Hochschule Karlsruhe
University of
Applied Sciences

Fakultät für
Informatik und
Wirtschaftsinformatik

Bachelor-Thesis

Name:	Philipp-Rene Bott
Thema:	Sprache zu Code als Plugin für die Rider IDE von JetBrains
Arbeitsplatz:	bluehands GmbH & Co.mmunication KG, Karlsruhe
Referent:	Prof. Dr. Körner
Korreferent:	Prof. Schwarz
Abgabetermin:	08.04.2023

Karlsruhe, 06.01.2023

Der Vorsitzende des Prüfungsausschusses

Prof. Dr. Heiko Körner

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Änderungen entnommen wurde.

Karlsruhe, 08.04.2023

. . . . . .

(Philipp-Rene Bott)

Zusammenfassung

Sprachassistenz ist ein wichtiges Thema zur Effizienzsteigerung auch in der Entwicklung von Software, nicht zuletzt um die Arbeitsunfähigkeit bei Verletzungen der Hände zu vermeiden. Im Rahmen dieser Arbeit wird speziell die Sprachassistenz beim Entwickeln von Programmcode betrachtet. Moderne Lösungen zur Sprachassistenz in der Softwareentwicklung erfüllen zwei Aufgaben, einerseits die Automatisierung und Steuerung der Entwicklungsumgebung mittels Sprachbefehlen und andererseits die Synthese und Eingabe von Programmcode mittels Diktat. Auf theoretische Lösungen zur Synthese von Programmcode aus der Beschreibung der gewünschten Funktionen in natürlicher Sprache wird nicht eingegangen.
Als Referenz für den aktuellen Stand der Technik dient *Serenade* [7]. *Serenade* hat eine große Schwäche bei der Synthese von Programmcode. Mit *Serenade* ist es über Sprachsteuerung nur möglich, die standardisierten Methoden und Klassen einer Programmiersprache zu verwenden. Da Entwicklungsprojekte aber mit steigernder Komplexität immer weniger auf diese Klassen und Methoden zugreifen, sondern vermehrt projektspezifische Klassen und Methoden nutzen, kann die Codesynthese von *Serenade* immer weniger genutzt werden. Diese Beschränkung liegt in der Art und Weise begründet, wie *Serenade* Programmcode synthetisiert. *Serenade* nutzt ein auf die Erkennung von Programmcode trainiertes neuronales Netz. Da ein fertig trainiertes Netz verwendet wird, können neue oder extrem spezifische Informationen bei der Erkennung nicht berücksichtigt werden.
Ziel dieser Arbeit ist es, eine Codesynthese zu entwickeln, die eine dynamische Berücksichtigung von fallspezifischen Informationen zulässt. Aufgrund der Rahmenbedingungen der Bachelorarbeit wird dies als Plugin für die Entwicklungsumgebung *Rider* [8] für die Sprache C# implementiert. Moderne Sprachtranskriptionsdienste für natürliche Sprache, wie sie von *Google* oder auch von *Microsoft Azure* angeboten werden, bieten die Möglichkeit, gezielt Schlüsselworte aus einer gegebenen Liste zu erkennen, auch wenn diese nicht Teil der jeweiligen natürlichen Sprache sind. Da diese Listen sich dynamisch für jede Anfrage anpassen lassen, ist dieser Ansatz der vielversprechendste um spezifische Informationen bei der Spracherkennung zu berücksichtigen.
Der in dieser Bachelorarbeit verfolgte Ansatz zur Generierung einer Liste mit gewünschten Eingaben ist, diese über ein verbreitetes Assistenzsystem für Entwickler zu generieren, der Autovervollständigung. Moderne Entwicklungsumgebungen zeichnen sich unter anderem durch fortgeschrittene Autovervollständigungswerkzeuge aus, die an vielen Stellen im Code sehr gute Vorschläge zur Eingabe erzeugen. Es wird gezeigt, dass mit diesen Vorschlägen der Autovervollständigung und einem Spracherkennungsdienst für natürliche Sprache eine höhere Erkennungsrate bei der Synthese von typischem Programmcode erreicht werden kann, als mit einem speziell auf eine Programmiersprache trainierten neuronalen Netz.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Glossar

API Application Programming Interface, eine Schnittstelle zur Progammierung von Anwendungen

Camel Case Ein Stil zur Formatierung von zusammengesetzten Wörtern, bei dem der Anfangsbuchstabe jedes Teilwortes groß geschrieben wird

Caret Der Textcursor in einem Editor-Programm, typischerweise dargestellt durch einen blinkenden vertikalen Strich

KI Künstliche Intelligenz. Ein Programm oder Service, der auf maschinellem Lernen aufbaut

1. Einleitung

Für Programmierer mit einer körperlichen Einschränkung, wie dem *Repetitive Strain Injury Syndrom* oder auch einer Verletzung wie einem Knochenbruch, gibt es Werkzeuge um den Arbeitsablauf mithilfe einer Sprachsteuerung zu automatisieren und den Arbeitsablauf zu erleichtern [18].
Viele dieser Werkzeuge basieren auf Kommandos, die es ermöglichen, den Computer mit Sprachbefehlen zu steuern. Es gibt auch Werkzeuge zur Synthese von Programmcode, diese funktionieren aber nicht immer wie gewünscht. Insbesondere im Vergleich zu den Assistenzsystemen, die für die Codeeingabe mittels Tastatur existieren, wie die leistungsstarken Entwicklungsumgebungen der *IntelliJ IDEA* oder Plugins wie *ReSharper* der Firma *JetBrains* [8].
Ziel dieser Arbeit ist es, eine Methode zu finden, die Sprachsynthese zu verbessern, um nicht nur syntaktisch korrekten Programmcode zu erzeugen, sondern insbesondere den Kontext des Programmcodes, wie zum Beispiel importierte Klassen und verwendete Frameworks bei der Spracherkennung zu berücksichtigen.

Bluehands

Diese Bachelorarbeit wird bei der Firma *bluehands GmbH & Co.mmunication KG* erstellt. *Bluehands* ist eine in Karlsruhe ansässige Softwareentwicklungsfirma, die unter anderem auf die Entwicklung in C# spezialisiert ist und den Status Goldpartner der Firma *Microsoft* hat [2]. In diesem Rahmen fokussiert sich diese Arbeit auf die Synthese von Programmcode in C#. Auch bei *bluehands* gibt es einen Mitarbeiter, der seine Hände nur eingeschränkt nutzen kann. Dieser benutzt bevorzugt die C# Entwicklungsumgebung *Rider* von *JetBrains*. Daher wird die Lösung zur Sprachsynthese als Plugin für *Rider* entwickelt, die verwendeten Funktionen sind aber bei vielen Entwicklungsumgebungen verbreitet. Das Konzept sollte also auch auf andere Umgebungen übertragbar sein.

Vorgehen

Zunächst wird das Sprachassistenzprogramm *Serenade* analysiert, eine Open Source Lösung zur Entwicklung mittels Spracheingabe [7]. Es wird kurz auf die Funktionsweise und die Schwächen von *Serenade* eingegangen. Lösungsansätze für diese Schwächen werden vorgestellt und die endgültige Wahl für den verwendeten Ansatz erläutert.
Danach wird der Entwurf für das Plugin vorgestellt, dass diesen Lösungsansatz implementieren soll. Es werden die verwendeten Werkzeuge vorgestellt, die zur Implementierung verwendet wurden. Die Implementierung und Funktionsweise der einzelnen Komponenten des Plugins werden ausführlich beschrieben, sowie Ansätze, wie die implementierte

Codesynthese weiter verbessert werden könnte.
Die Effektivität und Effizienz der implementierten Lösung zur Codesynthese und der Ansätze zur Verbesserung der Erkennung werden gemessen, mit der Codesynthese von *Serenade* verglichen und abschließend bewertet.

2. Sprachassistenzprogramme

Künstliche Intelligenz (KI) hat in den letzten Jahren sehr große Fortschritte gemacht. Programme, die darauf basieren, werden in zunehmenden Maße für große Teile der Bevölkerung zugänglich. Die Barriere wird nicht nur dadurch gesenkt, dass die Programme für Laien leicht verständlich und einfach zu bedienen sind, sondern auch vergleichsweise gute Ergebnisse liefern.
Da verbreitete Spracherkennungsdienste, wie die von *Google Cloud* [3] oder *Microsoft Azure Cognitive Services* [1] ebenfalls auf KI basieren, sind auch hier in den letzten Jahren Fortschritte erkennbar.

2.1. *Serenade*

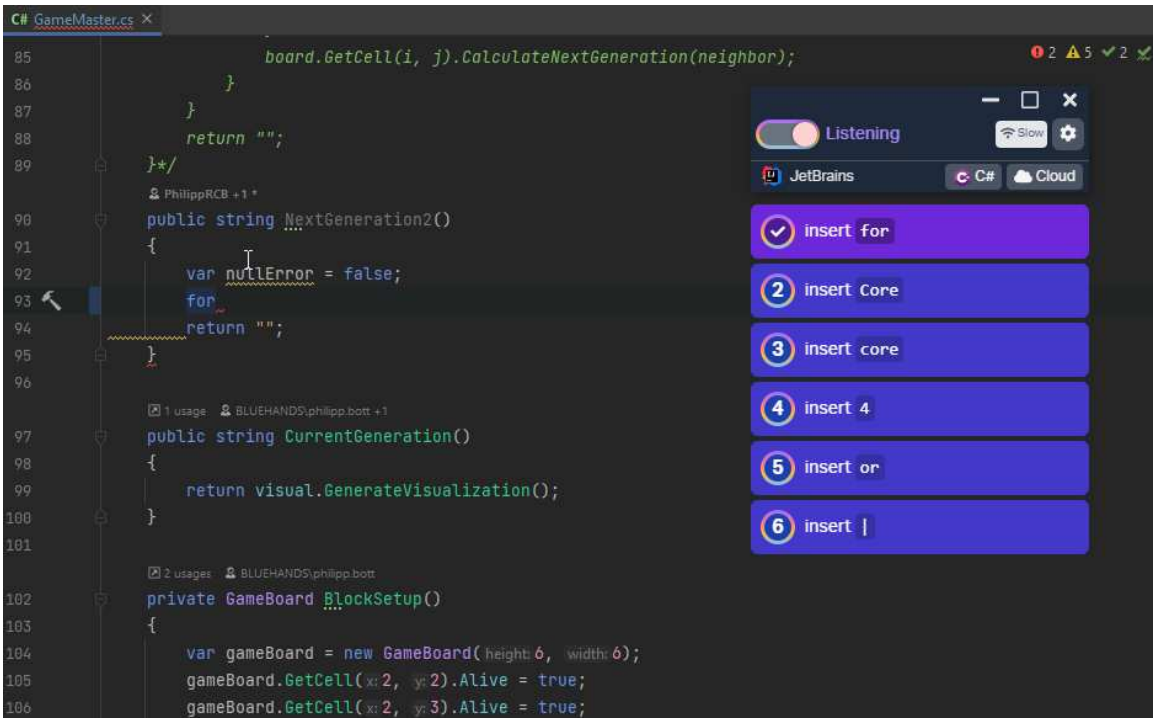

Abbildung 2.1.: Screenshot von *Serenade* im Einsatz mit der *Rider* Entwicklungsumgebung von *JetBrains*

Serenade ist ein Open Source Sprachassistent, der speziell zum Programmieren mit Spracheingabe entwickelt wurde. Zum Zeitpunkt der Veröffentlichung von *Serenade* waren Sprachassistenten vor allem auf natürliche Sprachen optimiert und die Möglichkeiten zur

Synthese von Programmcode beschränkten sich wie beispielsweise auf Sprachbefehle für bestimmte Tasten oder Buchstabieren mithilfe des NATO-Alphabets.
Serenade dagegen benutzt ein speziell für Programmiersprachen trainiertes Sprachmodell und erkennt typische Schlüsselworte und allgemeinen Funktionen von Programmiersprachen, wie 'enum' oder 'if'. Außerdem sind die in *Serenade* implementierten Kommandos auf das Bedienen von Entwicklungsumgebungen spezialisiert, was einen deutlichen Fortschritt gegenüber allgemeineren Sprachassistenzprogrammen zur Automatisierung der Computernutzung darstellt [7].
Die Entwicklung von *Serenade* war ein großer Fortschritt im Bereich der sprachgestützten Programmierung, hat allerdings auch seine Schwächen. Typischerweise wird moderner Programmcode nicht im Vakuum mit den Grundfunktionen einer Programmiersprache geschrieben. Insbesondere wenn das Programm größer und komplexer wird, werden immer weniger allgemeine Funktionen der Programmiersprache genutzt. Es werden zunehmend Klassen und spezielle Methoden, die von einem Framework oder einem Interface bereitgestellt werden, verwendet. Diese spezifischen Schlüsselworte kennt *Serenade* nicht und mit zunehmender Komplexität der Programme sinkt die Effektivität von *Serenade*.
Um diese projektspezifischen Schlüsselworte zu erkennen, muss ein anderer Ansatz gewählt werden. Es ist unrealistisch, alle existierenden Frameworks und Bibliotheken abzudecken, da je nach Sprache der Umfang enorm ist. Selbst dies wäre eine unvollständige Lösung, da die projektspezifischen Klassen und Methoden fehlen. Ein Modell speziell auf den Kontext eines Projekts zu trainieren wäre sehr ineffizient, da beim Programmieren der verfügbare Kontext stetig erweitert wird. Das Modell müsste also fortlaufend trainiert werden, was einen enormen zusätzlichen Rechenaufwand bedeuten würde.

3. Entwurf

Im Rahmen dieser Arbeit soll ein Sprachassistent als Plugin für die Entwicklungsumgebung *Rider* von *JetBrains* entwickelt werden. Der Fokus liegt hierbei auf der Entwicklung einer effektiveren Codesynthese.

3.1. Entwurf einer effektiveren Codesynthese

Ziel ist es, eine Codesynthese zu entwickeln, die bessere Ergebnisse liefert als eine spezialisierte KI, wie die von *Serenade*, insbesondere für die Arbeit in komplexen Projekten mit einem großen Netz an Abhängigkeiten und spezialisierten Methoden und Klassen. Dafür muss ein Weg gefunden werden, den aktuellen Kontext des Projektes zu berücksichtigen, der ständig erweitert wird. Dies geschieht auch durch den Nutzer der Codesynthese. Außerdem muss dieser Kontext nicht nur ermittelt, sondern auch bei der Spracherkennung berücksichtigt werden. Das kann bedeuten, dass jede Anfrage an die Spracherkennung einen anderen Kontext hat.

Spracherkennung mit den *Cognitive Services* der *Microsoft Azure Cloud*

Bei *bluehands* sind bereits Ressourcen zur Nutzung der *Microsoft Azure Cloud* vorhanden. Darin bietet *Microsoft* mit den *Cognitive Services* eine Vielzahl an Werkzeugen an, die auf KI basieren.
Einer der angebotenen Services ist ein Dienst, der Sprache in Text umwandelt. Der Dienst wird in der *Microsoft Azure Cloud* schlicht *Speech service* genannt [1]. Im Rahmen dieser Arbeit wird *Speech service* als Grundlage zur Spracherkennung verwendet. Der *Speech service* ist auf natürliche Sprache trainiert. Dies ist aber nicht unbedingt ein Nachteil, da moderne Bibliotheken und Frameworks oft Methoden und Klassennamen wählen, die auf natürlicher Sprache basieren, um die Verwendung und Wartung zu vereinfachen.

Spracherkennung mit priorisierten Schlüsselworten

Moderne Sprachengines, wie der *Speech service* in der *Microsoft Azure Cloud* bieten die Möglichkeit, über eine Liste an bevorzugten Schlüsselworten zu beeinflussen, welches Ergebnis der Spracherkennungsdienst liefert, ohne gleich eine vollständige eigenständige Sprache zu definieren oder eine KI darauf zu trainieren. Über eine *Phrase List* lässt sich Just-In-Time zu jeder Erkennungsanfrage sicherstellen, dass die gewünschten Schlüsselworte bevorzugt erkannt werden [10].
Diese dynamische Priorisierung von Schlüsselworten ist ideal zur Progammsynthese, wenn es möglich ist, zu jedem Schritt eine Liste mit wahrscheinlichen Eingaben für den jeweiligen Kontext zu generieren.

Programmkontext

Abhängig davon, in welchem Entwicklungsprojekt oder in welchem Teil eines Projektes programmiert wird, existieren andere Schlüsselworte. Unterschiedliche Projekte nutzen unterschiedliche Namen für Klassen und Methoden und auch innerhalb eines Projektes können sich die Namen von Variablen und Parametern je nach Programmkontext stark unterscheiden. Um eine bessere projektspezifische Codesynthese zu generieren, muss dieser individuelle Kontext berücksichtigt werden.
Dies effizient zu ermitteln, ist ein sehr großes und aufwändiges Problem. Da diese komplexen Rahmenbedingungen auch für menschliche Entwickler eine Herausforderung sein können, existieren bereits Assistenzsysteme, die diese Kontextinformationen ermitteln und verarbeiten. Eines dieser Systeme ist die Autovervollständigung in modernen Entwicklungsumgebungen.

Autovervollständigung

Die Tools zur Autovervollständigung von Programmcode generieren aus dem Kontext der Position des Caret eine Liste an möglichen Eingaben, die das Tool für sinnvoll hält. Typischerweise werden die Vorschläge durch den Start einer Eingabe ausgelöst und die generierten Vorschläge basierend auf den bereits eingegebenen Zeichen. Es ist aber auch möglich, eine Liste an Vorschlägen ohne eine vorige Eingabe zu generieren.
Die Autovervollständigung ist ein mächtiges und verbreitetes Werkzeug moderner Entwicklungsumgebungen und aus dem Arbeitsalltag von Programmierern nicht mehr wegzudenken. Wenn es möglich ist, darauf zuzugreifen, ist die Vorschlagsliste einer Autovervollständigung ein idealer Kandidat für priorisierte Schlüsselworte.

Zusammenfassung des Entwurfs für die Codesynthese

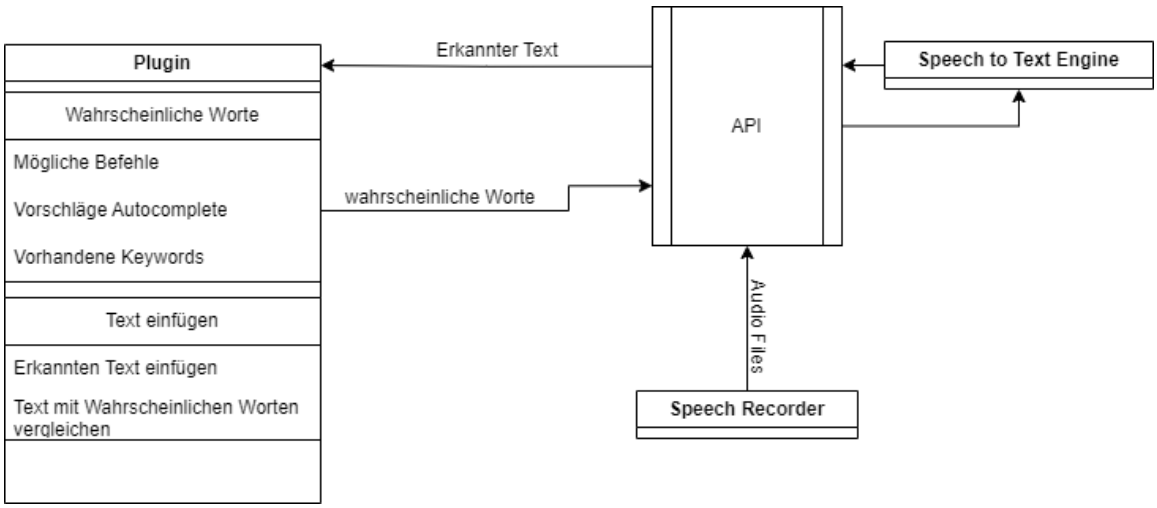

Abbildung 3.1.: Entwurf der Codesynthese

Es wird, den Rahmenbedingungen entsprechend, ein Plugin für die Entwicklungsumgebung *Rider* von *JetBrains* entwickelt. Dieses Plugin soll die Vorschläge der Autovervollstän-

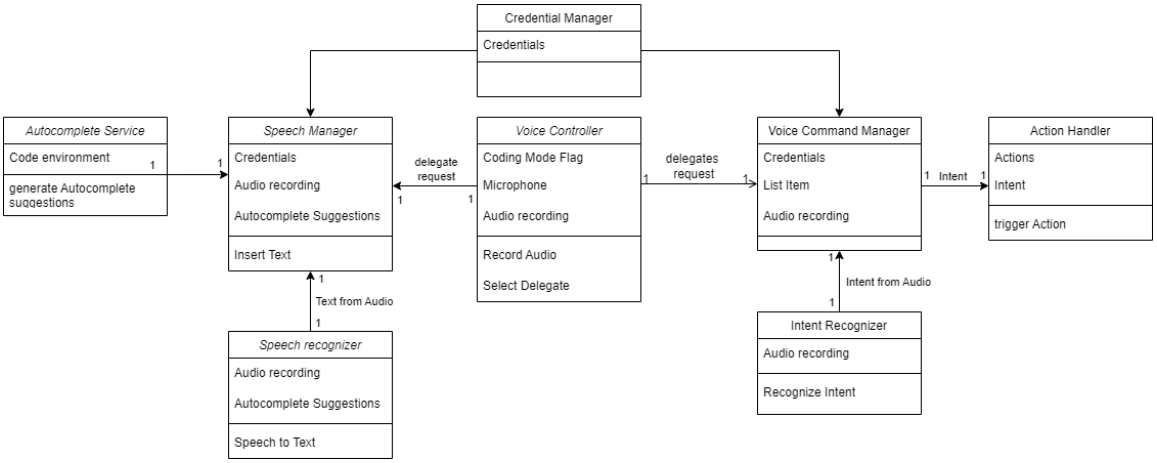

Abbildung 3.2.: Klassendiagramm des Plugin-Entwurfs

digung auslesen, um daraus eine Liste von möglichen Schlüsselworten zu generieren. Es muss außerdem eine Spracheingabe erkannt und in ein für die Spracherkennung passendes Format gebracht werden.
Diese verarbeitete Spracheingabe muss schließlich zusammen mit der Liste von Schlüsselworten an den Spracherkennungsdienst *Speech service* in der *Microsoft Azure Cloud* übermittelt werden, der daraus Text generiert und zurücksendet. Der erkannte Text muss abschließend an der korrekten Stelle im Editor von *Rider* eingefügt werden und das Plugin soll danach bereit sein, die nächste Spracheingabe zu erkennen.

3.2. Entwurf des Plugins

Basierend auf dem in Abschnitt 3.1 vorgestellten Entwurf zur Codesynthese muss das Plugin mehrere Aufgaben erfüllen. Es muss eine Spracheingabe erkannt und verarbeitet werden, eine Vorschlagsliste für die Spracherkennung generiert und eine Anfrage an einen cloudbasierten Spracherkennungsdienst gestellt werden.
Da nicht jeder ohne weiteres Anfragen an Services der *Microsoft Azure Cloud* senden kann, müssen zur Authentifizierung Zugangsdaten verwaltet werden. Außerdem wäre es wünschenswert, dass das Plugin neben der Codesynthese auch eine Steuerung der Entwicklungsumgebungen zulässt, ähnlich dem, was *Serenade* leistet. Aus diesen Überlegungen wurde für das Plugin ein Klassendiagramm (Abbildung 3.2) entworfen.
Das Programm soll über ein zentralen *Voice Controller* gesteuert werden, der die Spracheingaben des Nutzers erkennt. Abhängig davon, ob es sich bei der Spracheingabe um ein Kommando oder Programmcode handelt, wird die Verarbeitung an eine von zwei Klassen delegiert. Kommandos sollen im Voice Command Manager verarbeitet werden und die Absicht des Nutzers an eine Action-Manager Klasse kommuniziert werden. Die Action-Manager Klasse soll dann die zu dem Kommando gehörenden Prozesse auslösen. Bei einer Spracheingabe von Programmcode soll diese im Speech Service verarbeitet werden. Mithilfe der Autovervollständigung der Entwicklungsumgebung und dem Spracherkennungsdienst soll der gewünschte Programmcode synthetisiert und eingefügt werden.

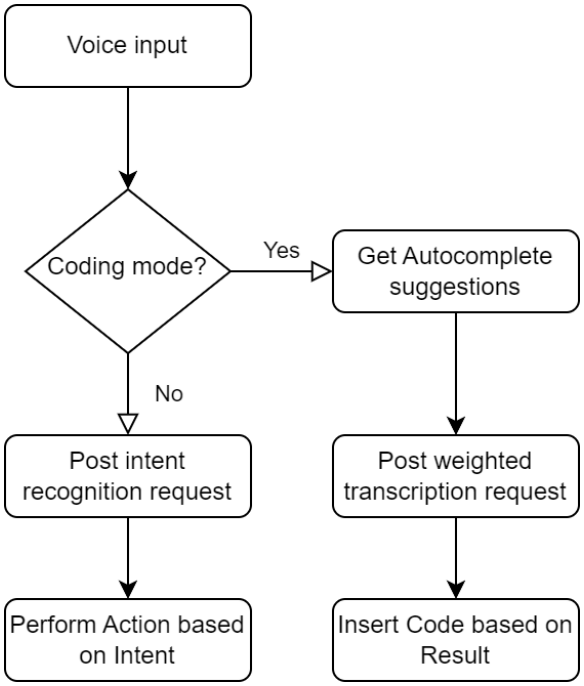

Abbildung 3.3.: Ablauf bei Spracheingabe

Sprachsteuerung

In *Serenade* fängt jede Spracheingabe mit einem Befehl an. Dies bedeutet allerdings, dass für jede Eingabe ein zusätzliches Wort ausgesprochen werden muss. Da der Fokus des Plugins auf der Codesynthese und nicht auf der Automatisierung liegt, wäre ein solcher Bedienungsprozess zu umständlich.
Stattdessen soll das Programm über einen Zustand entscheiden, ob es sich bei einer Spracheingabe um einen Befehl oder eine Codeeingabe handelt. Der geplante Ablauf eines einzelnen Sprachbefehls wird im Diagramm 3.3 veranschaulicht.

4. Implementierung

Dieses Kapitel behandelt die Implementierung des in Kapitel 3 beschriebenen Entwurfes. Zunächst wird dabei allgemein auf die Entwicklung eines Plugins für *Rider* eingegangen, die vorhandenen Werkzeuge und Frameworks, sowie die Programmiersprache *Kotlin*. Danach wird die Implementierung der einzelnen Komponenten erläutert. Diese sind die Tonaufnahme, die Funktionsweise und Methoden zum Auslesen der Autovervollständigung, die Kommandoinfrastruktur und wie die API des *Speech service* angesprochen wurde.

4.1. Plugin-Entwicklung für *Rider*

Die Entwicklungsumgebung *Rider* von *JetBrains* basiert auf *ReSharper* und *IntelliJ* [9]. Entsprechend kann ein Plugin für *Rider* sowohl die Funktionen von dem auf *ReSharper* basierenden Teil, als auch die Funktionen des auf *IntelliJ* basierenden Teils erweitern [12]. *ReSharper* ist ein Tool zur Codeanalyse und Unterstützung bei der Codeeingabe [8]. *Rider* nutzt *ReSharper* zur Codeanalyse und Steuerung der Assistenzsysteme, während die Benutzeroberfläche auf *IntelliJ* basiert [9].
Da das Plugin die Funktionen von *ReSharper* zwar nutzen, aber nicht modifizieren soll, wird ausschließlich das *IntelliJ* Frontend erweitert.

Template für Rider Plugins

Zur Entwicklung von Plugins für *Rider* stellt *JetBrains* ein Template zur Verfügung, das als Grundlage für jegliche Plugin-Entwicklung genutzt werden kann. Das Template bietet die Möglichkeit beide Hälften, die *ReSharper* und die auf *IntelliJ* basierende *Rider*-Hälfte zu erweitern und stellt vorkonfigurierte Build-Skripte zur Verfügung [12].

Die *IntelliJ Platform Plugin SDK*

Für die Entwicklung von Plugins für auf *IntelliJ* basierenden Entwicklungsumgebungen bietet *JetBrains* in der *IntelliJ Platform Plugin SDK* eine Reihe mächtiger Schnittstellen und Erweiterungsmöglichkeiten [9].
Zur Steuerung dieses Plugins wurde die *Action*-Schnittstelle und die *Listener*-Schnittstelle genutzt. Die Eingabe des generierten Codes erfolgte über die Editor-Schnittstelle. Plugins für *IntelliJ* können entweder in der Programmiersprache *Java* oder *Kotlin* geschrieben werden [9].
Da es bei der Entwicklung des mit dem Template für *Rider*-Plugins Probleme beim Kompilieren von *Java*-Code gab, wurde das gesamte Plugin in *Kotlin* entwickelt.

Action

Die Action-Schnittstelle ist dazu gedacht, eine neue Nutzerinteraktion zu ermöglichen. Actions implementieren eines von mehreren Interfaces, wie zum Beispiel *AnAction* oder *DumbAwareAction*, um eine Nutzerinteraktion über ein Menü oder eine Tastenkombination zu ermöglichen [9].

Listener

Listener dienen dazu, auf Vorgänge im Programmablauf zu reagieren. Eine Klasse, die ein Listener-Interface implementiert, kann auf dem internen Message-Bus registriert werden, um dann im Programmablauf entsprechend aufgerufen zu werden. Auf welche Ereignisse man mit einem Listener reagieren kann, kann der Dokumentation der *IntelliJ*-Plattform entnommen werden [9].

Kotlin

Kotlin ist eine von *JetBrains* entwickelte Programmiersprache. Ein großes Anwendungsgebiet für *Kotlin* ist die Entwicklung von Apps für mobile Endgeräte und ist sogar die von *Google* empfohlene Programmiersprache für Apps in *Android* [11]. *Kotlin* ist eine der möglichen Programmiersprachen für die Entwicklung von Plugins für *IntelliJ*-basierte Entwicklungsumgebungen. Da mit *Java* technische Probleme aufgetreten sind, fiel die Wahl für dieses Projekt auf *Kotlin*. *Kotlin* bietet die Möglichkeit, *Java*-Bibliotheken direkt einzubinden. Daher kann trotz dieses Umstiegs die komplette Bandbreite der *Java*-Landschaft genutzt werden [11].

Besonderheiten von Kotlin

Kotlin hat ein paar Besonderheiten gegenüber *Java*. Zunächst werden in *Kotlin* viele überflüssige Steuerzeichen eingespart, indem Statements mit einem Zeilenumbruch beendet werden, statt mit einem Strichpunkt. Variablen werden in *Kotlin* primär über veränderbar – über Deklaration mit ‘var’ – und statisch – über die Deklaration mit ‘val’ – getrennt, statt wie in *Java* über ihren Datentyp.
Der Datentyp einer Variablen ergibt sich in *Kotlin* bei der Initialisierung. Es ist weiterhin möglich, bei der Deklaration einer Variablen einen Datentyp zu definieren und wird insbesondere bei Parametern auch getan. Oft kann dies aber eingespart werden.
Insgesamt erzielt *Kotlin* durch diese Einsparungen ihr Ziel, den Code besser lesbar zu machen [11]. Eine für das Plugin relevante Umstellung ist das Fehlen von statischen Klassen. Stattdessen gibt es sogenannte Objects, die eine ähnliche Funktion erfüllen. Objects sind Klassen, die beim Start des Programms einmalig initialisiert werden. Alle Referenzen auf das Object greifen auf diese Instanz zu.

4.2. Implementierung des Voice Controllers

Die Aufgabe des Voice Controllers ist es, eine Spracheingabe zu erkennen und entweder einen Prozess, der daraus ein Kommando verarbeitet oder ein Prozess, der daraus Code generiert anzustoßen. Der Voice Controller verwaltet das Mikrofon und misst regelmäßig den Geräuschpegel. Wird ein bestimmter Geräuschpegel über einen gewissen Zeitraum überschritten, wurde eine Spracheingabe erkannt.

Implentierte Actions zum Voice Controller

Es gibt zwei Actions, die das Verhalten des Voice Controllers steuern. Die erste Action aktiviert bzw. deaktiviert den Voice Controller. Ist der Voice Controller aktiv, wird auf das Mikrofon zugegriffen, regelmäßig der Geräuschpegel geprüft und die Verarbeitung einer Spracheingabe angestoßen. Wird der Voice Controller deaktiviert, werden noch laufende Prozesse zur Verarbeitung von Spracheingaben zu Ende geführt und dann das Mikrofon wieder freigegeben. Die zweite Action zum Steuern des Voice Controllers legt fest, ob eine Spracheingabe als Kommando interpretiert wird oder ob daraus Code generiert werden soll.
Im Rahmen der Verbesserung der Erkennungsrate der Codesynthese wurde noch eine weitere Action zum Voice Controller hinzugefügt, die eine Spracheingabe von Freitext in Camel Case ermöglicht (vgl. Abschnitt 5.3).

Speech service API

Die Verarbeitung der Spracheingabe zu einem Befehl oder in eine Zeichenfolge wird über den Spracherkennungsdienst *Speech service* von *Microsoft Azure* realisiert. Der Spracherkennungsdienst bietet hierzu zwei grundsätzliche Vorgehensweisen.
Die erste Möglichkeit ist, die Steuerung des Mikrofons an das Application Programming Interface (API) zu übergeben und so die Spracheingabe direkt zu übertragen. Die zweite Möglichkeit ist, die Spracheingabe lokal in einer Audiodatei abzuspeichern und diese Datei zur Erkennung an den Dienst zu senden [1].
Nach Tests dieser beiden Methoden mit den vorhandenen Mikrofonen hat sich gezeigt, dass insbesondere Eingaben mit leiser Stimme über die erste Methode oft nicht registriert wurden, daher wurde die Methode mit lokaler Sprachaufnahme und Senden der Tonaufnahme gewählt.

Tonaufnahme mit der *Java Sound* API

Da *Kotlin* das Verwenden von *Java*-Bibliotheken erlaubt (vgl. Abschnitt 4.1), ist die Implementierung einer Tonaufnahme vergleichsweise simpel. Das Paket *javax.sound.sampled* bietet alle notwendigen Funktionen, um das Mikrofon anzusprechen, den Datenstrom auszulesen und in einer Datei zu speichern, die mit dem Spracherkennungsdienst kompatibel ist [16].
Da das Paket nicht nur das Speichern des Datenstroms in einer Datei ermöglicht, sondern

auch Zugriff auf den Datenstrom während der Aufnahme, kann auf Basis dieses Pakets auch die Steuerung über den Geräuschpegel implementiert werden.

Steuerung über Geräuschpegel

Die Tonaufnahmen sollen möglichst nur die Spracheingabe aufnehmen. Es sollen keine langen Abschnitte auftreten, die nur Stille oder leise Hintergrundgeräusche beinhalten. Daher wurde als Auslöser für die Tonaufnahme festgelegt, dass ein gewisser Geräuschpegel überschritten werden muss.
Zum Messen des Geräuschpegels wird regelmäßig ein kurzer Abschnitt aus dem Datenstrom des Mikrofons ausgelesen. Zur Bestimmung der Lautstärke in diesem Abschnitt wird über diesen Datensatz das quadratische Mittel gebildet.

$$x = \sqrt{\frac{1}{n} * (x_1^2 + x_2^2 + \cdots + x_n^2)}$$

Überschreitet die so berechnete Lautstärke einen festgelegten Wert, so wird eine Tonaufnahme gestartet. Auch während der Tonaufnahme regelmäßig die Lautstärke geprüft. Wird hier der Grenzwert der Lautstärke unterschritten, so wird die Aufnahme wieder beendet. Wurde auf diese Weise eine Spracheingabe aufgenommen, wird anschließend die Verarbeitung dieser Aufnahme angestoßen.

4.3. Implementierung von Sprachbefehlen in *Microsoft Azure*

Microsoft Azure hat zwei Lösungen zur Interpretation von Sprachbefehlen. Der *Speech service* bietet die Möglichkeit, über reguläre Ausdrücke feste Sprachbefehle zu definieren und zu erkennen. Die zweite Lösung zur Interpretation von Sprachbefehlen ist *Language Understanding* oder *LUIS*. *LUIS* ist eine auf KI basierende Lösung zur Interpretation von Sprachbefehlen, die es ermöglicht aus natürlicher Sprache die Absicht des Nutzers und relevante Parameter zu extrahieren [13].
Da das Definieren eines Modells für die Erkennung von Sprachbefehlen zur Automatisierung der Entwicklungsumgebung den Rahmen dieser Arbeit weit übersteigt, wurden die Sprachbefehle für dieses Plugin über den *Speech service* implementiert.

Erkennung von Sprachbefehlen mit *Speech service*

Mit dem *Speech service* lassen sich Sprachbefehle über Mustererkennung implementieren. Dafür muss jeder valide Befehl über einem regulären Ausdruck definiert werden.

Reguläre Ausdrücke zur Definition von Befehlen

Listing 4.1: Beispiele für Definitionen von Befehlsmustern

```
1   val pattern1 = PatternMatchingIntent("MOVE","[Row | Line | Move] {direction}
        [{distance}] [lines | rows]")
2   val pattern2 = PatternMatchingIntent("FILE", "[Create | Make] new {fileType} {fileName}")
```

```
3     val pattern3 = PatternMatchingIntent("AUTOCOMPLETE", "Autocomplete", "Complete")
4     val entity1 = CreateIntegerEntity("distance")
5     val entity2 = CreateListEntity("direction", PatternMatchingEntity.EntityMatchMode.Strict,
6                                "up", "down", "left", "right")
7     val entity3 = CreateListEntity("fileType", PatternMatchingEntity.EntityMatchMode.Strict,
8                                "class", "interface", "record", "struct", "enum")
```

Ein gültiges Befehlsmuster hat eine ID, über die nach der Spracherkennung der entsprechende Befehl zugeordnet wird und mindestens einen regulären Ausdruck, der das Muster beschreibt.
Diese regulären Ausdrücke bestehen aus Schlüsselworten und Parametern. Parameter werden über geschweifte Klammern markiert. Alle Zeichenfolgen, die nicht in geschweiften Klammern sind, sind Schlüsselworte. Parameter werden entweder über einen Datentyp oder eine Liste möglicher Werte definiert. Sowohl Schlüsselworte als auch Parameter können optional sein, dies wird über eckige Klammern markiert.
In Listing 4.1 werden beispielhaft drei Befehlsmuster und die dazugehörigen Parameter definiert. Das Befehlsmuster mit der ID *MOVE* dient zur Erkennung von einem Bewegungsbefehl. Es besteht aus vier Worten, wovon nur der Parameter *direction* zwingend notwendig ist. Mögliche Werte für *direction* sind ‘up’, ‘down’, ‘left’ und ‘right’.
Optional können weitere Schlüssel, wie ‘row’, ‘line’ oder ‘move’ vom Nutzer gesprochen werden, um die Erkennungsrate zu verbessern. Außerdem kann über den Parameter *distance* festgelegt werden, wie groß die Bewegung sein soll.

Erkennung der Befehlsmuster

Um einen Sprachbefehl zu erkennen, werden diese Befehlsmuster zusammen mit der Sprachaufnahme an die *Speech Service* API übergeben. Der Dienst antwortet nach Verarbeitung der Aufnahme auf die Anfrage mit einem Objekt, dass die ID des erkannten Befehlsmusters, sowie ein Wertepaar aus Parameter-ID und Parameterwert für jeden einzelnen Parameter enthält. Basierend auf der ID des Befehls und den Parametern können die entsprechenden Funktionen ausgeführt werden.

Durchführen der Befehle

Listing 4.2: Implementierung des MOVE-Befehls

```
1   private fun moveIntentExecution(direction: String?, distance: Int?) {
2      Logger.write("Moved $direction by $distance.")
3      if (direction != null) {
4         val dist = distance ?: 1
5         DataManager.getInstance().dataContextFromFocusAsync.onSuccess {context:
              DataContext? ->
6             val caret = context?.getData(CommonDataKeys.EDITOR)?.caretModel
7
8            when (direction) {
9               "up" -> caret?.moveCaretRelatively(0, (-dist),false, false, true)
10              "down" -> caret?.moveCaretRelatively(0, dist,false, false, true)
11              "left" -> caret?.moveCaretRelatively((-dist), 0,false, false, true)
```

```
12                "right" -> caret?.moveCaretRelatively(dist, 0,false, false, true)
13            }
14
15        }
16    }
17 }
```

Über eine einfache Fallunterscheidung der Befehls-IDs kann ausgewählt werden, welche Funktion durchgeführt werden soll. Viele Befehle, die über eine solche Sprachsteuerung ausgelöst werden sollen, dienen zur Automatisierung der Entwicklungsumgebung.
Die benötigten Funktionen für alle denkbar sinnvollen Befehle zu implementieren ist aufwändig und erfordert eine sehr gute Kenntnis der *IntelliJ Platform Plugin SDK*. Daher wurde sich im Rahmen dieser Arbeit primär darauf beschränkt, die Befehle zu Demonstrationszwecken zu erkennen. Eine Implementierung der gewünschten Befehle kann nachträglich durchgeführt werden.
Auch eine Erweiterung der Befehlsmuster ist mit dem implementierten Rahmen leicht möglich. In Listing 4.2 wird als Beispiel die Implementierung des *MOVE*-Befehls gezeigt, der lediglich das Caret innerhalb des Editors relativ zur aktuellen Position bewegt.

4.4. Implementierung der Codesynthese

Die in Abschnitt 3.1 skizzierte Implementierung für die Codesynthese sieht eine natürliche Spracherkennung mit priorisierten Schlüsselworten vor. Dieser Prozess gliedert sich in drei Stufen.
Die erste Stufe ist das Generieren der priorisierten Schlüsselworte aus der Autovervollständigung. Die zweite Stufe ist die Verarbeitung der im Voice Controller generierten Aufnahme (vgl. Abschnitt 4.2) mit dem Spracherkennungsdienst. Und die letzte Stufe ist das Nachbearbeiten, Verifizieren und Einfügen der Ergebnisse im Editor.

Autovervollständigung in IntelliJ-basierten Entwicklungsumgebungen

Die Autovervollständigung in *Rider* entspricht der Implementierung in *IntelliJ* und anderen Entwicklungsumgebungen, die auf der *IntelliJ Platform Plugin SDK* basieren (vgl. Abschnitt 4.1).
Sie basiert auf einem Provider-Consumer-Modell, bei dem aus mehreren Quellen alle möglichen passenden Vorschläge generiert und in einem zentralen Consumer gesammelt werden [9]. Dieser Prozess wird von einem Service angestoßen, der mit diesen Vorschlägen ein Popup befüllt [4]. In der Dokumentation zur *IntelliJ Platform Plugin SDK* sind mehrere Möglichkeiten vorgesehen, wie ein Plugin mit diesem Prozess interagieren kann.
Ein Plugin kann eigene Provider definieren, um zusätzliche Vorschläge zu generieren. Es ist möglich, bestimmte Ergebnisse herauszufiltern, sodass sie nicht im Popup auftauchen. Außerdem ist es möglich den Prozess der Autovervollständigung programmatisch anzustoßen und ein gefülltes Popup mit Vorschlägen zu generieren.
Für die Zwecke des Plugins ist nur die Funktion relevant, den Autocomplete-Prozess anzustoßen. Es wäre theoretisch möglich einen eigenen Consumer zu entwickeln, der das

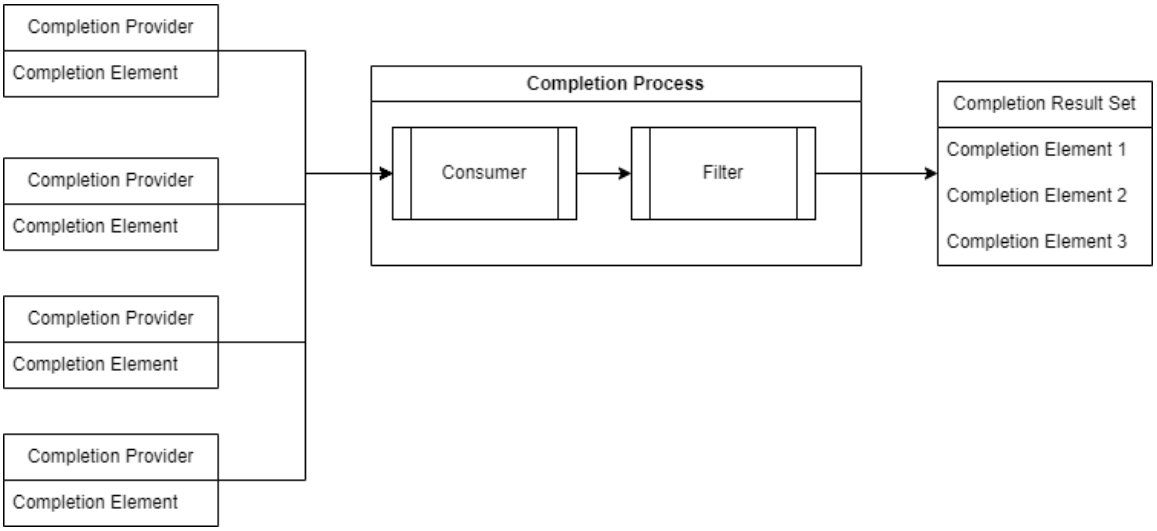

Abbildung 4.1.: Generierung der Wortvorschläge im Autovervollständigungsprozess

Plugin mit den gewünschten Informationen versorgt, es wurde jedoch keine Möglichkeit gefunden, einen weiteren Consumer auf den Nachrichtenbus zu registrieren. Stattdessen wurde der Prozess zur Generierung des Popups erweitert, um die Ergebnisse an das Plugin zu übermitteln.

Auslesen der Ergebnisse der Autovervollständigung

Für den Autovervollständigungsprozess gibt es eine Listener-Schnittstelle, die es ermöglicht, auf bestimmte Ereignisse im Popup der Autovervollständigung zu reagieren [9]. Das Ereignis, auf das reagiert wird, ist das Aktualisieren des Popups.
Ändern sich die Ergebnisse der Autovervollständigung zwischen zwei Aktualisierungen nicht, wird davon ausgegangen, dass die Ergebnisliste vollständig ist, sofern sie nicht leer ist. In diesem Fall wird die Ergebnisliste für die Codesynthese kopiert und zwischengespeichert.

Verarbeitung der Spracheingabe mit Wortvorschlägen

Das Hinzufügen der Wortvorschläge für die Anfrage an den Spracherkennungsdienst ist in der Dokumentation der Funktion beschrieben und die Implementierung ist mit den zwischengespeicherten Wortvorschlägen möglich [10]. Es muss allerdings zwingend darauf geachtet werden, dass die Nutzeroberfläche der Entwicklungsumgebung weiterhin responsiv bleibt, während auf das Ergebnis des Autovervollständigungsprozesses gewartet wird. Die Entwicklungsumgebung wäre sonst dauerhaft blockiert. Der Autovervollständigungsprozess arbeitet nur, wenn die Benutzeroberfläche nicht blockiert ist. Daher muss in einem seperaten Prozessthread auf das Ergebnis gewartet werden.
Dazu wird der Autovervollständigungsprozess um eine Semaphore erweitert, die signalisiert, ob die Ergebnisliste für die Verwendung zur Spracherkennung vorliegt. Das Aufrufen des Spracherkennungsdienstes direkt aus dem Autovervollständigungsprozess ist zwar

möglich, führt aber oft zu Fehlern, die einen Absturz der Autovervollständigung verursachen und nur durch einen Neustart der Entwicklungsumgebung behoben werden können.

Nachbearbeitung der Transkription und Zuordnung zu einem Schlüsselwort

Aus der Transkription der Spracherkennungsschnittstelle werden zunächst alle Satz- und Leerzeichen entfernt. Weitere Sonderzeichen sind nicht zu erwarten, da die Spracherkennungsschnittstelle diese nicht ohne weiteres generiert.
Die so bearbeitete Transkription wird nun mit allen Schlüsselworten verglichen. Wird eine exakte Übereinstimmung gefunden, so wird das korrespondierende Ergebnis der Autovervollständigung eingefügt. So wird sichergestellt, dass die Großschreibung des Schlüsselworts korrekt ist. Wenn die Transkription mit keinem Schlüsselwort exakt übereinstimmt, aber die Buchstabenfolge der Transkription in einem oder mehreren Schlüsselworten vorkommt, so wird das ebenfalls als Erfolg gewertet. Im Fall von nur einer gefundenen Buchstabenfolge wird das entsprechende Ergebnis der Autovervollständigung eingefügt. Im Fall von mehreren Übereinstimmungen wird stattdessen die nachbearbeitete Transkription eingefügt. Wird keine Übereinstimmung gefunden, so wird das Ergebnis zunächst ebenfalls eingefügt. Eine Unterscheidung zwischen einer mehrdeutigen Lösung und einer Sprachsynthese ohne Erkennung ist für die spätere Implementierung von weiteren Nachbearbeitungsmethoden wichtig.
Nach dem Einfügen wird das Popup der Autovervollständigung aufgerufen, um dem Nutzer alle möglichen Varianten zu präsentieren, die zu seiner Eingabe passen. Dem Nutzer wird so die Möglichkeit gegeben, die passende Variante zu wählen und Steuerzeichen automatisch generieren zu lassen.

5. Ansätze zur Verbesserung der Codesynthese

Idealerweise wird bei der Spracherkennung eines der vorgegebenen Schlüsselworte erkannt. Dies ist nicht unbedingt der Fall, sei es, weil der Nutzer nur ein Teilwort eines längeren Namens gesprochen hat, oder weil er bei undeutlicher Aussprache falsch verstanden wurde.
Im Folgenden werden Methoden vorgestellt, die helfen sollen, die Eingabe einem Schlüsselwort zuzuordnen, wenn eine direkte Zuordnung nicht möglich war.

5.1. Überprüfung von Homofonen

Listing 5.1: Auszug aus einer beispielhaften Liste von Homofonen `Homophones.txt` [17]

```
cedar,seeder
cede,seed
ceiling,sealing
cell,sell
cellar,seller
censor,sensor
cent,scent,sent
cents,scents,sense
cereal,serial
Ceres,series
cession,session
```

Verschiedene Wörter, die eine identisch ausgesprochen werden, bezeichnet man als Homofon. Eine der Methoden, die verwendet werden, um die Trefferrate der Spracherkennung zu erhöhen, ist das Überprüfen aller möglichen Homofone auf eine Übereinstimmung mit einem existierenden Schlüsselwort. Dabei werden alle einzeln erkannten Worte der Transkription darauf überprüft, ob es dazu ein Homofon gibt.
Aus diesen Homofonen werden alle möglichen Variationen der Transkription generiert und der Reihe nach geprüft, wenn zur ursprünglichen Transkription kein Schlüsselwort gefunden wurde. Als Referenz dient eine Liste von häufig verwendeten Homofonen der englischen Sprache [17].

5.2. Zuordnung von Wortteilen über Textdistanz

Eine Alternative zu den homofonen Variationen stellt die Zuordnung der Transkription zu einem Wortteil über Textdistanz dar. Hierbei werden alle Wortteile der Schlüsselworte

mit der Transkription erneut verglichen und zu jedem die Abweichung zur Transkription, die Textdistanz, errechnet. Es wird der Wortteil gesucht, der die kleinste Distanz zur Transkription aufweist.
Haben mehrere Wortteile die gleiche Textdistanz, so wird derjenige bevorzugt, der zuerst gefunden wurde. Überschreitet die so gefundene kleinste Distanz einen vorher festgelegten Schwellwert, wird das Ergebnis verworfen.

Textdistanz nach Hamming

Die am schnellsten zu ermittelnde Textdistanz ist die Hammingdistanz. Die Hammingdistanz beschreibt die Anzahl an unterschiedlichen Zeichen bei zwei Zeichenfolgen gleicher Länge [14].
Zum Ermitteln der Hammingdistanz werden zu jedem Schlüsselwort alle Wortteile einer festen Länge, die Länge der Transkription entspricht, Zeichen für Zeichen mit der Transkription verglichen und die abweichenden Zeichen gezählt. Schlüsselworte, die kürzer sind als die Transkription, werden hierbei übersprungen.

Textdistanz nach Damerau und Levenshtein

Die Levenshtein-Distanz beschreibt die Anzahl an Editiervorgängen, die vorgenommen werden müssen, um eine Zeichenfolge in eine andere zu überführen [15].
Bei der Levenshtein-Distanz sind die gültigen Operationen das Ersetzen eines Zeichens durch ein anderes, das Löschen eines Zeichens und das Einfügen eines Zeichens. Die Damerau-Levenshtein-Distanz stellt eine Erweiterung der Levenshtein-Distanz dar. Als weitere gültige Operation kommt noch die Transposition – also das Vertauschen – benachbarter Zeichen hinzu [5].
Zur Berechnung der Distanz wird ein dynamischer Algorithmus verwendet, wie er von Navarro für die Levenshtein-Distanz beschrieben wurde [15].
Dieser ist zwar Ineffizient, mit einer Abschätzung von $O(|x||y|)$ für die beiden Zeichenfolgen x und y. Da aber nur relativ kurze Zeichenfolgen verglichen werden, ist der Effizienznachteil nicht schwerwiegend. Der Algorithmus hat den Vorteil, dass er leicht erweiterbar ist. Aus diesem Grund wird er für die Damerau-Levenshtein Distanz verwendet.

Dynamischer Algorithmus zur Berechnung der Levenshtein-Distanz

Der im Folgenden demonstrierte Algorithmus wurde in A *Guided Tour to Approximate String Matching* [15] beschrieben. Die Levenshtein-Distanz wird mit diesem Algorithmus schrittweise berechnet. Das Vorgehen wird hier am Beispiel der Distanz des Wortes ‘Furche’ zum Wort ‘Frucht’ verdeutlicht.
Zunächst wird eine leere Matrix $C_{0\ldots|x|,\,0\ldots|y|}$ generiert, wobei |x| der Länge des ersten Wortes X entspricht, im Beispiel ‘Furche’ also 6. |y| entspricht der Länge des Wortes Y, im Beispiel ‘Frucht’ ebenfalls 6. Im nächsten Schritt wird die erste Zeile und die erste Spalte mit Index 0 jeweils mit dem Zeilen- und Spaltenindex gefüllt (vgl. Tabelle 5.1).

$$C_{i,0} = i$$

		F	u	r	c	h	e
	0	1	2	3	4	5	6
F	1	0	0	0	0	0	0
r	2	0	0	0	0	0	0
u	3	0	0	0	0	0	0
c	4	0	0	0	0	0	0
h	5	0	0	0	0	0	0
t	6	0	0	0	0	0	0

Tabelle 5.1.: Distanzberechnung von Furche zu Frucht: Startbedingungen

		F	u	r	c	h	e
	0	1	2	3	4	5	6
F	1	**0**	**1**	2	3	4	5
r	2	**1**	**1**	**1**	2	3	4
u	3	2	**1**	**2**	3	4	5
c	4	3	2	2	**2**	3	4
h	5	4	3	3	3	**2**	3
t	6	5	4	4	4	3	**3**

Tabelle 5.2.: Distanzberechnung von Furche zu Frucht: Ergebnis

$$C_{0,j} = j$$

Diese Einträge repräsentieren den Abstand eines leeren Wortes zum Wortteil bis zu diesem Punkt. Der Wert, der beispielsweise in $C_{0,2}$ steht, entspricht der Levenshtein-Distanz der ersten beiden Zeichen des Wortes Y, also ‘Fr’, zum leeren Wort.
Im nächsten Schritt wird die Matrix zeilenweise von links oben nach rechts unten gefüllt. Der Wert eines Feldes ergibt sich dabei aus seinen drei Vorgängern, dem linken Nachbarn, dem oberen Nachbarn und dem Nachbarn diagonal links oben. Welchen Wert das Feld $C_{a,b}$ annimmt, hängt davon ab, ob das Zeichen von X an der Stelle a mit dem Zeichen von Y an der Stelle b übereinstimmt.
Wenn man im Beispiel das Feld $C_{1,1}$ betrachtet, dann ist das der Fall, da ‘Furche’ und ‘Frucht’ beide mit dem Zeichen ‘F’ beginnen. In diesem Fall wird der Wert des diagonalen Nachbarn oben links unverändert übernommen. In allen anderen Fällen wird der kleinste Wert der drei Vorgänger genommen und um 1 erhöht.

$$C_{i,j} = \begin{cases} C_{i-1,j-1} & \text{wenn } x_i = y_j \\ 1 + \min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1}) & \text{sonst} \end{cases}$$

Auf diese Weise wird die Matrix vollständig gefüllt. Die gesuchte Levenshtein-Distanz entspricht dem finalen Wert im letzten Feld unten rechts. Für das Beispiel ergibt sich für die Distanz zwischen ‘Furche’ und ‘Frucht’ ein Wert von 3 (vgl. Tabelle 5.2).

		F	u	r	c	h	e
	0	1	2	3	4	5	6
F	1	**0**	1	2	3	4	5
r	2	1	1	2	3	4	5
u	3	2	1	**1**	2	3	4
c	4	3	2	2	**1**	2	3
h	5	4	3	3	2	**1**	2
t	6	5	4	4	3	2	**2**

Tabelle 5.3.: Distanzberechnung von Furche zu Frucht mit Transposition.

Erweiterung zur Berechnung der Damerau-Levenshtein-Distanz

Um mit diesem Algorithmus die Damerau-Levenshtein-Distanz zu errechnen, muss lediglich die Transposition als zusätzliche mögliche Operation zum Entscheidungsknoten hinzugefügt werden.
Die Transposition wird durchgeführt, wenn die beiden aktuellen Zeichen x_i und y_j untereinander verschieden sind, aber beide Identisch zum jeweils vorigen Zeichen des anderen Wortes, also $x_i = y_{j-1}$ und $y_j = x_{i-1}$ gilt. In diesem Fall wird zusätzlich zu den anderen Optionen, das Feld $C_{i-2,j-2}$ als Ausgangspunkt für eine Operation berücksichtigt. Daraus ergibt sich für die modifizierte Beschreibung:

$$C_{i,0} = i$$

$$C_{0,j} = j$$

$$C_{i,j} = \begin{cases} C_{i-1,j-1} & \text{wenn } x_i = y_j \\ 1 + \min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1}, C_{i-2,j-2}) & \text{wenn } x_i = y_{j-1} \wedge x_{i-1} = y_j \\ 1 + \min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1}) & \text{sonst} \end{cases}$$

Um das ganze nochmal mit dem Beispiel ‘Furche’ zu ‘Frucht’ zu veranschaulichen:
Das zweite Zeichen stimmt jeweils mit dem dritten Zeichen des anderen Wortes überein. Hier kann also mit einer Transposition ein Editiervorgang eingespart werden und es ergibt sich eine neue Distanz von 2 (vgl. Tabelle 5.3).

5.3. Generieren von Namen

Der Ansatz zur besseren Generierung von Namen von Variablen und Methoden geht in die entgegengesetzte Richtung wie die Methoden zur Verbesserung der Trefferrate.
Bei der Benennung müssen Namen gewählt werden, die im aktuellen Kontext nicht vorhanden sind. Eine Zuordnung macht in diesem Rahmen keinen Sinn. Stattdessen wird eine Funktion eingebaut, die es ermöglicht, erkannte natürliche Sprache direkt einzufügen.
Die natürliche Sprache wird direkt transkribiert, anschließend um Satzzeichen bereinigt und nach dem Camel Case-Schema formatiert. Von dieser Umgehung des implementierten Prozesses wird nicht unbedingt eine höhere Trefferrate erwartet, sondern viel mehr ein Performanzgewinn bei der Benennung.

6. Evaluation

Zur Bewertung der Effektivität und Effizienz der implementierte Codesynthese und der Methoden zur Verbesserung der Erkennung, wurden mehr als 500 Sprachaufnahmen von verschiedenen Schlüsselworten mit einem Mirkofon angefertigt.
Mit diesen Aufnahmen wurde im jeweils korrekten Kontext geprüft und dokumentiert, ob das jeweilige Schlüsselwort ohne zusätzliche Nachbearbeitung erkannt wurde, ob es mit Homofonerkennung erkannt wurde und ob es mit Homofonerkennung und Zuordnung über Textdistanz nach Damerau und Levenshtein (vgl. Abschnitt 5.2) erkannt wurde. Die akzeptierte Textdistanz wurde im Rahmen dieser Auswertung auf 20 % der Wortlänge, aber mindestens 1 festgelegt.
Neben der Erkennungsrate wurde auch die Zeit gestoppt, die zwischen dem Anstoßen des Erkennungsprozesses und dem Einfügen des synthetisierten Programmcodes vergeht. Als Referenzpunkt dient *Serenade*. Da *Serenade* keine Schnittstelle für Audioaufnahmen hat, wurden die Aufnahmen stattdessen über ein simuliertes Mikrofon eingegeben.

6.1. Trefferrate

Das Ziel dieser Arbeit war der Entwurf und die Implementierung einer besseren Codesynthese. Der Fokus lag dabei auf der Verbesserung der Trefferrate. Wie man in Abbildung 6.1 sehr deutlich erkennen kann, wurde dieses Ziel erreicht.

Vergleich der implementierten Codesynthese mit einer trainierten KI

Im Vergleich zu *Serenade* konnte die Erkennungsrate um mindestens 30 Prozentpunkte verbessert werden, von durchschnittlich 43,65 % auf mindestens 74 % ohne weitere Verbesserung durch Postprocessing.
Schon vor der Messung wurde ein deutlicher Anstieg der Erkennungsrate erwartet, weil die Codesynthese von *Serenade* nur auf allgemeine oder häufige Schlüsselworte einer Programmiersprache trainiert ist und die spezifischen Namen von Variablen, Methoden und Klassen unmöglich kennen und damit erkennen kann.
Insbesondere wenn nach modernen Standards zur Strukturierung von Programmcode entwickelt wird, wie zum Beispiel nach den Prinzipien von *Clean Code* [6], ist die Verwendung von projektspezifischen Schlüsselworten ein sehr großer Bestandteil der Entwicklungsarbeit.

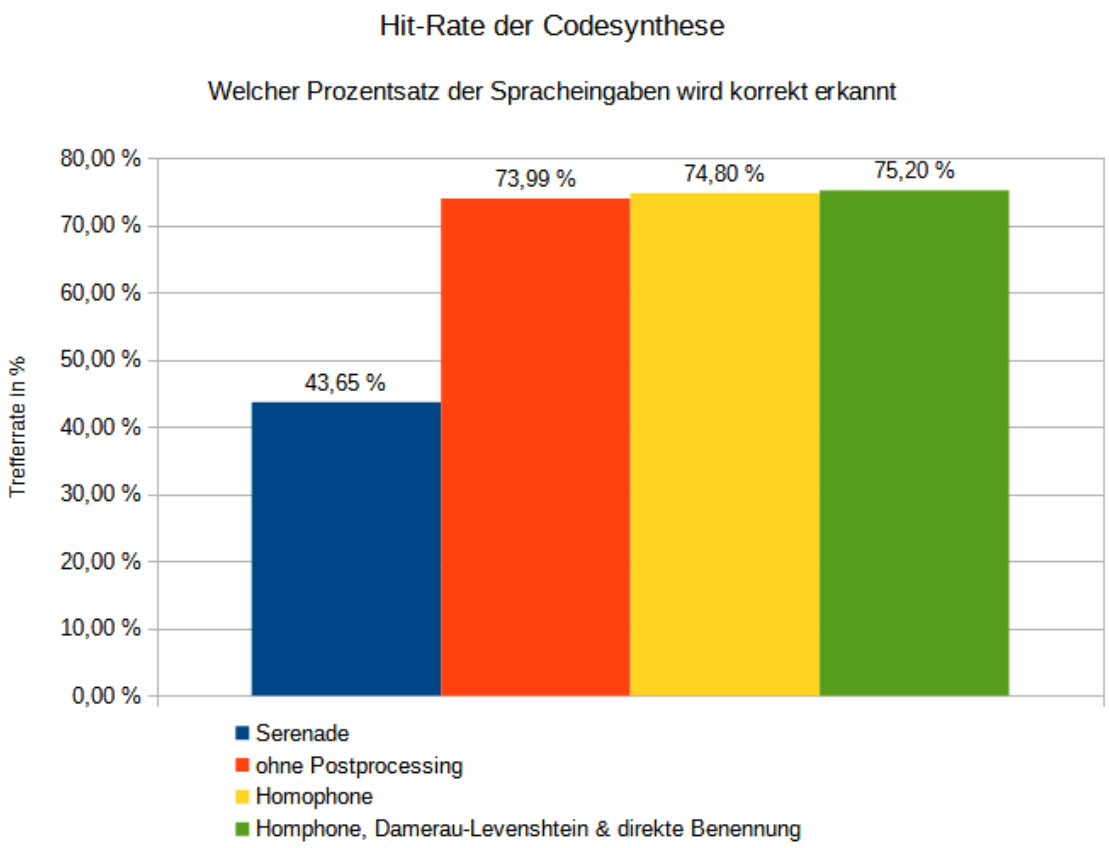

Abbildung 6.1.: Visualisierung der gemessenen Trefferrate

Effektivität der zusätzlichen Methoden zur Verbesserung der Erkennung

Die Auswirkungen der in Kapitel 5 vorgestellten Methoden zur Erkennungsrate der Trefferrate sind dagegen eher gering. Die Verwendung von Homofonen verbessert die Trefferrate um 0.8 Prozentpunkte und der Textdistanz-Algorithmus nach Damerau und Levenshtein verbessert die Trefferrate lediglich um weitere 0.4 Prozentpunkte. Bei etwas mehr als 500 Datenpunkten können diese Verbesserungen leicht als Messfehler erklärt werden. Hier wäre eine größere Messreihe mit deutlich mehr Datenpunkten im Kontext verschiedener Entwicklungsprojekte notwendig um ein abschließendes Urteil treffen zu können, ob die verwendeten Methoden überhaupt einen Einfluss auf die Trefferrate haben.
Insbesondere bei den Homofonen wäre auch interessant zu prüfen, ob die verwendete Liste (siehe Anhang A) umfangreich genug war oder ob eine umfangreichere Liste zu einer merklichen Steigerung der Trefferrate betragen würde.

6.2. Performanz

Die Messung der Reaktionszeit wurde manuell mit einer Stopuhr durchgeführt. Aufgrund der nicht konstanten menschlichen Reaktionszeit sind also erhöhte Schwankungen zu erwarten. Der Versuch einer Implementierung über eine programmatische Zeitmessung mithilfe einer Listener-Schnittstelle hat zu Störungen im Programmablauf geführt. Da vor der Synthese durch den Spracherkennungsdienst auf die Ergebnisse der Autovervollständigung gewartet werden muss, wurde mit einer Verschlechterung der Performanz gerechnet.

Der direkte Vergleich mit *Serenade* ist nur beschränkt möglich, da *Serenade* keine direkte Eingabe von Sprachaufnahmen zulässt. Die Spracheingaben wurden stattdessen über ein simuliertes Mikrofon eingegeben. Um dies zu kompensieren, wurde von der gemessenen Reaktionszeit die durchschnittliche Länge der Spracheingaben abgezogen. Damit ergibt sich eine Steigerung der Reaktionszeit gegenüber *Serenade* von etwa 300 ms. Die durchschnittliche Performanz sinkt also nur wenig.
Allerdings gibt es Kontexte, in denen es bis zu 6 s dauern kann, bis der Autovervollständigungsprozess abgeschlossen ist und mit der Codesynthese begonnen werden kann. Die Performanz von *Serenade* dagegen bleibt sehr konstant.

Die gemessene durchschnittliche Reaktionszeit ist bei Verwendung von Homofonen im Postprocessing um etwa 50 ms länger. Dieser Wert ist so klein, dass er leicht auf Messungenauigkeiten zurückgeführt werden kann und keine wirkliche Aussage über die Auswirkungen der Methode auf die Performanz zulässt.

Das Verfahren zur direkten Bennenung von Variablen war dagegen sehr effektiv. Obwohl nur vergleichsweise wenige aufgenommene Spracheingaben in den Beispieldaten den Zweck der Bennenung von Variablen hatten, wurde die durchschnittliche Reaktionszeit um 100 ms gesenkt. Im direkten Vergleich war die Reaktionszeit fast eine Sekunde schneller. Da allerdings eine zusätzliche Eingabe über Hotkeys gemacht werden muss, um

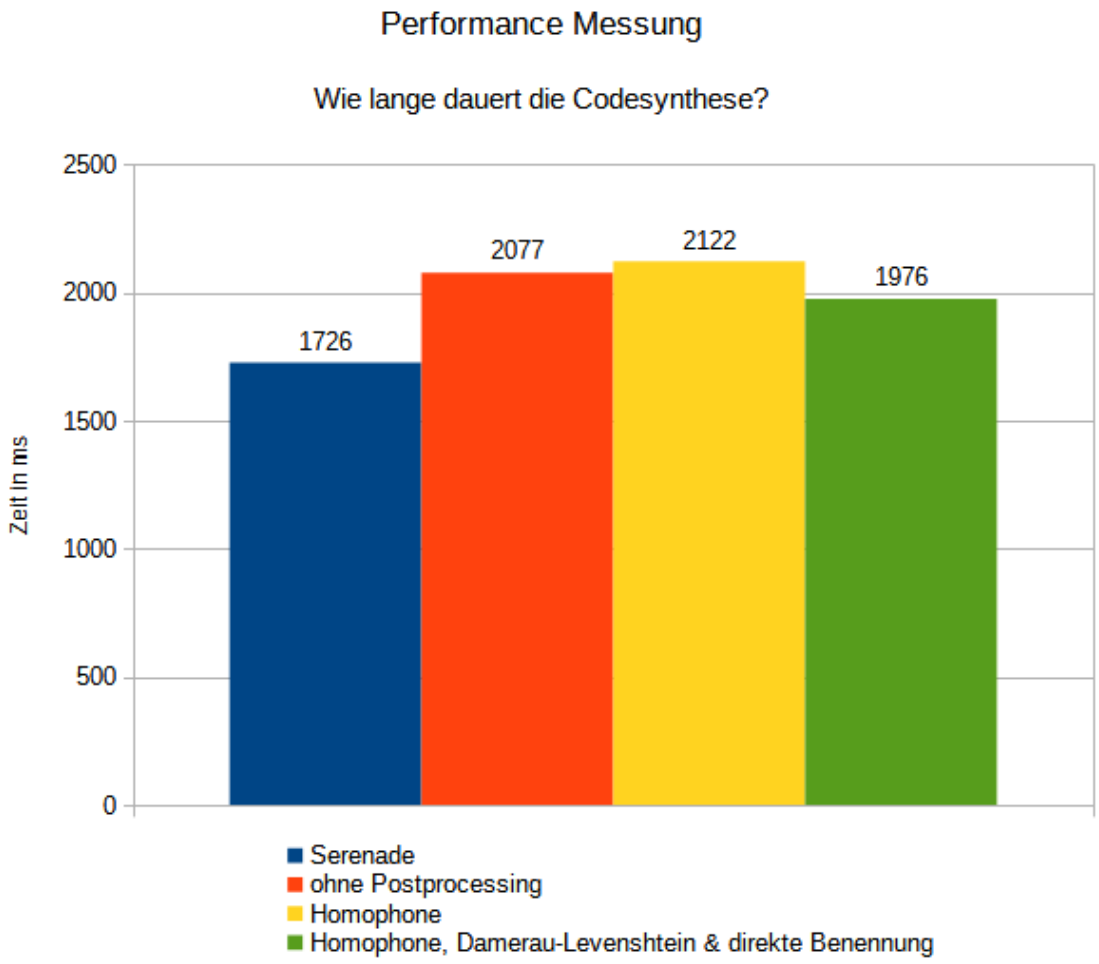

Abbildung 6.2.: Visualisierung der gemessenen Reaktionszeit

die Codesynthese in den Benenungszustand zu bringen, ist fraglich, ob diese Verbesserung lohnenswert ist.

6.3. Bewertung der Ergebnisse

Insgesamt konnte gezeigt werden, dass die entwickelte Methode zur Codesynthese eine deutliche Verbesserung gegeüber dem aktuellen Stand der Technik, wie er in *Serenade* implementiert ist, darstellt. Zwar ist die Reaktionszeit höher, der Anstieg bewegt sich aber in einem akzeptablen Rahmen. Die Steigerung der Erkennungsrate ist dafür bemerkenswert. Es konnten mit der neuen Methode fast doppelt so viele Schlüsselworte korrekt erkannt werden.
Dagegen haben die getesteten Methoden im Postprocessing zur Verbesserung der Erkennung leider keine zweifelfrei messbare Verbesserung demonstrieren können. Hier wären weitere Messreihen in mehr Kontexten hilfreich, um die Effektivität zweifelfrei bewerten zu können.

7. Ausblick

Kommandos

Mit der in dieser Arbeit implementierten Codesynthese kann die Spracheingabe von Programmcode deutlich effektiver gestaltet werden. Allerdings ist die Codesynthese nur ein Aspekt der Sprachassistenz. Die große Stärke von *Serenade* ist nicht die Codesynthese, sondern die Steuerung der Entwicklungsumgebung mit Kommandos.
Im Rahmen der Arbeit wurde zwar ein Rahmen für die Implementierung von Sprachkommandos geschaffen, allerdings ist die Implementierung von diesen Kommandos nicht trivial, sondern erfordert ein tiefes Verständnis der Funktionsweise der *IntelliJ*-Plattform. Da der Quellcode von *Serenade* öffentlich und Open Source ist [7], gibt es zwei Möglichkeiten für das weitere Vorgehen.
Die erste Möglichkeit ist, die Implementierung der Befehle des *IntelliJ*-Plugins von *Serenade* zu kopieren. Dies ist allerdings aufwändig und nicht unbedingt sinnvoll, da jedes neue Kommando von *Serenade* erneut kopiert werden muss, um die gleichen Features zu bieten.
Die zweite Möglichkeit ist, das implementierte Plugin auf die Nutzung als Ergänzung zu *Serenade* zu optimieren. Es ist möglich, eigene Sprachebefehle in *Serenade* zu definieren. Man könnte also mit diesem Ansatz die sehr gute Kommandoinfrastruktur von *Serenade* zur Bedienung der Entwicklungsumgebung nutzen und die verbesserte Codesynthese über ein Kommando auslösen. Der Nachteil dieses Ansatzes ist, dass man sich von einem anderen Programm abhängig macht und das Risiko eingeht, dass der Support für *Serenade* eingestellt wird.

Die *IntelliJ*-Plattform

Das entwicklete Plugin verwendet von der *Rider*-Plugin-Infrastruktur ausschließlich die Funktionen der *IntelliJ Platform Plugin SDK*. Das Plugin kann also, nach Verständnis des Autors, ohne Funktionsverluste in ein allgemeineres *IntelliJ*-Plugin umgewandelt werden. Der Vorteil, das *Rider*-Plugin in ein *IntelliJ*-Plugin umzuwandeln, ist, dass es damit mit allen Entwicklungsumgebungen genutzt werden kann, die auf der *IntelliJ*-Plattform basieren. Aktuell basieren laut *JetBrains* 14 verschiedene Entwicklungsumgebungen auf der *IntelliJ*-Plattform [9], der Umstieg wäre also ein deutlicher Gewinn an Nutzbarkeit.

Literatur

[1] *Azure Cognitive Services documentation*. https://learn.microsoft.com/en-us/azure/cognitive-services/. Accessed: 2022-12-15.

[2] bluehands GmbH & Co.mmunication KG. *bluehands Softwareentwicklung*. https://www.bluehands.de/. Accessed: 2023-03-29.

[3] *Cloud Speech-to-Text*. https://cloud.google.com/speech-to-text?hl=de. Accessed: 2022-12-15.

[4] 878 verschiedene Contributors. *intellij–community Programmcode*. https://github.com/JetBrains/intellij-community/. 2023.

[5] Fred J. Damerau. "A Technique for Computer Detection and Correction of Spelling Errors". In: *Communications of the ACM* 7.3 (1964).

[6] One Man Think Tank EOOD. *Clean Code Developer*. https://clean-code-developer.de/. Accessed: 2023-03-29.

[7] Serenade Labs Inc. *Serenade Blog*. https://serenade.ai/blog/. Accessed: 2023-03-15.

[8] JetBrains s.r.o. *Essential tools for software developers and teams*. https://www.jetbrains.com/. Accessed: 2023-03-29.

[9] JetBrains s.r.o. *IntelliJ Platform SDK*. https://plugins.jetbrains.com/docs/intellij. Accessed: 2023-01-05.

[10] ut-karsh, eric-urban. *Improve recognition accuracy with phrase list*. https://learn.microsoft.com/en-us/azure/cognitive-services/speech-service/improve-accuracy-phrase-list. Accessed: 2023-01-15. 2023.

[11] *Kotlin*. https://kotlinlang.org/. Accessed: 2023-03-29.

[12] Matthias Koch. *Writing plugins for ReSharper and Rider*. https://blog.jetbrains.com/dotnet/2019/02/14/writing-plugins-resharper-rider/. Accessed: 2022-12-15. 2019. DOI: 2019.02.14.

[13] Microsoft. *Language Understanding (LUIS)*. https://www.luis.ai/. Accessed: 2022-12-15.

[14] Technology & Standards Division National Commmunications System. *Telecommunications: Glossary of Telecommunication Terms*. General Services Administriation, Infromation Technology Services, 1996.

[15] Gonzalo Navarro. *A Guided Tour to Approximate String Matching*. Blanco Encalada 2120 - Santiago - Chile.

[16] Oracle. *Java Platform, Standard Edition Documentation*. https://docs.oracle.com/en/java/javase/. Accessed: 2022-12-18.

[17] English Tutor. *Homophones List: 400+ Common Homophones in English for ESL Learners!* https://myenglishtutors.org/homophones-list/. 2019. DOI: 2019.01.09.

[18] Zipp. *Programmieren mit Spracherkennung*. https://www.repetitive-strain-injury.de/erfahrungsbericht-programmieren-mit-spracherkennung.php. Accessed: 2023-03-15. 2020.

A. Anhang

Listing A.1: Verwendete Homphonliste `Homophones.txt` [17]

```
%List of Homophones
%Source: https://myenglishtutors.org/homophones-list/
accede,exceed
accept,except
addition,edition
adds,adz,ads
affect,effect
affected,effected
ale,ail
all,awl
ant,aunt
apatite,appetite
apprise,apprize
arc,ark
ariel,aerial
ark,arc
arrant,errant
ascent,assent
assistance,assistants
ate,eight
atom,Adam
ax,acts
axel,axle
axes,axis
aye,eye,I
ayes,eyes
baa,bah
baal,bail,bale
bait,bate
baited,bated
bald,balled,bawled
bale,baal,bail
ball,bawl
balled,bawled,bald
basal,basil
base,bass
based,baste
bases,basis
basil,basal
basis,bases
bask,basque
bass,base
baste,based
```

```
bate,bait
bated,baited
bawl,ball
been,bin
beer,bier
beet,beat
bell,belle
berry,bury
berth,birth
better,bettor
bib,bibb
bight,bite,byte
billed,build
bin,been
bird,burred
birth,berth
bite,byte,bight
bizarre,bazaar
blew,blue
bloc,block
bolder,boulder
bomb,bombe,balm
bootie,booty
border,boarder
bore,boar
bored,board
born,borne
borough,burro,burrow
bough,bow
bouillon,bullion
braid,brayed
braise,brays
brake,break
brayed,braid
brays,braise
breach,breech
bread,bred
break,brake
bred,bread
breech,breach
brewed,brood
brews,bruise
bridal,bridle
broach,brooch
burro,burrow,borough
bury,berry
bussed,bust
but,butt
buy,by,bye
byte,bight,bite
cache,cash
caddie,caddy
Cain,cane
calendar,calender
```

```
caster,castor
cause,caws
cedar,seeder
cede,seed
ceiling,sealing
cell,sell
cellar,seller
censor,sensor
cent,scent,sent
cents,scents,sense
cereal,serial
Ceres,series
cession,session
chance,chants
chased,chaste
chauffeur,shofar
cheap,cheep
check,Czech
cheep,cheap
chews,choose
chic,sheik
Chile,chilly,chili
choir,quire
choose,chews
choral,coral
chord,cord,cored
chute,shoot
cite,sight,site
cited,sided,sighted
clack,claque
clamber,clammer,clamor
claque,clack
clause,claws
clew,clue
click,clique
climb,clime
clique,click
close,clothes,cloze
clue,clew
coal,cole
coarse,course
coarser,courser
coat,cote
coax,cokes
coffers,coughers
cokes,coax
cole,coal
collard,collared
colonel,kernel
coolie,coulee
coop,coupe
cops,copse
coral,choral
cord,cored,chord
```

```
core,corps
cored,chord,cord
corps,core
coughers,coffers
coulee,coolie
council,counsel
coup,coo
course,coarse
courser,coarser
cousin,cozen
coward,cowered
cozen,cousin
craft,kraft
crape,crepe
crawl,kraal
creak,creek
crepe,crape
crewel,cruel
crews,cruise
cruel,crewel
cruise,crews
dam,damn
Dane,deign
days,daze
dear,deer
defused,diffused
deign,Dane
dense,dents
descent,dissent
desert (abandon),dessert
dew,do,due
die,dye
diffused,defused
disburse,disperse
discreet,discrete
disperse,disburse
dissent,descent
duct,ducked
ducts,ducks
due,dew,do
duel,dual
dun,done
dye,die
dyeing,dying
edition,addition
educe,adduce
eek,eke
effect,affect
effected,affected
effects,affects
eight,ate
eke,eek
elicit,illicit
elude,allude
```

```
errant,arrant
eve,eave
ewe,you,yew,u
ewes,yews,use
exceed,accede
except,accept
facts,fax
faint,feint
fair,fare
fairy,ferry
fare,fair
fate,fete
faun,fawn
fax,facts
faze,phase
feat,feet
feint,faint
fends,fens
ferry,fairy
fete,fate
few,phew
flair,flare
flea,flee
flew,flu,flue
flier,flyer
flocks,phlox
floe,flow
flour,flower
flow,floe
flower,flour
flu,flue,flew
flyer,flier
foaled,fold
fort,forte
forth,fourth
forward,foreword
foul,fowl
four,fore,for,4
fourth,forth
fowl,foul
franc,frankfur,fir
gaff,gaffe
Gail,gale
gait,gate
gale,Gail
gamble,gambol
gel,jell
gene,jean
gibe,jibe
gnu,knew,new
gofer,gopher
gored,gourd
gorilla,guerilla
gourd,gored
```

```
grade,grayed
graft,graphed
hail,hale
hair,hare
hale,hail
hall,haul
halve,have
handmade,handmaid
handsome,hansom
hangar,hanger
hay,hey
hays,haze
heal,heel
hear,here
heard,herd
heart,hart
heroin,heroine
hertz,hurts
hew,hue,Hugh
hey,hay
hi,high
higher,hire
him,hymn
hire,higher
ho,hoe
hoard,horde
hoarse,horse
hoe,ho
hoes,hose
hold,holed
hole,whole
holed,hold
hour,our
hue,Hugh,hew
humerus,humorous
hurts,hertz
hymn,him
I,aye,eye
aisle,isle
idle,idol,idyll
illicit,elicit
illusion,allusion
illusive,allusive,elusive
incite,insight
inn,in
innocence,innocents
knap,nap
knave,nave
ladder,latter
lade,laid
lain,lane
lays,laze,leis
lea,lee
leach,leech
```

```
lead,led
leak,leek
lean,lien
leased,least
led,lead
lee,lea
leech,leach
liar,lier,lyre
madder,matter
made,maid
mail,male
main,mane,Maine
maize,maze
medal,metal,mettle,meddle
mete,meat,meet
meteor,meatier
mints,mince
missal,missile
missed,mist
misses,Mrs.
missile,missal
mist,missed
mite,might
moan,mown
moat,mote
mode,mowed
mood,mooed
moose,mousse
morn,mourn
morning,mourning
mote,moat
mourn,morn
mourning,morning
mousse,moose
mowed,mode
mown,moan
Mrs.,misses
mucous,mucus
mule,mewl
muscle,mussel
mustard,mustered
nap,knap
naval,navel
nave,knave
navel,naval
nay,neigh
need,knead,kneed
neigh,nay
new,gnu,knew
nice,gneiss
Nice,niece
pain,pane
pair,pare,pear
palate,palette,pallet
```

```
pale,pail
parish,perish
parlay,parley
passed,past
paste,paced
patience,patients
patted,padded
pea,pee
peace,piece
peak,peek,pique
peal,peel
pie,pi
piece,peace
pier,peer
pigeon,pidgin
Pilate,pilot
pique,peak,peek
pistil,pistol
plum,plumb
read,red
read,reed
real,reel
red,read
reed,read
reek,wreak
reel,real
reign,rein,rain
residence,residents
seam,seem
sear,seer,sere
seas,sees,seize
see,sea
seed,cede
sighed,side
sighs,size
sight,site,cite
sighted,cited,sided
stayed,staid
steak,stake
steal,steel
there,their
weighs,ways
weight,wait
were,whir
wet,whet
wether,weather,whether
why,y
on,one,1
to,two,2
equals,==
is,=
x,ex,eggs
jay,j
```

Hits	Datensatz 1	Datensatz 2	Datensatz 3	Datensatz 4	Datensatz 5	Datensatz 6	Datensatz 7	Datensatz 8	Datensatz 9	Datensatz 10
File 1	0	0	0	0	0	0	0	0	1	1
File 2	1	1	1	0	0	0	1	1	1	1
File 3	0	0	0	0	0	0	0	0	0	0
File 4	0	0	0	0	0	0	0	0	0	0
File 5	0	0	0	0	0	0	0	0	1	1
File 6	0	0	0	0	0	1	0	0	0	1
File 7	1	0	1	1	0	0	0	0	0	0
File 8	0	0	1	0	0	0	0	0	0	0
File 9	1	1	1	0	0	0	0	0	0	1
File 10	1	0	0	1	1	1	0	0	0	0
File 11	1	1	1	0	0	0	0	0	0	1
File 12	1	1	0	1	1	1	0	1	1	1
File 13	1	0	1	0	1	0	1	0	1	1
File 14	1	1	1	0	0	0	1	1	1	1
File 15	0	0	0	0	0	1	1	1	0	1
File 16	0	0	1	1	1	1	1	1	0	0
File 17	1	1	1	1	0	1	0	0	0	0
File 18	1	0	1	0	0	1	0	0	0	0
File 19	0	0	0	0	1	0	0	1	1	1
File 20	0	1	1	1	0	1	1	0	0	1
File 21	1	0	0	1	0	0	0	0	0	0
File 22	1	1	1	0	1	1	0	1	1	1
File 23	1	0	1		1	0	1	1	1	1
File 24	1						1	1	1	1
File 25							1	1	0	0
File 26							0			

Tabelle A.1.: Treffer für Datenpaket 1 in Serenade (1 bedeutet korrekt erkannt, 0 nicht erkannt)

Hits	Datensatz 1	Datensatz 2	Datensatz 3	Datensatz 4	Datensatz 5	Datensatz 6	Datensatz 7	Datensatz 8	Datensatz 9	Datensatz 10
File 1	1	1	1	1	1	1	1	1	1	1
File 2	0	0	0	0	0	0	0	1	0	0
File 3	0	1	0	0	0	0	0	0	0	0
File 4	0	0	0	1	1	1	1	0	0	0
File 5	0	0	0	0	0	0	0	0	0	0
File 6	0	1	0	1	1	1	0	1	1	1
File 7	0	0	1	0	0	0	0	0	0	0
File 8	1	1	0	0	0	0	1	0	0	1
File 9	1	1	0	0	0	0	0	0	0	0
File 10	0	0	0	0	0	0	1	0	0	0
File 11	0	0	1	0	0	0	0	0	0	0
File 12	1	1	0	0	0	1	1	0	0	1
File 13	1	0	0	0	0	1	0	1	1	0
File 14	0	0	1	1	1	0	1	1	1	1
File 15	1	0	1	1	1	0	1	0	0	1
File 16	0	0	0	0	0	0	0	0	0	0
File 17	0	0	0	1	0	1	1	0	0	1
File 18	1	0	0	1	0	1	1	0	1	1
File 19	0	1	1	1	1	0	1	1	1	1
File 20	1	1	0	1	1	1	1	0	1	0
File 21	1	1	1	1	1	1	0	0	0	0
File 22	0	0	0	0	0	0	0	0	1	1
File 23	1	1	1	0	0	1	1	0	0	0
File 24	1	1	1	1	1	1	0	1	1	1
File 25	1	0	1	0	0	0	0	1	0	0
File 26	1	1	0	1	1	0	0	0	0	0
File 27	1	1	1	1	0	1	0	0	1	0
File 28	1	0	0	0	0	1	0	1	1	1
File 29	0	0	0	0	0	1	1	1	1	1
File 30		1	1	1	1	1	1	0	0	1
File 31		1	1	0	1	1	1	0	1	0
File 32		1	0		1	0	1	1	0	1
File 33		0	1		1		0	0		0
File 34		1	1		0					
File 35		0	0							

Tabelle A.2.: Treffer für Datenpaket 2 in Serenade (1 bedeutet korrekt erkannt, 0 nicht erkannt)

Time per File	Datensatz 1	Datensatz 2	Datensatz 3	Datensatz 4	Datensatz 5	Datensatz 6	Datensatz 7	Datensatz 8	Datensatz 9	Datensatz 10
File 1	1735	1828	2000	2016	1875	1812	1985	1922	3266	2031
File 2	1906	1735	2907	4609	1781	3125	1985	2328	1875	2531
File 3	1828	2687	2062	1953	3047	1922	1812	4141	2265	2079
File 4	3578	1891	1953	1937	2172	2500	3469	3516	3578	2562
File 5	1953	2203	3016	2750	1985	1797	1797	1594	2234	2203
File 6	1907	1985	1766	1890	3110	2093	2328	1859	3312	3125
File 7	1938	3593	2203	2109	3109	2031	1609	2312	2140	2157
File 8	2047	1938	2766	1953	1922	2156	2719	9516	2250	2219
File 9	1859	2188	1687	2141	1829	1859	1781	1969	2172	2140
File 10	1797	1828	1422	1844	1844	1891	1906	2172	1922	2422
File 11	2422	2500	1703	1750	1609	1610	3218	2172	1922	2015
File 12	1860	2063	7515	1672	1969	1687	3250	2625	1859	1890
File 13	1657	1907	1953	1875	1562	2000	1922	1985	1906	2016
File 14	1656	1672	2032	1672	2391	2297	2016	1984	1906	2047
File 15	2125	2250	1781	2062	2047	1719	2375	2031	1968	2391
File 16	1984	1906	1813	2000	1937	1828	1968	2172	1672	2079
File 17	2188	2609	2953	1719	1781	1781	1797	2047	2031	2235
File 18	2047	1766	1390	1703	2000	1812	1750	2172	2140	2094
File 19	1891	2078	1797	1766	1719	1735	2406	2015	2219	1938
File 20	2031	2250	3063	1828	1922	1735	1891	2234	2172	2985
File 21	2297	1969	1953	2812	2859	2438	1547	1907		2047
File 22	2578	1781		1907	2093	2500	2016	2015		
File 23	1922	2859					4687	2313		
File 24	3750	1907					2500	2234		
File 25							3094			
File 26							8234			

Tabelle A.3.: Abspieldauer und Reaktionszeit in ms für Datenpaket 1 in Serenade

Time per File	Datensatz 1	Datensatz 2	Datensatz 3	Datensatz 4	Datensatz 5	Datensatz 6	Datensatz 7	Datensatz 8	Datensatz 9	Datensatz 10
File 1	2047	2188	2125	2453	2187	1875	1922	1843	2031	2109
File 2	2141	1985	2282	2109	2188	2109	2063	2875	2109	2312
File 3	2172	1985	1969	1812	2437	2391	2344	3141	1891	1953
File 4	2094	1922	2500	1984	2141	2063	2032	1890	1859	2328
File 5	2187	2375	2250	1938	2250	1890	2125	2219	2328	2562
File 6	2000	1625	1875	2610	1844	1922	2234	2969	1609	2109
File 7	2219	2000	2343	2437	2266	2125	2312	2437	2391	2641
File 8	2281	1875	1937	1656	3625	1797	2141	2203	1968	2375
File 9	3109	2172	2437	3484	1906	1953	2140	1953	1984	2390
File 10	1781	1578	2359	2063	2109	2203	2281	2375	2266	1953
File 11	1969	1813	3047	2484	2188	2203	2093	2156	2781	2094
File 12	2297	1688	2359	2390	1797	2109	1844	2031	1969	2375
File 13	2235	1907	2094	2172	1735	2093	1953	2235	2532	2219
File 14	2188	1781	1688	2312	2032	2203	1922	1829	2140	2250
File 15	2750	1907	2047	1734	1735	2735	3219	2656	2344	2375
File 16	1828	2375	2610	2375	2032	3313	2250	2672	2719	1969
File 17	2062	2125	2046	2140	2046	2187	1906	2750	1984	1641
File 18	2141	2188	1954	2641	2140	2156	2188	3547	2203	1734
File 19	1954	1828	1797		2031	2422	2187	2438	8000	1781
File 20	1781	1969	2203		1953	2218	1953	2265	2360	2156
File 21	2656	2235	2047		2532	2156	2109	3156	2047	2344
File 22	532	1719	2328		1781	2625	2266	2203	1922	2266
File 23	2297	2032	2203		2172	2297	2781	2422	2547	2062
File 24	1984	2156			2125	3859	2250	2266	2500	2141
File 25	2141				2234	2156	2484	2032	1875	2844
File 26	1813				2516	1984	3125	2391	2312	2125
File 27	2219				1781	2093	1953	2360	2015	1953
File 28	1781				1922	2297	1937		1969	2171
File 29	2047				2141	1969	1860		2219	2156
File 30	2282				1813	2156	2079		2312	2297
File 31					2094		2172		2109	2469
File 32					2234				2312	

Tabelle A.4.: Abspieldauer und Reaktionszeit in ms für Datenpaket 2 in Serenade

Hits	Datensatz 1	Datensatz 2	Datensatz 3	Datensatz 4	Datensatz 5	Datensatz 6	Datensatz 7	Datensatz 8	Datensatz 9	Datensatz 10
File 1	1	1	0	0	1	0	0	0	0	0
File 2	0	1	0	1	1	1	0	0	0	0
File 3	1	1	1	1	1	0	0	1	1	0
File 4	1	1	1	1	0	1	1	1	1	1
File 5	0	1	1	1	1	1	0	0	0	0
File 6	0	0	1	1	0	0	0	0	0	0
File 7	1	1	1	0	1	1	1	1	1	0
File 8	1	1	1	1	0	1	1	1	1	1
File 9	1	1	0	1	1	1	0	1	1	1
File 10	1	1	0	1	1	1	1	1	1	1
File 11	0	0	0	1	0	0	1	0	1	1
File 12	1	1	1	0	1	1	1	1	1	1
File 13	1	1	0	1	1	0	1	1	1	0
File 14	0	0	1	1	0	0	0	1	1	1
File 15	1	1	1	0	1	1	1	1	0	1
File 16	1	1	1	1	1	1	1	1	1	0
File 17	0	0	1	1	1	1	0	1	0	1
File 18	1	1	1	1	1	1	1	1	1	1
File 19	1	1	0	1	1	1	0	1	1	1
File 20	1	1	0	1	0	1	1	1	1	1
File 21	1	0	1	0	1	0	1	0	0	1
File 22	1	1	1	1	1	1	1	1	1	0
File 23	1	0		1		0	1	1	1	1
File 24	0	1					0	0		0
File 25	1	0					1	1		
File 26							1	1		

Tabelle A.5.: Treffer für Datenpaket 1 ohne Postprocessing (1 bedeutet korrekt erkannt, 0 nicht erkannt)

Hits	Datensatz 1	Datensatz 2	Datensatz 3	Datensatz 4	Datensatz 5	Datensatz 6	Datensatz 7	Datensatz 8	Datensatz 9	Datensatz 10
File 1	1	1	1	1	1	1	1	1	1	1
File 2	1	1	1	1	1	1	1	1	1	1
File 3	1	1	0	1	1	1	1	0	1	1
File 4	1	0	0	0	1	1	1	1	1	0
File 5	1	1	0	0	1	1	1	0	0	1
File 6	0	1	0	1	0	0	0	0	1	0
File 7	1	1	1	1	1	1	1	1	1	1
File 8	1	0	1	1	1	1	1	1	1	1
File 9	1	1	1	1	1	1	1	1	1	1
File 10	1	1	1	0	1	1	1	1	1	1
File 11	0	1	1	0	1	0	1	1	0	1
File 12	0	1	0	1	0	0	0	0	0	0
File 13	1	0	1	1	0	0	1	1	0	0
File 14	1	1	1	1	1	1	1	0	1	1
File 15	1	1	1	1	1	1	1	1	1	0
File 16	1	1	1	1	1	1	1	1	0	1
File 17	1	1	1	1	1	0	1	0	1	1
File 18	1	1	1	0	1	1	1	1	1	0
File 19	1	1	1	1	1	1	1	1	0	1
File 20	1	0	1	0	1	1	1	0	1	1
File 21	1	1	1	1	1	1	1	1	1	0
File 22	0	1	1	1	1	1	0	1	1	1
File 23	1	1	1	1	1	1	1	1	0	1
File 24	1	1	1	1	0	1	0	0	1	0
File 25	1	1	1	1	1	1	1	1	0	1
File 26	1	1	1	1	1	1	1	1	1	0
File 27	1	1	1		1	1	1	1	1	1
File 28	1	1	0		1	1	1	1	1	1
File 29	1	1			1	1	1	1	1	1
File 30	1	0			1	1	0	0	1	1
File 31	1				1	1	1	1	0	1
File 32	0				1	1	1	1	1	0
File 33					1	1			1	1
File 34					1	0				1

Tabelle A.6.: Treffer für Datenpaket 2 ohne Postprocessing (1 bedeutet korrekt erkannt, 0 nicht erkannt)

Time per File	Datensatz 1	Datensatz 2	Datensatz 3	Datensatz 4	Datensatz 5	Datensatz 6	Datensatz 7	Datensatz 8	Datensatz 9	Datensatz 10
File 1	2035	1766	1922	1906	1969	1984	1704	1875	2281	1860
File 2	1938	1718	4812	4984	4656	4968	4750	2531	1703	1703
File 3	4907	4921	2016	1860	1734	2250	1781	5312	5109	4891
File 4	1860	1843	1796	1828	1969	4750	1875	2203	1704	1781
File 5	1937	1890	4750	5063	5563	1704	1578	1796	2344	2000
File 6	1766	1609	2046	1625	1906	1734	4953	1782	1594	1610
File 7	5062	4922	1938	1891	1438	1485	1891	4062	2188	5188
File 8	2000	1891	1531	1578	1922	1860	1812	1735	1578	1750
File 9	2140	2047	1656	1828	1719	1641	1562	2032	2360	1859
File 10	1703	1547	2109	2281	1875	1828	2079	1562	1875	1610
File 11	2016	2296	2062	1750	1828	1781	1766	2328	2297	2266
File 12	1750	1844	1906	2125	1844	1750	2094	2203	1859	2047
File 13	2156	2438	1719	1843	2109	1860	2062	2235	1953	1984
File 14	1969	4656	1828	1891	1781	1859	1813	3203	2016	1953
File 15	1828	7375	1937	1843	1907	1875	1828	2093	1891	1859
File 16	1984	1797	1875	1813	1765	1907	1750	1843	2109	1953
File 17	2031	1907	2141	1828	1844	1797	2016	1953	1859	1828
File 18	1984	2062	1766	1562	1812	1531	1859	2000	1968	1891
File 19	2031	1703	1625	1860	1922	1781	2531	2109	1922	2187
File 20	2296	1985	1953	1859	1875	1875	2156	1937	1812	1922
File 21	2047	1734	2093	1890	1953	1797	1656	1656	1875	1984
File 22	1593	2063	1688		1813	2031	2250	2016		1938
File 23	1985	2078	1875				2203	2094		1906
File 24	1813	1953					1891	1781		2125
File 25	2329						1828	1859		

Tabelle A.7.: Reaktionszeit in ms für Datenpaket 1 ohne Postprocessing

Time per File	Datensatz 1	Datensatz 2	Datensatz 3	Datensatz 4	Datensatz 5	Datensatz 6	Datensatz 7	Datensatz 8	Datensatz 9	Datensatz 10
File 1	2047	2063	2218	1812	1953	1968	1984	1906	2156	1656
File 2	2000	2032	1922	2297	2391	2187	2110	2187	1937	2078
File 3	1750	1906	1703	1657	1813	1828	1859	1797	1750	1703
File 4	1969	2079	1782	1890	2000	1953	1843	1969	2141	2062
File 5	1610	1532	1609	1609	1547	1657	1594	1656	1734	1703
File 6	1937	2281	1859	2000	1844	2078	1906	1953	1859	2172
File 7	2031	1453	1859	1969	2234	2110	1922	2016	2141	2235
File 8	2000	1500	1875	2109	1984	2031	2078	2093	2000	2172
File 9	2109	2094	1953	2125	2281	1906	2047	2000	2125	2000
File 10	1719	1625	1922	1922	2079	2047	2046	2157	1625	2422
File 11	2516	2016	1531	2047	1656	1640	1625	1734	1906	1656
File 12	1953	1906	1797	2078	1953	1922	2297	1797	2000	2093
File 13	2359	1500	1953	1641	1891	2031	1656	1906	1968	2297
File 14	1750	1828	1562	2157	1672	1593	1844	1797	2000	2328
File 15	2156	2093	1750	1593	1953	1719	2344	1969	2078	2125
File 16	1875	1500	2171	1891	2063	1859	2031	1563	1953	2094
File 17	1953	1937	1532	1969	1515	1531	1718	2000	2094	2015
File 18	1719	2110	1828	1656	1953	1907	1953	2250	2062	2079
File 19	2125	1563	2078	2344	2031	2360	2344	1578	1781	1672
File 20	2063	2110	1688	2109	1562	1937	1640	1937	1719	2203
File 21	1656	2171	1500	2203	2125	1859	1922	2172	2109	2031
File 22	1875	2219	2109	1703	2078	1687	1968	1782	1875	1594
File 23	1906	2063	1657	1688	1641	2063	1610	11297	1891	2594
File 24	1562	1547	2062	1844	1969	2078	2047	1609	2125	2375
File 25	2031	1953	1515	2109	1734	1562	1890	2250	1765	1734
File 26	3844	2062	1812	2063	2078	2031	1578	2328	2016	2281
File 27	1672	1531	2047		1875	1891	2016	1593	1687	2141
File 28	2250	1719			1515	1578	2031	1985	1422	2110
File 29	2032	1985			1875	1968	1609	2250	1860	2343
File 30	2031	1703			1875	1938	1859	2125	2062	2172
File 31					1875	1640	2156		2344	
File 32					2047	1781	2000			
File 33					2062	2032				
File 34						1890				

Tabelle A.8.: Reaktionszeit in ms für Datenpaket 2 ohne Postprocessing

Hits	Datensatz 1	Datensatz 2	Datensatz 3	Datensatz 4	Datensatz 5	Datensatz 6	Datensatz 7	Datensatz 8	Datensatz 9	Datensatz 10
File 1	1	1	0	0	1	0	0	0	0	0
File 2	0	1	1	1	1	1	0	0	0	0
File 3	0	1	1	1	1	1	0	0	0	1
File 4	1	1	1	1	1	1	1	1	1	0
File 5	0	1	0	1	1	1	0	0	1	0
File 6	0	0	1	1	0	0	0	0	0	0
File 7	1	1	1	0	1	1	1	0	1	1
File 8	1	1	1	1	0	1	0	1	0	1
File 9	1	1	1	1	1	1	1	1	1	1
File 10	1	1	0	1	1	1	1	1	1	1
File 11	0	0	0	1	0	0	1	0	1	1
File 12	1	1	0	1	1	1	1	1	1	1
File 13	1	1	1	0	1	0	1	1	1	1
File 14	1	0	0	1	0	0	1	1	1	0
File 15	1	1	1	1	1	1	1	1	0	1
File 16	1	1	1	0	0	1	0	1	1	1
File 17	0	0	1	1	1	1	1	1	1	1
File 18	1	1	1	1	1	1	0	1	1	1
File 19	1	0	1	1	1	1	1	1	1	1
File 20	1	1	0	1	1	1	1	1	1	1
File 21	0	1	0	1	0	0	1	0	1	1
File 22	1	1	1	1	1	1	1	1	1	0
File 23	1	1	1	1	1	0	1	1		
File 24	1	1					1	1		
File 25	1							1		
File 26	1							1		

Tabelle A.9.: Treffer für Datenpaket 1 mit Homofonen (1 bedeutet korrekt erkannt, 0 nicht erkannt)

Hits	Datensatz 1	Datensatz 2	Datensatz 3	Datensatz 4	Datensatz 5	Datensatz 6	Datensatz 7	Datensatz 8	Datensatz 9	Datensatz 10
File 1	1	1	1	1	1	1	1	1	1	1
File 2	1	1	1	1	1	1	1	1	1	1
File 3	1	0	0	1	1	1	1	0	1	1
File 4	1	1	0	0	1	1	1	1	1	0
File 5	0	0	0	0	1	1	1	0	1	1
File 6	0	1	0	1	0	0	0	0	0	0
File 7	1	1	1	1	1	1	1	1	1	1
File 8	1	0	1	1	1	1	1	1	1	1
File 9	1	1	1	1	1	1	1	1	1	1
File 10	1	1	1	0	1	1	1	1	1	1
File 11	0	1	1	0	1	0	1	1	1	1
File 12	0	1	0	1	0	0	0	0	0	0
File 13	0	1	1	1	0	0	0	1	0	0
File 14	1	1	1	1	1	1	0	0	0	1
File 15	0	1	1	1	1	1	1	1	1	0
File 16	1	1	1	1	1	1	1	1	1	1
File 17	1	1	1	1	1	0	1	0	0	1
File 18	1	1	0	0	1	1	1	1	1	0
File 19	1	0	1	1	1	0	1	1	1	1
File 20	1	0	1	0	1	1	1	0	0	1
File 21	1	1	1	0	1	1	1	1	0	0
File 22	1	1	1	1	1	1	0	1	1	1
File 23	1	1	1	1	1	1	1	1	0	0
File 24	1	1	1	1	0	1	1	0	1	1
File 25	1	1	1	1	1	1	1	0	0	0
File 26	1	1	1	1	1	1	1	1	1	1
File 27	1	1	1	1	1	1	1	1	1	1
File 28	1	1			1	1	1	1	1	1
File 29	1				1	1	1	1	1	1
File 30	1				1	1	0	1	1	1
File 31	1				1	1	1	0	0	0
File 32	1				1	1	1	1	1	1
File 33					1	1		1	1	1
File 34					1					

Tabelle A.10.: Treffer für Datenpaket 2 mit Homofonen (1 bedeutet korrekt erkannt, 0 nicht erkannt)

Time per File	Datensatz 1	Datensatz 2	Datensatz 3	Datensatz 4	Datensatz 5	Datensatz 6	Datensatz 7	Datensatz 8	Datensatz 9	Datensatz 10
File 1	1500	1700	2031	1859	1922	2172	1813	1781	2235	1688
File 2	1400	1500	7766	7578	7250	7328	1719	1547	1938	2078
File 3	6200	6500	2406	1859	1703	1687	1828	1766	1985	1563
File 4	1300	1900	2546	1735	2015	7750	2063	1844	2390	2563
File 5	1500	2000	5610	7406	1625	2046	2015	1516	1875	1922
File 6	1900	1500	1797	4390	344	1734	1531	1891	6032	3234
File 7	6000	6000	2000	2250	1641	1609	2437	1813	2032	1812
File 8	2000	1800	1860	2093	2610	1859	1609	1360	2047	2063
File 9	1600	2000	1828	1922	2375	1671	1859	1875	1687	1500
File 10	2000	1500	1828	1578	2140	2016	1844	1922	2109	1969
File 11	2400	2000	1985	2203	2312	2328	1843	2328	2281	2203
File 12	1700	2000	2062	1922	2188	1687	2031	2063	1969	2110
File 13	1800	2100	2015	1891	2156	1891	1828	1766	2672	2110
File 14	1300	1900	1938	1719	2203	4828	2843	2031	1984	2125
File 15	1800	1800	2235	2204	2469	2422	1797	1860	2078	1985
File 16	1500	1900	1953	1922	2093	1938	1875	3219	2421	2234
File 17	2000	2500	2047	2375	2094	1890	1750	1922	2343	1734
File 18	2000	1800	2047	2078	1938	1782	2031	2343	1969	2016
File 19	1500	1900	1578	1594	1843	1468	1968	2219	1734	2281
File 20	1900	1700	1844	2031	1891	1829	1625	4000	1937	1828
File 21	2000	2000	1859	2359	2172	1922	1797	2000	1750	1766
File 22	1600	1900	1781	2016	2046	2953		2187	1859	1656
File 23	1900	1900	1860		1860	1922		2016		2813
File 24	1800							1922		
File 25	1900									
File 26	2000									

Tabelle A.11.: Reaktionszeit in ms für Datenpaket 1 mit Homofonen

Time per File	Datensatz 1	Datensatz 2	Datensatz 3	Datensatz 4	Datensatz 5	Datensatz 6	Datensatz 7	Datensatz 8	Datensatz 9	Datensatz 10
File 1	2360	1812	2047	1797	1968	4375	1797	2000	1656	1781
File 2	1844	2062	2406	2015	2031	2031	2094	2204	1922	2032
File 3	1609	2031	1765	1844	1828	1938	1860	1922	1594	1843
File 4	2203	1828	2203	1703	1954	2094	1734	2015	1703	1891
File 5	1891	2407	1797	2047	1703	2062	1594	1796	1719	1531
File 6	1828	3500	2109	2282	2110	2063	2625	1813	1953	1797
File 7	2453	2156	2219	1938	2468	2109	1969	2234	2125	2047
File 8	2078	1562	2157	2062	1985	2188	2047	2016	1969	1796
File 9	2406	2125	1797	2078	1922	2140	2032	1922	1812	2015
File 10	1750	2219	2000	2015	2016	1610	1906	1985	1828	1735
File 11	2843	2140	4593	1954	1656	2141	1610	1766	1984	1734
File 12	2312	2016	1609	2078	2094	2140	2125	2000	1890	1906
File 13	2359	1907	1875	1953	2359	2063	1907	1766	2078	2187
File 14	2719	1844	2094	2203	1703	2109	1938	1734	1969	1610
File 15	2219	2156	1781	2047	1860	1672	1875	1656	2015	1922
File 16	2141	1750	2000	2360	1750	1797	1734	1953	1687	1969
File 17	1938	2031	1907	1593	1578	2156	1937	2204	1781	1765
File 18	2094	2328	1937	2031	1828	2796	2125	1844	1734	2078
File 19	2047	1609	3453	1734	2187	1953	1610	1703	1610	1969
File 20	2500	2500	2969	1985	1844	2031	2016	1750	1984	1609
File 21	2453	1609	2375	2235	1906	1672	3016	1953	1328	1890
File 22	1969	2016	1609		2093	2000	1922	2110	2015	2829
File 23	1859	2156	2078		1891	1891	1813	1672	1515	1640
File 24	3391	2141	1953		1938	1640	1500	1718	1844	1968
File 25	1734	2110	2032		2109	1797	1875	1860	1750	2188
File 26	2250	1953	2266		1938	2016	1796	2141	2187	1532
File 27	1797	2094			1860	1781		1718	1922	1984
File 28	1610				1703			3062	2047	1937
File 29	2000				1860			2031	1656	1859
File 30	2469				1734			2234	1890	2063
File 31	1813				1844				1750	2031
File 32					1719				1875	1797
File 33					2375					

Tabelle A.12.: Reaktionszeit in ms für Datenpaket 2 mit Homofonen

Hits	Datensatz 1	Datensatz 2	Datensatz 3	Datensatz 4	Datensatz 5	Datensatz 6	Datensatz 7	Datensatz 8	Datensatz 9	Datensatz 10
File 1	0	1	0	0	1	0	0	0	0	0
File 2	0	1	1	1	1	1	0	0	0	1
File 3	1	1	1	1	1	0	1	1	1	0
File 4	1	1	1	1	1	1	1	1	1	0
File 5	0	1	0	1	1	1	1	0	0	0
File 6	0	0	1	1	0	1	0	0	0	0
File 7	1	1	1	0	1	1	0	1	1	1
File 8	1	1	0	1	0	1	1	1	1	1
File 9	1	1	1	1	1	1	0	1	1	1
File 10	0	1	1	1	1	1	1	1	1	1
File 11	0	0	1	1	0	0	1	0	1	1
File 12	1	1	0	0	1	1	1	1	1	1
File 13	1	1	0	1	1	0	1	1	1	1
File 14	1	0	1	1	0	0	1	1	1	1
File 15	1	1	0	0	1	1	1	1	0	0
File 16	0	1	1	1	0	1	0	1	1	1
File 17	1	0	1	1	1	1	1	0	1	1
File 18	1	1	1	1	1	1	0	1	1	1
File 19	1	0	1	1	1	1	1	1	1	1
File 20	0	1	0	1	1	1	1	1	1	1
File 21	1	1	0	1	0	0	1	0	1	1
File 22	1	1	1	0	1	1	1	1	1	1
File 23	1	0	1	0	1	0	1	1	1	0
File 24	1	1		0			1	1		
File 25	1	0					1	1		
File 26								1		

Tabelle A.13.: Treffer für Datenpaket 1 mit Homofonen, Textdistanz und direkter Benennung (1 bedeutet korrekt erkannt, 0 nicht erkannt)

Hits	Datensatz 1	Datensatz 2	Datensatz 3	Datensatz 4	Datensatz 5	Datensatz 6	Datensatz 7	Datensatz 8	Datensatz 9	Datensatz 10
File 1	1	1	1	1	1	1	1	1	1	1
File 2	1	1	1	1	1	1	1	1	1	1
File 3	1	1	0	1	1	1	1	0	1	1
File 4	1	1	0	0	1	1	1	1	1	0
File 5	0	0	0	0	1	1	1	0	1	1
File 6	0	1	0	1	0	0	0	0	0	0
File 7	1	1	1	1	1	1	1	1	1	1
File 8	1	1	1	1	1	1	1	1	1	1
File 9	1	0	1	1	1	1	1	1	1	1
File 10	1	1	1	0	1	1	1	1	1	1
File 11	0	1	1	0	1	0	1	1	1	1
File 12	0	1	0	1	0	0	0	0	0	0
File 13	0	1	1	1	0	0	1	1	0	0
File 14	1	0	1	1	1	1	0	0	0	1
File 15	1	1	1	1	1	1	1	1	1	0
File 16	1	1	1	1	1	1	0	1	1	1
File 17	0	1	1	1	1	1	1	0	0	1
File 18	1	1	1	1	1	0	1	1	1	0
File 19	1	1	1	1	1	1	1	1	0	1
File 20	1	1	1	0	1	1	1	0	1	1
File 21	1	0	1	1	1	1	1	1	1	0
File 22	0	0	1	1	1	1	0	1	0	1
File 23	1	1	0	1	1	1	1	1	1	0
File 24	1	1	1	1	1	1	0	1	0	0
File 25	1	1	1	1	1	1	1	1	1	0
File 26	1	1	1	1	1	1	1	1	1	0
File 27	1	1	1	1	1	1	1	1	1	1
File 28	1	1	1		1	1	1	1	1	1
File 29	1	1	1		1	1	1	1	1	1
File 30	1	1			1	1	0	0	0	1
File 31	1	1			1	1	1	1	1	1
File 32	1				1	1	1	1	1	1
File 33					1	1				1
File 34						1				1

Tabelle A.14.: Treffer für Datenpaket 2 mit Homofonen, Textdistanz und direkter Benennung (1 bedeutet korrekt erkannt, 0 nicht erkannt)

Time per File	Datensatz 1	Datensatz 2	Datensatz 3	Datensatz 4	Datensatz 5	Datensatz 6	Datensatz 7	Datensatz 8	Datensatz 9	Datensatz 10
File 1	1906	1750	1875	1734	1969	1766	1656	1828	2563	1641
File 2	1578	1718	5110	4953	2235	4579	1922	1687	1640	1188
File 3	5500	4531	1578	1766	4797	4609	4766	4735	4922	5922
File 4	1969	1531	1984	1938	1656	1891	1797	1765	2375	1641
File 5	2188	1688	1546	4875	2453	1781	1765	1734	1484	2078
File 6	1469	2953	1719	1687	1891	924	1563	3766	2079	1250
File 7	5953	4547	784	1922	774	1782	1766	4703	847	1875
File 8	1843	1750	1671	735	1968	1672	1750	1985		1813
File 9	1937	1875	1875	2063	1531	1875	886	1813	1297	750
File 10	1937	1485	3079	1547	2031	1812	2250	819	1953	1953
File 11	1891	1813	1906	1797	1954	1750	1484	1719	2218	2469
File 12	1719	1938	2297	1843	1719	1875	2031	1922	1829	1797
File 13	1797	1797	2562	1875	1969	1907	2046	1922	1875	1907
File 14	2063	1719	1781	1922	1969	1781	1985	1797	1922	1828
File 15	1906	1907	1890	2046	1922	1921	1985	2000	2156	1734
File 16	1734	2109	1765	1843	1797	1719	2094	1890	1812	1656
File 17	1906	1562	1813	1890	1891	1750	2110	1782	1781	1922
File 18	1656	1375	1782	1828	1515	1812	1875	1828	2484	1796
File 19	1938	1547	1578	1500	1703	1813	1640	1609	1531	1594
File 20	1781	1704	2187	1766	1687	1906	3172	1750	5016	1766
File 21	1688	1609		1750	2812		2078	1890	1631	1766
File 22	1788	2079			1844			1875	1532	
File 23		1875						2063		
File 24		1578								

Tabelle A.15.: Reaktionszeit in ms für Datenpaket 1 mit Homofonen, Textdistanz und direkter Benennung

Time per File	Datensatz 1	Datensatz 2	Datensatz 3	Datensatz 4	Datensatz 5	Datensatz 6	Datensatz 7	Datensatz 8	Datensatz 9	Datensatz 10
File 1	1844	1734	1953	1781	2016	1734	2016	2032	1984	1781
File 2	1860	1875	1985	1765	1781	1734	1719	2063	2000	2047
File 3	1922	2875	752	910	744	720	868	796	737	750
File 4	1953	1813	1781	1172	1813	1907	1844	1921	2906	3672
File 5	825	1797	1172	13469		5688				
File 6	1922	1735	2094		1781	1563	1719	1953	1734	1844
File 7	2141	750	1922	2406	2141	1688	2031	2437	1828	2078
File 8	1516	2172	1813	2000	1750	2031	2094	2078	2265	2188
File 9	2015	2187	1781	2125	1843	2187	2000	1922	1969	2000
File 10	1922	1968	1719	1484	2109	1922	1734	1625	2313	2016
File 11	1531	1594	1781	1812	1781	2110	1610	1797	1531	1610
File 12	1922	2344	1500	1610	1860	1844	1921	1984	3047	1938
File 13	1828	1625	1562	1828	1797	1422	1890	1735	2094	1812
File 14	1875	1828	2203	1437	1563	1907	1579	2109	1828	1672
File 15	2093	1469	1484	1640	1719	1734	1984	1515	2469	1781
File 16	1532	1984	1766	1922	1969	1453	2109	1906	1625	1563
File 17	1890	4344	2093	1609	1484	1687	1735	2000	2047	1813
File 18	1500	1750	1688	1578	1828	1843	1516	1609	1609	1859
File 19	1672	1813	1813	1781	1766	1656	1547	1766	1906	1437
File 20	1797	1985	1672	1735	1469	1953	1875	1640	1671	2046
File 21	1484		1875	1844	1610	1516	1672	1828	2016	1719
File 22	2016		1828		2062	1812	2343	1516	2047	2172
File 23	1844				1656	1453	1484	2016	1531	2016
File 24	1547				1531	2734	1906	1922	1843	1875
File 25	1813				1656	2094	1781	1453	1844	1657
File 26	1828				1906	1469	1485	1500	2203	1875
File 27	1750				1469	2094	2094	1938		1875
File 28					1750	1953	2563	2172		1781
File 29					1844	1485				2390
File 30					1500	1953				1844
File 31					2078	1860				
File 32					1609					
File 33					1750					

Tabelle A.16.: Reaktionszeit in ms für Datenpaket 2 mit Homofonen, Textdistanz und direkter Benennung