eintreffen einer neuen Nachricht die Methode printMessage(String nachricht) aufgerufen wird. Diese Methode ist im GlassUpConnector zwar implementiert, druckt die Nachricht allerdings nicht direkt auf der Brille aus, sondern ruft die private Methode addMessageToLineList(String nachricht) auf, welche die Nachricht, im folgenden messageBuffer genannt, in einzelne Zeilen teilt und diese der String-Liste linesToPrint hinzufügt.

Die addMessageToLineList-Methode Im folgenden ist der Algorithmus dargestellt, welcher eine Nachricht in einzelne Zeilen zerlegt.

```
1 [caption={Die Methode addMessageToLineList(String Message) der Klasse
      GlassUpPrinter}\label{lst:GlassUpPrinter: addMessageToLineList(
      String message)},captionpos=t]
2 /**
```

```
3  * method takes a message and splits it in lines of 17 chars
4  * to be printed to the AR device. The lines are added to this.
       linesToPrint
5  * @param messageBuffer
6  */
7  private boolean addMessageToLineList(String messageBuffer){
8    int counter = 0;
9
10   if(messageBuffer == null){
11     return false;
12   }
13
14   while(counter < messageBuffer.length()) {
15
16     if (messageBuffer.length() - (counter + 17) <= 0) {
17       //rest of buffer is smaller than one line, -> prepare buffer and
             send
18       //then break
19       this.linesToPrint.add(messageBuffer.substring(counter));
20       break;
21     }
22     //after 17 signs there is a space --> perfect line
23     if (messageBuffer.charAt(counter + 17) == ' ') {
24       this.linesToPrint.add(messageBuffer.substring(counter, counter +
             18));
25       counter += 18;
26     } else {
27       //check next ' ' before 17
28       boolean foundSpace = false;
29
30       for (int i = counter + 17; i > counter; i--) {
31         //space found?
32         if (messageBuffer.charAt(i) == ' ') {
33           this.linesToPrint.add(messageBuffer.substring(counter, i+1));
34           counter = i + 1;
35           foundSpace = true;
36           break;
37         }
38       }
39
40       //check if a space was found in the line
41       if (!foundSpace) {
42         //if no space in whole line just break on letter 17
43         this.linesToPrint.add(messageBuffer.substring(counter,  counter
               +17));
44         counter += 17;
45       }
```

```
46      }
47    }
48 return true;
49 }
```

Erläuterungen:
Innerhalb der while-Schleife tastet die Variable Counter die gesamte Nachricht ab.
Beim Zeichen mit Index 0 angefangen, wird in der ersten If-Clause überprüft, ob der Buffer länger als eine Zeilenlänge ist. Ist messageBuffer.length - (counter + 17) kleiner oder gleich 0, so sind die übrigen abzuarbeitenden Zeichen im Buffer weniger, als eine darstellbare Zeile und können dem Zeilenbuffer hinzugefügt werden, durch this.linesToPrint.add(messageBuffer.substring(counter)). Danach wird die while-Schleife durch break beendet. Ist die Zahl der noch abzuarbeitenden Zeichen größer als eine Zeilenlänge, wird die zweite If-Clause erreicht.
Diese überprüft ob sich an der Stelle counter + 17, also am 18. Zeichen, da die charAt(int i)-Methode das erste Zeichen bei Index 0 findet, ein Leerzeichen befindet. Ist dies der Fall, endet ein Wort genau am Zeilenende und alle Zeichen zwischen counter und counter+18 können als String der Liste linesToPrint hinzugefügt werden.
Ist auch dies nicht der Fall, muss nach dem nächsten Leerzeichen von counter+17 anfangend in Richtung counter gesucht werden. Dies geschieht durch eine for-Schleife. Wird innerhalb der for-Schleife ein Leerzeichen gefunden, werden alle Zeichen von counter bis Zeichen i der Zeilenliste linesToPrint hinzugefügt, die Hilfsvariable foundSpace auf true gesetzt und die for-Schleife beendet.
Ereicht die for-Schleife das Ende, ohne ein Leerzeichen zu finden, bedeutet dies, dass das Wort länger als eine Zeile ist und somit über zwei Zeilen verteilt werden muss.
So wird, wenn foundSpace false ist, das Wort nach 17 Zeichen hart getrennt und alle Zeichen zwischen counter und counter + 17 als String dem Zeilenbuffer hinzugefügt.
Immer nach der Abarbeitung der Zeichen einer Zeile, wird der counter um die entsprechende Zeichenzahl erhöht, um im nächsten Schleifen-Durchlauf mit dem erhöhten counter zu beginnen.

Die sendLinesToGlassUp-Methode Durch die bishher erläuterten Funktionalitäten werden Nachrichten, die vom Sprache zu Text Konvertierungsmodul an den GlassUpConnector gesendet werden, innerhalb der AbstractPrinter Klasse einem Buffer hinzugefügt. Ein Thread innerhalbt der ApstarctPrinter Klasse, welcher ständig prüft ob der Buffer neue Nachrichten enthält, bemerkt die Nachricht und ruft die Methode printMessage der Subklasse GlassUpConnector auf. In dieser Methode wird die Nachricht durch die Hilfsmethode addMessageToLineList in Zeilen aufgeteilt und eine String-Liste hinzugefügt.
Der noch fehlende Teil ist eine Methode, welche immer wenn Zeilen in der Liste sind,

alle Zeilen der Liste, höchstens jedoch sech Zeilen, zu einem String zusammen fügt und diese an die GlassUp Service Applikation sendet, welche sie auf der Brille abbildet. Diese Funktionalität wird in der Methode sendLinesToGlassUp() implementiert.
Die Methode wird innerhalb der von AbstractPrinter überschriebenen startPrinting() Methode in einem eigenen Thread gestartet, nachdem super.startPrinting() ausgeführt wurde. Innerhalb der sendLinesToGlassUp-Methode wird eine while-Schleife durchlaufen, solange bis die Aufnahme beendet wurde und sich keine Zeilen mehr in der Liste linesToPrint befinden.

Der Connection Listener Durch den Connection Listener wird die Applikation über Änderungen des Verbindungszustands des GlassUp AR Gerätes mit dem Smartphone benachrichtigt. Bricht die Verbindung während einer Konvertierung ab, wird der Konvertierungsvorgang beendet und der User über den Verbindungsabbruch informiert. Wird der GlassUp Connector ausgewählt ohne, dass eine Verbindung zum GlassUp Gerät besteht, ist der Button zum Starten einer Konvertierung deaktiviert, bis eine Verbindung hergestellt wurde.

Der GlassUp Emulator Alle Entwicklungsarbeiten und Tests wurden mit Hilfe des von GlassUp als Teil der GlassUp SDK zur Verfügung gestellten Emulators durchgeführt [glab]. Der Emulator muss auf einem Computer installiert werden, welcher sich im gleichen Netzwerk wie das Testsmartphone befindet. Mittels der Service Application, welche ebenfalls Teil der SDK ist und auf dem Testsmartphone installiert werden muss, verbindet sich das Mobiltelefon automatisch mit dem GlassUp Augmented Reality Brillen Emulator.

5.3 Der TextField Printer

6 Speech-To-Text-Konvertierung

6.1 Der Android Speech Recognition Client

6.2 Speech-To-Text Konvertierung durch die Google Cloud API

Aufbau des Moduls Die Klassen des Google Speech Recognition Moduls befinden sich im package de.dhbw.studienarbeit.hearItApp.recorder.googleSpeechRecognition. Zum aufzeichnen des Tons dient die Klasse VoiceRecorder welche das in XXX beschriebene IRecorder Interface implementiert. Die Klasse VoiceRecorder wird der Klasse MainActivity als Recorder Instanz zugeordnet, wenn im User Interface die Google Speech Recognition als Option gewählt wurde. Sie implementiert die Interface Methoden startRecording(), stopRecording() und shutdown(). In ihrem Konstruktor wird die Klasse GoogleSpeechConverter instanziiert. Diese Klasse ist verantwortlich für die Konvertierung der vom VoiceRecorder aufgenommenen Audiodaten. Sie kommuniziert mit den Google Servern, sendet Audiodaten und empfängt das Konvertierungsergebnis.
Diese Komponenten werden im folgenden näher erläutert.

6.2.1 Der Voice Recorder

Zur Aufnahme von Ton bietet Android zwei Optionen**Recording Optionen**

1. Die Erste ist der Android Media Recorder. Dieser bietet ein einfaches Interface, um mit wenigen Code Zeilen eine Audio Aufnahme zu starten und die Audio Daten in eine Datei zu schreiben. Nach beendigung der Aufnahme kann die Datei abgespielt oder weiter verarbeitet werden.

2. Die Zweite und für diesen Zweck passendere Methode ist die Audio Aufnahme durch ein Objekt der Klasse AudioRecord. Die Programmierung des AudioRecords findet eine Ebene tiefer statt, als die des MediaRecorders. Nachdem dem Objekt durch den Konstruktor die Aufnahmequelle, das Sampling, die Anzahl der Channels, das Encoding und die Buffergröße zugewiesen wurden, können nach dem Starten der

Aufnahme in einer Schleife die aufgenommenen Audio Daten in Form von Byte oder Short Arrays ausgelesen werden.

Ein Code-Beispiel zur Initialisierung und Aufnahme durch ein AudioRecord Objekt sieht so aus:

```
AudioRecord androidRecord = new AudioRecord(MediaRecorder.AudioSource.
    MIC,
    VoiceRecorder.SAMPLING, VoiceRecorder.RECORDER_CHANNELS,
    VoiceRecorder.RECORDER_AUDIO_ENCODING, VoiceRecorder.BUFFER_SIZE);

androidRecord.startRecording();
boolean isRecording = true;

while (isRecording) {

  short[] shortBuffer = new short[VoiceRecorder.BUFFER_SIZE / 2];

  //read ist return code, shortBuffer der Buffer in den geschrieben wird
  int read = this.androidRecord.read(shortBuffer, 0, shortBuffer.length)
      ;

  //verarbeite auio daten in shortBuffer

}
```

Begrifflichkeiten Die im Konstruktor übergebenen Daten sind die Audioquelle, das Sampling, die Anzahl der Channels, das Encoding und die Buffergröße.
Die Audio Quelle ist das Gerät, von welchem Ton aufgezeichnet werden soll und die Anzahl der Channels gibt an, über wie viele Kanäle der Ton aufgenommen wird (Mono, Stereo, usw).
Unter dem Sampling versteht man die Anzahl der Werte die pro Sekunde aus dem Analogen Signal einer Audioquelle entommen werden sollen. Weil ein Analoges Signal eine Welle bestehend aus unendlich vielen Werten darstellt, muss das Signal zur digitalen Speicherung auf eine definierte Punktezahl reduziert werden. Diese Zahl nennt sich Sampling. Für dieses Projekt wurde ein 16 000 Sampling gewählt, da ein Kompromiss zwischen Übertragungszeit und Qualität gefunden werden musste. Je höher das Sampling, umso größer die Daten Menge, umso höher die Übertragungszeit.
Das Encoding gibt die Zahl der Bits an, die jeder gespeicherte Wert hat. Bei einem 16 Bit Encoding, hat jeder gespeicherte Wert eine größe von 2 Byte.
Die Buffergröße ist die Größe des Arrays, in welchen man die Daten später einliest. Die Buffergröße gibt an, wie viele Sekunden jeweils vom Mikrophon ausgelesen werden sollen.

Herleitung Buffer Größe:
Pro Schleifendurchlauf sollen 80 aufgezeichnete Millisekunden vom AudioRecord gelesen werden.
Ein Sampling von 16 000 mit Encoding 16 Bit, bedeutet, dass pro Sekunde 16 000 Werte a 16 Bit gespeichert werden, was 16 000 Werten von je 2 Byte also 32 000 Byte = 32 kByte enstspricht. Multipliziert man diesen Wert mit der gewünschten Aufnahme Zeit ergibt sich die Größe eines Buffers in Byte.
Die Speichergröße errechnet sich dann wie folgend:

$$Speichergrö„e = Sampling * \frac{Encoding}{8} * ZeitSpeichergrö„e = 16000 * \frac{16}{8} * 0,8 = 25600Byte \tag{6.1}$$

Nun kann also entweder ein Byte Array mit größe 25 600 an den AudioRecord übergeben werden, oder aber ein Short Array mit der halben größe, da ein Short-Wert eine größe von 2 Byte hat.

Funktionaler Aufbau des Voice Recorders Nachdem die Option Google Speech Conversion im User Interface gewählt wurde, wird durch die RecorderFactory Klasse eine Instanz des VoiceRecorders erzeugt. Im Konstruktor werden sowohl ein AudioRecord Objekt, als auch ein GoogleSpeechConverter Objekt, welches im Verlauf noch näher erläutert wird, inizialisiert. Danach steht der Recorder zur Verwendung bereit.
Wird im User Interface der Button zum starten der Konvertierung gedrückt, ruft die Klasse MainActivity die Methode startRecording() des Recorders auf. Innerhalb dieser Methode wird die Methode readAudioInput() der eigenen Klasse in einem eigenen Thread gestartet. ReadAudioInput() setzt das Attribut isRecording auf true und beginnt eine while-Schleife, mit isRecording = false als Abbruchskriterium. Innerhalb der Schleife wird pro Durchlauf ein AudioBuffer mit Audio Daten gefüllt. Der Audio Buffer ist vom Typ short[]. Dieser Buffer wird der MainActivity-Methode showSoundAnimation übergeben, welche in Abhängigkeit der aus den Daten errechneten Lautstärke eine Animation erzeugt, um die laufende Aufnahme für den Benutzer zu visualisieren.
Danach wird der short[]-Buffer durch die Methode convertShortToByte[] in einen byte[]-Buffer konvertiert und der Methode recognizeBytes der Klasse GoogleSpeechConverter übergeben, welche die Daten an Google sendet, um gesprochene Sprache zu erkennen und diese in Text zu konvertieren.
Die Konvertierung von short zu byte ist notwendig, da die Google Streaming Speech API lediglich byte[]-Buffer als Übergabewert akzeptiert. Die Methode showSoundAnimation verwertet allerdings short-Werte, da bei einem 16-Bit Encoding jeder einzelne Audiowert die länge von 2 byte, also einer short hat. Bei der Konvertierung wird jeder short-Wert im

Buffer einmal mit 0x00FF maskiert und das Ergebnis in einen byte-Array geschrieben. Die Maske 0x00FF setzt die oberen 8 Bit auf 0 und übernimmt die Werte der untern 8 Bit. Es bleibt eine 1 Byte große Zahl, welche den Wert der unteren Hälfte der short-Zahl hat. Danach wird die short-Zahl mit » 8 um 8 Bit nach rechts geshiftet. Dies hat zur Folge, dass die untern 8 Bit verschwinden, während die oberen 8 Bit an die untere Stelle rücken. Die oberen 8 Bit werden mit nullen aufgefüllt. Wieder bleibt eine 8 Bit lange Zahl, welche die Werte der oberen Hälfte der short-Zahl enthält. Diese wird an die darauf folgende Stelle im byte-Buffer geschrieben.
Es entsteht ein Array der Form:

> buffer[0] = low Nibble Short[0] buffer[1] = high Nibble Short[0] buffer[2] = low Nibble Short[1] buffer[3] = high Nibble Short[1] ...

Quellcode der convertShortToByte-Methode:

```
1  private byte[] convertShortToByte(short[] buffer) {
2
3    byte[] byteBuffer = new byte[buffer.length * 2];
4
5    for (int i = 0; i < buffer.length; i++) {
6      //mask to set high 8 bit of short 0, let low 8 bit as is, add to
           array
7      byteBuffer[i * 2] = (byte) (buffer[i] & 0x00FF);
8      //shift right to get high 8 bit down, and add high bits to array
9      byteBuffer[(i * 2) + 1] = (byte) (buffer[i] >> 8);
10   }
11 return byteBuffer;
12
13 }
```

Wird der UI-Button zum Stoppen der Aufnahme gedrückt, oder passiert im Connector, GoogleSpeechConverter oder Recorder selbst ein schwerwiegende Fehler, wird die Recorder-Methode stopRecording() ausgeführt. Diese Methode Methode setzt das Attribut isRecording auf false, was zur Beendigung der Schleife führt. Auserdem wird die Methode stop() des AudioRecord Objektes ausgeführt, welche den Aufnahmevorgang stoppt.
Innerhalb der shutdown() Methode, welche beim Wechsel der Aufnahmemethode zu einem anderen Recorder-Modul aufgerufen wird, werden vom AudioRecord belegte Resourcen durch die Methode release() wieder frei gegeben.

6.2.2 Der Google Speech Converter

Der GoogleSpeechConverter ist der mit der Google Speech API kommunizierende Modulteil. Er sendet Audio Daten und empfängt die Konvertierungsergebnisse, welche er dann an die

MainActivity Klasse weitergibt, die diese an einen AR-Connector weiterleitet. Um den Aufbau der GoogleSpeechConverter Klasse darzustellen wird im folgenden zunächst die API der Google Cloud Funktion für Sprachkonvertierung erläutert. Im Anschluss wird auf die Implementierung innerhalb der Applikation eingegangen.

Die Google Cloud Speech API Zum Zugriff auf die Google Cloud Speech API bestehen drei Möglichkeiten **API Doc**

1. Synchrone Konvertierung
   Im ersten Fall wird eine Audiodatei an Google gesendet, welche konvertiert wird. Erst nach Abschluss des Konvertierungsvorgangs erhält der Client eine Antwort in Form eines JSON Strings, welcher den erkannten gesprochenen Text enthält.

2. Asynchrone Konvertierung
   Im zweitenen Fall, der asynchronen Konvertierung werden dem Client Teilergebnisse der Konvertierung der gesendeten Audiodatei schon mitgeteilt, sobald diese verfügbar sind und nicht erst nach Abschluss der Konvertierung.

3. Konvertierung durch Live Streaming
   Weil im Falle der hier thematisierten Applikation keine Audio Datei zur Verfügungn steht, sondern ein in echtzeit vom Mikrophon aufgenommener Audiostream, wird hier von der dritten Möglichkeit, der Live Streaming Variante gebrauch genommen. Bei dieser Option, sendet ein Client Audiopakete über einen Http-Stream an Google und empfängt über einen Antwort-Stream die Teilergebnisse der bisher konvertierten Daten.
   Die Audiodaten müssen hierfür in nahezu Echtzeit gesendet werden. Dies bedeutet, dass in jeder Sekunde nach Start der Konvertierung, Audiodaten einer Sekunde beim Google Server eingegangen sein müssen. Dies macht deutlich, dass die Übertragungsgeschwindigkeit hierbei eine große Rolle spielt. Werden vom Client im Abstand von 0,5 Sekunden Audiopakete mit Audiodaten der letzen 0,5 Sekunden gesendet, diese Pakete benötigen allerdings, aufgrund schlechter Übertragunsgeschwindigkeit 4 Sekunden zur Übertragung, erhält der Google Server lediglich alle 4 Sekunden Audio Daten von 0,5 Sekunden, wojdurch ein Konvertieren nicht mehr möglich ist.

Authentifizierung bei der Google Speech API Laut Googles API Dokumentation ist die sicherste Variante der Authentifizierung bei der Google Cloud die Authentifizierung durch einen Service Account. **https://cloud.google.com/speech/docs/common/auth** Ein Service Account ist ein Google Account, über den durch Apps Google Services genutzt werden können. Zur Authentifizierung bei einem Account dient ein asynchrones Private-Public-Key-Verfahren. Nachdem Anlegen des Accounts, genereiert ein Entwickler zunächst

ein Service Account Key-Paar in Form von JSON Strings. Der Key kann für eine oder mehrere Google APIs zugelassen werden. Im Falle dieses Projekts wurde er auf die Speech API beschränkt um Missbrauch zu vermeiden. Der JSON String wird als Datei im Projekt hinterlegt. Bei der Authentifizierung generiert Google ein syncrones Auth-Token, welches in der weiteren Kommunikation zur Identifikation des Authotisierten Clients.

OAuth Authentifizierung in Java Der sich authentifizierende Client verwendet einen InputStream um den im Projekt hinterlegten Key einzulesen. Er initialisiert ein Objekt der Klasse GoogleCredentials, welche im Package com.google.auth.oauth2 der Library com.google.auth:google-auth-library-oauth2-http:0.3.0 zu finden ist, und weißt ihm den Key-String zu. Dem GoogleCredentials Objekt wird ein Scope hinzugefügt, also ein Ziel, für welches die Credentials bestimmt sind. In diesem Fall ist das https://www.googleapis.com/auth/cloud-platform.
Durch die Klasse OkHttpChannelProvider und OkHttpChannelBuilder der Library io.grpc:grpc-okhttp, kann ein ManagedChannel Objekt erzeugt werden, welchem das zuvor generierte GoogleCredentials als parameter hinzugefügt wird. Das Objekt der Klasse ManagedChannel abstrahiert die Authentifizierung bei Google und stellt eine Verbindung zur Google API her. Jede Kommunikation mit der API erfolgt über diesen Channel.

Die Google Speech RPC API Zum Zugriff auf Funktionen der Speech API, werden zwei Optionen Angeboten: Zugriff via REST und Zugriff via RPC. Da die REST API nur für non-Streaming Use Cases verwendet werden kann, wird in diesem Projekt von der RPC API gebrauch gemacht **https://cloud.google.com/speech/docs/apis** Zugriffe über das Remote Procedure Call Protokoll bieten sich in diesem Fall an, da durch Remote Procedures bidirektionales Streaming ermöglich wird, während ein Client bei REST Zugriffen nach dem senden einer Http-Anfrage so lange wartet, bis eine Antwort eintrifft. Für Streaming während in festen Abständen immer wieder neue Audiodaten gesendet werden, ist es wichtig, dass die Antworten über einen zweiten Stream asynchron empfangen werden können. Auf RPC Methoden der Google Speech Cloud kann in Java durch die Library google.cloud.speech.v1beta1 zugegriffen werden. Die RPC Schnittstelle von Google wird auch gRPC API genannt **tps://cloud.google.com/speech/docs/apis**
Folgende Funktionen werden von der RPC API angeboten:

1. Das Interface Speech definiert, dass eine rpc Anfrage StreamingRecognize mit einem StreamingRecognizeRequest als Anfrage mit einem StreamingRecognizeResponse Objekt antwortet.

2. Um eine StreamingRecognize Anfrage zu stellen, muss ein StreamingRecognizeRequest Objekt erzeugt werden. Diesem Objekt wird entweder ein Parameter streaming_config oder audio_config hinzugefügt.

3. Im ersten StreamingRecognizeRequest müssen der API die Attribute der Gesendeten Audiodaten mitgeteilt werden. Dies geschieht durch ein StreamingRecognitionConfig Objekt, welches dem StreamingRecognizeRequest hinzugefügt wird.
   Das StreamingRecogntitionConfig Objekt enthält ein RecognitionConfig Objekt, in welchem Informationen über das verwendete Encoding, Sampling und die Sprache gespeichert sind. Außerdem wird durch die boolschen Parameter single_utterance und interim_results definiert, ob die Konvertierung bei einer Sprechpause beendet werden soll und ob auch geratene Ergebnisse mitgeteilt werden sollen. Geratene Ergebnisse entstehen während der Konvertierung einer Phrase, bevor deren Ende noch nicht erreicht ist. Geratene Ergebnisse sind zu einer mitgesendeten Wahrscheinlichkeit korrekt. Endgültige Ergebnisse haben eine Sicherheit von 98+

4. Alle weiteren StreamingRecognizeRequests enthalten nur noch audio_content, in Form eines byte Arrays von Audio Daten, welchen im zuvor spezifizierten Format codiert sind.

5. Auf jede StreamingRecognize Anfrage antwortet der Server mit einem StreamingRecognizeResponse Objekt in einem gesonderten Stream. Die StreamingRecognizeResponse ist ein JSON String. Dieser kann Flags zum hinweis auf den Beginn oder die Beendigung der Konvertierung enthalten. Innerhalb einer Konvertierung enthält der String ein oder mehrere alternatives Objekte. Diese bestehen aus einem "transcriptWert, also einer Übersetzung und einer ßtability", also der Wahrscheinlichkeit mit der die Übersetzung korrekt ist. Die alternatives Objekte sind in absteigender Wahrscheinlichkeit sortiert.
   ein is_final: true flag gibt an, dasss die Übersetzte Phrase nun final übersetzt wurde und sich nicht mehr ändert. Setzt man alle Ergebnisse, die dieses Flag enthalten zusammen, erhält man das Gesamtergebnis. Wurde der interim_results Parameter innerhalb der RecognitionConfig auf false gesetzt, werden nur finale Ergebnisse mitgeteilt.

https://cloud.google.com/speech/reference/rpc/google.cloud.speech.v1beta1

Implementierung des Google Speech API Zugriffes Die Funktionen bezüglich des Zugriffs auf die Google Speech API sind in der Klasse GoogleSpeechConverter im package de.dhbw.studienarbeit.hearItApp.recorder.googleSpeechRecognition gebündelt. Der zuvor

bereits thematisierte VoiceRecorder ruft im Konstruktor die private Methode initializeSpeechConverter() auf. Innerhalb dieser Methode wird eine neue Instanz des GoogleSpeechConverters erzeugt und dem VoiceRecorder als Attribut hinzugefügt.
Im Konstruktor des GoogleSpeechConverters wird ein Thread gestartet, welcher ständig die aktuelle Übertragungsgeschwindigkeit überprüft, da bei einer zu kleinen Geschwindigkeit keine Konvertierung stattfinden kann, weil Audiodaten in Echtzeit gestreamt werden müssen.
Danach wird durch die Methode createChannel() ein ManagedChannel nach dem bereits beschriebenen Ablauf erzeugt.

```
1  private Channel createChannel()
2   throws IOException, GeneralSecurityException {
3
4   InputStream credentials =this.recorder.getMainView()
5     .getAssets().open(authentication\_key);
6
7   GoogleCredentials creds = GoogleCredentials.fromStream(credentials)
8     .createScoped(this.OAUTH2\_SCOPES);
9
10  OkHttpChannelProvider provider = new OkHttpChannelProvider();
11  OkHttpChannelBuilder builder = provider.builderForAddress(
12    this.HOST, this.PORT);
13
14  ManagedChannel channel = builder.intercept(
15    new ClientAuthInterceptor(
16    creds, Executors.newSingleThreadExecutor()))
17    .build();
18
19  credentials.close();
20
21  return channel;
22 }
```

Mit diesem Channel wird ein SpeechGrpc.SpeechStub Objekt initialisiert durch die Methode SpeechGrpc.newStub(channel).
Wird dann eine Aufnahme gestartet, ruft der VoiceRecorder nach jedem lesen von Audio-Bytes die Methode recognizeBytes(byte[] buffer, int size) auf.
Innerhalb dieser Methode erfolgt eine Überprüfung, ob der Converter bereits initialisiert wurde. Wenn nicht, wird die Methode initializeStreaming() aufgerufen. Hier wird dem im Konstruktor initialisierten SpeechStub durch die Methode speechRPCStub.streamingRecognize(this) der ResponseStreamObserver zugewiesen.
Der Zugewiesene ResponseStreamObserver wird vom SpeechStub benachrichtigt, sobald eine Antwort von der SpeechAPI erhalten wird. Um als solcher zu funkieren, muss die

Klasse das Interface io.grpc.stub.StreamObserver<StreamingRecognizeResponse> und die zugehörigen, selbsterklärenden Methoden onNext(StreamingRecognizeResponse response), onError(Throwable error) und onCompleted() implementieren. Diese Methoden werden im jeweiligen Fall vom SpeechStub aufgerufen.
Die Methode speechRPCStub.streamingRecognize(this) gibt ein Objekt der Klasse StreamObserver<StreamingRecognizeRequest>, welches im Attribut requestObserver gespeichert wird, zurück. Über den RequestObeserver Stream können Daten an die Google API gesendet werden.
Sind beide Streams initialisiert, wird nach den zuvor beschriebenen API Regeln eine RecognitionConfig in eine StreamingRecognitionConfig eingebettet und daraus ein StreamingRecognizeRequest erzeugt, welches durch die Methode requestObserver.onNext(requehst) an die API gesendet wird, um die Konvertierung zu initialisieren. Der Programmcode zum generieren der Konfiguration ist im Anschluss abgebildet.

```
1  RecognitionConfig audioConfig =
2    RecognitionConfig.newBuilder()
3      .setEncoding( RecognitionConfig.AudioEncoding.LINEAR16 )
4      .setSampleRate( VoiceRecorder.SAMPLING )
5      .build();
6
7  StreamingRecognitionConfig streamingConfig =
8    StreamingRecognitionConfig.newBuilder()
9      .setConfig( audioConfig )
10     .setInterimResults( false )
11     .setSingleUtterance( false )
12     .build();
13
14 StreamingRecognizeRequest initialRequest =
15   StreamingRecognizeRequest
16     .newBuilder()
17     .setStreamingConfig( streamingConfig )
18     .build();
19
20 //send initial request with config
21 this.requestObserver.onNext(initialRequest);
```

Nachdem die Konvertierung nun initialisiert wurde, wird in der recognizeBytes Methode das isInitialized Flag auf true gesetzt, um eine erneute Initialisierung zu vermeiden.
Um die Audio Daten zu Google zu senden, wird erneut ein StreamingRecognizeRequest initialisiert. Statt einer StreamingRecognitionConfig, werden diesem aber durch die Methode setAudioContent die Übergebenen Audio Bytes zugewiesen. Durch requestObserver.onNext(request) wird der Request gesendet.

```
1  public void recognizeBytes(byte[] buffer, int size){
```

```
2
3   if (!this.isInititalized) {
4     this.initializeStreaming();
5     this.isInititalized = true;
6   }
7   try {
8     StreamingRecognizeRequest request =
9       StreamingRecognizeRequest.newBuilder()
10        .setAudioContent(ByteString
11          .copyFrom(buffer, 0, size))
12        .build();
13    requestObserver.onNext(request);
14
15  } catch (RuntimeException e){
16    Log.e(MainActivity.LOG\_TAF, "Error while recognizing speech.
          Stopping."
17      + e.getMessage());
18    requestObserver.onError(e);
19    throw e;
20  }
21}
```

Verarbeitung der Antworten Die Verarbeitung der Antworten, passiert im dem Speech-Stub zugewiesenen ResponseStreamObserver, welcher ebenfalls in der GoogleSpeechConverter Klasse implementiert ist.
Bei einem Fehler wird die onError Methode mit einem Throwable als Parameter aufgerufen. Innerhalb der Methode, wird der Fehler in die Log Datei geschrieben und durch this.recorder.stopRecording() die Aufnahme beendet.
Bei einem positiven Ergebnis wird innerhalb der aufgerufenen onNext(StreamingRecognizeRespone respone) Methode die Liste der in der response entahltenen StreamingRecognitionResults ausgelesen. Sind Ergebnisse in der Antwort enthalten, werden vom ersten Ergebnis, welches den Index 0 hat, durch getAlternativesList() die Übersetzungsalternativen gewonnen. Die erste alternative, die die Wahrscheinlichste Übersetzung darstellt, wird durch this.recorder.getMainView().receiveResult(alternative.getTranscript()) and die MainActivity weiter geleitet, welche den ausgewählten Connector mit der Anzeige des Textes beauftragt.
Eine Prüfung des is_final Flags in der Antwort ist nicht notwendig, da durch das setzen der Einstellung interrim_results=true während der Initialisierung des Streamings sichergestellt wurde, dass alle Antworten nur finale Phrasen enthalten.

Überprüfung der Übertragungsgeschwindigkeit

Zeitliche Analyse der Konvertierung

Maximaler Streaming Zeitraum

6.3 Der TextFieldRecorder / Konvertierer

Literatur

[anda] android.com. *Intents and Intent Filters*. Techn. Ber. URL: https://developer.android.com/guide/components/intents-filters.html (besucht am 09. 04. 2017).

[andb] android.com. *Services in Android*. Techn. Ber. URL: https://developer.android.com/guide/components/services.html (besucht am 09. 04. 2017).

[Buc13] Jürgen Buck. *Radiologie Träger des Fortschritts: Festschrift für Friedrich H.W. Heuck*. Springer-Verlag, März 2013. ISBN: 978-3-642-80128-0.

[glaa] glassup.net. *FAQ GlassUp*. URL: http://www.glassup.net/en/faq/ (besucht am 06. 04. 2017).

[glab] glassup.net. *GlassUp Emulator*. URL: https://github.com/GlassUp/Sdk/tree/master/Emulators (besucht am 04. 04. 2017).

[glac] glassup.net. *GlassUp Sdk*. URL: https://github.com/GlassUp/Sdk (besucht am 04. 04. 2017).

[glad] glassup.net. *Home GlassUp*. URL: http://www.glassup.net/en/ (besucht am 06. 04. 2017).

[Gla15] GlassUp. *GlassUp Developer Guide-Android.pdf*. English. Techn. Ber. März 2015. URL: https://github.com/GlassUp/Sdk/blob/master/Docs/GlassUp%20Developer%20Guide-Android.pdf.

[goo] google.com. *Google Glass Camera*. URL: https://developers.google.com/glass/develop/gdk/camera (besucht am 09. 04. 2017).

[KR12] Greg Kipper und Joseph Rampolla. *Augmented Reality: An Emerging Technologies Guide to AR*. Elsevier, Dez. 2012. ISBN: 978-1-59749-734-3.

Anhang

(Beispielhafter Anhang)

A. Assignment

B. List of CD Contents

C. CD

B. Auflistung der Begleitmaterial-Archivdatei

Die Archivdatei wurde zusammen mit der Online-Version dieser Ausarbeitung auf die Lernplattform hochgeladen.

```
⊢ Literature/
|    ⊢ Citavi-Project(incl pdfs)/   ⇒ Citavi (bibliography software) project with
|    |                               almost all found sources relating to this report.
|    |                               The PDFs linked to bibliography items therein
|    |                               are in the sub-directory 'CitaviFiles'
|    |    – bibliography.bib        ⇒ Exported Bibliography file with all sources
|    |    – Studienarbeit.ctv4      ⇒ Citavi Project file
|    |    ⊢ CitaviCovers/           ⇒ Images of bibliography cover pages
|    |    ⊢ CitaviFiles/            ⇒ Cited and most other found PDF resources
|    ⊢ eBooks/
|    ⊢ JournalArticles/
|    ⊢ Standards/
|    ⊢ Websites/
|
⊢ Presentation/
|    –presentation.pptx
|    –presentation.pdf
|
⊢ Report/
     – Aufgabenstellung.pdf
     – Studienarbeit2.pdf
     ⊢ Latex-Files/ ⇒ editable LaTeX files and other included files for this report
          ⊢ ads/                  ⇒ Front- and Backmatter
          ⊢ content/              ⇒ Main part
          ⊢ images/               ⇒ All used images
          ⊢ lang/                 ⇒ Language files for LaTeX template
```