Automatic speech recongnition toolkit for Audio and Video - astAV (Feinentwurf)

Patrick Löw

6. August 2022

Projekt: Automatic speech recongnition toolkit for Audio and Video - astAV

Auftraggeber: Technology Arts Sciences TH Köln

Version	**Datum**	**Autor**
1.8	6. August 2022	Patrick Löw

Inhaltsverzeichnis

1 Daten des Systems

Das System braucht selber keine persistenten Daten, auf die es zugreift. Für die Spracherkennung der Video- oder Audiodatei muss lediglich auf diese zugegriffen werden. Für die Spracherkennungsbibliotheken und `Formatoren` können je nach Programmierung weitere Zugriffe nötig seien. Die `DeepSpeech` und `Vosk` Spracherkennungsbibliotheken brauchen zum Beispiel Zugriff auf ihre Sprachmodelle die in einem Ordner oder einer Datei gespeichert sind.

2 Aktivitätsdiagramm Benutzeroberfläche

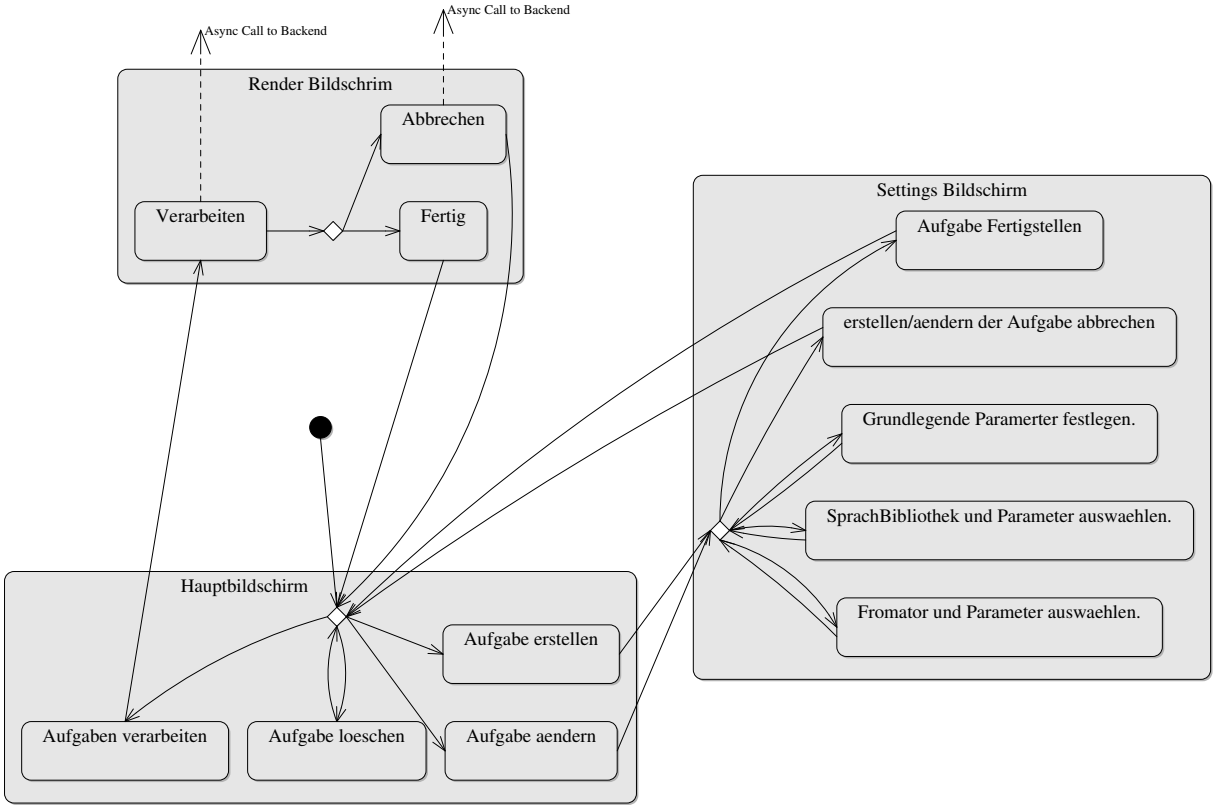


Abbildung 1: Alle Möglichkeiten der Benutzerführung

Die Benutzeroberfläche besteht aus drei Bildschirmen die für verschiedene Aufgaben genutzt werden.

Hauptbildschirm

Im Hauptbildschirm sind erstellte Aufgaben, also Video- oder Audiodateien, aus denen der gesprochene Text extrahiert wird, aufgelistet. Diese können hier auch einzeln gelöscht werden. Von dem Hauptbildschirm kann in den Render-Bildschirm oder den Settings-Bildschirm navigiert werden. Um vom Settings in den Render-Bildschirm zu gelangen, muss man aber erst über den Hauptbildschirm navigieren.

Settings-Bildschirm

Im Settings-Bildschirm werden die Parameter für eine neue Aufgabe vom Benutzer ausgewählt. Er kann dort sowohl grundlegende Parameter auswählen, wie z.B. die Video- oder Audiodatei, welche Spracherkennung und welcher `Formator` genutzt wird, als auch Parameter, die die Spracherkennung selber betreffen.

Render-Bildschirm

Im Render-Bildschirm wird der Fortschritt der einzelnen Aufgaben bei der Verarbeitung angezeigt. Dieser Vorgang kann aber auch abgebrochen werden. Wenn alle Aufgaben

durchgelaufen sind, wird ein kleines Fenster angezeigt, in dem alle Aufgaben und deren Erfolg oder Misserfolg aufgelistet sind. Wenn der Benutzer aus diesem Bildschirm in den Hauptbildschirm zurückkehrt, werden alle Aufgaben, die Erfolgreich beendet wurden, gelöscht.

3 Aktivitätsdiagramm Backend

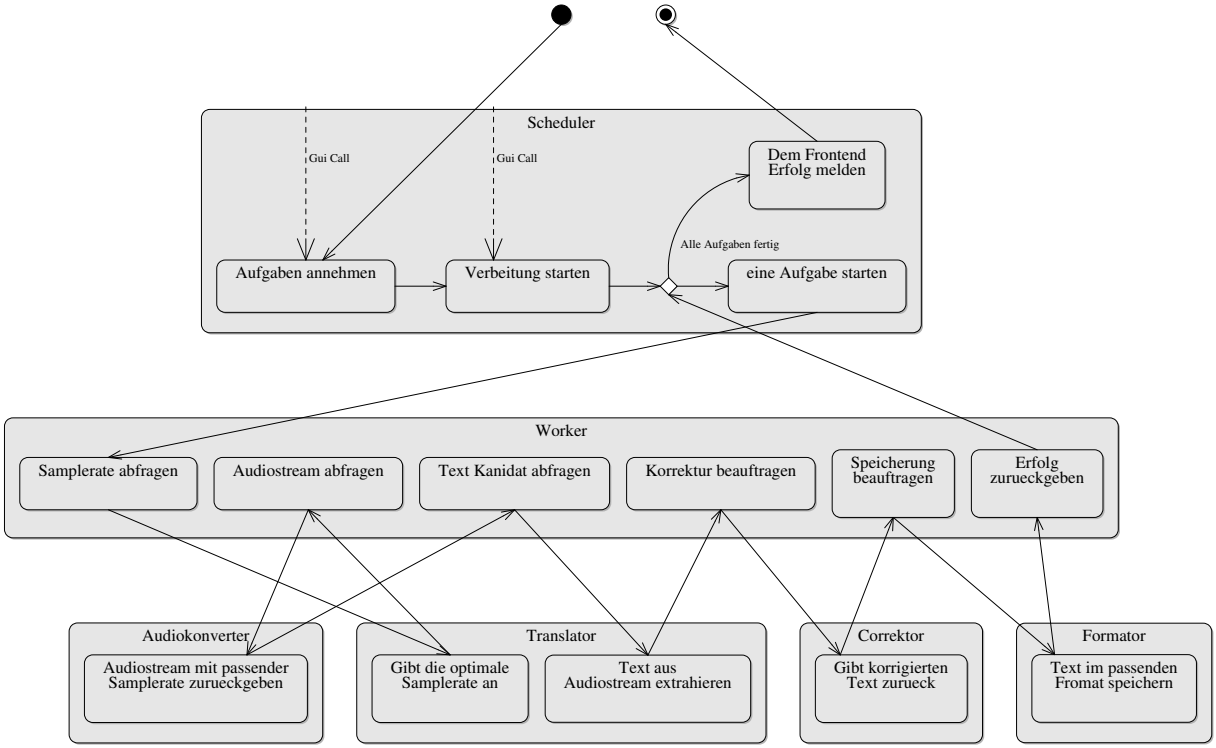
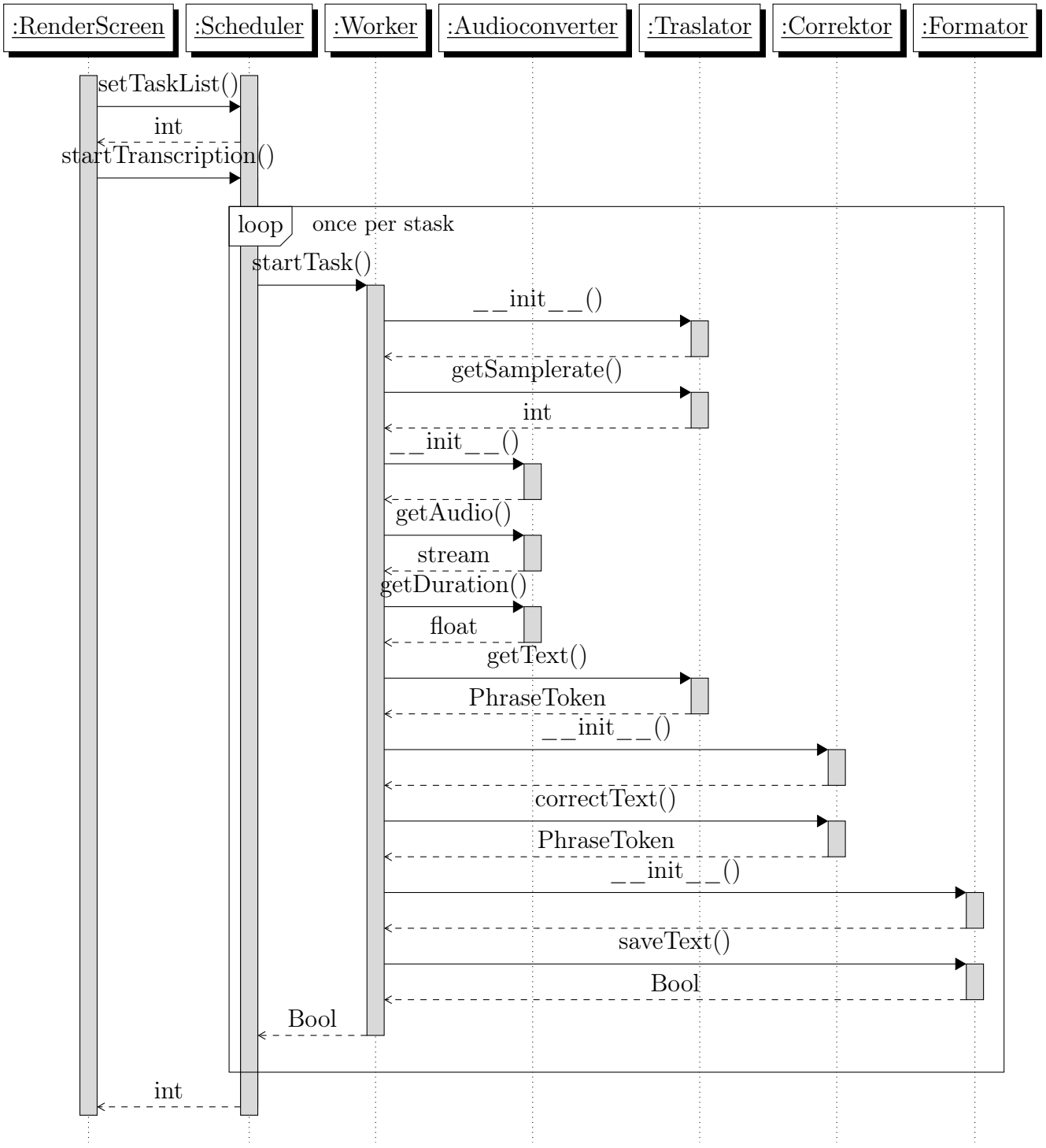


Abbildung 2: Aktivitätsdiagramm Backend

Das Backend nimmt die Aufgaben der Benutzeroberfläche an und verarbeitet sie nach einander. Der `Scheduler` ist dabei für die Reihenfolge der Aufgaben zuständig und übergibt diesen dem `Worker`. Der `Worker` hat die Kontrolle über die Arbeitsschritte einer Aufgabe und erledigt diese nacheinander.

4 Sequenzdiagramm vom Backend



Das hier gezeigte Sequenzdiagramm ist eine grobe Übersicht über die Abfolge der Aufgaben im Backend.

5 Bootloader

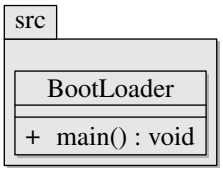


Abbildung 3: Bootloader

BootLoader

Die `Bootloader`-Klasse bietet den Einsprungpunkt in das Programm. Sie wird aktuell nur die Benutzeroberfläche öffnen, kann aber auch in der Zukunft erweitert werden.

Verhalten

main() Die Einsprungsfunktion in das Programm ruft die Funktion `__init__()` der `Gui`-Klasse auf.

6 Grenzklassen

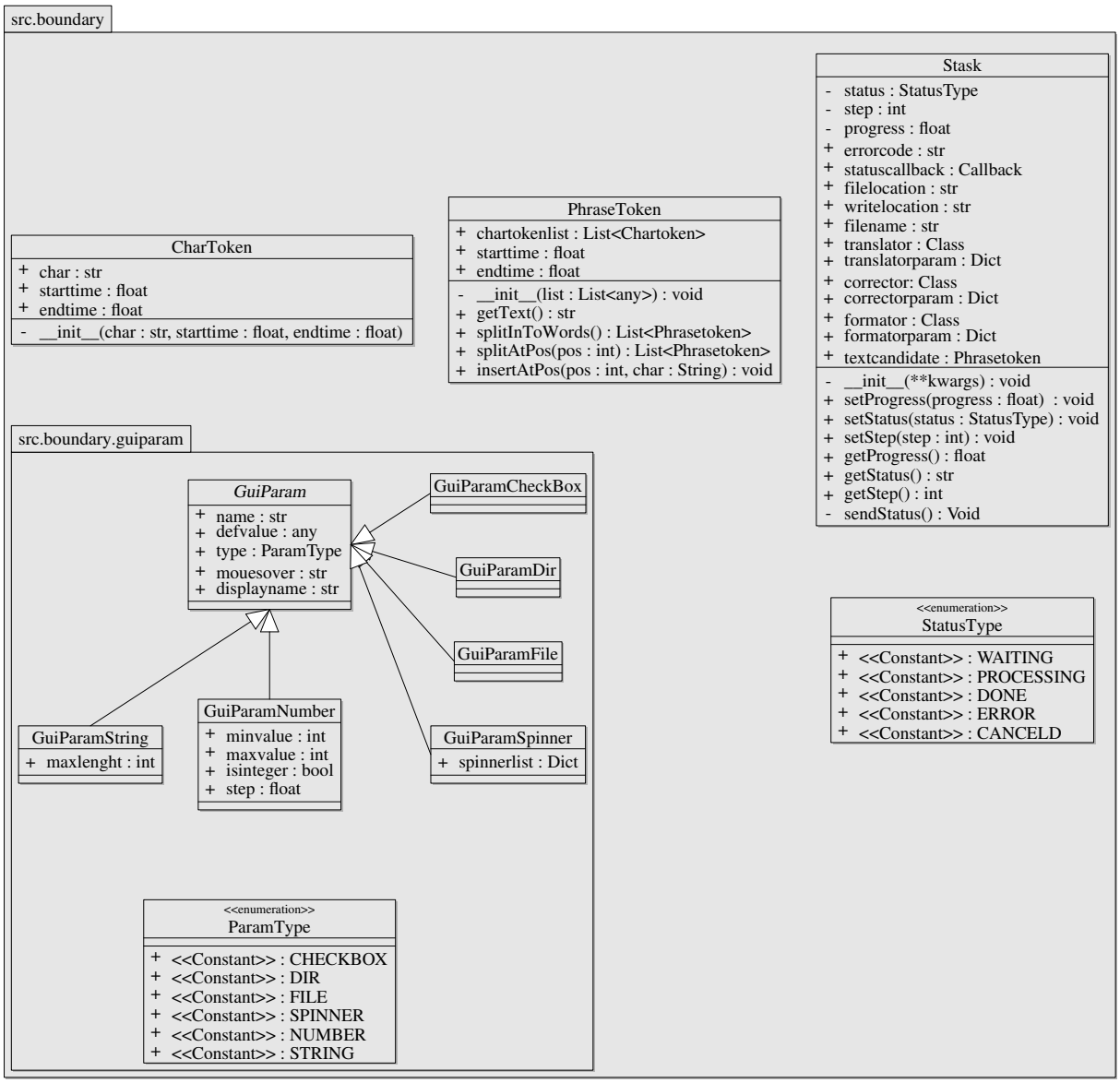


Abbildung 4: Alle Grenzklassen des Systems

6.1 Stask

`Stask` ist eine Klasse die eine einzelne Aufgabe abbildet. Sie wird von der Benutzeroberfläche erzeugt und an alle Klassen gereicht, die Information zur Aufgabe brauchen. Im weiteren Verlauf wird, wenn von Aufgaben die Rede ist, sich immer auf die `Stask`-Klasse bezogen.

Verhalten

`__init__`(kwargs)** Bei der Erzeugung einer Instanz der `Stask`-Klasse können ihr Parameter übergeben werden. Wenn die Namen den öffentlichen Variablen entsprechen werden diese Parameter übernommen. Wenn die Argumente nicht vorhanden sind, werden

sie durch leere Listen bzw. Lexika ersetzt. Die privaten Variablen bekommen folgende Werte:
`status` = WAITING,
`step` = 1,
`progress` = 0.0,

setProgress(progress : float) Der Funktion wird ein float Wert zwischen 0 und 1 übergeben. Werte über diese Grenzen werden auf den jeweiligen Grenzwert gesetzt und in die private `progress` Variable gesetzt. Zum Schluss wird `sendStatus()` aufgerufen.

setStatus(status : StatusType) Der `status` wird in die Instanz übernommen. `sendStatus()` wird aufgerufen.

setStep(step : int) Der `step` Parameter wird in der Instanz übernommen. Falls der Parameter < 1 ist, wird 1 übernommen. Die Variable `progress` wird auf 0.0 gesetzt. `sendStatus()` wird aufgerufen.

getProgress() Gibt die Variable `progress` zurück.

getStatus() Gibt die Variable `status` zurück.

getStep() Gibt die Variable `step` zurück.

sendStatus() Ruft die in `statuscallback` hinterlegte Funktion auf. Diese wird von der Benutzeroberfläche hinterlegt. Siehe Kapitel 7.

6.2 CharToken

Die Klasse enthält den Start- und Endzeitpunkt eines erkannten Zeichens.

Verhalten

__init__(char : str, starttime : float, endtime : float) Bei der Erzeugung Instanz müssen das Zeichen, der Startzeitpunkt und der Endzeitpunkt übergeben werden. Falls mehrere Zeichen übergeben werden, wird nur das erste übernommen. Zeichen für einen Zeilenumbruch sind erlaubt.

6.3 PhraseToken

Ist eine Grenzklasse die zwischen der `Translator`-Klasse (Abbildung 8) und der `Formator`-Klasse (Abbildung 9) ausgetauscht wird. Diese beinhalten den erkannten Text als auch die passenden Zeitstempel im `CharToken`-Format.

Verhalten

__init__(list : List<any>) Mit dieser Funktion können Listen von zwei verschiedenen Klassen übergeben werden. Die Klassen sind `phraselist` und `charlist`. Wenn eine Liste mit den Klassen `charlist` übergeben wird, werden nur diese verarbeitet, sonst nur die anderen.

Verarbeitung für charlist Eine neue Instanz kann aus mehreren Instanzen der Klasse `CharToken` erzeugt werden. Die `CharToken` werden sortiert nach Startzeitpunkt und Endzeitpunkt übernommen. Start- und Endzeitpunkt der PhraseToken-Instanz wird dem minimalen Wert des Startzeitpunkts und dem maximalen Wert des Endzeitpunkts aller `CharToken` gleichgesetzt.

Verarbeitung für phraselist Eine neue Instanz kann aus mehreren Instanzen der gleichen Klasse erzeugt werden. Die `CharToken` der alten Instanzen werden sortiert nach Startzeitpunkt und Endzeitpunkt übernommen. Start- und Endzeitpunkt der `PhraseToken` Instanz wird dem minimalen Wert des Startzeitpunkts und dem maximalen Wert des Endzeitpunkts aller `CharToken` gleichgesetzt. Zwischen den `CharToken` der Instanzen wird jeweils ein `CharToken` mit Leerzeichen eingefügt.

getText() : str Die Funktion gibt den einen String mit allen Zeichen der `CharToken` in der Liste zurück.

splitInToWords() Die `CharToken` werden an den Leerzeichen aufgeteilt und als eine Liste von `PhraseToken` zurückgegeben. Leerzeichen werden nicht übernommen.

splitAtPos(pos : int) Die `CharToken` werden an der Position pos in zwei `PhraseToken` aufgeteilt und als Liste zurückgegeben. Vorangestellte oder nachfolgende Leerzeichen werden nicht übernommen.

insertAtPos(pos : int, char : str) Es wird ein `CharToken` aus dem übergebenden String erzeugt. Start- und Endzeitpunkt werden aus den `CharToken` davor und dahinter entnommen.

6.4 GuiParam

`GuiParam` wird von der `Formator`- und der `Translator`-Klasse verwendet, um die benötigten Parameter der Benutzeroberfläche mitzuteilen. Dabei wird Art, Parametergrenzen und der Name des Parameters übergeben. Die Klasse dient als Basisklasse für weitere Klassen, die ebenfalls keine Funktionen implementieren. Der `type` wird bei der Initialisierung der Klasse festgelegt.

7 Benutzeroberfläche

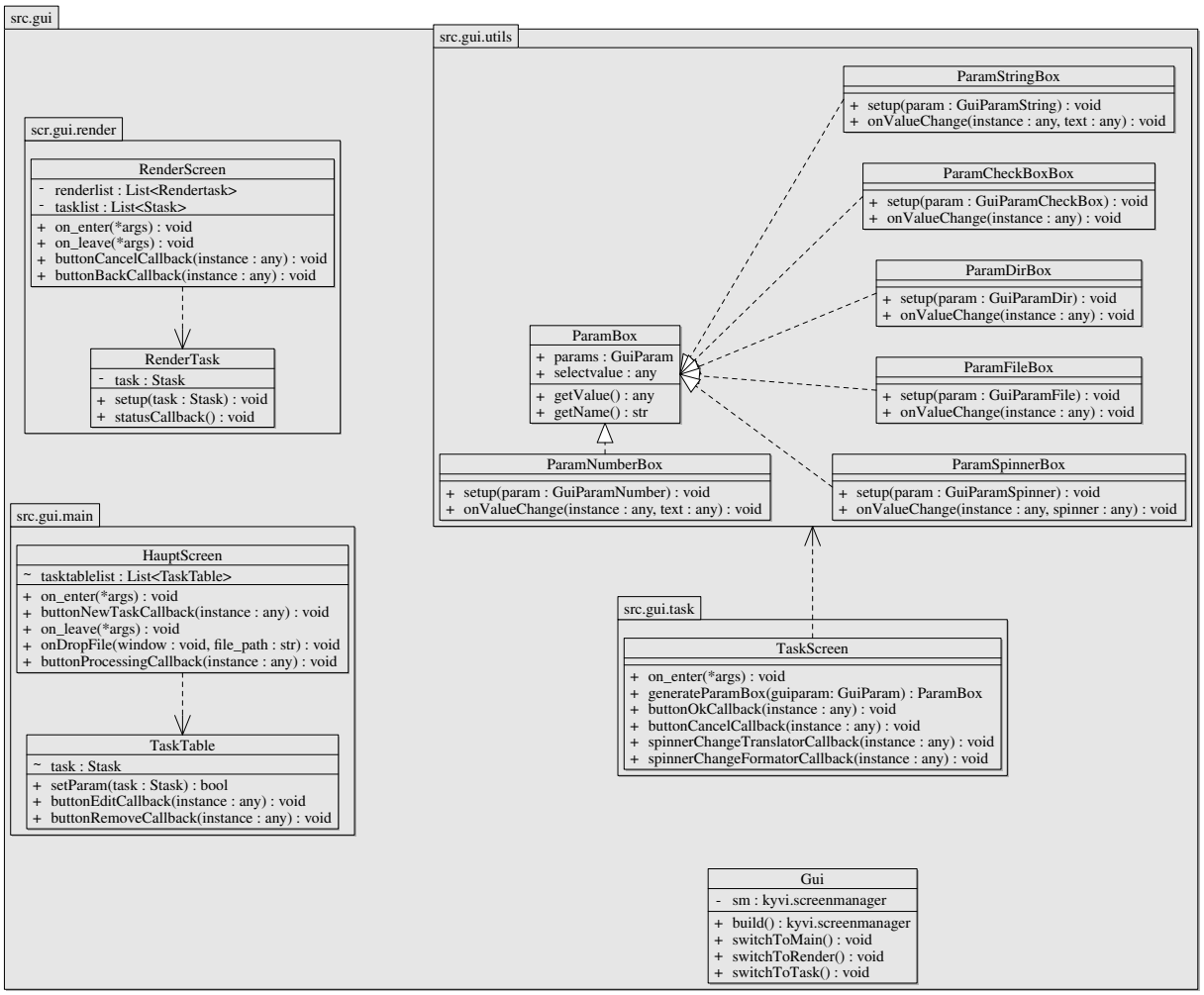


Abbildung 5: Übersicht der Benutzeroberfläche

Die Benutzeroberfläche besteht aus 3 verschiedenen Ansichten. Jede wird in einer separaten Klasse implementiert. Die Benutzeroberfläche wird mit dem Kivy Framework implementiert. Weiter Informationen sind Kapitel 2 zu entnehmen.

7.1 Gui

Die Klasse `Gui` implementiert die Schnittstelle `IGuiActivate`.

Verhalten

build() Erzeugt einen Kivy Screenmanager des `HauptScreens`.

switchToMain() Ruft den `HauptScreen` auf.

switchToRender() Ruft den `RenderScreen` auf.

switchToTask() Ruft den `TaskScreen` auf.

7.2 RenderScreen

Diese Ansicht zeigt eine Auflistung mit allen Aufgaben und den Status der Verarbeitung.

Verhalten

on_enter(*args) Diese Funktion ist eine Einsprungsfunktion. Sobald die Ansicht aufgerufen wird, fragt die Instanz die Liste aller Aufgaben der Klasse `Scheduler` ab (siehe Abbildung 6). Mithilfe der Klasse `RenderTask` wird eine Liste aller Aufgaben mit hilfreichen Parametern angezeigt. Alle buttonCallbacks werden an die entsprechenden Schaltflächen gebunden. Anschließend wird dem `Scheduler` der Befehl zum Starten der Verarbeitung gegeben. Die Verarbeitung verläuft nebenläufig.

on_leave(*args) Sobalt diese Ansicht verlassen wird, wird die `clearCallback`-Methode des Schedulers (siehe Kapitel 6) aufgerufen.

buttonCancelCallback(instance : any) Der `StatusType` aller Aufgaben in der `tasklist` wird von WAITING oder PROCESSING nach CANCELD gesetzt. Aufgaben, die andere `StatusTyps` haben sind nicht betroffen.

buttonBackCallback(instance : any) Wenn keine Aufgabe mehr in der `tasklist` den `StatusType` WAITING oder PROCESSING hat, kann diese Ansicht verlassen werden und der Hauptbildschirm aufgerufen werden. Vorher wird die Funktion `removeTaskDone()` des `Schedulers` aufgerufen (Siehe Abbildung 6).

7.3 RenderTask

Zeigt hilfreiche Informationen über den Verarbeitungsstatus einer Aufgabe an. Information sind: Name der Aufgabe, Status, Arbeitsschritt und ein Fortschrittsbalken.

Verhalten

setup(task : Stask) Diese Funktion erstellt einen Listeneintrag für die übergebende Aufgabe und übergibt der Aufgabe die Funktion `statusCallback()`.

statusCallback() Diese Funktion übernimmt die Änderungen der oben genannten Informationen in den Listeneintrag. Der Fortschrittsbalken ändert die Farbe in Abhängigkeit vom `status` der Aufgabe WAITING und PROCESSING blau, DONE grün, ERROR rot und CANCELD grau.

7.4 HauptScreen

Hier wird eine Übersicht über alle Aufgaben angezeigt, die zu verarbeiten sind.

Verhalten

on_enter(*args) Diese Funktion ist die Einsprungsfunktion in die HauptScreen-Ansicht. Sobald die Ansicht aufgerufen wird, fragt die Instanz die Liste aller Aufgaben der Klasse `Scheduler` ab (siehe Abbildung 6). Diese werden mithilfe der Klasse `TaskTable` in einer

List dargestellt. Falls keine Aufgaben existieren, wird in Englisch oder Deutsch der Satz, „Drop File here“ zu lesen sein. Die `onDropFile()` Funktion wird an das Fenster gebunden und alle buttonCallbacks an ihre entsprechenden Schaltflächen.

buttonNewTaskCallback(instance : any) Es wird eine neue Aufgabe mit Standardwerten erzeugt und der Scheduler Klasse als `temptask` übergeben. Anschließend wird die `TaskScreen` Ansicht aufgerufen.

on_leave(*args) Sobald das Fenster verlassen wird, wird die `onDropFile()`-Funktion von dem Fenster gelöst.

onDropFile(window : void, file_path : str) Diese Funktion wird aufgerufen, wenn der Benutzer eine Datei auf das Programmfenster zieht. Es wird eine Aufgabe erzeugt mit allen Parametern, die man aus dem übergebenden Dateipfad entnehmen kann. Diese Instanz wird der `Scheduler`-Klasse als `temptask` übergeben. Anschließend wird die `TaskScreen`-Ansicht aufgerufen.

buttonProcessingCallback(instance : any) Die Funktion ruft die `clearStatus()`-Funktion des `Schedulers` auf. Anschließend wird die `RenderScreen`-Ansicht aufgerufen.

7.5 Tasktable

Stellt wesentliche Informationen einer unbearbeiteten Aufgabe in einem Listeneintrag da. Zu den Informationen gehören: Dateiname, Translator- und Formatorname. In dem Listeneintrag gibt es noch einen Button zum Editieren und Löschen.

Verhalten

setParam(task : Stask) Der Instanz wird eine Aufgabe als Parameter übergeben, um die Informationen im Listeneintrag anzuzeigen. Dabei werden auch die buttonCallbacks mit den passenden Schaltflächen verknüpft.

buttonEditCallback(instance : any) Es wird die in der Instanz gespeicherte Aufgabe als `temptask` dem `Scheduler` übergeben. Anschließend wird die `TaskScreen` Ansicht aufgerufen.

buttonRemoveCallback(instance : any) Die in der Instanz gespeicherte Aufgabe wird dem `Scheduler`, in der Funktion `removeTask()` übergeben. Anschließend wird eine neue Instanz des `HauptScreens` erzeugt und aufgerufen.

7.6 TaskScreen

In dieser Ansicht sollen alle Parameter für eine neue Aufgabe ausgewählt werden oder Parameter einer bereits existierenden Aufgabe verändert werden können. Dazu gehören neben den Standardwerten, auch die Parameter der `Formatoren` und der `Translatoren`.

Verhalten

on_enter(*args) Die Einsprungsfunktion erzeugt mithilfe der `ParamBox` Implementation, eine Reihe von Eingabefeldern für verschiedene Parameter. Dazu gehören: Dateipfad, Speicherort, Name der zu speichernden Datei, der `Translator` mit passenden Parametern und der `Formator` mit passenden Parametern. Die Parameter für den `Translator` werden der `ITraslatorGuiParam`-Schnittstelle entnommen (Siehe Abbildung 8). Die Parameter für den `Formator` werden der `IFormatorGuiParam`-Schnittstelle entnommen (Siehe Abbildung 9). Die `button`- und `spinnerCallback`-Funktionen werden an die entsprechenden Schaltflächen gebunden.

generateParamBox(guiparam : GuiParam) Diese Funktion erzeugt, in Abhängigkeit zur übergebenen `GuiParam`-Instanz, eine passende Unterklasse des Typs `ParamBox` und gibt diese zurück.

buttonOKCallback(instance : any) In dieser Funktion wird mithilfe der Parameter aus den Eingabefeldern eine neue Aufgabe erzeugt. Diese wird der Funktion `insertTask()` im `Scheduler` übergeben. Anschließend wird die `HauptScreen`-Ansicht aufgerufen.

buttonCancelCallback(instance : any) Es wird die `HauptScreen`-Ansicht aufgerufen, nichts wird übernommen.

spinnerChangeTranslatorCallback(instance : any) Sobald die eine andere `Translator` Implementation ausgewählt wurde, werden die zum `Translator` gehörigen Eingabeflächen gelöscht. Über die Schnittstelle `ITraslatorGuiParam` (Siehe Abbildung 8) werden die benötigten Parameter für den aktuell ausgewählten `Translator` abgefragt und als Eingabefelder neu gezeichnet.

spinnerChangeFormatorCallback(instance : any) Sobald die eine andere `Formator` Implementation ausgewählt wurde, werden die zum `Formator` gehörigen Eingabeflächen gelöscht. Über die Schnittstelle `IFormatorGuiParam` (Siehe Abbildung 9) werden die benötigten Parameter für den aktuell ausgewählten `Formator` abgefragt und als Eingabefelder neu gezeichnet.

7.7 ParamBox

`ParamBox` dient als Basisklasse für alle Eingabefelder.

Verhalten

getValue() Die Funktion gibt den aktuell ausgewählten Wert `selectvalue` zurück.

getName() Die Funktion gibt den Namen des Eingabefeldes aus der lokalen `GuiParam`-Klasse zurück. Dieser Name dient als Key für das Erzeugen bestimmter Parameter in Aufgaben.

7.8 ParamStringBox

Die Klasse `ParamStringBox` ist eine abgeleitete Klasse von ParamBox.

Verhalten

setup(param : GuiParamString) Die Instanz erzeugt ein einzelnes Eingabefeld. In dies kann eine beliebige Zeichenfolge eingetragen werden. Die `onValueChange()` Funktion wird an das `text` Event gebunden.

onValueChange(instance : any, text : any) Die aktuelle Zeichenfolge wird in `selectvalue` der Basisklasse gespeichert. Die Zeichenlänge kann durch den `maxleght` Parameter, der übergebenden `GuiParamString`-Klasse, begrenzt werden.

7.9 ParamCheckBoxBox

Die Klasse `ParamCheckBoxBox` ist eine abgeleitete Klasse von ParamBox.

Verhalten

setup(param : GuiParamCheckBox) Die Funktion erzeugt eine CheckBox-Instanz mit dem Namen, die die Instanz von der `getName()` Funktion der Basisklasse bekommt. Die `onValueChange()` Funktion wird an das `active`-Event gebunden.

onValueChange(instance : any) Der aktuelle Zustand der `CheckBox` wird als bool in `selectvalue` der Basisklasse gespeichert.

7.10 ParamDirBox

Die Klasse `ParamDirBox` ist eine abgeleitete Klasse von ParamBox.

Verhalten

setup(param : GuiParamDir) Die Funktion erzeugt ein Label mit einem Button. Die `onValueChange()` Funktion wird an das `on_press` Event des Buttons gebunden.

onValueChange(instance : any) Die Funktioon öffnet einen Filedialog, in dem ein Ordner ausgewählt werden kann. Der Ordnerpfad wird im Label angezeigt und im `selectvalue` der Basisklasse gespeichert. Wenn bereits ein Ordnerpfad im `selectvalue` gespeichert wurde, wird der Filedialog an der entsprechenden Stelle geöffnet.

7.11 ParamFileBox

Die Klasse `ParamFileBox` ist eine abgeleitete Klasse von ParamBox.

Verhalten

setup(param : GuiParamFile) Die Funktion erzeugt ein Label mit einem Button. Die `onValueChange()` Funktion wird an das `on_press` Event des Buttons gebunden.

onValueChange(instance : any) Die Funktion öffnet einen Filedialog, in dem eine Datei ausgewählt werden kann. Der Dateipfad wird im Label angezeigt und im `selectvalue` der Basisklasse gespeichert. Wenn bereits ein Dateipfad im `selectvalue` gespeichert wurde, wird der Filedialog an der entsprechenden Stelle geöffnet.

7.12 ParamSpinnerBox

Die Klasse `ParamSpinnerBox` ist eine abgeleitete Klasse von ParamBox.

Verhalten

setup(param : GuiParamSpinner) Die Funktion erzeugt ein Spinner Menü. Die Optionen die ausgewählt werden können, werden aus der übergebenen `GuiParamSpinner`-Klasse übernommen. Der aus der `param.defvalue` entnommene `value` der aktuell ausgewählte Option, wird im `selectvalue` Parameter der Basisklasse gespeichert. Die `onValueChange()` Funktion wird an das `text`-Event des Spinners gebunden.

onValueChange(instance : any, spinner : any) Der `value` der aktuell ausgewählten Option wird in der `selectvalue` Variable der Basisklasse gespeichert und angezeigt.

7.13 ParamNumberBox

Die Klasse `ParamNumberBox` ist eine abgeleitete Klasse von ParamBox.

Verhalten

setup(param : GuiParamNumber) Die Funktion erzeugt einen Slider und ein Eingabefeld. Der Default und die Grenzwerte werden aus der mit übergebenen `GuiParamNumber` Klasse entnommen. Je nach Parameter der `GuiParamNumber` Klasse wird der input_filter auf float oder int gestellt. Der Slider wird auf eine Schrittweite eingestellt, die der `step`-Variable der lokalen `GuiParamNumber` Instanz entspricht. Die `onValueChange()` Funktion wird an das `text` Event des Eingabefeldes gebunden.

onValueChange(instance : any, text : any) Die Aktuell ausgewählt Nummer wird auf ihre Korrektheit geprüft und entweder durch einen der Grenzwerte oder, bei keiner erkennbaren Zahl, durch den Standardwert ersetzt. Dabei ist zu beachten, ob der Wert als float oder int abgespeichert werden soll.

8 Backend

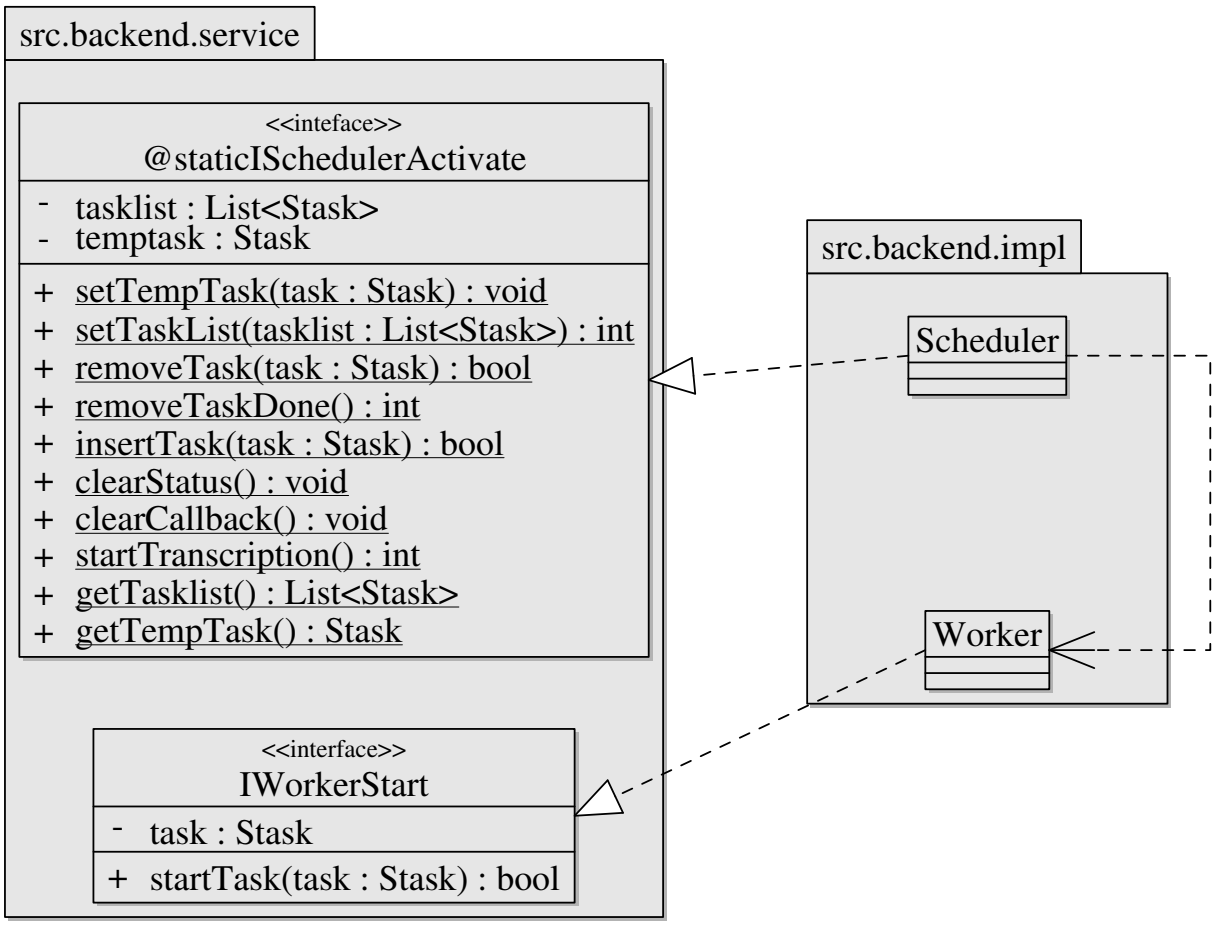


Abbildung 6: Übersicht der Steuerungsklassen im Backend

Die Steuerungsklassen übernehmen sowohl die Reihenfolge der Aufgaben, als auch die Reihenfolge der Verarbeitungsschritte. Weiter Informationen sind Kapitel 3 zu entnehmen.

8.1 ISchedulerActivate

Die Schnittstelle wird durch die `Scheduler`-Klasse implementiert. Diese Klasse ist statisch.

Verhalten

setTempTask(task : Stask) Die Funktion wird die übergebene Aufgabe in der Variable `temptask` speichern.

setTaskList(stasklist : List<Stask>) Die Funktion speichert die übergebene Liste in der Variable `stasklist` und gibt die Anzahl der Einträge zurück.

removeTask(task : Stask) Sucht die übergebene Aufgabe in `stasklist` und löscht diese aus der Liste. Wenn eine passende Aufgabe gefunden wurde, wird True zurückgegeben, sonst der Wert False.

removeTaskDone() Die Funktion sucht in der Liste `stasklist` alle Aufgaben, die den `StatusType` DONE enthalten. Diese werden aus der Liste gelöscht und die Anzahl der gelöschten Elemente zurückgegeben.

insertTask(task : Stask) Die Aufgabe in der Variable `temptask` wird in der `stasklist`-Liste falls vorhanden gelöscht. Anschließend wird die übergebene Aufgabe in die Liste eingefügt.

clearStatus() Der `StatusType` aller Aufgaben in `stasklist` wird auf WAITING gesetzt.

clearCallback() Der `statuscallback` aller Aufgaben wird durch eine leere Liste ersetzt.

startTranscription() Die Funktion geht in einer Schleife alle Aufgaben der Liste `stasklist` durch. Für jede Aufgabe wird eine Instanz der Klasse `Worker` erstellt und die Funktion `startTask()` aufgerufen. Die Anzahl der True Werte aller Funktionen werden zurückgegeben.

getTasklist() Die Funktion gibt die Liste `stasklist` zurück.

getTempTask() Die Funktion gibt die Variable `temptask` zurück.

8.2 IWorkerStart

Die Schnittstelle wird durch die `Worker`-Klasse implementiert. Sie kontrolliert die Verarbeitung einer Aufgabe.

Verhalten

startTask(task : Stask) Die Funktion übernimmt die übergebene Aufgabe. Es werden nun mehrere Schritte durchgeführt.
1. Die Funktion initialisiert die in der Aufgabe gespeicherte `Translator`-Klasse.
2. Die Funktion `getSamplerate()` der `Translator`-Instanz wird aufgerufen (Siehe Abbildung 8).
3. Eine `Audiokonverter`-Klasse wird mit der Samplerate und der Aufgabe initialisiert.
4. Die Funktionen `getAudio()` und `getDuration()` des Audiokonverters werden aufgerufen (Siehe Abbildung 7).
5. Mit dem `Audiostream` und der `Duration`, wird die Funktion `getText()` der `Translator`-Instanz aufgerufen. Der Rückgabewert wird in die `textcandidate`-Variable der `task`-Instanz gespeichert.
6. Die in der Aufgabe gespeicherte `Corrector`-Klasse wird initialisiert.
7. Die Funktion `correctText` der `Corrector`-Instanz wird mit dem Rückgabewert aus Punkt 5 aufgerufen. Der Rückgabewert wird in die `textcandidate`-Variable der `task`-Instanz gespeichert.
8. Die in der Aufgabe gespeicherte `Formator`-Klasse wird initialisiert.
9. Die Funktion `saveText()` der `Formator`-Instanz (siehe Abbildung 9) wird mit dem Rückgabewert aus Punkt 7. aufgerufen.

Wenn alle Punkte ohne Fehler durchgeführt wurden, wird der Wert True zurückgegeben und der `status` der Aufgabe auf DONE gesetzt. Die Funktion fängt mögliche Exceptions, der aufgerufenen Klassen ab und schreibt den Fehler in den `errorcode` der Aufgabe. Sie setzt zudem den `status` auf ERROR und gibt False zurück.

9 Audiokonverter

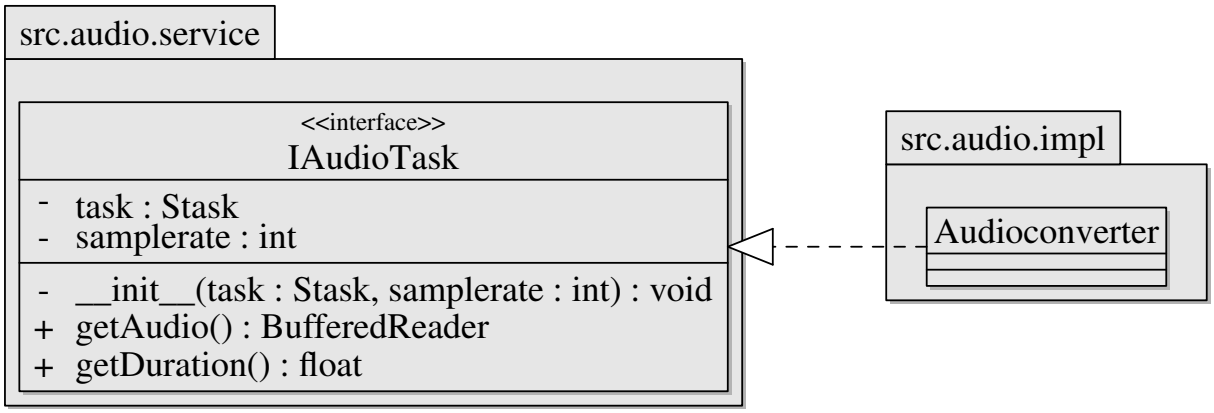


Abbildung 7: Komponente für das Erzeugen eines Audiostreams

Da die Spracherkennungsbibliotheken auf eine bestimmte Audiofrequenz optimiert werden, diese sollte der Spracherkennungsbibliothek vorliegen. Die Klasse entnimmt der Audio- oder Videodatei die Audiospur und wandelt diese in einem Stream mit der passenden Audiofrequenz um.

9.1 IAudioTask

Die Schnittstelle wird durch die `Audiokonverter`-Klasse implementiert.

Verhalten

__init__(task : Stask, samplerate : int) Die Funktion speichert die übergebenen Parameter in den lokalen Variablen.

getAudio() Die Funktion entnimmt den in der Aufgabe gespeicherten Dateipfad zur Video- oder Audiodatei. Dieser Datei wird mithilfe von FFmpeg die Audiospur entnommen, in die angegebene Samplerate konvertiert und als BufferedReader zurückgegeben.

getDuration() Die Funktion nimmt den in der Aufgabe gespeicherten Dateipfad zur Video- oder Audiodatei. FFmpeg wird die Abspiellänge dieser Datei heraussuchen und diese in Sekundeneinheiten als float zurückgegeben.

10 Translator

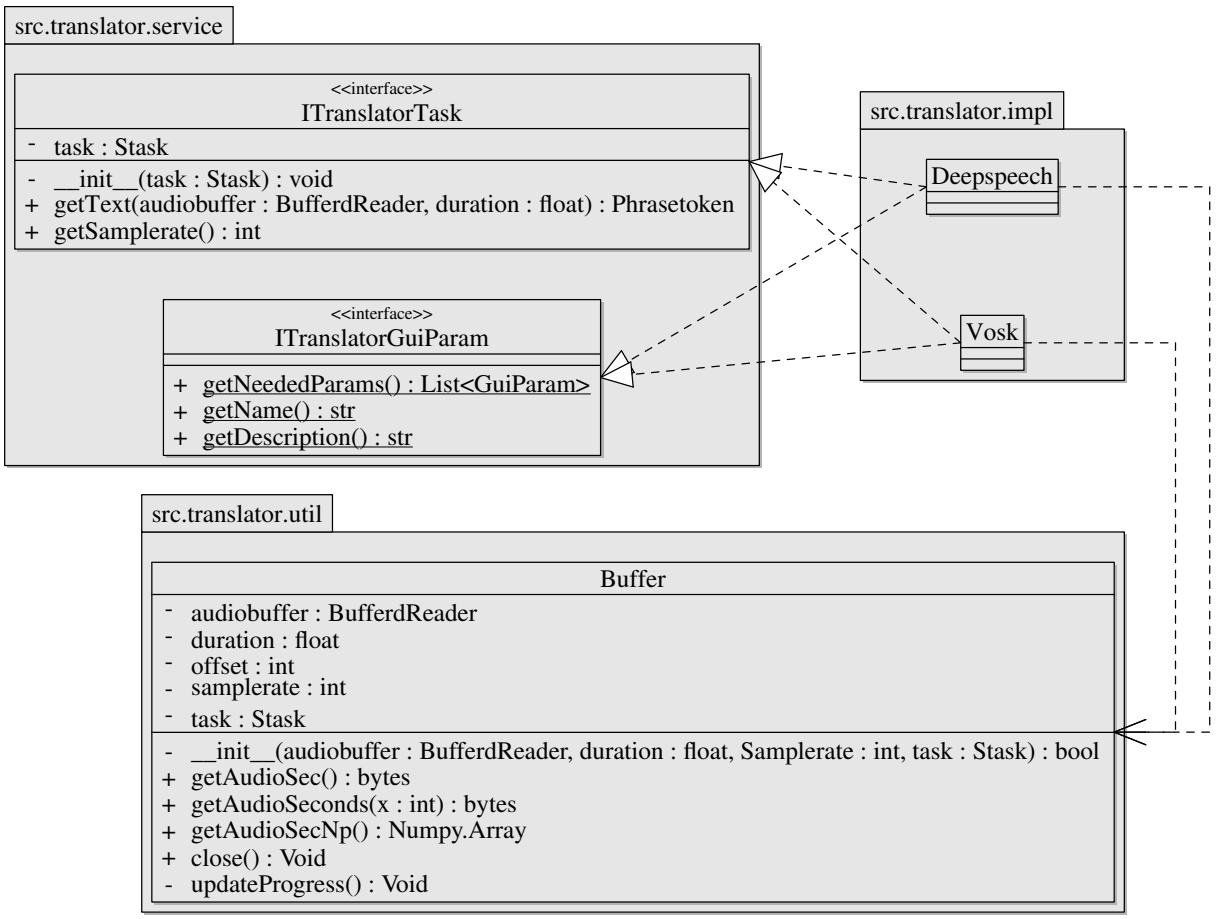


Abbildung 8: Struktur der Translator-Komponente

Die Struktur der `Translator`-Komponente ist so aufgebaut, dass weitere Implementationen dem Programm hinzugefügt werden können, ohne das andere Klassen verändert werden müssen. So können neue Spracherkennungsbibliotheken sehr einfach hinzugefügt werden. Die Implementation sollte die `Buffer`-Klasse nutzen, um sowohl die Nutzung des `Audiostreams`, als auch die Aktualisierung der Fortschrittsanzeige durch vorgefertigte Funktionen zu erleichtern.

10.1 ITranslatorTask

Die Schnittstelle kann durch Klassen im `src.translator.impl` Ordner implementiert werden.

Verhalten

`__init__(task : Stask)` Die Funktion wird zum Initialisieren der Spracherkennungsbibliotheken genutzt, so sollen z.B. die KI-Modelle geladen und parametrisiert werden. Die nötigen Parameter, die genutzt werden, sind der `translatorparam`-Variable der Aufgabe zu entnehmen.

getText(audiobuffer : Stream, duration : float) Mit den übergebenen Werten kann eine `Bufferklasse` initialisiert werden. Aus dieser kann die Spracherkennungsbibliothek mit kleine Abschnitten des Audiostreams versorgt werden. Der erkannte Text wird in eine `PhraseToken`-Klasse (Siehe Abbildung 4) gespeichert und zurückgegeben.

getSamplerate() Die Funktion gibt die bevorzugte `Samplerate` des zu verarbeitenden `Audiostreams` aus.

10.2 ITranslatorGuiParam

Die Schnittstelle kann durch Klassen im `src.translator.impl` Ordner implementiert werden.

Verhalten

getNeededParams() Die Funktion erzeugt Instanzen der Klasse `GuiParam`. Diese werden mit den für die Spracherkennungsbibliothek nötigen Parametern gefüllt, um diese in der Benutzeroberfläche anzuzeigen. Die Liste an `GuiParam`-Klassen wird zurückgegeben.

getName() Gibt den Namen des `Translators` zurück, unter dem er angezeigt werden soll.

getDescription() Gibt eine Beschreibung der aktuellen Implementation zurück.

10.3 Buffer

Diese Klasse kann genutzt werden, um die Nutzung des Audiostreams einfacher zu gestalten. Durch das Nutzen der Klasse wird automatisch die Aktualisierung der Fortschrittsanzeige vorgenommen.

Verhalten

__init__(audiobuffer : BufferdReader, duration : float, Samplerate : int, task : Stask) Die übergebenen Werte werden intern gespeichert. `offset` wird auf 0 gesetzt.

getAudioSec() Es wird ein bytearray zurückgegeben das der nächsten Sekunde im `audiobuffer` entspricht. `offset` wird um den Wert 1 erhöht und `updateProgress()` wird aufgerufen.

getAudioSeconds(x : int) Wenn `x` kleiner als eins ist, wird x auf eins gesetzt. Es wird ein bytearray zurückgegeben, das den nächsten `x` Sekunden im `audiobuffer` entspricht. `offset` wird um den Wert `x` erhöht und `updateProgress()` wird aufgerufen.

getAudioNp() Es wird ein NumPy.Array zurückgegeben, das der nächsten Sekunde im `audiobuffer` entspricht. `offset` wird um den Wert 1 erhöht und `updateProgress()` wird aufgerufen.

close() Schließt den lokal gespeicherten `audiobuffer`.

updateProgress() Die Funktion ruft `setProgress()` des lokalen `task` auf. Übergibt als Wert `offset / duration`.

11 Formator

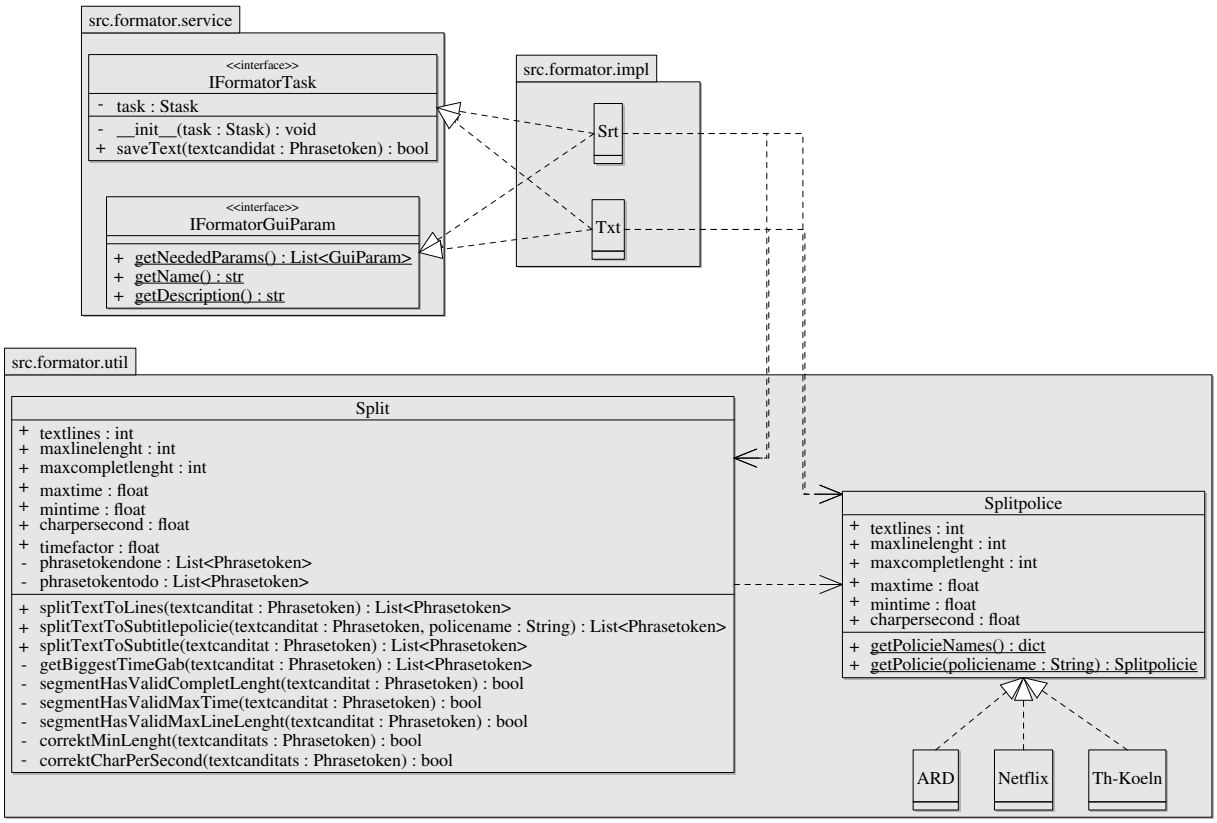


Abbildung 9: Struktur der Formator-Komponente

Die Struktur der `Formator`-Komponente ist so aufgebaut, dass weitere Implementationen dem Programm hinzugefügt werden können, ohne das andere Klassen verändert werden müssen. So können neue `Formatoren` sehr einfach hinzugefügt werden. Die `Formatoren` können bei der Erzeugung von Untertiteln auf die `Split`-Klasse zugreifen, die Funktionen wie das Unterteilen des Textes bereithalten.

11.1 IFormatorTask

Die Schnittstelle kann durch Klassen im `src.formator.impl` Ordner implementiert werden.

Verhalten

`__init__`(task : Stask) Die Funktion wird zum Initialisieren des `Formators` genutzt. Die nötigen Parameter die genutzt werden, sind der `formatorparam`-Variable der Aufgabe zu entnehmen.

saveText(textcandidate : PhraseToken) Der übergebene `PhraseToken` wird nun gemäße der Implementation verarbeitet und abgespeichert. Dabei kann die Klasse `Split` zu Hilfe genommen werden. Der Speicherort ist der `filelocation`-Variable der Aufgabe zu entnehmen, der Dateiname ist der `filename`-Variable zu entnehmen. Dem Dateinamen ist eine Endung hinzuzufügen. Welche ist von der Implementation abhängig.

11.2 IFormatorGuiParam

Die Schnittstelle kann durch Klassen im `src.formator.impl` Ordner implementiert werden.

Verhalten

getNeededParam() Die Funktion erzeugt Instanzen der Klasse `GuiParam`. In diese werden mit den für die Spracherkennungsbibliothek nötigen Parametern gefüllt, um diese in der Benutzeroberfläche anzuzeigen. Eine Liste an `GuiParam`-Klassen wird zurückgegeben.

getName() Die Funktion gibt den Namen des `Formators` zurück, unter dem er angezeigt werden soll.

getDescription() Die Funktion gibt eine Beschreibung der aktuellen Implementation zurück.

11.3 Split

Die Klasse ist eine Hilfsklasse und kann zum Trennen von Texten genutzt werden. Die in der Klasse enthaltenden Variablen sind Standardwerte, können aber auch verändert werden. Die Funktionen nutzten diese Werte in ihren Funktionen.

Verhalten

splitTextToLines(textcandidate : PhraseToken) Die Funktion teilt eine `PhraseToken`-Instanz in eine Liste aus `PhraseTokens` auf. Die maximale Textlänge eines `PhraseTokens` ist gleich der `maxlinelenght` der `Split`-Klasse. Die `PhraseToken` werden immer nur an `CharToken` mit Leerzeichen getrennt. Wenn es kein Leerzeichen gibt, wird es an der `maxlineleght -1` getrennt und mit einem Bindestrich versehen.

splitTextToSubtitelpolicie(textcandidate : PhraseToken, policiename : str) Ruft mit den Parameter `policiename` die Methode `getPolicies` der Klasse `Splittpolicie` auf und bekommt Klasse mit allen Parametern der `Split`-Klasse zurück. Diese Parameter werden in die `Split`-Klasse übernommen. Anschließend wird eigene Methode `splitTextToSubtitel` aufgerufen.

splitTextToSubtitel(textcandidate : PhraseToken) Diese Funktion teilt den `textcandidate` in mehrere `PhraseTokens` auf. Jeder `PhraseToken` symbolisiert am Ende eine auf dem `Bildschirm` zu sehende Zeichenfolge. Dafür werden folgende Schritte genutzt. Dafür wird eine Art DFS-Algorithmus genutzt.
1. Dieser Punkt wird beim ersten Durchlauf übersprungen. Mithilfe der Funktion `getBiggestTimeGab()` wird der `PhraseToken` in zwei `PhraseToken` geteilt. Der erste Token wird als aktueller `PhraseToken` weiter benutzt, der zweite wird an die erste Stelle der `phraseTokentodo` Liste gesetzt.
2. Der aktuelle `PhraseToken` wird mithilfe von `correctMinLenght()` an die minimale Zeit angepasst, die ein Text zusehen seinen darf.
3. Der aktuelle `PhraseToken` wird mithilfe von `correctCharPerSecond()` an die maximale zulässige Lesegeschwindigkeit versucht anzupassen.

4. Nun wird der aktuelle `PhraseToken` mithilfe von `segmentHasValidCompletLenght()` auf die maximale Länge geprüft. Wenn die Prüfung fehlschlägt, wird mit dem aktuellen `PhraseToken` von Punkt 1 weiter gemacht.
5. Der aktuelle `PhraseToken` wird mithilfe von `segmentHasValidMaxTime()` auf die maximale Zeit geprüft. Wenn die Prüfung fehlschlägt, wird mit dem aktuellen `PhraseToken` von Punkt 1 weiter gemacht.
6. Der aktuelle `PhraseToken` wird mithilfe von `segmentHasValidMaxLineLeght()` auf den maximalen Zeicheninhalt von jeder Zeile geprüft. Wenn die Prüfung fehlschlägt, wird mit dem aktuellen `PhraseToken` von Punkt 1 weiter gemacht.
7. Der aktuelle `PhraseToken` wird an die `phraseTokendone`-Listen angehängt. Mit dem ersten Eintrag aus der `phraseTokentodo`-Liste wird ab Punkt 2 weiter gemacht. Wenn keine Einträge mehr in der `phraseTokentodo`-Listen zu finden sind, wird die `phraseTokendone`-Liste zurückgegeben.

getBiggestTimeGab(textcandidate : PhraseToken) Alle `CharToken` in der `PhraseToken`-Instanz, werden nach sämtlichen Leerzeichen durchsucht. Die Zeiten der Leerzeichen werden mit folgender Funktion multipliziert:

$$y = (-(\frac{Chatoken.starttime - PhraseToken.starttime}{PhraseToken.endtime - PhraseToken.starttime} - 0.5)^2 + 0.5) \cdot timefactor + 1$$

und mit einer weiteren Funktion addiert.

$$y = \frac{(-(\frac{Chatoken.starttime - PhraseToken.starttime}{PhraseToken.endtime - PhraseToken.starttime} - 0.5)^2 + 0.5) \cdot timefactor}{10}$$

Die `PhraseToken`-Instanz wird an dem Punkt getrennt, wo das Leerzeichen den höchsten Wert aufweist. Beide `PhraseToken` werden in passender Reihenfolge als Liste zurückgegeben.

segmentHasValidCompletLeght(textcandidate : PhraseToken) Es wird geprüft, ob die Länge des in `PhraseToken` enthaltenden Textes, länger ist als die `maxcompletleght`-Variable vorgibt. Wenn der Text zu lang ist, wird False zurückgegeben, wenn nicht True.

segmentHasValidMaxTime(textcandidate : PhraseToken) Die Funktion prüft, ob die Länge der `PhraseToken` länger ist als die `maxtime`-Variable. Wenn die Länge eingehalten wurde, wird ein True zurückgegeben sonst False.

segmentHasValidMaxLineLenght(textcandidate : PhraseToken) Die Funktion prüft, ob die Wörter des `PhraseToken` so aufgeteilt werden können, dass die maximale Zeilenanzahl (`textlines`) eingehalten wird und in jeder Zeile die `maxlinelenght` eingehalten werden kann. Wenn dies der Fall ist, wird an der passenden Stelle ein `CharToken` mit einem Zeilenumbruch eingesetzt und True zurückgegeben, wenn nicht wird nur False zurückgegeben.

correktMinLenght(textcandidate : PhraseToken) Wenn die Länge von dem `PhraseToken` geringer ist als die `mintime`, wird versucht die `starttime` und die `endtime` von `PhraseToken` so zu versetzen, dass die `mintime` erreicht wird. Dabei soll auf den vorherigen `PhraseToken` (letzter Eintrag in `phraseTokendone`) und den nachfolgenden (erster Eintrag in `phraseTokentodo`) geachtet werden.

cerrektCharPerSecond(textcandidate : PhraseToken) Wenn in dem `PhraseToken` mehr Zeichen pro Sekunde gelesen werden müssen, als `charpersecond` vorgibt, wird versucht die `starttime` und die `endtime` weiter so zu versetzen, dass `charpersecond` unterschritten wird. Dabei soll auf den vorherigen `PhraseToken` (letzter Eintrag in `phraseTokendone`) und den nachfolgenden (erster Eintrag in `phraseTokentodo`) geachtet werden.

11.4 Splitpolicie

Verwaltet die verschiedenen Trennrichtlinien für die verschiedenen Plattformen.

Verhalten

getPolicieName() Gibt ein dict aus den einzigartigen Strings und den Displaynamen aller verfügbaren Trennungsrichtlinien an.

getPolicie(policiename : String) Bekommt den einzigartigen String einer Trennungsrichtlinie übergeben und gibt eine Klasse mit allen Parametern für die Trennungsrichtlinien zurück.

12 Corrector
Noch zu implementieren

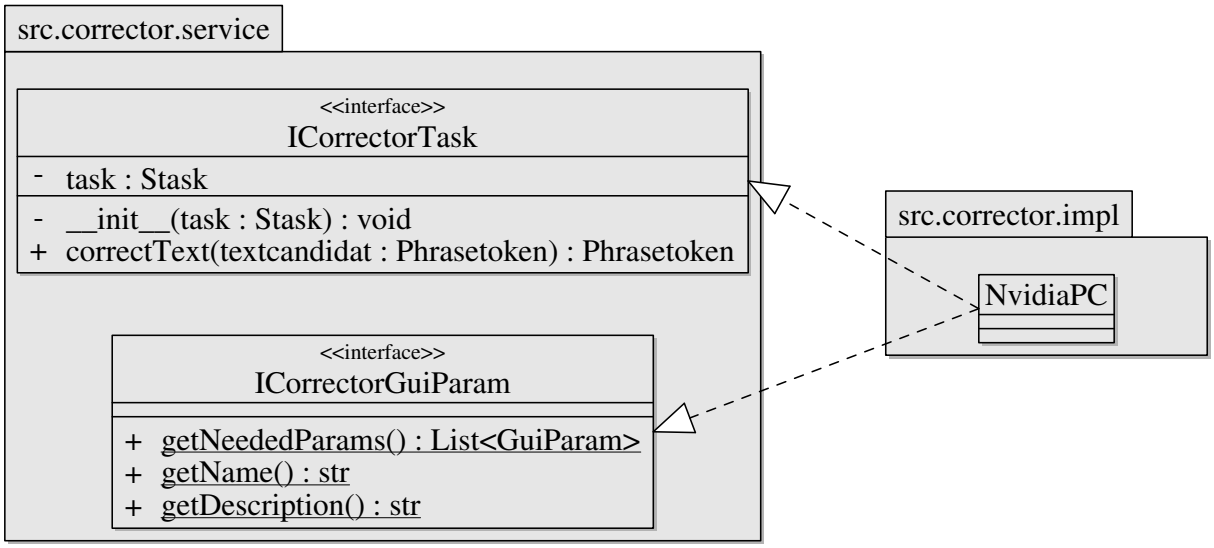


Abbildung 10: Struktur der Corrector-Komponente

Die Struktur der `Corrector`-Komponente ist so aufgebaut, dass weitere Implementationen dem Programm hinzugefügt werden können, ohne das andere Klassen verändert werden müssen. So können neue `Correctoren` sehr einfach hinzugefügt werden. Die `Correctoren` können den von dem Translatoren erkannten Text korrigieren und als `textcandidate` zurückgeben.

12.1 ICorrectorTask

Die Schnittstelle kann durch Klassen im `src.corrector.impl` Ordner implementiert werden.

Verhalten

__init__(task : Stask) Die Funktion wird zum Initialisieren des `Correctors` genutzt. Die nötigen Parameter die genutzt werden, sind der `correctorparam`-Variable der Aufgabe zu entnehmen.

correctText(textcandidate : PhraseToken) Der übergebene `PhraseToken` wird nun gemäße der Implementation korrigiert. Der überarbeitetet Text, wird als `PhraseToken` zurückgegeben.

12.2 ICorrectorGuiParam

Die Schnittstelle kann durch Klassen im `src.corrector.impl` Ordner implementiert werden.

Verhalten

getNeededParam() Die Funktion erzeugt Instanzen der Klasse `GuiParam`. In diese werden mit den für den Corrector nötigen Parametern gefüllt, um diese in der Benutzeroberfläche anzuzeigen. Eine Liste an `GuiParam`-Klassen wird zurückgegeben.

getName() Die Funktion gibt den Namen des `Correctors` zurück, unter dem er angezeigt werden soll.

getDescription() Die Funktion gibt eine Beschreibung der aktuellen Implementation zurück.