Masterarbeit

Implementierung Automatisierter Audio Text Timing Transkription mit Automatischer Spracherkennung in Windows

Masterarbeit für das
Institut für Softwaretechnologie

Bajramovic Faruk

Masterarbeit

Implementierung Automatisierter Audio Text Timing Transkription mit Automatischer Spracherkennung in Windows

Masterarbeit für das
Institut für Softwaretechnologie

Masterarbeit

an der

Technischen Universität Graz

eingereicht von

Bajramovic Faruk

Institut für Softwaretechnologie (IST),
Graz Technischen Universität Graz
A-8010 Graz, Austria

01 Jan 2011-2012

Begutachter: Univ.-Prof. DI. Dr. Wolfgang Slany

Master's Thesis

Implementation of Automated Audio Text Timing Transcription with Automated Speech Recognition in Windows

Master's thesis for the Institute of Technology, Graz University of Technology

Master's Thesis

at

Graz University of Technology

submitted by

Bajramovic Faruk

Institute for Softwaretechnology (IST),
Graz University of Technology
A-8010 Graz

1. January 2011-2012

This thesis is written in german.

Advisor: Univ.-Prof. DI Dr. Wolfgang Slany

Abstract

This master's thesis objective was originally the research and development of up to three possible methods to demonstrate how it would be possible to create text timing data about a given text in any given audio file. While doing the resarch three acceptable methods of solving this problem were probed. DSP (Digital Signal Processing) with heuristics to match words with audio recognition, SPR (Speech Recognition) with DSP, and the possibility of applying avaliable Speech Recognition Engines. The first method had very poor results and faced very big obstacles. The second method required the training of neural networks and would in theory provide only good results with trained and very well developed dictionaries. The last approach showed very promising results early on. It was decided to change the scope of the thesis and advance with the development of a software solution using the Microsoft Speech API in .NET. The selected work methodology is TDD (Test-Driven-Development cite) with an agile development strategy.

Kurzfassung

Das Ziel dieser Masterarbeit war ursprünglich das Erforschen und Ausarbeiten von drei Möglichkeiten, die demonstrieren sollten, wie es möglich wäre Text-Timing Daten über einen bestimmten Text einer Audiodatei zu gewinnen. Während der Recherchearbeit wurden drei prinzipielle Ansätze untersucht, dieses Problem zu lösen: DSP (Digitale Signalverarbeitung) mit Heuristik um Wörter entsprechend ihrer Länge und auftretenden Audiosignalen zu übereinstimmen, Spracherkennung mit Neuronalen Netzen unterstützt durch DSP, und die Möglichkeit der Anwendung von einer frei zugänglichen Spracherkennungs-API. Die erste Methode hatte bereits bei den ersten Tests sehr schlechte Ergebnisse und große technische Schwierigkeiten offenbart. Die zweite Methode stellte voraus, dass Neuronale Netze trainiert werden, und dass eine große Bibliothek mit Daten vorhanden ist. Der letzten Ansatz erwies sich schon zu Beginn erster Tests als sehr aussichtsreich. Es wurde entschieden, den Umfang und die Zielsetzung der Masterarbeit zu ändern und mit der Implementierung einer konkreten Software zu beginnen. Zum Einsatz kam die Microsoft Speech API in .NET. Als Arbeitsmethode wurde nach dem Agilen Entwicklungsprinzip mit Test-Driven-Development vorgegangen.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Ort	Datum	Unterschrift

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Place	Date	Signature

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Danksagung

An dieser Stelle möchte ich mich bei all den Personen bedanken, die mir auf meinem Weg eine große Hilfe waren.

Ich möchte mich zu aller erst beim Hr. Prof. Dr. Slany bedanken, der nicht nur durch seine überragende Kompetenz, sondern auch durch seine Geduld immer eine unentbehrliche Hilfe gewesen ist, wenn Hilfe nötig war. Danke.

Besonders möchte ich meinen Eltern danken, Dzevad und Subhija Bajramovic und meinem Bruder Fuad. Sie haben mich immer kompromisslos unterstützt. *Hvala, volim vas.* Was ich für meine Freundin Christiane fühle, die stets ein großer Rückhalt an meiner Seite war, kann ich gar nicht in Worte fassen. Ohne sie wäre das alles unmöglich gewesen. Und zuletzt möchte mich bei Igor Skoric bedanken, für seine Hilfe beim Testen und dafür, dass er seit Jahren ein treuer Freund ist.

Bajramovic Faruk
Graz, Austria, April 2012

Kapitel 1

Einleitung

" The true sign of intelligence is not knowledge but imagination. Logic will get you from A to B. Imagination will take you everywhere. "

[Albert Einstein, 1879-1955]

Sprachlernlernsoftware kommt in den letzten Jahren eine immer größere Bedeutung zu. Obwohl die Ursprünge moderner Spracherkenner in den achtziger Jahren liegen, ist eine Anwendung erst durch die hohe Rechenleistung moderner Computersysteme möglich geworden. Die Möglichkeiten der Anwendung sind sehr breit gefächert, und sind in der heutigen Welt der Anwendungen spärlich vertreten. Besonders hohe Erwartungen an die Anwendung von Spracherkennung werden in den Bereichen der medizinischen Betreuung gelegt, und der sprachlichen Bedienung von Computersystemen [Markowitz und Scholz, 2010].

Ende 2010 begann die Arbeit an diesem Werk, welches zum Ziel hatte, eine Audio-Text-Timing Software zu erstellen. Dabei sollte aus gegebenen Audiodateien und einem gegebenen Text ein Output produziert werden. Dieser Output wird in einer XML Datei gespeichert (siehe W3C [2011]), und enthält Informationen über die Länge, Position und ausgewählte Metadaten. Im Kapitel 4 wird der Inhalt der XML-Datei näher beschrieben.

Die Vilango GmbH produziert Sprachlernsoftware nach der Birkenbihl Methode (Birkenbihl [2004]), und war der Auftraggeber der Arbeit. Um die Erstellung der Lernkapitel zu vereinfachen wurde ein Programm benötigt, dass die Erstellung von Audio-Text-Timings automatisiert durchführt. Verwirklicht wurde das Programm 'Vilango SpeechApp' in Zusammenarbeit mit dem Institut für Softwaretechnologie, an der Technischen Universität Graz mit Betreuung durch Univ.-Prof. DI Dr. Wolfgang Slany. Die zugrunde liegenden Technologien, die benutzt wurden sind das Microsoft Windows .NET Framework (Microsoft .NET [2011]) und das Windows 7 Software Development Kit (Microsoft SDK [2011]) mit der Microsoft Speech API (Microsoft SAPI [2011]). Auf die spezifischen Einzelheiten über die Technologien wird im Kapitel 2 näher eingegangen.

Da das Projekt als exploratorische Arbeit definiert ist, um im Rahmen einer Machbarkeitsstudie festzustellen, ob eine Lösung realisiert werden kann oder nicht, wurden verschiedene Lösungsansätze untersucht und Vergleiche gezogen. Im zweiten Schritt sollen die gefundenen Möglichkeiten auf ihren Nutzen hin verglichen werden.

Durchgeführt wurde die Arbeit unter den Prinzipien des Test-Driven-Development wie beschrieben in Link und Fröhlich [2002]; Madeyski [2010]; Beck [2002]; Bender und McWherter [2011]; Astels [2003] und Agiler Softwareentwicklungsprozesse (Beck und Andres [2004];

Jeffries [2004]), die im Kapitel 3 beschrieben werden. Das Design und die Architektur werden im Kapitel 5 im Allgemeinen vorgestellt. Auf wichtige Designdetails wird anschließend in Kapitel 6 eingegangen. Mit Kapitel 7 über die Codemetriken und die Testdokumentation der Arbeitsweise wird der technische Teil der Arbeit abgeschlossen.

Den Schlusspunkt bildet das Fazit mit Vorschlägen, welche Verbesserungen und Erweiterungen möglich wären, um das Programm noch effizienter zu machen, die im Rahmen dieser Masterarbeit aus Gründen des zeitlichen Aufwandes nicht durchgeführt werden konnten. Schließlich wird im Appendix A eine kurze Demonstration und Anleitung des Programmes vorgestellt.

Abbildung 1.1 illustriert grob die Problemstellung und die Anforderungen an das Programm. Es stellt den Zusammenhang aus Input und erwarteten Output dar. Die Birkenbihl Methode ist ein Zusammenspiel aus De-Kodieren und Aktiv Hören ([Birkenbihl, 2004]). Die Vilango Sprachlernsoftware visualisiert dabei die zu hörenden Laute und Wörter während der Lernende diese mitliest. Dies ist die Gehirn-gerecht/Birkenbihl-Methode, wie beschrieben in Birkenbihl [2004].

1.1 Problemstellung

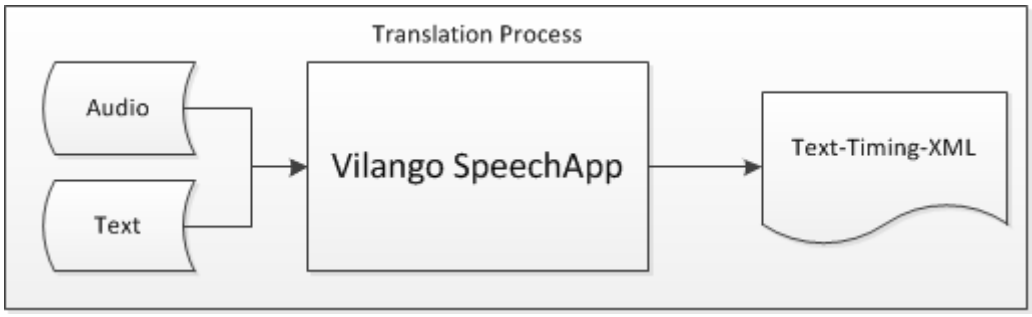


Abbildung 1.1: Ein Übersetzungsprozess von Text-Audio-Input zu Timings-XML stellt den Zusammenhang aus Input und erwarteten Output dar.

Für die Umsetzung dieser Methode muss die audiovisuelle Übereinstimmung zwischen gesprochenem Wort und sichtbaren Lauten möglichst genau sein. Vilango hat bisher händisches Übersetzen benutzt. Dabei bedienten sie sich eines nicht näher bekanntes Programmes als Hilfestellung. Dieser Prozess soll nun vollautomatisiert umgesetzt werden. Als Input dient eine Audiodatei und ein Text, dessen Wörter und die dazugehörigen Timings, in der richtigen Reihenfolge gefunden werden müssen. Danach werden die Daten in einer definierten XML-Datei ausgegeben.

Dabei werden insbesondere folgende Schwierigkeiten im Kapitel 6 näher betrachtet: lange Stille-Phasen im Audio, unterschiedliche Sprecher mit unterschiedlichen Dialekten, Laute - insbesondere Tierlaute - und durcheinander Reden in Gruppen. Da für die Lösung die In-Process Speech Recognition Engine von Microsoft in einem **untrainierten** Zustand benutzt wird, stellen vor allem die letzten zwei Problempunkte eine große Herausforderung dar, die nicht zur Gänze gelöst werden konnten (Siehe Kapitel 6).

Das Programm soll zudem auch Zwischenergebnisse von erfolgreich übersetzten Audio-Text-Timings produzieren, selbst wenn Teile der Datei nicht erfolgreich analysiert werden konnten (Siehe Kapitel 6).

1.2 Untersuchte Ansätze

Für die Lösung der beschriebenen Problemstellung werden zunächst die verschiedenen Ansätze wie Heuristik, Digitale Signalverarbeitung und Spracherkenner betrachtet. Nach der Gegenüberstellung der einzelnen Vor- und Nachteile wird gemeinsam mit dem Kunden die Entscheidung getroffen, welcher Ansatz als Lösung in Frage kommt.

1.2.1 Heuristik

Die einfachste und billigste Methode die zur Auswahl stand, war eine rein heuristische Methode. Dabei wurde der Text in einzelne Wort-Token umgewandelt und es wurde versucht aufgrund der Wortlänge die Wörter in der richtigen Reihenfolge zu platzieren.

Aufgrund der schlechten Genauigkeit wurde diese Idee verworfen. Abbildung 1.2 veranschaulicht den Satz:

(1) Now what am I going to do all there by myself?

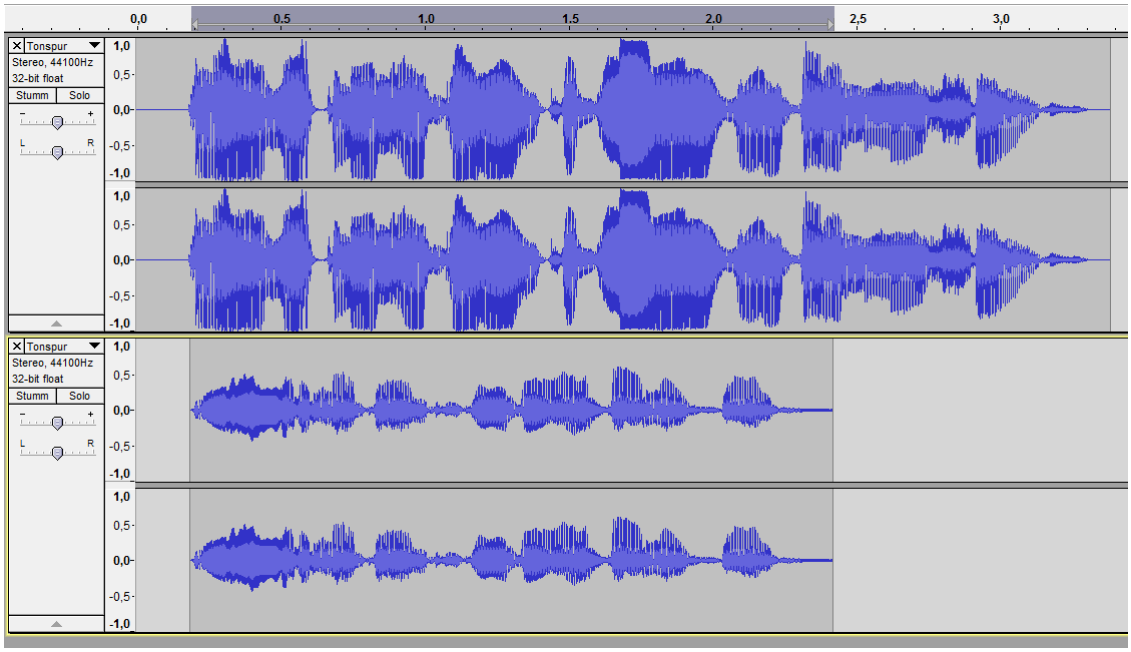


Abbildung 1.2: Wellenform des Satzes (1). Einmal TTS-Generiert (Oben) einmal natürliche Sprache (Unten). Visualisiert durch Audacity. (siehe Team Audacity [2011])

Die gesamte Satzlänge beträgt ungefähr zwei Sekunden in natürlicher Sprache. Die Referenzlänge des Text-to-Speech Audio beträgt knapp drei Sekunden. Es entsteht eine Abweichung von 50 Prozent. Diese Abweichung übersteigt die Anforderung um ein Vielfaches. Die Abweichung sollte mit dem Auge nicht sichtbar sein [Birkenbihl, 2004].

1.2.2 Digital Signal Processing

Der zweite Ansatz war ein Versuch, die Worte aufgrund der Audiofrequenzen zu trennen. Dies war jedoch ohne Spracherkennung nicht möglich. Man muss Geräusche, Tierlaute und andere

Audiosignale von Sprache trennen. Dieser Prozess wird Voice Activation Detection genannt, und ist ein Teil moderner Spracherkennungstechnik (Vergleiche Nilsson und Ejnarsson [2002]). Daraus ergab sich der endgültige und gewählte Ansatz: Ein durch Spracherkennung gestütztes, heuristisches Auswahlverfahren.

1.2.3 Gewählte Methode: Timing Extraktion gestützt durch Spracherkennung

Die Grundidee beim gewählten Ansatz ergab sich aus der Untersuchung der verfügbaren Technologien zum NLP (Natural Language Processing). In Kapitel 2 sind die untersuchten, frei verfügbaren Technologien aufgeführt, die in Betracht gezogen wurden.

Es wurde die Speech API In-ProcEngine von Microsoft gewählt, die Hypothesen über den gesprochenen Inhalt in der Audioaufnahme aufstellt. Mit Hilfe des vorgegebenen Textes wird der Searchspace eingeschränkt um die Genauigkeit zu erhöhen. Danach wird durch geeignete Auswahl aus einem Hypothesenraum erkannter Textpassagen ausgesucht und der restliche zu untersuchende Text reduziert. Auf die Grundidee bei der Lösung wird in Kapitel 5 näher eingegangen.

Kapitel 2

Wichtigste Technologien und Ansätze

Dieses Kapitel beschreibt zunächst die Grundlagen der Spracherkennung, und stellt anschließend die wichtigsten frei verfügbaren Technologien im Bereich der automatisierten Spracherkennung vor.

2.1 Grundlagen der Spracherkennung

Dieses Unterkapitel soll keine detaillierte Grundlage zur automatischen Spracherkennung liefern. Es soll in Grundzügen erklären, wie Audiosignale mit Phonem–Erkennung oder Ganzwort–Erkennung analysiert werden, und wie dabei Resultate entstehen. Im besonderen wird der auf Hidden-Markov-Models (HMM) basierte Ansatz vorgestellt. Zum HMM basierten Ansatz gibt es mehrere Veröffentlichungen (Vergleiche Gales und Young [2007]; Nilsson und Ejnarsson [2002]; Hosom [2011]; Lawrence und Rabiner [1989]; Levinson et al. [1983]; Paul [1990]; Jurafsky und Martin [2000]; Carstensen et al. [2001]). Andere Ansätze und Modelle sind Motor Theory, Multiple-Cue Modell, Dynamic Time Warping, und Fletcher-Allen Modell. Nach Hosom [2011] existiert kein Beweis, dass die auf HMMs basierte Lösung auch die beste sei.

2.1.1 Hidde-Markov-Model basierte Automatische Spracherkennung Systeme

Gales und Young [2007]; Nilsson und Ejnarsson [2002]; Hosom [2011]; Lawrence und Rabiner [1989]; Levinson et al. [1983]; Paul [1990]; Jurafsky und Martin [2000]; Carstensen et al. [2001]; Hosom et al. [1999] bilden die Grundlage des präsentierten Wissens in diesem Unterkapitel. Es folgt eine semantische Zusammenfassung der wichtigsten Prinzipien der automatisierten Spracherkennung. Die Entwicklungsarbeit bei der *Vilango SpeechApp* basiert auf einem Lösungsansatz, der aus dem Verständnis der Funktionsweise von modernen Spracherkennern hervorgeht. Es ist ausdrücklich nicht die Absicht, oder das Ziel, dieser Masterarbeit eine fundierte und ausführliche Auseinandersetzung mit moderner Sprachtechnologie zu führen.

Erkennung in vier Grundschritten

Die Grundoperationen eines HMM-Erkenners können in vier Schritte gegliedert werden (Abbildung 2.1).

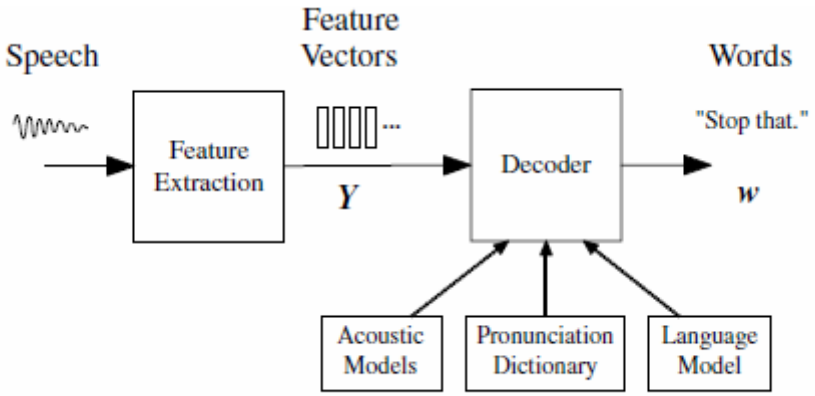


Abbildung 2.1: Grundoperationen eines HMM Erkenners. Quelle: Gales und Young [2007]

Erster Schritt: Digitalisieren des Audiosignals

Vor der Merkmalsextraktion wird die gesprochene Sprache digitalisiert und von der Zeitdomäne in die Frequenzdomäne überführt.

Zweiter Schritt: Merkmalsextraktion (Zum Beispiel Hauptkomponentenalayse)

Die Merkmalsextraktion ist eine Form der Dimensionsreduktion. Dieser Schritt sorgt dafür, dass Noise und unwichtige Information gefiltert werden. Die Merkmale werden aus dem Spektrogramm des Signals generiert, und in sogenannten "Frames" gespeichert. Das bedeutet, dass Merkmale in eine Sequenz von festgelegten akustischen Vektoren konvertiert werden. Codiert werden diese Merkmale am häufigsten über *mel-frequency cepstral coefficients (MFCCs)* [Gales und Young, 2007; Hosom et al., 1999; Paul, 1990; Jurafsky und Martin, 2000; Nilsson und Ejnarsson, 2002].

Dritter Schritt: Klassifizieren der extrahierten Merkmale aufgrund von akustischen Modellen

In diesem Schritt wird ein Neuronales Netzwerk benutzt um eine Klassifikation der erfassten Merkmale durchzuführen, und eine Summe von Sequenzen von akzeptierten Modellen zu erstellen. Das Ziel ist es, die Reihenfolge und Klassifikation der erkannten Phoneme (Laute) durchzuführen. Diese unterscheiden sich, zum Beispiel, aufgrund unterschiedlicher Aussprache (Kulturunterschiede; als Beispiel: Britisches Englisch und US Englisch), die im Aussprache-Lexikon gespeichert ist. Jeder Laut ist als ein Wahrscheinlichkeitsmodell der Übergänge seiner Phoneme (HMM) dargestellt [Gales und Young, 2007; Hosom et al., 1999; Paul, 1990; Jurafsky und Martin, 2000; Nilsson und Ejnarsson, 2002].

Vierter Schritt: Viterbi–Suche der Wörter

Im letzten Schritt wird eine Viterbi–Suche nach den Wörtern, die im Wörterbuch vorhanden sind, durchgeführt. Alternativ wird auch "Forward Algorithm" und "Backward Algorithm" benutzt. (Es resultieren verschiedene Hypothesen von erkannten Wortsequenzen. Eine gängige Methode, diese zu speichern, ist in Wort-Gittern. (siehe Abbildung 2.2). Jedes Wort und jede Wortsequenz hat ein bestimmtes, ausgerechnetes Vertrauen (Confidence). Liegt eine Hypothese über einer bestimmten Vertrauensschwelle, dann wird die resultierende Wortkette als erfolg-

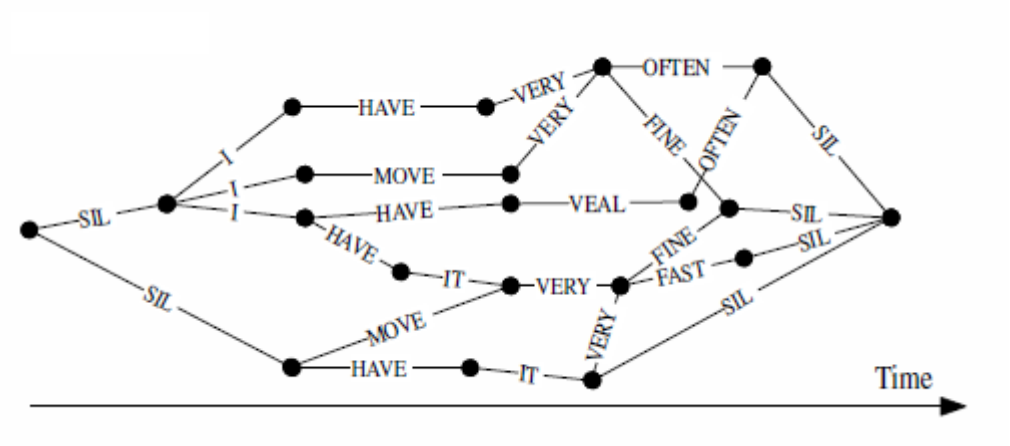


Abbildung 2.2: Wort-Gitter. Quelle: Gales und Young [2007]

reich erkannt angesehen. Das heißt auch, dass eine reduzierte Anzahl an Worten, und eine vorgegebene Reihenfolge das Vertrauen in die Erkennung erheblich steigern. Das ist wichtig für die Funktion des Programmes, weil eine komplett untrainierte Engine benutzt wird. Im folgenden Unterkapitel werden die wichtigsten frei verfügbaren Spracherkenner vorgestellt [Gales und Young, 2007; Hosom et al., 1999; Paul, 1990; Jurafsky und Martin, 2000; Nilsson und Ejnarsson, 2002].

2.2 Open Source Projekte - CMU Sphinx mit VoxForge

Das Carnegie Mellon University Projekt Sphinx ist das bekannteste frei verfügbare System zur Spracherkennung. Es ist Open–Source und in Java programmiert. Geliefert wird es mit einem Sprachmodell für Englisch [Carnegie Mellon University, 2011].

Ergänzt werden kann das System durch VoxForge. Auf VoxForge finden sich verschiedene akustische und sprachliche Modelle in verschiedenen Sprachen. [VoxForge, 2011].

2.3 HTK - Hidden Markov Model Toolkit

Das HTK Toolkit wurde 1989 im Cambridge University Engineering Department entwickelt. Es ist eine der bekanntesten und grundlegenden Bibliotheken zum modellieren von HMMs in Bezug auf automatische Spracherkennung. Es wurde Ende der neunziger Jahre von Microsoft erworben [HTK Toolkit, 1989].

2.4 Microsoft Speech API

Die Microsoft Speech API ist Teil des Windows 7 Software Development Kit. Die SAPI (Speech API) ist für die Verwendung unter Windows vorgesehen. Jede Windows 7 Version wird mit einer Spracherkennung–Engine geliefert. Diese enthält sowohl das akustische als auch sprachliche Modell, das für Spracherkennung notwendig ist. Standardmäßig werden diese Modelle zusammen mit dem Sprachpaket für Windows installiert. Die Vilanog SpeechApp basiert auf

der SAPI von Microsoft [Microsoft SAPI, 2011; Microsoft SDK, 2011]. Die Microsoft SAPI [2011] wurde für die Umsetzung des Programmes eingesetzt.

Kapitel 3

Arbeitsmethodik - Test Driven Development (TDD) und Agile Entwicklung

“ Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius – and a lot of courage – to move in the opposite direction. ”

[Albert Einstein, 1879-1955]

In diesem Kapitel wird Bezug auf die Arbeitsmethodik genommen, die bei der Entwicklung der Software eingesetzt wurde. Dieses Kapitel soll die wichtigsten Eckpunkte der Agilen Softwareentwicklung und des Test-Driven-Developments (TDD) anführen und erklären. Lechner [2010], und Schindler [2010] sind zwei Untersuchungen, die sich mit der Agilen Entwicklung in der Praxis auseinandersetzen. Obwohl das Konzept des TDD aus der Agilen Softwareentwicklung hervorging, sind sie nur bedingt miteinander verbunden. TDD kann auch außerhalb der Agilen Softwareentwicklung eingesetzt werden. TDD ist eine Technik, die aus den Testpraktiken des Extreme-Programming entstanden ist (Siehe: Beck und Andres [2004]). Die konkreten Beispiele, wie das TDD bei der Entwicklung der Software eingesetzt wurde, werden in Kapitel 6 demonstriert.

3.1 TDD in der Agilen Softwareentwicklung

Test Driven Development ist ein Ansatz bei der Agilen Entwicklung (siehe: Beck und Andres [2004]). Kent Beck, Erschaffer des Extreme Programming Konzeptes, ist einer der wichtigsten geistigen Väter dieser Entwicklungsmethode. In Beck und Andres [2004] vertritt Beck die Ansicht, dass TDD zu einfachem Design und nachweislich funktionierendem Code führt.

Seitdem wurden zahlreiche empirische Studien durchgeführt, welche die positiven Effekte vom TF-Ansatz (Test-First-Ansatz) belegen. In Madeyski [2010] findet sich eine Tabelle, die aktuelle Studien zum Einfluss von TF auf den Entwicklungsprozess anführt. Die Daten in Abbildung 3.1 sind Madeyski [2010] entnommen. Wie unschwer zu erkennen ist, sind die Ergebnisse widersprüchlich und sehr unterschiedlich. In seinem Werk stell Madeyski [2010] fest,

(übersetzt) “[..] dass die widersprüchlichen und unterschiedlichen empirischen Ergebnisse dadurch erklärt werden könnten, dass die Studien in unterschiedlichen

> Kontexten durchgeführt wurden, und es schwer ist die Effekte des TF von anderen Variablen zu isolieren. [..]"

Aufgrund eigener Forschung und näherer Untersuchung der veröffentlichten Studien präsentiert Madeyski [2010] seine eigenen Ergebnisse.

Demnach extrapoliert Madeyski in seinem Werk folgende sechs "Rules of thumb":

1. Thumb-Rule TF1: The TF practice has a positive impact on defect rate
2. Thumb-Rule TF2: The TF practice has a little impact on the percentage of acceptance tests passed
3. Thumb-Rule TF3: The TF practice has a little impact on the number of acceptance tests passed per hour
4. Thumb-Rule TF4: The TF practice has a medium (but close to large) and positive impact on the mean coupling between object classes
5. Thumb-Rule TF5: The TF practice has a little impact on the mean value of weighted methods per class
6. Thumb-Rule TF6: The TF practice has a little impact on the mean value of response for a class

Quelle: Madeyski [2010]

3.1.1 Prinzipien der Agilen Softwareentwicklung

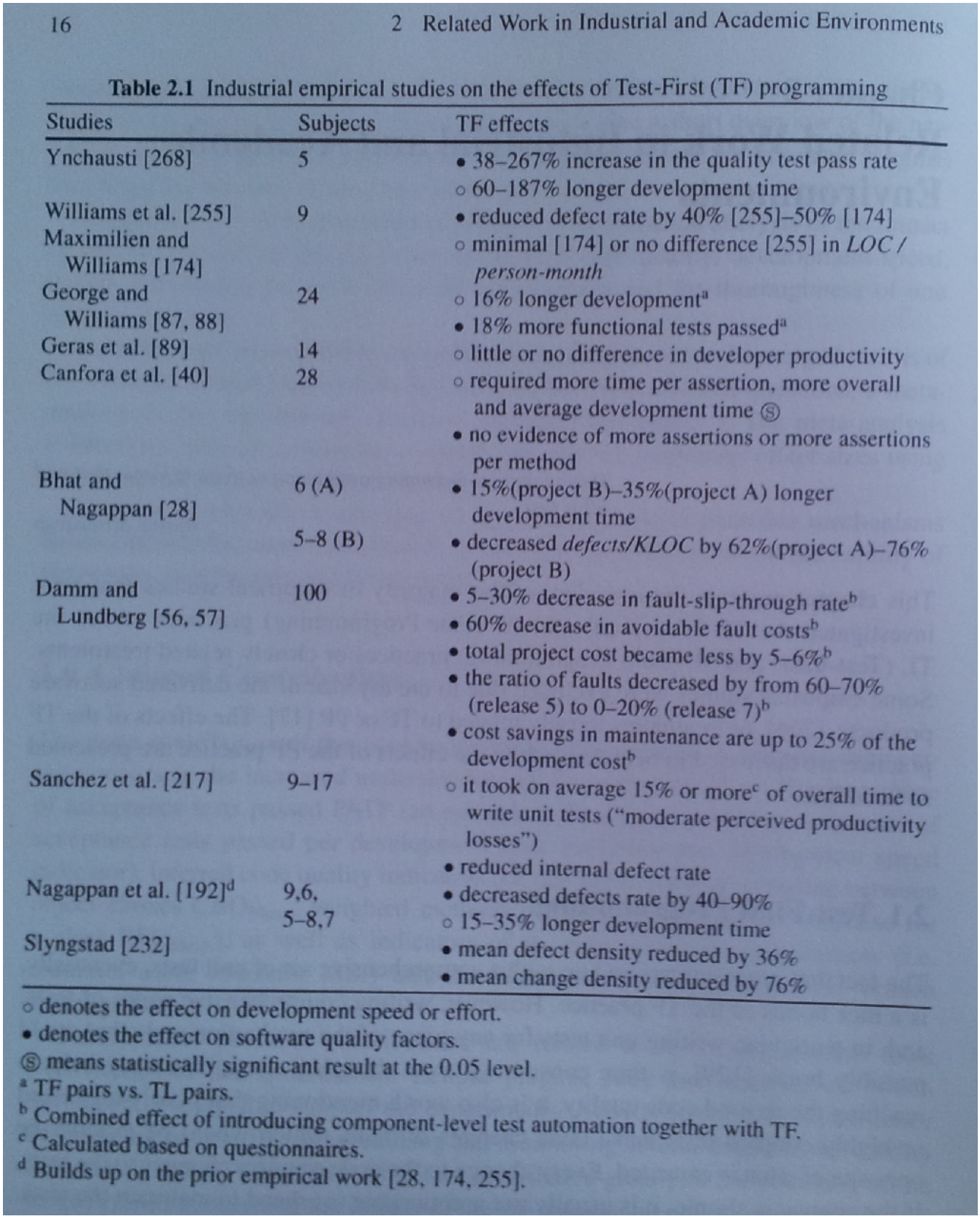
Da im Zuge dieser Arbeit nicht alle Konzepte und Werte der Agilen Softwareentwicklung umsetzbar sind, werden in den folgenden Unterkapitel die wichtigsten Prinzipien angesprochen, die Anwendung bei der Entwicklung der Software fanden. Die gewählte Entwicklungsmethode ist Test-getriebene Entwicklung (TDD). [Beck und Andres, 2004; Shore und Warden, 2007]

Weil Flexibilität und Agilität bei der Agilen Softwareentwicklung das oberste Prinzip darstellen, macht der Ansatz der Agilen Entwicklung auch nicht vor der Modifikation der eigenen Auslegung halt (siehe "Break the Rules"–Prinzip in Beck und Andres [2004]; Shore und Warden [2007]). Selbst der Agile Entwicklungsprozess ist flexibel an die Situation anzupassen. So ist das Agile Prinzip der Softwareentwicklung als allgemeine Philosophie zu verstehen, und die konkrete Umsetzung des Prinzipes wäre eine Kombination aus Kontinuierliche Integration, Simples Design und TDD (unter anderem). Das Agile Manifest ist in Abbildung 3.2 dargestellt. [Chon, 2010; Wolf und Bleek, 2010; Shore und Warden, 2007]

Um einen guten Gesamteindruck über die Agile Softwareentwicklung zu bekommen, empfiehlt es sich Beck und Andres [2004]; Shore und Warden [2007] zu lesen.

3.1.2 Kontinuierliche Integration

Kontinuierliche Integration bedeutet, dass ausgehend von einem einfachen Prototyp, zum Beispiel einer sehr rudimentären GUI, entwickelt wird. In einem Intervall von wenigen Stunden soll das Programm erfolgreich kompiliert, gestartet und dadurch neuer Code integriert werden.



16 2 Related Work in Industrial and Academic Environments

Table 2.1 Industrial empirical studies on the effects of Test-First (TF) programming

Studies	Subjects	TF effects
Ynchausti [268]	5	• 38–267% increase in the quality test pass rate ○ 60–187% longer development time
Williams et al. [255]	9	• reduced defect rate by 40% [255]–50% [174]
Maximilien and Williams [174]		○ minimal [174] or no difference [255] in *LOC / person-month*
George and Williams [87, 88]	24	○ 16% longer development[a] • 18% more functional tests passed[a]
Geras et al. [89]	14	○ little or no difference in developer productivity
Canfora et al. [40]	28	○ required more time per assertion, more overall and average development time Ⓢ • no evidence of more assertions or more assertions per method
Bhat and Nagappan [28]	6 (A)	• 15%(project B)–35%(project A) longer development time
	5–8 (B)	• decreased *defects/KLOC* by 62%(project A)–76% (project B)
Damm and Lundberg [56, 57]	100	• 5–30% decrease in fault-slip-through rate[b] • 60% decrease in avoidable fault costs[b] • total project cost became less by 5–6%[b] • the ratio of faults decreased by from 60–70% (release 5) to 0–20% (release 7)[b] • cost savings in maintenance are up to 25% of the development cost[b]
Sanchez et al. [217]	9–17	○ it took on average 15% or more[c] of overall time to write unit tests ("moderate perceived productivity losses") • reduced internal defect rate
Nagappan et al. [192][d]	9,6, 5–8,7	• decreased defects rate by 40–90% ○ 15–35% longer development time
Slyngstad [232]		• mean defect density reduced by 36% • mean change density reduced by 76%

○ denotes the effect on development speed or effort.
• denotes the effect on software quality factors.
Ⓢ means statistically significant result at the 0.05 level.
[a] TF pairs vs. TL pairs.
[b] Combined effect of introducing component-level test automation together with TF.
[c] Calculated based on questionnaires.
[d] Builds up on the prior empirical work [28, 174, 255].

Abbildung 3.1: Studien über den Einfluss von TF-Programmierung. Quelle: Madeyski [2010, S. 16–17]

Das Programm ist jederzeit lauffähig und lieferbar. Zum Testen wird zusätzlich eine Kontrast–Maschine verwendet, um sicher zu stellen, dass alle Voraussetzungen für den Code bekannt sind. Es muss immer kompilieren.

Dadurch ergibt sich der Vorteil, dass die Entwicklungsumgebung gut bekannt ist. Probleme können einfach und effektiv eingegrenzt werden. Das Eingrenzen geschieht durch die Gewissheit, dass nur die kürzlich durchgeführten Aktionen mitverantwortlich sein müssen, für die neuen Fehler. Die zweite Testmaschine, die dafür zuständig ist eine andere Umgebung zu simulieren, sorgt dafür, dass sich keine versteckten Anforderungen eingeschlichen haben. [Beck und Andres, 2004; Bender und McWherter, 2011; Shore und Warden, 2007]

3.1.3 Simples Design

Simples Design bedeutet nicht, dass auf bewährte Muster und Architekturmethoden verzichtet wird. Es bedeutet, dass sie kontinuierlich angewendet werden, wenn sich eine Situation präsentiert, die eine Anwendung eines bestimmten Designprinzips fordert. [Beck und Andres, 2004; Shore und Warden, 2007]

An dieser Stelle werden drei der wichtigsten Elemente des *Simples Design*–Prinzips aufgezählt:

- **Selbst–Dokumentierender Code:** Code soll lesbar und selbsterklärend sein.
- **YAGNI – You Aren't Gonna Need It:** Kein Code soll für spekulative Anforderungen, die in der Zukunft möglich sein könnten, geschrieben werden.
- **Einmal, und nur einmal:** Kein doppelter Code für dieselbe Aufgabe soll entstehen. Wenn die Situation auftritt, dass ein Objekt in zwei verschiedenen Versionen auftreten muss, dann ist das Objekt falsch konzipiert, und muss überarbeitet werden.

Erich Gamma, bekannt durch sein Werk Gamma et al. [1994] über Design Muster und Mitglied der "Gang of Four" (siehe: Gamma et al. [2011]), beschreibt das Prinzip des einfachen Designs in einem Gespräch mit Bill Venners (siehe: Gamma und Venners [2005]) folgender maßen:

> "[..] Bill Venners: The GoF book says, "The key to maximizing reuse lies in anticipating new requirements and changes to existing requirements, and in designing your systems so they can evolve accordingly. To design a system so that it's robust to such changes, you must consider how the system might need to change over its lifetime. A design that doesn't take change into account risks major redesign in the future." That seems contradictory to the XP philosophy.
>
> Erich Gamma: It contradicts absolutely with XP. It says you should think ahead. You should speculate. You should speculate about flexibility. Well yes, I matured too and XP reminds us that it is expensive to speculate about flexibility, so I probably wouldn't write this exactly this way anymore. To add flexibility, you really have to be able to justify it by a requirement. If you don't have a requirement up front, then I wouldn't put a hook for flexibility in my system up front.
>
> But I don't think XP and patterns are conflicting. It's how you use patterns. The XP guys have patterns in their toolbox, it's just that they refactor to the patterns

> once they need the flexibility. Whereas we said in the book ten years ago, no, you can also anticipate. You start your design and you use them there up-front. In your up-front design you use patterns, and the XP guys don't do that.
>
> Bill Venners: So what do the XP guys do first, if they don't use patterns? They just write the code?
>
> Erich Gamma: They write a test.
>
> Bill Venners: Yes, they code up the test. And then when they implement it, they just implement the code to make the test work. Then when they look back, they refactor, and maybe implement a pattern?
>
> Erich Gamma: Or when there's a new requirement. I really like flexibility that's requirement driven. That's also what we do in Eclipse. When it comes to exposing more API, we do that on demand. We expose API gradually. When clients tell us, Öh, I had to use or duplicate all these internal classes. I really don't want to do that,"when we see the need, then we say, OK, we'll make the investment of publishing this as an API, make it a commitment. So I really think about it in smaller steps, we do not want to commit to an API before its time. [..]"

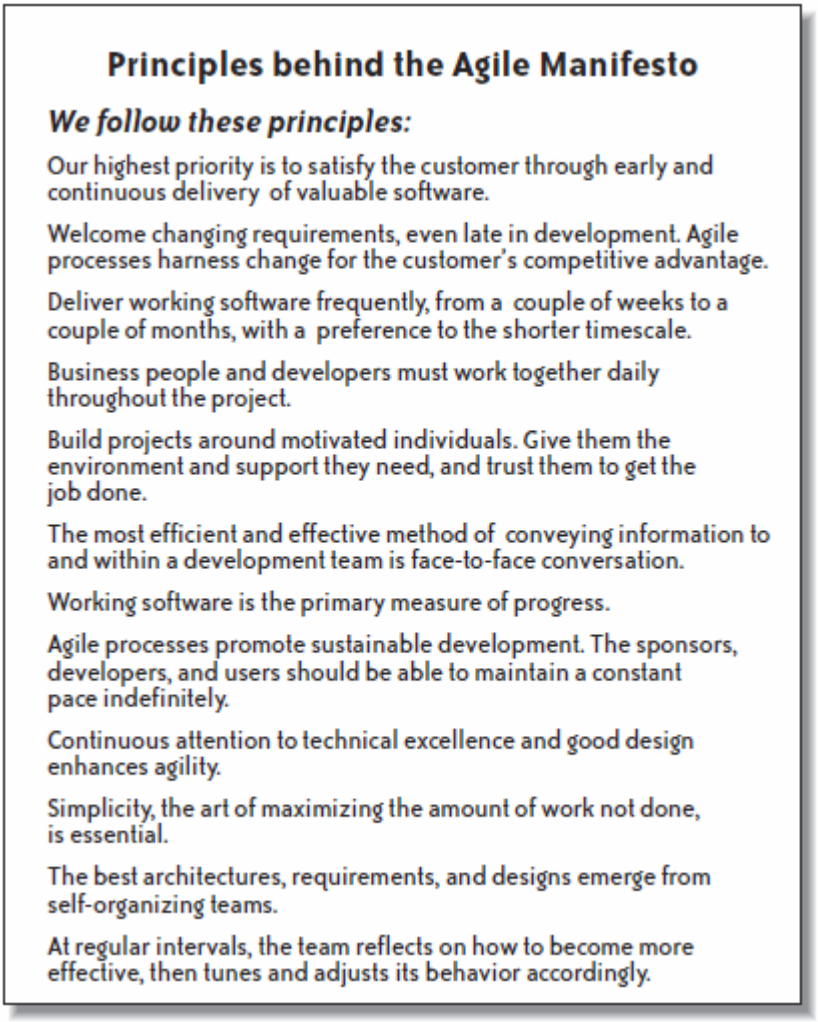
Gamma beschreibt in diesem Ausschnitt eines längeren Interviews in Gamma und Venners [2005] den größten Unterschied zwischen XP und dem traditionellen Entwicklungszyklus von Software. Traditionell wurde der Entwickler gezwungen vorausschauend zu denken, und zu Spekulieren. XP versucht die Spekulation wegzulassen, und durch eine Kombination von Agilität und Refactoring zu ersetzen. Gamma erklärt auch, dass XP–Entwickler sehr wohl auf Designmuster angewiesen sind. Sie werden aber nicht vorausschauend angewendet, denn *refactorn* soll zum benötigten Designmuster führen.

3.1.4 Inkrementelles Design und Architektur

Inkrementelles Design und Architektur ist eine große Herausforderung, da es von einem Programmierer verlangt, ein Design nur für eine kleine Aufgabe zu überlegen. Design entsteht als Evolution von kleinen Designschritten, die aus den Anforderungen von *User Stories* entstehen. Jede weitere Anforderung, die ein Designelement betrifft, ändert und erweitert das Design, und macht es abstrakter. Jede weitere Iteration – sofern richtig angewendet – sollte zu weniger, bis gar keiner Änderung am Design führen. Keine Änderungen am Design durchführen zu müssen, wenn Anforderungen wachsen, ist ein Zeichen von gutem Design. Dieser Aspekt der Agilen Softwareentwicklung verlangt also vom Programmierer, dass er das Objekt–Orientierte Denken verinnerlicht hat, und Designmuster nicht nur gut anwenden kann, sondern auch richtig versteht. Auch in ganz kleinen Schritten. [Beck und Andres, 2004; Jeffries, 2004; Wolf und Bleek, 2010; Shore und Warden, 2007]

3.2 TDD - Test Driven Development als Agile Methode

TDD ist als eigenständige Methode aus dem *eXtreme Programming* hervorgegangen. In diesem Unterkapitel folgt eine Auseinandersetzung mit den wichtigsten Schritten bei der Entwicklung durch TDD, der Methodik an sich, und den Designmustern des TDD. Weiteres wird gezeigt, wie sich die Philosophie der Agilen Entwicklung im Prozess auswirkt. Bei der Entwicklung



Principles behind the Agile Manifesto

We follow these principles:

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

Simplicity, the art of maximizing the amount of work not done, is essential.

The best architectures, requirements, and designs emerge from self-organizing teams.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Abbildung 3.2: Agile Prinzipien. Quelle: Shore und Warden [2007]; Agile Alliance [2001]

der Vilango SpeechApp wurde das TDD als Agile Methode angewendet. Beim traditionellen *eXtreme Programming* ist TDD nur ein Teil der Entwicklung (Test-First-Ansatz).

Das grundlegende Konzept ist in 3.3 dargestellt, und setzt sich aus folgenden Schritten zusammen (vergleiche: Astels [2003]; Beck [2002]):

1. Der Programmierer überlegt sich eine Anforderung
2. Es wird zuerst der Test geschrieben. Der erste Test schlägt immer fehl.
3. Wenn der Test fehlschlägt, implementiere Funktionalität. Wenn der Test erfolgreich verläuft, gehe zu (1).
4. Nach jeder Implementationsphase (Änderungsphase) führe alle Tests durch. Wähle nächsten Schritt entsprechend (3).

Dadurch wird inkrementell der Test-First Ansatz umgesetzt. Keine Anforderung oder Funktionalität wird implementiert, ohne einen Test dafür zu schreiben. Es entsteht eine verständliche und zusammenhängende Testdokumentation. [Beck, 2002; Astels, 2003] Das Ziel ist es, eine Kultur des automatisierten Testens zu etablieren. Ein wesentlicher Punkt des TDD wurde noch nicht angesprochen: Refactoring. Es ist ein essentieller Bestandteil der Test-getriebenen Entwicklung und wird im nächsten Schritt genauer beschrieben. Im weiteren Verlauf dieser Arbeit werden sowohl Begriffe wie “User Story”, “Unit Test”, und andere dem Testen verwandte Fachbegriffe, als auch grundlegende Vertrautheit mit Agiler Softwareentwicklung, vorausgesetzt.

3.2.1 Wichtige Charakteristika des TDD

Es ist wichtig zu beachten, dass der Softwareentwicklungsprozess und der Test-getriebene Prozess, nicht gleichzusetzen sind. Viel mehr ist die TDD Methode im gesamten Softwareentwicklungsprozess einzuordnen. Planung, Design und Anforderungsanalyse werden trotzdem durchgeführt, wenngleich dies entsprechend den Prinzipien der Agilen Entwicklung möglichst einfach gehalten wird. Die im Zuge des TDD entwickelten und geplanten Tests ersetzen nicht den Bedarf nach weiteren gezielten Tests. Die entwickelten Tests sollen derart einfach und leicht lesbar sein, so dass sie als Ersatz für eine umfangreiche Dokumentation dienen können. [Beck, 2002; Astels, 2003; Jeffries, 2004; Bender und McWherter, 2011]

Ausgehend von Abbildung 3.3 wird das Konzept nun in mehr Detail dargestellt. TDD besteht aus drei Phasen: Der *Roten Phase* (Test schlägt fehl), der *Grünen Phase* (alle Test gelingen) und der *Refactoring Phase*. Das Hauptaugenmerk liegt darin, von der roten Phase in die grüne Phase zu kommen. Dies kann durch verschiedene Techniken erreicht werden, unter anderem sind dies folgende:

1. Implementieren
2. Fake it till you make it
3. Triangulation

[Beck, 2002; Astels, 2003; Madeyski, 2010]

Eine genauere Betrachtung der möglichen Instrumente wird in den folgenden Unterkapitel vorgenommen. Zuerst wird der Prozess von Beginn bis zum Ende vorgestellt.

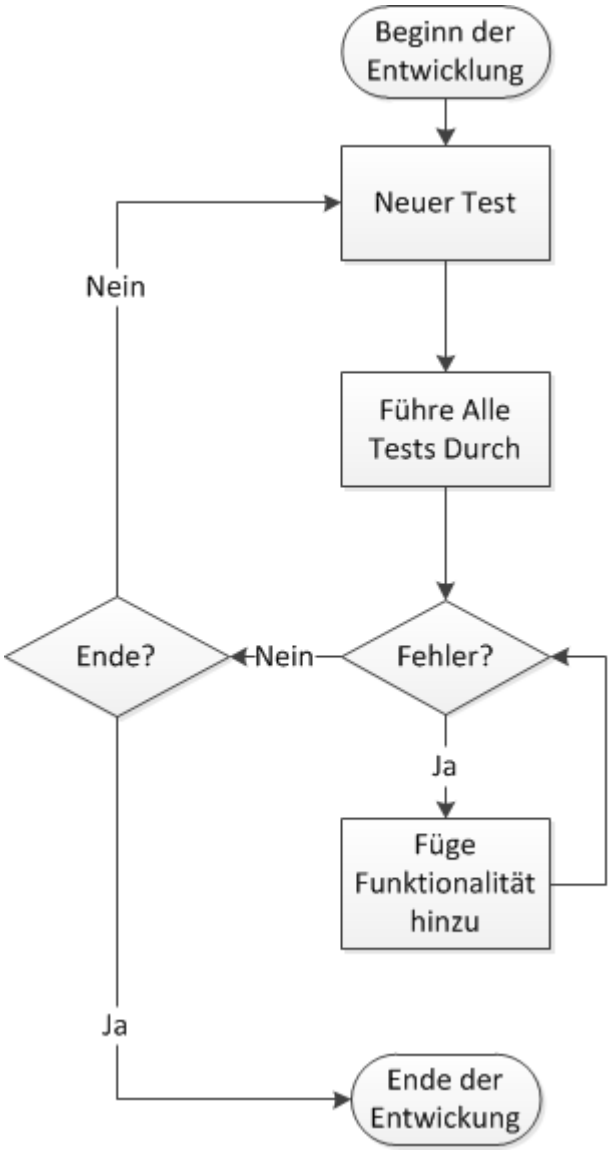


Abbildung 3.3: Einfacher Zyklus des TDD.

3.2.2 Den Anfang macht die Testliste

Wie startet man den Prozess der Test–Getriebenen–Entwicklung? Diese Frage ist auf den ersten Blick offensichtlich einfach zu beantworten: Mit einem Test. Aber welchen Test? In diesem Fall empfiehlt Beck [2002] erst eine Liste von Tests zu schreiben, von denen der Entwickler weiß, dass sie benötigen werden. Das setzt voraus, dass die Programmierer sich vorab mit den grundlegenden Anforderungen des Programmes auseinandersetzen. Dadurch erfüllt TDD eine ganz wichtige Eigenschaft der Agilen Entwicklung: inkrementelles, minimalistisches Design, das mit dem Programm wächst, und sich in den Refactoring–Phasen agil ändern lässt. Dabei können natürlich auch Tests wegfallen, wenn sich durch Vereinfachen, oder dem KISS Prinzip, Code reduzieren lässt.(“KISS – Keep it simple stupid”; Deutsch: “Halte es einfach, Dummkopf!”) [Beck, 2002; Shore und Warden, 2007; Chon, 2010].

Diese Liste von Tests begleitet den Programmierer ständig durch die Entwicklungszeit. Alle Änderungen und Probleme werden in Form von notwendigen Tests in dieser Liste eingetragen.

Im konkreten Fall der Entwicklung der Vilango SpeechApp wurde folgender Ansatz gewählt:

1. Erstellen einer Liste von funktionalen-, Benutzer- und nicht-funktionalen Anforderungen. Dieser Schritt bezieht den Kunden mit ein.
2. Erstellen eines erstes Flow-Chart Diagrammes aufgrund dieser Anforderungsliste, welches die ersten essentiellen Anforderungen umsetzt.
3. Konzipieren von Teil–Flow-Charts, User-Stories und Tests zu jedem Schritt des Haupt–Flow–Chart. Hinzufügen der Tests zur Testliste.
4. Verfeinern des Flow-Chart und der Anforderungen.

Tabelle 3.1: Arbeitsschritte bei der Entwicklung der Vilango SpeechApp.

Durch die Wahl dieses Ansatzes, der in Bender und McWherter [2011] demonstriert wird, ist das Problem der nötigen Schrittlänge bei der Wahl der Tests auch gelöst. Die Schrittlänge definiert, wie viel “Strecke” jeder Test im optimalen Fall abdecken sollte. Die Frage zum Thema der optimalen Schrittlänge, wie sie in Beck [2002] gestellt wird, kann von der Literatur nicht einheitlich beantwortet werden. Zum Zeitpunkt dieser Masterarbeit konnten keine empirischen Untersuchungen zu diesem Thema gefunden werden.

Nachdem diese Phase abgeschlossen ist, folgt der Rot–Grün–Refactor Rhythmus. Dies ist der Vorgang, den Entwickler vornehmen, wenn sie Test-getrieben implementieren. Abbildung 3.4 zeigt den gesamte Ablauf [Bender und McWherter, 2011; Beck, 2002; Astels, 2003; Madeyski, 2010].

3.2.3 Rote Phase

Diese Phase ist jene Phase, welche für die meisten Entwickler sehr gewöhnungsbedürftig ist. Es stellt sich die Frage: Wie wird ein Test für Code, der noch gar nicht existiert, geschrieben?

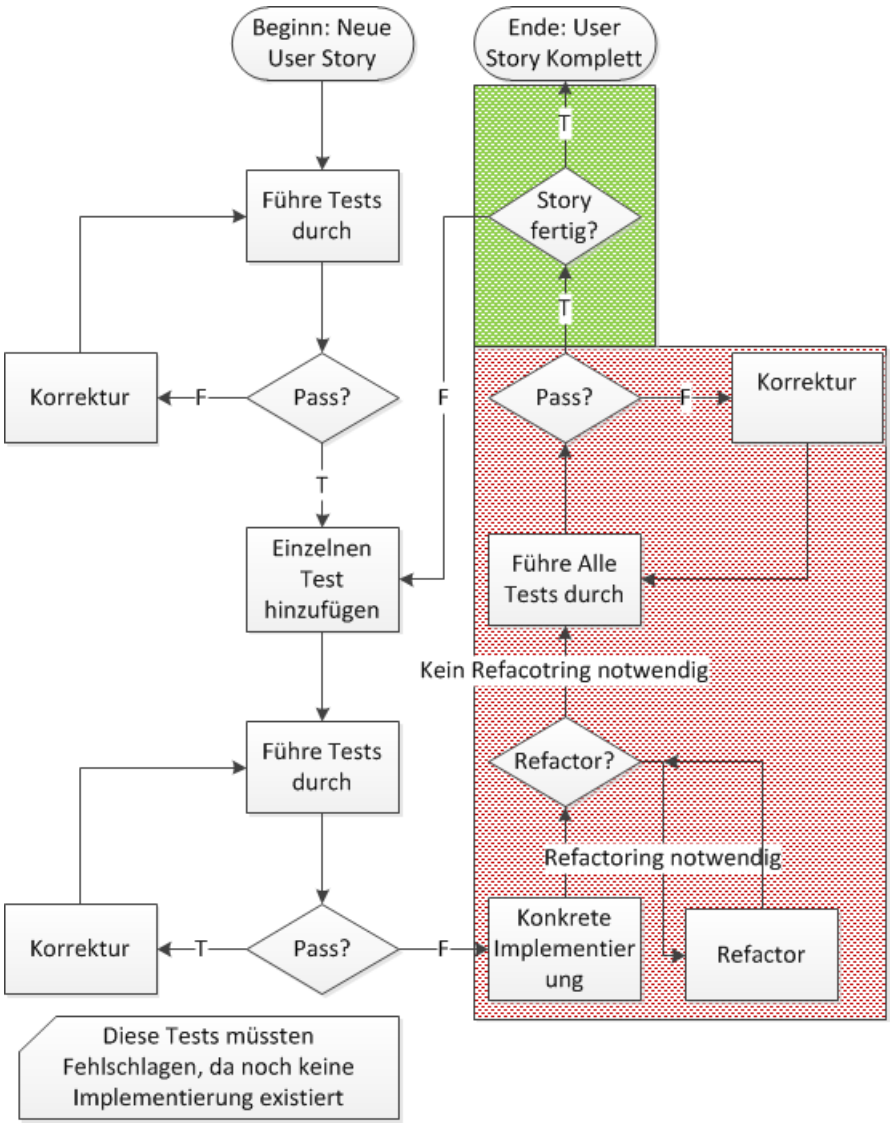


Abbildung 3.4: Detaillierter Entwicklungszyklus des TDD. Vergleiche Madeyski [2010]

Das bedeutet natürlich, dass der Test scheitern wird. Moderne Entwicklungsumgebungen bieten standardmäßige Unterstützung für Testumgebungen, die eine automatisierte Testausführung erlauben, an. NUnit, VisualStudio–TestUnit und JUnit sind Beispiele, unter anderen, für solche Werkzeuge. Dabei hat sich Rot als Signal für einen scheiternden Test durchgesetzt. [Beck, 2002; Astels, 2003; Bender und McWherter, 2011; Madeyski, 2010]

Die Kunst besteht darin schrittweise Funktionalität durch Tests zu definieren, und durch die Implementierung der Methoden in die grüne Phase zu überführen. (Siehe Abbildung 3.4) Jede Entwicklung durch TDD beginnt in der roten Phase. In dieser Phase werden die "Red Bar Patterns" angewendet. Sie beschäftigen sich mit dem richtigen Zeitpunkt Tests zu schreiben, dem Ort an dem Tests zu schreiben sind, und wann genug Tests geschrieben wurden [Beck, 2002].

Jede Methode, Interface, oder Operation wird durch einen Test vordefiniert, der scheitert. Jedem inkrementellen Hinzufügen von weiteren Anforderungen an die Elemente geht ein neuer Test voraus [Bender und McWherter, 2011; Beck, 2002].

Die verschiedenen Muster werden in Beck [2002] beschrieben. Dieses Werk wird in der Beispielsektion auf ausgewählte, bei der Entwicklung von Vilango SpeechApp verwendete, Muster eingehen.

3.2.4 Grüne Phase

Jeder Test der fehlschlägt weist auf einen Mangel in der Implementierung hin. In dieser Phase werden die "Green Bar Patterns" angewendet. Die Herausforderung ist, die Rote Phase zu überbrücken, alle Anforderungen zu erfüllen, und trotzdem in der Grünen Phase zu bleiben. [Bender und McWherter, 2011; Beck, 2002]

Eine detaillierte und ausführlichere Vorstellung der "Green Bar Patterns" findet sich in Beck [2002]. Die wichtigsten Muster sind folgend definiert:

Obvious Implementation wird zum implementieren einfacher Operationen benutzt.

Fake it till you make it ist ein Vorgang, bei dem der Entwickler inkrementell, Implementationsdetails weglässt, und vorgibt, dass sie existieren. Es wird eine Konstante benutzt (oder andere äquivalente Techniken), die vorgaukelt (Engl.: "fakes it"), dass der Fehler nicht mehr existiert. Die Konstante wird graduell durch funktionierenden Code ersetzt, und es wird darauf geachtet, dass die laufenden Tests "grün" bleiben.

Triangulierung bietet sich an, wenn die konkrete Implementierung von der Abstraktion nicht klar ist. Bei diesem Vorgang wird durch das Hinzufügen von Assertions eine schrittweise Annäherung an die richtige Umsetzung durchgeführt [Beck, 2002].

3.2.5 Refactoring – Gründe für Refactoring

Dieses Unterkapitel setzt das Verständnis der Techniken des Refactoring voraus, und behandelt Refactoring im Kontext des TDD als Methodik. Refactoring ist nicht nur beim TDD ein essentieller Schritt, sondern ein wichtiges Instrument der Softwareentwicklung. Es gibt verschiedene Gründe für Refactoring in TDD, die in der Literatur als "Bad Smells" betitelt werden [Astels, 2003; Beck, 2002; Bender und McWherter, 2011; Fowler et al., 1999].

Refactoring allgemein kann unterschiedlich definiert werden:

(2) Nach Astels [2003] ist Refactoring ein Prozess, der ändert *wie* ein Prozess funktioniert, nicht *was* er macht, mit dem Ziel die innere Struktur zu verbessern.

(3) Nach Fowler et al. [1999] ist Refactoring ein Prozess, der die innere Struktur verändert, sodass die Software leichter zu verstehen ist, und es billiger wird sie zu modifizieren.

Wenn von Refacotring im Zusammenhang mit TDD gesprochen wird, muss zwischen dem Refactoring von der Testsuite, und dem Refactoring von der Software unterschieden werden. Beides folgt den gleichen Prinzipien beschrieben in (2) und (3). Allerdings gibt es zwischen dem Refactoring der Software und der Testsuite keine lose Kopplung. Änderungen in der Software sollen keine Änderung der Tests nach sich ziehen. Dieser Umstand ist Teil der Designentwicklung durch den Test-First Ansatz, da die Tests die Rahmenbedingungen des Designs festlegen.

Bezogen auf das TDD bedeutet Refactoring auch eine Aufräumarbeit. Durch die kleinst möglichen Schritte bei der Entwicklung der Software, entsteht redundanter Code, der das Prinzip der Lesbarkeit verletzt. Je detaillierter eine User Story implementiert wird, desto mehr Tests entstehen. Viele davon werden später nicht mehr benötigt, und werden zu Ballast, weil das Durchführen der Testliste immer mehr Zeit in Anspruch nimmt.

Verbesserung des Designs

Wie Abbildung 3.4 vermuten lässt, entsteht beim Test-Driven-Development über den Verlauf der Entwicklungszyklen sehr viel Code. Die Struktur droht zwangsläufig unüberschaubar zu werden. Nach Beck [2002]; Fowler et al. [1999] sorgt regelmäßiges Refactoring dafür, dass die Struktur und Lesbarkeit der Software erhalten bleiben.

Refactoring reduziert Codezeilen. Reduktion von Codezeilen führt nicht notwendigerweise zu einer besseren Performance der Software. Reduktion von Codezeilen hat aber einen großen Einfluss, wenn versucht wird den Code zu modifizieren. Weniger Codezeilen sind leichter zu verändern, und Fehler sind leichter zu vermeiden. Weniger Codezeilen kann die Fehlerfindung erleichtern [Fowler et al., 1999].

Lesbarkeit

Entwickler neigen dazu zu vergessen, dass nach der Entwicklung vor der Entwicklung ist. Änderungen, Erweiterungen der Software und Fehler, die erst nach Monaten der Verwendung auftreten, sind ständige Begleiter von Softwareprodukten [Fowler et al., 1999].

Die Programmierer, die versuchen werden Änderungen vorzunehmen, sind dankbar, wenn sie einen leserlichen und leicht verständlichen Code vorfinden. Es wird der nötige Zeitaufwand reduziert, und Änderungen werden besser umgesetzt. Nach Fowler et al. [1999] muss der Code zum Programmierer sprechen. Es entsteht eine Art Unterhaltung. Fragen wie: “Wie funktioniert der Input?”, sollen durch Statements, in quasi-natürlicher Sprache beantwortet werden. Codebeispiel 3.1 zeigt wie das Aussehen kann.

```
1 private void Open_Click(object sender, RoutedEventArgs e)
2         {
3             file_ = new OpenFileDialog();
4             file_.Filter = Constants.WaveFileFilter;
5
```

```
6                 if (file_.ShowDialog() == true)
7                 {
8                     String input_ = InputBox(TextToTranscribePrompt, ""
                          , "", 100, 100);
9                     TranscriptionEngine.handleInput(file_.FileName,
                          input_);
10                }
11            }
```

Listing 3.1: Codebeispiel in leicht verständlicher, quasi-natürlicher Sprache.

Auffinden von Fehlern

Fowler et al. [1999] sagt, dass verständlicher Code auch Fehler verständlicher macht. In anderen Worten gefasst: Wenn der Code verständlich zu lesen ist, dann ist der Fehler auch leicht, verständlich und natürlich zu verstehen. Verständliche Struktur sorgt dafür, dass gemachte Fehler aus der Struktur verständlich hervorgehen.

3.2.6 Refactoring – Wann anwenden? Die “Bad Smells in Code”

Als *Bad Smells in Code* (Englisch für: *Der Code stinkt.*) werden Codeabschnitte bezeichnet, die für Verwirrung sorgen.[Astels, 2003] Nach Fowler et al. [1999] gibt es keine festen Kriterien, wann Refactoring nötig ist. Der Bedarf Änderungen vorzunehmen ergibt sich durch den Gestank, der vom Code ausgeht. Die Sprache in diesem Abschnitt ist bewusst gewählt. Es existiert eine Anekdote, die erklärt wie es zu dem Begriff *stinkender Code* kommt. Kent Beck hatte zum Zeitpunkt der Diskussion über Refactoring mit Fowler eine neugeborene Tochter, und war wohl inspiriert durch regelmäßiges Windeln wechseln, denn um es in seine Worte zu fassen: “Wenn der Code stinkt, wechsle ihn.”[Fowler et al., 1999].

Nachfolgende Aufzählung zeigt die *Bad Smells*, die im Originalwerk von Fowler et al. [1999] angeführt werden.

1. Doppelter Code: Gleicher Ausdruck in zwei Methoden der gleichen Klasse oder verwandten Klassen.
2. Überlange Methoden.
3. Überlange Klassen: Klassen, die viele Instanzvariablen besitzen, sind häufig ein Zeichen von nötigem Refactoring.
4. Lange Parameterliste: Methoden die zu viele Parameter annehmen, sind häufig überlange Methoden.
5. Divergente Änderung: Software soll derart strukturiert werden, dass ein Änderung immer einen offensichtlichen Ansatzpunkt bietet. Fehlt dieser, ist Refactoring nötig.
6. Schrotflinten Operation: Das Gegenteil von “Divergente Änderung”. Im Gegensatz zu “Divergente Änderung”, bei der ein klarer Ansatzpunkt fehlt, gibt es bei “Schrotflinten Operation” zu viele Ansatzpunkte.

7. Feature Eifersucht: Dieser Umstand ist zu beobachten, wenn eine Methode in einer Klasse fehl am Platz scheint, da sie mehr Interesse an den Eigenschaften einer anderen Klasse zeigt, als an den eigenen.
8. Datenklumpen: Parameter die häufig miteinander auftreten, sollten in eine eigene Klasse extrahiert werden.
9. Obsession mit elementaren Datentypen: Werte und Einheiten werden nicht in Miniklassen definiert, sondern werden immer mit Strings und Integer–Datentypen ausgedrückt.
10. Switch-Statements: Switch-Statements sollten mit Polymorphismus ergänzt werden.
11. Parallele Vererbungshierarchie: Dieser Fall äußert sich dadurch, dass bei jeder neuen Vererbung von einer Klasse A, auch eine Vererbung einer anderen Klasse B erstellt werden muss.
12. Faule Klasse: Klassen die zu wenig leisten um ihre Existenz zu rechtfertigen.
13. Spekulative Allgemeingültigkeit: Dieser Fall bezeichnet die nicht notwendige Implementierung von spekulativen, zukünftigen Anforderungen. Vergleiche YAGNI-Prinzip des XP in Jeffries [2004]; YAGNI [2006]. (YAGNI – "you ain't gonna need it"; Deutsch: " Du wirst es nicht brauchen")
14. Temporäre Klassenfelder.
15. Nachrichtenketten: Eine Klasse A fragt bei Klasse B für Klasse C nach. A sollte direkt C ansprechen können.
16. Mittelsmann: Anfragen von Klasse A nach Klasse C werden über Methoden der Klasse B durchgeführt.
17. Unangemessene Intimität: Klassen haben direkten Zugriff zu internen Variablen.
18. Alternative Klassen mit verschiedenen Interfaces: Methoden sollen nicht doppelt in unterschiedlichen Klassen dieselbe Arbeit verrichten.
19. Mangelhafte Bibliothek: Wiederverwenden von Code in verschiedenen Klassen soll vermieden werden (siehe: Agiles *write once* Prinzip).
20. Datenklasse: Klassen, die lediglich Felder setzen und ausgeben, sind unnötig, und sollten in Datenstrukturen ausgelagert werden.
21. Kommentare: Ein wichtiger Indikator für mangelnde Lesbarkeit sind lange Kommentare, die eine Methode ausführlich beschreiben.
22. Abgelehntes Erbe: Vererbung ist fehl am Platz, wenn ein Großteil der vererbten Methoden nicht benutzt wird. Delegieren ist in diesem Fall vorzuziehen.

3.2.7 Fortgeschrittene Test-getriebene Techniken

Da eine ausführliche Auseinandersetzung aller fortschrittlicher Techniken den Rahmen dieser Arbeit übersteigen würde, beschränkt sich die Auswahl auf die wichtigsten Muster: *Mock Object*, *Child Test*, *Ultra-Thin GUI* und *Self Shunt* [Beck, 2002; Astels, 2003].

Mock Object

Mock Objects (engl. “Mock” für Attrappe) ist die Definition einer Attrappe, die alle Eigenschaften eines realen Objektes nachahmt. Beim Testen entstehen Situationen, die es erfordern, komplexe Systeme, die im Hintergrund einer Applikation stehen, miteinzubeziehen. Ein klassisches Beispiel ist eine Datenbank. Da Tests schnell und effektiv sein sollen, und aus Gründen der Sicherheit, kann nicht immer auf diese Objekte zugegriffen werden. Aus diesem Grund werden Attrappen kreiert, die sich innerhalb der definierten Parameter dem Originalobjekt äquivalent verhalten. Dadurch ist es möglich auch variable Komponenten als Teil der Tests einfließen zu lassen [Beck, 2002; Astels, 2003].

Child Test

In Beck [2002] äußert sich der Autor oft über die, seiner Meinung nach, große Bedeutung des “metaphysischen Wohlbefindens” (siehe Beck [2002]). Dieses Wohlbefinden resultiert aus dem regelmäßigen Verweilen in der “Grünen Phase”, denn das bedeutet, dass der Entwickler auf dem richtigen Weg ist. Tests können aber in bestimmten Fällen, wenn viel Funktionalität auf einmal getestet werden soll, ausarten und zu groß werden. Dadurch wird es schwieriger diese richtig umzusetzen. Die vorgeschlagene Lösung in Beck [2002] ist, einen kleineren Test zu definieren, der einen Teil des gesamten Tests repräsentiert, und aus dem ursprünglichen Test auszulagern.

Self Shunt

Ein häufiges Anliegen der Softwareentwicklung ist die richtige Kommunikation zwischen Objekten zu gewährleisten. Um diese Funktion der Software zu Testen genügt es das Objekt, welches kommunizieren soll, mit einem Testfall kommunizieren zu lassen. Dadurch müssen allfällige Fehler im konkreten Objekt nicht mit getestet werde, und es entsteht die Gewissheit, dass die Kommunikation innerhalb definierter Parameter richtig ist [Beck, 2002].

Ultra-Thin Gui

GUIs (“Graphical User Interface” zu Deutsch: “Grafische Benutzer Oberfläche”) Test-getrieben zu entwickeln ist eine große Herausforderung, und es gibt viel Diskussion, ob es überhaupt machbar ist. [Astels, 2003; Beck, 2002; Jeffries, 2004; Bender und McWherter, 2011] Ein Ansatz wird in Astels [2003] vorgestellt, der sich aus der Definition von gutem GUI-Design ableitet. Gutes GUI-Design soll lediglich Aufrufe in das Modell, und das Setzen der Konfiguration, ermöglichen. Dadurch ergibt sich im optimalen Fall keine Logik innerhalb der GUI, die getestet werden muss. Das definierte Ziel dieser Methode ist ein Attrappen-GUI zu erstellen, welches die definierten Funktionsaufrufe ins Modell durchführt. Der Unterschied besteht in der Handhabung der Tests. Beim herkömmlichen *Mock Object* soll das Objekt eine Hilfestellung beim Testen sein. In diesem Fall wird die GUI-Attrappe getestet. Das bedeutet, dass nicht die Funktionsaufrufe, sondern die Antworten auf die definierte Attrappe und der Umgang der Attrappe mit den Antworten getestet wird. Dieser Test hilft die Zustände zu überprüfen, und macht einen manuellen Test der GUI nicht obsolet [Astels, 2003].

3.2.8 Die Test-getriebene Dokumentation

Test-getriebene Dokumentation ist eine von Anfängern und Kritikern oft missverstandene Eigenschaft. Leichtgewichtige Dokumentation von Design und agile Entwicklung bedeuten nicht, dass auf traditionelle Dokumentation verzichtet wird [Beck und Andres, 2004].

Test-getriebene Dokumentation ist die primäre Art mit TDD entwickelte Software zu dokumentieren, aber nicht die einzige. Flow-Charts, Anforderungsanalysen und UML-Diagramme sind ergänzend.

Die zwei Haupteigenschaften der Test-getriebenen Dokumentation sind die Testsuite und der Code selbst. In der Testsuite sollen die Testklassen und die Tests in natürlicher Sprache verständlich sein. Der Testcode muss den Test sinngemäß durch seine Anweisungen leserlich wiedergeben [Beck, 2002; Astels, 2003; Bender und McWherter, 2011; Jeffries, 2004].

Abbildung 3.5 zeigt wie die Namensgebung bei der Testsuite der Vilango SpeechApp aussieht. *TestTranscriptionControlClass* stellt den Namen der getesteten Komponente dar. “Test Transcription Control Class” (Deutsch: “Teste Transkriptions Kontroll Klasse”) und die dazugehörigen Tests werden durch ihren Testnamen bereits mit Semantik versehen. “Test Reconition Engine Setup” wird als Befehl an die Testsuite aufgefasst den Aufbau der Erkennungskomponente zu testen. Damit ist der erste Teil der Test-getriebenen Dokumentation abgeschlossen. Der zweite Teil ist im Code des Tests selbst. Beispiel 3.2 zeigt den Testcode, der in natürliche Sprache übersetzt werden kann.

```
1          [Test]
2          public void TestRecognitionEngineSetup()
3          {
4              EngineCoreComponent RecognitionTestEngine = new
                   EngineCoreComponent();
5              RecognitionTestEngine.SetTextInput(valid_test_input_);
6              RecognitionTestEngine.SetWavInput(valid_file_input_);
7
8              bool success = RecognitionTestEngine.
                   InitializeInProccessRecognitionEngineAtStreamPosition
                   (0);
9              Assert.IsTrue(success);
10         }
```

Listing 3.2: Codebeispiel für TestRecognitionEngineSetup-Test.

Das ausgewiesene Ziel, das Design der Software durch Tests und ihre Implementierung auszudrücken, wird erreicht, indem jede definierte Anforderung in den Tests wiederzufinden ist. Dadurch wird sichergestellt, dass die Software alle Anforderungen erfüllt, und dass die Dokumentation in der Lage ist, alle Fragen, die ein fremder Entwickler hat, zu beantworten. Zusätzlich wird eine sehr hohe Testabdeckung erreicht [Beck, 2002; Astels, 2003].

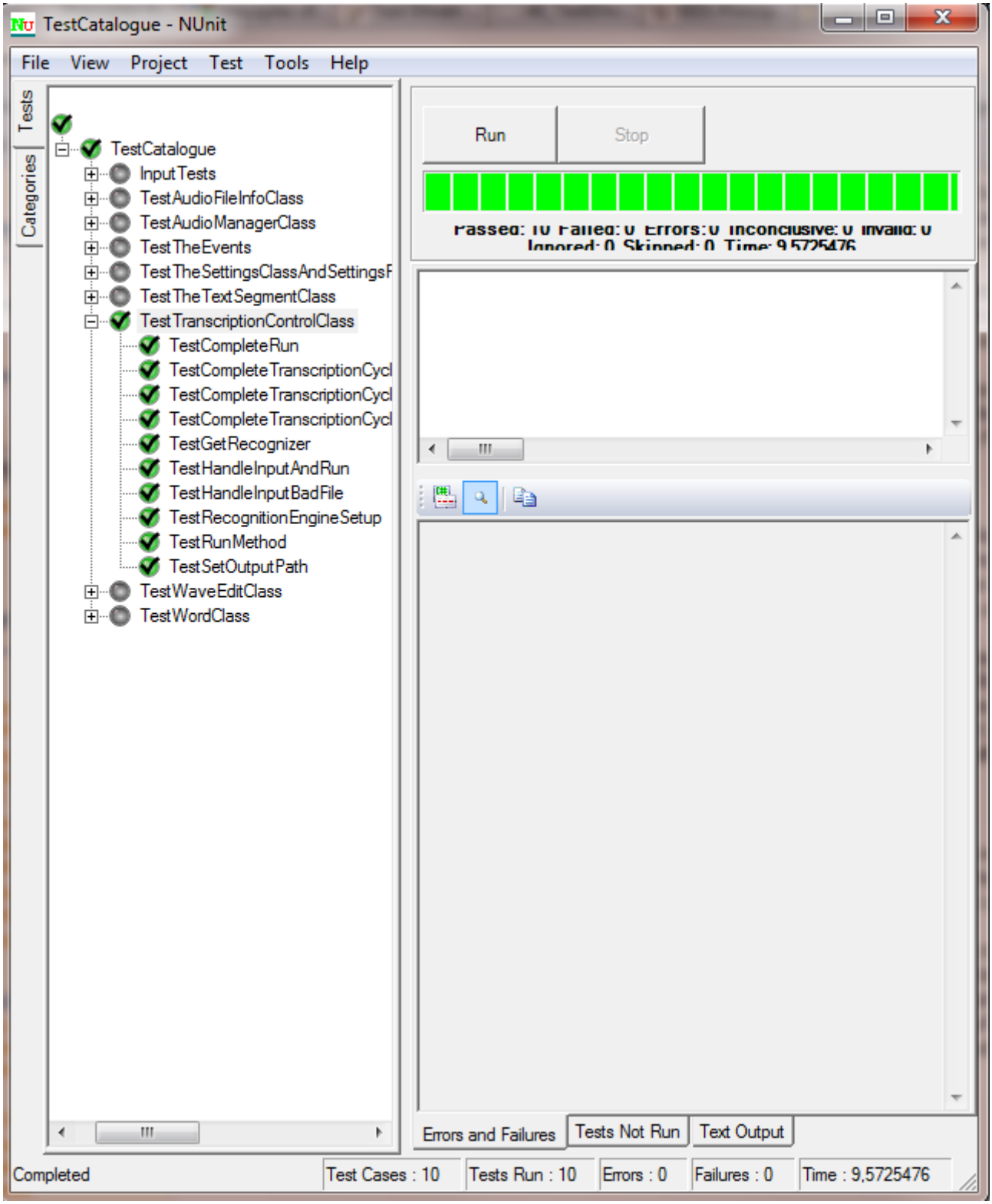


Abbildung 3.5: Testkatalog präsentiert in NUnit.org [2009]. Links: Testklassen, Zweig mit Tests.

3.3 Der Test-getriebene Prozess – Anwendung bei der Vilango SpeechApp

Folgendes Unterkapitel dokumentiert ausgewählte Beispiele aus der Entwicklung der Vilango SpeechApp. Es zeigt auf, wie Refactoring, Rote und Grüne Phasen, und die Test-getriebene Dokumentation praktisch umgesetzt sind. Zusätzlich werden zum Refactoring konkrete Beispiele für *Code Smells* angeführt, die das Refactoring beseitigt hat.

3.3.1 Funktionale Anforderung

Die Anforderungsanalyse stellt den ersten Schritt beim Entwickeln neuer Methoden dar. Es stellt sich die Frage: *Was möchte ich erreichen?*

Für diesen Fall wird ein, auf den ersten Blick, einfaches Beispiel als funktionale Anforderung gewählt: *Das Programm muss nach dem Erkennen den erkannten Text in der richtigen Reihenfolge vom Anfang entfernen.* Es wird angenommen, dass die Erkennung von Links nach Rechts geschieht, da zwei gleiche Wörter nacheinander nicht unterschieden werden können.

Mathematisch gilt folgende Beziehung:

$$Text = TextAnfang \backslash ErkannterText$$

Der Entwickler erstellt die erste Testliste aufgrund der formulierten Anforderung:

1. Text = Text ohne erkannten Text vom Anfang.
2. Text = Text, wenn erkannter Text leer ist.

3.3.2 Rot–Grün–Refactor Beispiel

Im nächsten Schritt wird ein Test erstellt. Beispiel 3.3 zeigt die Implementierung.

```
1         [Test]
2         public void TestGetRemainingTextAfterRecognition()
3         {
4             Assert.IsTrue(false);
5         }
```

Listing 3.3: Beispiel: Erster Test.

Der Test scheitert, denn es existiert weder die Methode, noch kann *Assert.IsTrue(false)* je erfolgreich abschließen. Der nächste Schritt besteht aus einer Implementierungsphase der konkreten Methode, und anschließendem Anpassen des Tests. Das nächste Beispiel 3.4 zeigt den Schritt.

```
1       public String getNextIterationInput()
2         {
3             string new_input_text = "";
4             return new_input_text;
```

```
5          }
6
7          [Test]
8          public void TestGetRemainingTextAfterRecognition()
9          {
10             Assert.IsEmpty(TestAudioManagerClass.
                   getNextIterationInput());
11         }
```

Listing 3.4: Beispiel: Erster Code.

Dieser Test gelingt. Nach einer neuen Überlegung wird festgestellt, dass keine Methode existiert, die erkannten Text zeigt. Es wird nun das *Fake it 'til you make it* Muster des TDD benutzt. Es wird die Testliste verfeinert, und die Methode angepasst. Bevor das Beispiel 3.4 erläutert wird, folgt zuerst eine kurze Erklärung. Es wird vorausgesetzt, dass eine funktionierende Klasse *TextSegment* existiert, die im Laufe des Refactoring und der Verfeinerung benutzt wird. Die neue Testliste sieht folgendermaßen aus:

1. Text = Text ohne erkannten Text vom Anfang.
2. Text = Text, wenn erkannter Text leer ist.
3. "holidays Time to relax leave work behind and enjoy" = "Finally my well deserved holidays Time to relax leave work behind and enjoy" – "Finally my well deserved".

```
1        public String getNextIterationInput()
2          {
3              string new_input_text = "";
4              string entire_transcription_text = "Finally my well
                   deserved holidays Time to relax leave work behind
                   and enjoy";
5              string recognized_text = "Finally my well deserved";
6              return "holidays Time to relax leave work behind and
                   enjoy";
7          }
8
9          [Test]
10         public void TestGetRemainingTextAfterRecognition()
11         {
12             Assert.IsEqual(TestAudioManagerClass.
                   getNextIterationInput(), "holidays Time to relax
                   leave work behind and enjoy");
13         }
```

Listing 3.5: Beispiel: Verfeinerung mit *Fake it 'til you make it* Muster.

Nach diesem Schritt entsteht eine neue Anforderung: Das Programm benötigt einen Algorithmus, der einen guten Text aus der Menge des erkannten Textes wählt. In diesem Fall wird dem Programm vorgegaukelt, dass ein solcher Algorithmus existiert. Der Test ist in dieser Form nicht zufriedenstellend, da noch Tests auf der Testliste nicht abgedeckt werden. Deswegen wird er noch einmal verfeinert.

```
1       public String getNextIterationInput(string
           entire_transcription_text, string recognized_text)
2         {
3             TextSegment entire_transcription_text_ = new
                 TextSegment(entire_transcription_text);
4             TextSegment recognized_text_ = new TextSegment(
                 recognized_text);
5             int i = 0;
6             if(recognized_text.getCount() <=
                 entire_transcription_text_){
7                 foreach(Word word_ in recognized_text){
8                       if(entire_transcription_text_.getWord(i) ==
                           recognized_text(i))
9                           entire_transcription_text_.remove(i);
10                      i = i+1;
11                }
12            }
13            return entire_transcription_text_.ToString();
14        }
15
16        [Test]
17        public void TestGetRemainingTextAfterRecognition()
18        {
19            String entire_text_input = "Finally my well deserved
                 holidays Time to relax leave work behind and enjoy";
20            String recognized_text_input = "Finally my well
                 deserved";
21            Assert.IsEqual(TestAudioManagerClass.
                 getNextIterationInput(entire_text_input,
                 recognized_text_input), "holidays Time to relax
                 leave work behind and enjoy");
22        }
```

Listing 3.6: Beispiel: Zweite Verfeinerung mit *Fake it 'til you make it* Muster.

Der Test gelingt abermals und der Balken bleibt grün. Die Testliste wird erweitert, da vor der nächsten Iteration des Erkenners, der zu erkennende Text mit dem verbliebenen Teil des Textes ersetzt werden muss. Von der Testliste können weitere Tests gestrichen werden. Die Methode wird zusehends schwerer zu verstehen und zu lesen, außerdem weist sie den Entwickler auf einen Mangel in *TextSegment* hin. Es liegt Nahe, dass *TextSegment* eine Methode benötigt, die es ermöglicht einen Text von einem gegebenen Index bis zu einer gegeben Länge zu entfernen. Es folgt ein Refactoring–Zyklus mit folgenden Schritten:

1. Hinzufügen einer Methode *RemoveRange(int,int)*.

2. Refactoring von *getNextIterationInput()*, sodass die Änderung von TextSegment umgesetzt wird.

3. Der Test bleibt gleich.

Abbildung 3.4 zeigt, dass der optimale Vorgang wieder in der Grünen Phase endet. Das Beispiel illustriert das psychologisch perfekte Szenario beschrieben in Beck [2002], da die Grüne Phase nie verlassen wird. In der Praxis allerdings wird der Test kleine Implementationsfehler – in diesem Fall wäre ein Off–By–One Fehler nicht unüblich – nicht verzeihen.

Die Testliste hat nun zwei, noch nicht umgesetzte, Tests. Einer davon ist neu:

1. ~~Text = Text ohne erkannten Text vom Anfang.~~
2. Text = Text, wenn erkannter Text leer ist.
3. ~~“holidays Time to relax leave work behind and enjoy” = “Finally my well deserved holidays Time to relax leave work behind and enjoy” – “Finally my well deserved”.~~
4. Nach dem Aufruf von *getNextIterationInput()* muss der interne Text auf den restlichen, unerkannt verbliebenen Text gesetzt werden.

Beispiel 3.7 zeigt das finale Ergebnis des Refactoring. Zusätzlich wurden die restlichen Tests implementiert.

```
1        public String getNextIterationInput(string
            entire_transcription_text, string recognized_text)
2          {
3              TextSegment entire_transcription_text_ = new
                  TextSegment(entire_transcription_text);
4              TextSegment recognized_text_ = new TextSegment(
                  recognized_text);
5              entire_transcription_text_.RemoveRange(0,
                  recognized_text_.getCount());
6              SetNewInputText(entire_transcription_text_);
7              return entire_transcription_text_.ToString();
8          }
9
10         [Test]
11         public void TestGetRemainingTextAfterRecognition()
12         {
13             String entire_text_input = "Finally my well deserved
                  holidays Time to relax leave work behind and enjoy";
14             String recognized_text_input = "Finally my well
                  deserved";
15             String remaining_text = "holidays Time to relax leave
                  work behind and enjoy";
16             Assert.IsEqual(TestAudioManagerClass.
                  getNextIterationInput(entire_text_input,
                  recognized_text_input), "holidays Time to relax
                  leave work behind and enjoy");
17             Assert.IsEqual(TestAudioManagerClass.getRemainingText()
                  , remaining_text);
18             Assert.IsEqual(TestAudioManagerClass.
                  getNextIterationInput(entire_text_input,""),
                  remaining_text);
```

```
19          }
```

Listing 3.7: Beispiel: Endversion der TDD–Implementierung

Am Ergebnis ist zu erkennen, dass die Testliste umgesetzt wurde. Alle Anforderungen an die Methode werden erfüllt, und zusätzlich sind neue Anforderungen entstanden. Die erste neue Anforderung ist ein Auswahlalgorithmus aus der Menge der erkannten Texte. Die zweite Anforderung ist eine Methode zum Setzen des restlichen Textes.

Das Beispiel illustriert, wie eine einfache Anforderung schrittweise umgesetzt wird, und wie durch Verfeinerung und Refactoring neue Tests und Anforderungen entstehen können. Diese Anforderungen prägen das Design und die endgültige Architektur. Jede Änderung erfordert zu keinem Zeitpunkt eine grundlegende Umstrukturierung des gesamten Ansatzes, sondern ist agil umsetzbar.

3.3.3 “Code Smells” Beispiele

Die folgenden Beispiele sind ausgewählte, dokumentierte Fälle. Alle Änderungen konnten mit minimalem Aufwand durchgeführt werden. Selbst große Designänderungen waren schnell und effektiv umsetzbar. Der nötige Quellcode wurde um bis zu 75% reduziert.

Mittelsmann

```
1
2  private string AnalyzeRecognitionData()
3          {
4              String ret_val = "";
5              ret_val = AMR_.getNextIterationInput();
6              RecognitionEngine.SetTextInput(ret_val);
7              return ret_val;
8          }
```

Listing 3.8: Code Smells: Mittelsmann, Klasse: TranscriptionControlClass

Beispiel 3.8 ist ein Fall, der sich erst nach einem Refactoring offenbart hat. In diesem Beispiel benutzt eine interne Methode die private Funktion *AnalyzeRecognitionData* um den Input für die nächste Iteration abzufragen. Diese Methode ist in Wirklichkeit obsolet.

Nachrichtenketten

```
1
2  private void SetupForProccessing(string in_string)
3          {
4              string input_ = CleanInput(in_string);
5              RecognitionEngine.SetInput(input_);
6              RecognitionEngine.AnalyzeAudio();
7              RecognitionEngine.SetInput(RecognitionEngine.GetInput()
                   );
```

```
8               RecognitionEngine.GetAudioManager().convertTextStream(
                    RecognitionEngine.GetInput());
9               stopconditionrule_ = RecognitionEngine.GetAudioManager
                    ().getRemainingText();
10              allow_start = true;
11              tmpfile_exists_ = false;
12              new_path_ = "";
13          }
```

Listing 3.9: Code Smells: Nachrichtenketten, Klasse: TranscriptionControlClass

Subtil zeigt sich in Beispiel 3.9, dass die Klasse *AudioManager* vielleicht Teil der *TranscriptionControlClass* sein sollte. Die weitere Analyse der Beziehungen ergibt, dass die AudioManager–Klasse aus der *RecognitionEngine* Klasse auszulagern und in die *TranscriptionControlClass* einzugliedern ist.

Überlange Methoden

Im folgenden Abschnitt wird ein Beispiel für eine überlange Methode demonstriert.

```
1
2                   AudioManager.MarshallXMLData();
3                   ArrayList rev_val = AudioManager.
                        displayInMilSecondsRec();
4                   for (int i = 0; i < rev_val.Count; i++)
5                   {
6                       StdOutProccessWindowEvent(rev_val[i].ToString()
                            );
7                       StdOutProccessWindowEvent(Environment.NewLine);
8                   }
9                   StdOutProccessWindowEvent("XML Data Written to: " +
                        getSavePath() + Environment.NewLine);
10                  last_output_location_ = getSavePath();
11                  StdOutProccessWindowEvent("DONE PROCESSING" +
                        Environment.NewLine);
12                  StdOutProccessWindowEvent("end transcription" +
                        Environment.NewLine);
13                  StdOutProccessWindowEvent("Text Count Remaining: "
                        + AudioManager.getRemainingTextCount() +
                        Environment.NewLine);
14                  StdOutProccessWindowEvent("Untranscribed Text
                        remaining is: " + AudioManager.getRemainingText
                        ());
15                  FinishedTranscription();
16                  RecognitionEngine.closeWavFile();
17                  RecognitionEngine = new Engine.EngineCoreComponent
                        ();
18                  setupEvents();
19                  allow_start = false;
```

Listing 3.10: Code Smells: Überlange Methoden, Klasse: TranscriptionControlClass

Beispiel 3.10 zeigt einen kleinen Ausschnitt aus einem Else-Zweig des *EndStreamEvent-Handler*. Wie der Name bereits impliziert, befindet sich das Programm am Ende einer Erkennungsiteration. Im Fall von 3.10 ist keine weitere Iteration möglich. Der Grund kann sein, dass der gesamte Text bereits erkannt wurde, oder dass eine weitere Erkennung keinen Erfolg mehr bringt.

In diesem Fall soll das Programm alle relevanten Daten in eine XML-Datei schreiben, und das Programm in seinen Anfangszustand bringen. Diese Operationen sind oben dargestellt, und stellten einen Teil der *sapi_SpeechEndStream*-Methode dar. Die primäre Funktion der Methode ist das Vorbereiten der nächsten Iteration. Zuerst wird aufgrund von Abbruchbedingungen festgestellt, ob eine weitere Iteration nötig ist. Dieser Else-Zweig ist nur der Schlusspunkt der gesamten Prozedur, und kann in eine eigene Methode *WrapUpTranscription* ausgelagert werden. Dadurch ergibt sich eine Reduktion der *sapi_SpeechEndStream*-Methode um 38%. Weitere Reduktionen, durch ausgelagerte Operationen in eigene Methoden, haben zu einer Gesamtreduktion der Codezeilen von *sapi_SpeechEndStream* um 75% geführt.

Solche überlangen Methoden entstehen immer wieder, wenn zu viele Tests entstehen, und eine Refactoring-Phase übersprungen wird. Das Umsetzen von zusätzlicher Funktionalität kann die Länge einer Methode schnell vervielfachen.

Kapitel 4

Spezifikation der Arbeit

Das Kapitel über die Spezifikation der Arbeit präsentiert einen genauen Überblick über die technischen Anforderungen an das Programm. Die Gesamtspezifikation gliedert sich in drei Grundschritte, wie beschrieben in Abbildung 4.1. Dabei wird im ersten Schritt der Input der Audio– und Textdaten analysiert. Im zweiten Schritt wird die Microsoft SAPI [2011] benutzt, um Erkennungsdaten über das Gesprochene zu sammeln. Danach wird eine Entscheidung über die beste Wahl der Hypothesen getroffen. Der letzte Schritt besteht aus der Ausgabe der gesammelten Daten in eine XML-Datei.

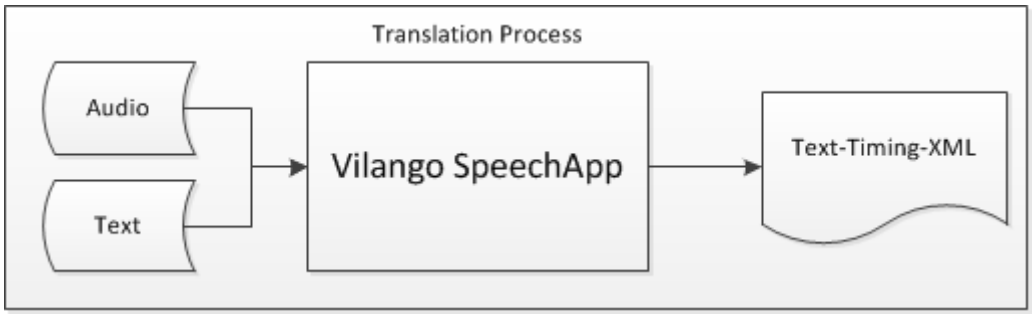


Abbildung 4.1: Übersetzungsprozess von Text-Audio-Input zu Timings-XML.

4.1 Grundidee des Lösungsansatzes

Die Grundidee für den Lösungsansatz ergibt sich aus der Forschung im Bereich der automatisierten Spracherkennung. Moderne Sprachsysteme benutzen statistische Methoden (Zum Beispiel *Hidden–Markov–Models*; siehe Kapitel 2) um Audiodaten, die durch digitale Signalanalyse gewonnen wurden, im Kontext (Kontext–Freie-Grammatik) von *Natural Language Understanding* und Laute–Profilen, dem wahrscheinlichsten Text zuzuordnen. [Amtrup, 1999; Jurafsky und Martin, 2000; Carstensen et al., 2001; Allen, 1987] Die Grundschritte zum Erkennungsprozess wurden bereits in Kapitel 2 erläutert.

An dieser Stelle wird besonders auf die statistische Auswertung eingegangen. Die möglichen Zustandsübergänge ergeben sich aus zwei Komponenten: Dem akustischen Modell der Sprache und dem benutzten Wörterbuch. Spracherkenner versuchen dabei den Übergang mit der höchsten Gesamtwahrscheinlichkeit zu finden. Jurafsky und Martin [2000]; Markowitz und

Scholz [2010]; Paul [1990]; Juang und Rabiner [1991]; Gales und Young [2007] sind gute Werke zum Thema der Automatisierten Spracherkennung, die diese Themen genauer behandeln.

Die Aufgabe dieser Arbeit war nicht die Entwicklung eines solchen Systems. Das Verständnis des zugrunde liegenden Prinzips der Spracherkennung mithilfe von HMMs, war die Grundlage für die Idee des Lösungsansatzes.

Die erzielbare Genauigkeit hängt von verschiedenen Faktoren ab. Einer davon ist die Qualität der akustischen Profile. Aus diesem Grund bieten alle modernen Erkenner die Möglichkeit an, diese Profile zu trainieren. Dabei wird ein vordefinierter Text vom Benutzer gesprochen. Der Erkenner verfeinert dabei das Modell für das Profil des Sprechers. Die Erkennung für den Sprecher wird an die Eigenheiten und Besonderheiten der Aussprache des Benutzers angepasst. Dies erhöht die Qualität der Spracherkennung für den trainierten Benutzer, und mindert gleichzeitig die Qualität für andere Benutzer.

Der Kunde hat für die Aufnahmen der Audiodateien verschiedene Sprecher benutzt. Es werden stets neue Sprecher auf Honorarbasis engagiert. Beim Erstellen der Vilango SpeechApp musste aus diesem Grund ein nicht–trainierter Erkenner benutzt werden. Da für die Audiodaten verschiedene Sprecher benutzt wurden, und in einer Aufnahme mehrere verschiedene Sprecher auftreten, ist ein vorab Trainieren des Erkenners auch ausgeschlossen.

Die zweite Möglichkeit um die Qualität einer Erkennung zu verbessern ist, den Raum, der erkennbaren Wörter, einzuschränken. Zudem ist die Reihenfolge der Wörter bereits durch das Transkript vorgegeben. Das bedeutet, dass auch falsch erkannte Wortketten ausgeschlossen werden können.

Vor jeder Erkennung wird der Erkenner konfiguriert. Im Falle der Microsoft SAPI [2011] ist dies möglich, indem eine *Command and Control* Konfiguration durchführt wird [Microsoft InProcRecognizer, 2010; Microsoft SAPI, 2011].

In diesem Modus reagiert der Erkenner auf vermutete Befehle. Wenn als Befehl der zu erkennenden Text definiert wird, entstehen Hypothesen über die Erkennung dieses Textes. Diese Hypothesen werden analysiert und die besten selektiert. Diese Hypothesen sind nur Teilerkennungen und auch Falscherkennungen. Es werden die Teilerkennungen von den Falscherkennungen durch Heuristik getrennt. Die beste Teilerkennung wird benutzt, um den Eingabetext zu reduzieren. Die Erkennung wird mit dem reduzierten Text neu konfiguriert. Letztendlich endet der Prozess, wenn der gesamte Text erkannt wird, oder die Erkennung keine weiteren Hypothesen aufstellt. Im letzteren Fall werden die Teile, die erkannt werden konnten, als Teilergebnis ausgegeben.

Um die Qualität der Erkennung weiter zu erhöhen, wird die Audiodatei geschnitten, und somit die Eingabe weiter reduziert. Die Microsoft SAPI [2011] bot zum Zeitpunkt des Verfassens dieser Arbeit standardmäßig ein akustisches Profil für folgende Sprachen an:

- Deutsch
- Englisch
- Französisch
- Spanisch
- Japanisch
- Einfaches Chinesisch

- Traditionelles Chinesisch

4.2 Anforderungen an die GUI und Eingabe

Die Texteingabe ist der erste Schritt der Konfiguration zur Erkennung. Abbildung 4.2 stellt den Eingabeprozess dar. Die Eingabe von Audiodateien muss überprüft werden. Nur kompatible Dateien zum *SpeechAudioFormatType* sind erlaubt [SAPI Audio Format, 2010].

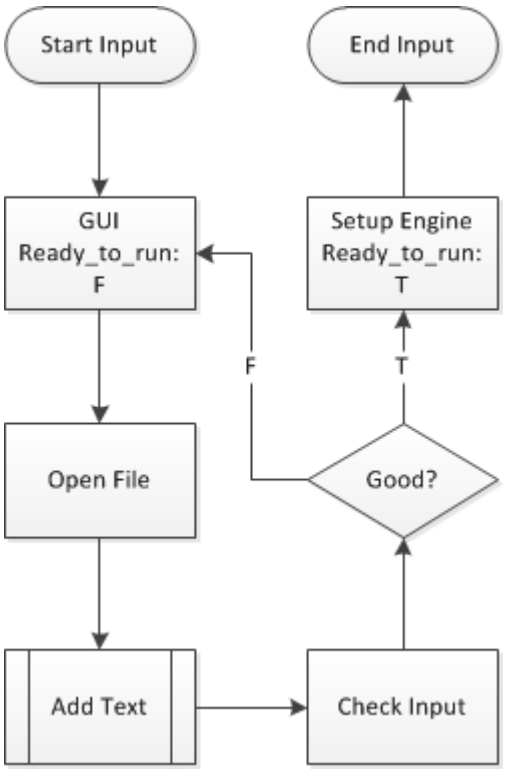


Abbildung 4.2: Flow–Chart Eingabeprozess.

- **UR 01:** GUI Zum einlesen von Daten.
- **UR 02:** Möglichkeit zum einlesen von Text als Eingabe.
- **UR 03:** Entfernen von unerlaubten Zeichen.
- **UR 04:** Unerlaubte Zeichen als Eingabe sind erlaubt.
- **UR 05:** Vorbereitung des Textes für die Verarbeitung.
- **UR 06:** Möglichkeit zum Öffnen von Dateien.
- **UR 07:** Audiodateien müssen auf Kompatibilität überprüft werden.
- **UR 08:** Vorbereitung des Textes für die Verarbeitung.
- **UR 09:** Das Programm soll über eine GUI konfigurierbar sein.

- **UR 10:** Einstellungen sollen einsehbar sein.
- **UR 11:** Einstellungen müssen änderbar sein.
- **UR 12:** Einstellungen sollen die Möglichkeit haben permanent gespeichert zu werden in einer Konfigurationsdatei.
- **UR 13:** Das Programm soll nur bei validen Eingabedaten startberechtigt sein.
- **UR 14:** Das Programm soll standardmäßig die gesamte Datei analysieren.
- **UR 15:** Das Programm soll in der Lage sein von einer bestimmten Stelle in der Audiodatei zu starten. Die Angabe der Startposition erfolgt in Millisekunden.
- **UR 16:** Der Arbeitsprozess soll in der GUI verfolgbar sein.
- **UR 17:** Die Ergebnisse sollen in der GUI sichtbar sein nachdem der Arbeitsprozess abgeschlossen ist.
- **UR 18:** Die GUI soll das Zugreifen auf die Ausgabe ermöglichen.
- **UR 19:** Das Schließen des Programmes soll erst nach einer Bestätigung erfolgen.
- **UR 20:** Der Ausgabeordner soll über ein Suchfenster definierbar sein.

4.3 Erstellen von Text-Timing Information mit SAPI

Abbildung 4.3 stellt den Translation–Prozess dar.

- **UR 21:** Der Prozess darf nicht starten ohne valide Daten.
- **UR 22:** Vor dem Starten des Erkenners muss der Erkenner konfiguriert werden.
- **UR 23:** Vor dem Starten muss die Analysekomponente konfiguriert werden.
- **UR 24:** Vor dem Starten des Erkenners muss er erfolgreich initialisiert werden als *InProcRecognitionEngine* [Microsoft InProcRecognizer, 2010].
- **UR 25:** Ausnahmefehler müssen abgefangen und behandelt werden.
- **UR 26:** Der Erkenner muss für die Aufnahme von Audiodateien konfiguriert sein.
- **UR 27:** Alle definierten Ereignisse müssen behandelt werden.
- **UR 28:** Alle Daten, erzeugt vom Erkenner, müssen gespeichert werden.
- **UR 29:** Alle Daten, erzeugt vom Erkenner, müssen analysiert werden.
- **UR 30:** Jeder Durchlauf führt zu einer Umgestaltung der Daten.
- **UR 31:** Jeder Durchlauf reinitialisiert den Erkenner mit neuen Daten.
- **UR 32:** Am Ende des Prozesses sollen alle temporären Dateien gelöscht werden.

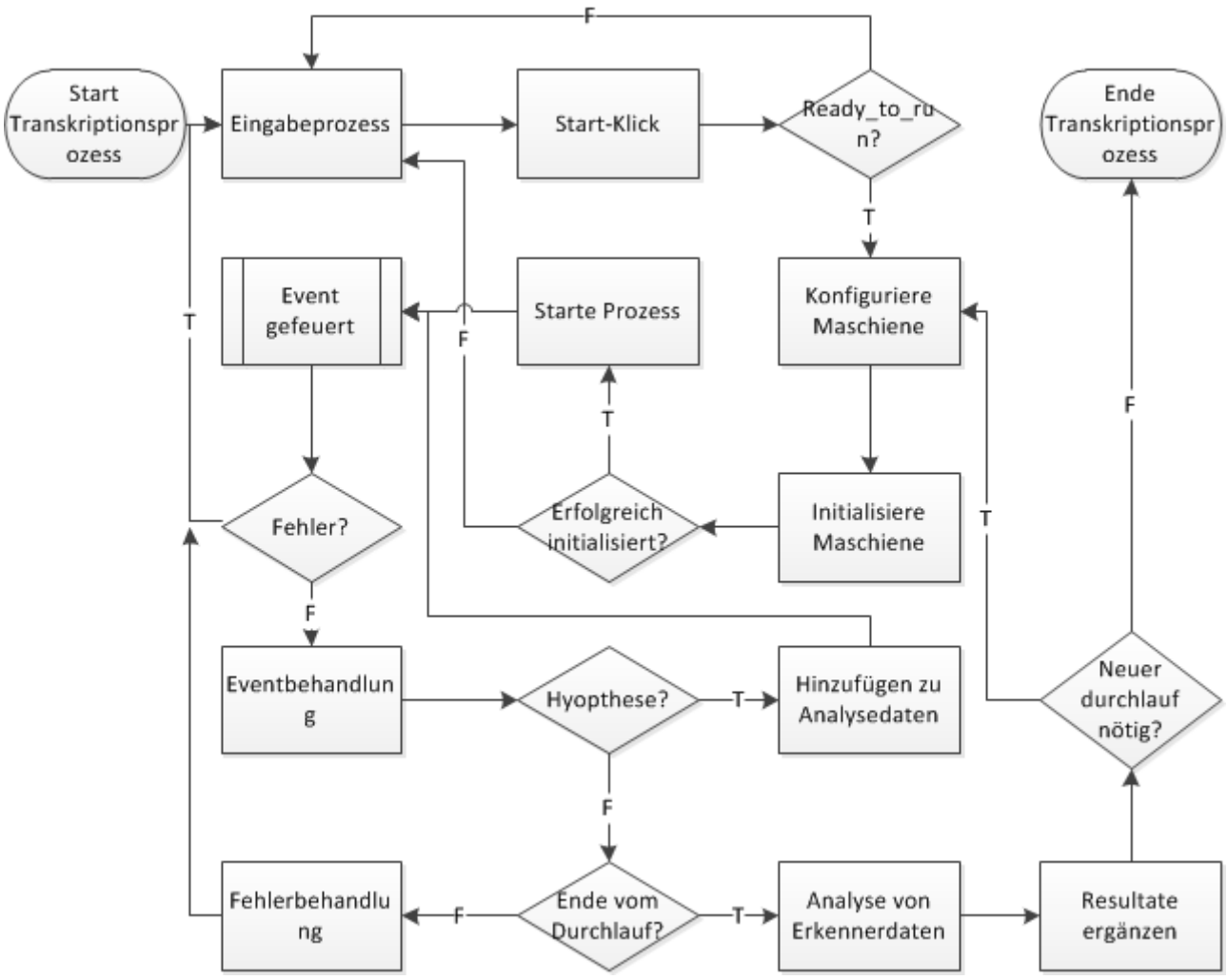


Abbildung 4.3: Flow–Chart Translation Prozess.

- **UR 33:** Am Ende des Prozesses sollen alle zwischengespeicherten Daten gelöscht werden.
- **UR 34:** Am Ende des Prozesses sollen alle Resultate in eine XML–Datei vom vereinbarten Format ausgegeben werden.

4.4 Nicht–Funktionale Anforderungen

Als Nicht-Funktionale Anforderung werden folgende Punkte definiert:

- **NFR 01:** Das Programm soll stabil sein und bei korrupten Daten nicht unkontrolliert beenden.
- **NFR 02:** Zwischenresultate sollen bei erfolgreicher Fehlerbehandlung ausgegeben werden.
- **NFR 03:** Zwischenresultate sollen ausgegeben werden, wenn eine komplette Erkennung nicht erfolgreich war.
- **NFR 04:** Alle Prozesse sollen nach der Eingabe durch den Benutzer vollautomatisiert ablaufen.
- **NFR 05:** Umprogrammieren des Auswahlalgorithmus soll einfach möglich sein.
- **NFR 06:** Hinzufügen neuer Auswahlalgorithmen soll einfach möglich sein.
- **NFR 07:** Einfache Bedienung.

Kapitel 5

Architektur und Design

“ Simplicity – the art of maximizing the amount of work not done – is essential. ”

[Agile Manifesto, 2001]

Die Art und Weise, wie die Architektur und das Design zustande kommen, wurde bereits im Kapitel 3 diskutiert. Dieses Kapitel stellt die benutzte Arbeitsumgebung, die allgemeine Architektur, Komponenten und die Interfaces vor. Auf spezifische Details zum Design wird in Kapitel 6 eingegangen.

5.1 Arbeitsumgebung

Die Arbeitsumgebung, die bei der Entwicklung eingesetzt wird, stellt sich aus folgenden Programmen zusammen:

1. Visual Studio 2010 Ultimate Service Pack 1. [Visual Studio, 2010]
2. NUnit Testumgebung. [NUnit.org, 2009]
3. TestDriven.Net, eine Erweiterung für Visual Studio. [Mutant Desing Ltd., 2011]
4. Sandcastle Documentation Builder Erweiterung für Visual Studio. [Sandcasle Help File Builder Team, 2011]

Tabelle 5.1: Zusammenstellung der Arbeitsumgebung.

5.2 Allgemeine Architektur von Vilango SpeechApp.

Der gesamte Arbeitsprozess der Vilango SpeechApp kann als eine lineare Verarbeitung von asynchroner Ereignisbehandlung gesehen werden. Die *Recognition–Engine* stellt über asynchrone Ereignisse Hypothesen zur Verfügung, die im ersten Schritt gespeichert, im zweiten

Schritt aussortiert und im dritten Schritt ergänzend zu den Ergebnissen hinzugefügt werden. Nach der Konfigurationsphase wird versucht den Rest der unerkannten Daten mit einem reduzierten Set von Eingabetext zu erkennen. Der Analyseprozess ist grob in Abbildung 5.1 dargestellt.

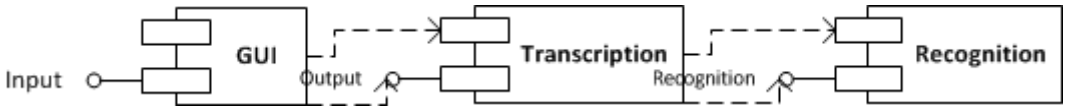


Abbildung 5.1: Der Analyseprozess.

Diese grobe Darstellung wird nachfolgend genauer erörtert.

5.2.1 Detaillierter Flow–Chart

Der detaillierte Flow–Chart, in Abbildung 5.2, stellt den Ablauf des Erkennungsprozesses dar.

5.2.2 Klassendiagramm von Vilango SpeechApp

Das Klassendiagramm stellt den strukturellen Zusammenhang des Programmes dar. Für eine genauere Beschreibung der Klassen und ihrer Struktur, wird auf den Anhang und die technische Dokumentation verwiesen. Das Klassendiagramm ist in Abbildung 5.3 dargestellt.

5.3 Komponenten

Das Unterkapitel Komponenten beschreibt die wichtigsten Komponenten des Programmes. Nicht jede Komponente bedarf einer Beschreibung, da Komponenten wie *PojektInstaller* für die Architektur nicht von Belang sind. Auf die wichtigsten Details der Komponenten wird im Kapitel 6 eingegangen.

5.3.1 Grafische Benutzeroberfläche (GUI)

Die GUI ist, wie bereits in Kapitel 3 beschrieben, eine *Ultra–Thin GUI*. Die einzige Aufgabe der GUI ist es, einen Einblick in die Daten zu gewähren, und Steuerelemente zur Verfügung zu stellen. Die GUI implementiert keine Logik. Sie dient nur zum Bedienen eines Interfaces.

Die GUI Elemente

Die wichtigsten Aufgaben der GUI sind es, dem Benutzer einen Einblick auf die aktuellen Einstellungen zu gewähren, den Input zu ermöglichen, und den produzierten Output zu betrachten. Die Eingabe von unerlaubten Zeichen wird kontrolliert. Zur Robustheit und Stabilität der Software wird im Kapitel 7 Stellung genommen. Abbildung 5.4 zeigt das Aussehen der GUI.

Settings–Window: Die GUI ermöglicht dem Benutzer über ein eigenes *Settings–Window* Einstellungen einzusehen und zu korrigieren.

Öffnen von Dateien: Die GUI ermöglicht dem Benutzer eine Datei für die Verarbeitung zu öffnen. Gleichzeitig wird die Eingabe eines passenden Textes erwartet.

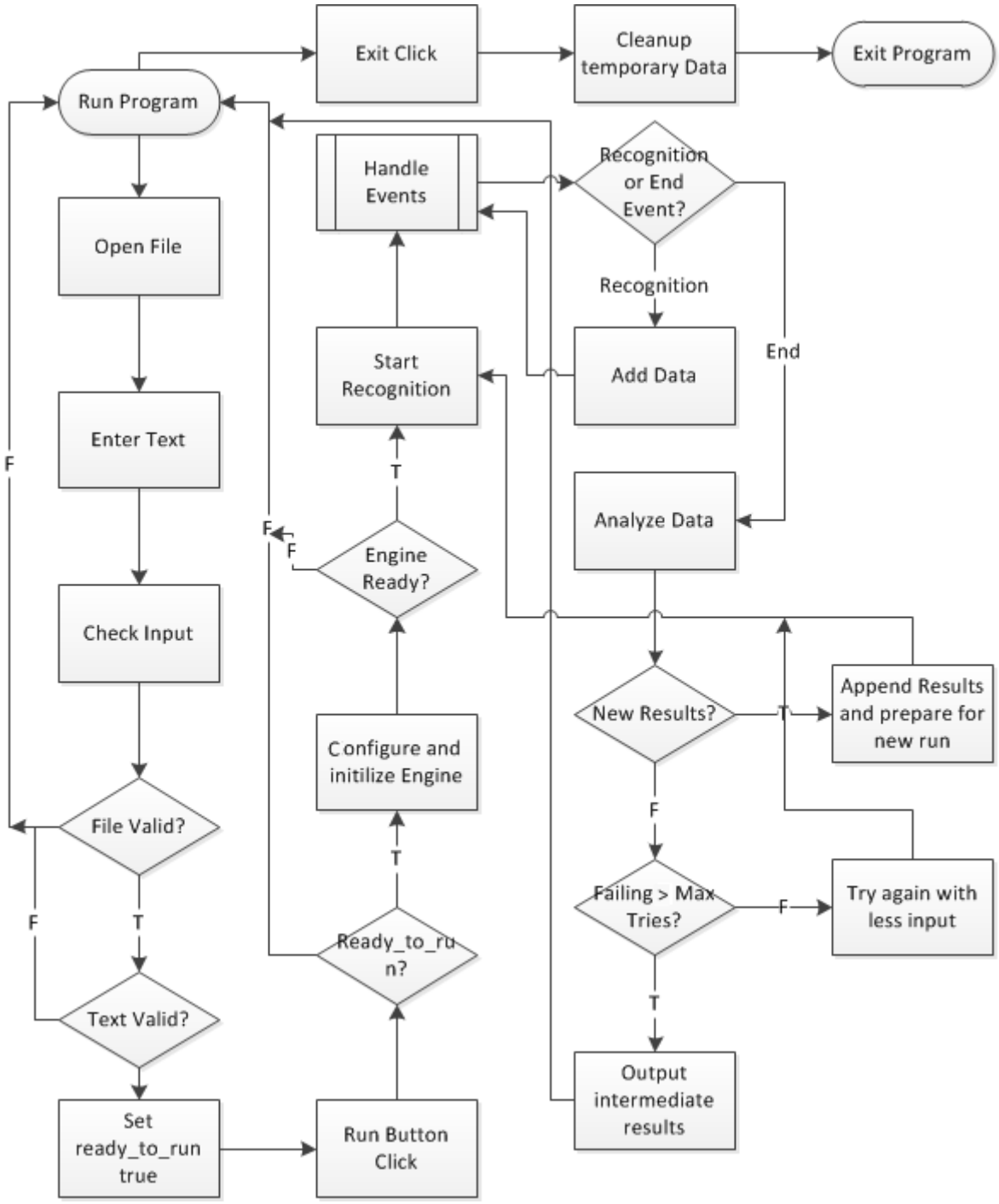


Abbildung 5.2: Detaillierter Flow–Chart des Erkennungsprozesses.

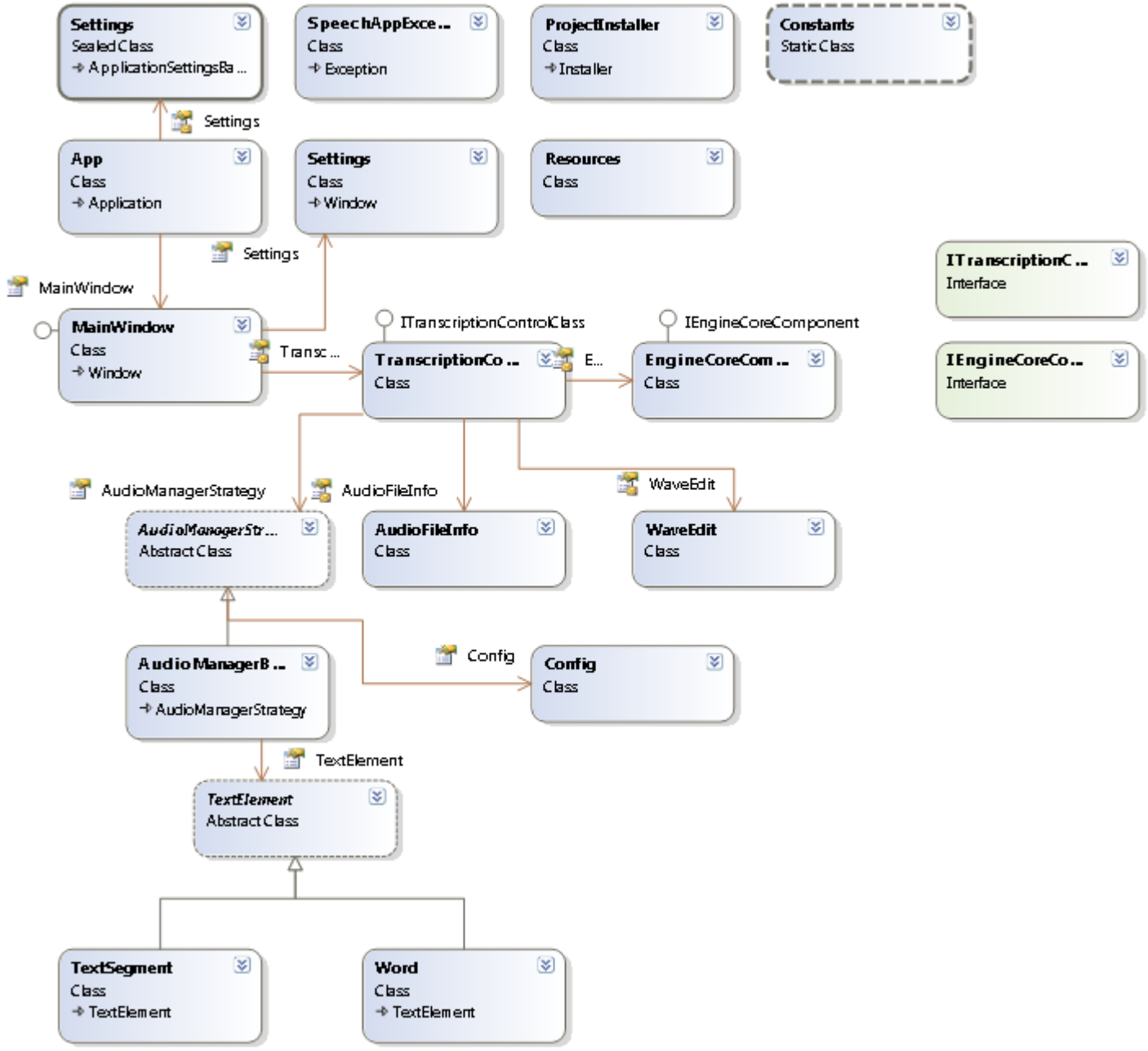


Abbildung 5.3: Klassendiagramm erstellt durch Visual Stidio. [Visual Studio, 2010]

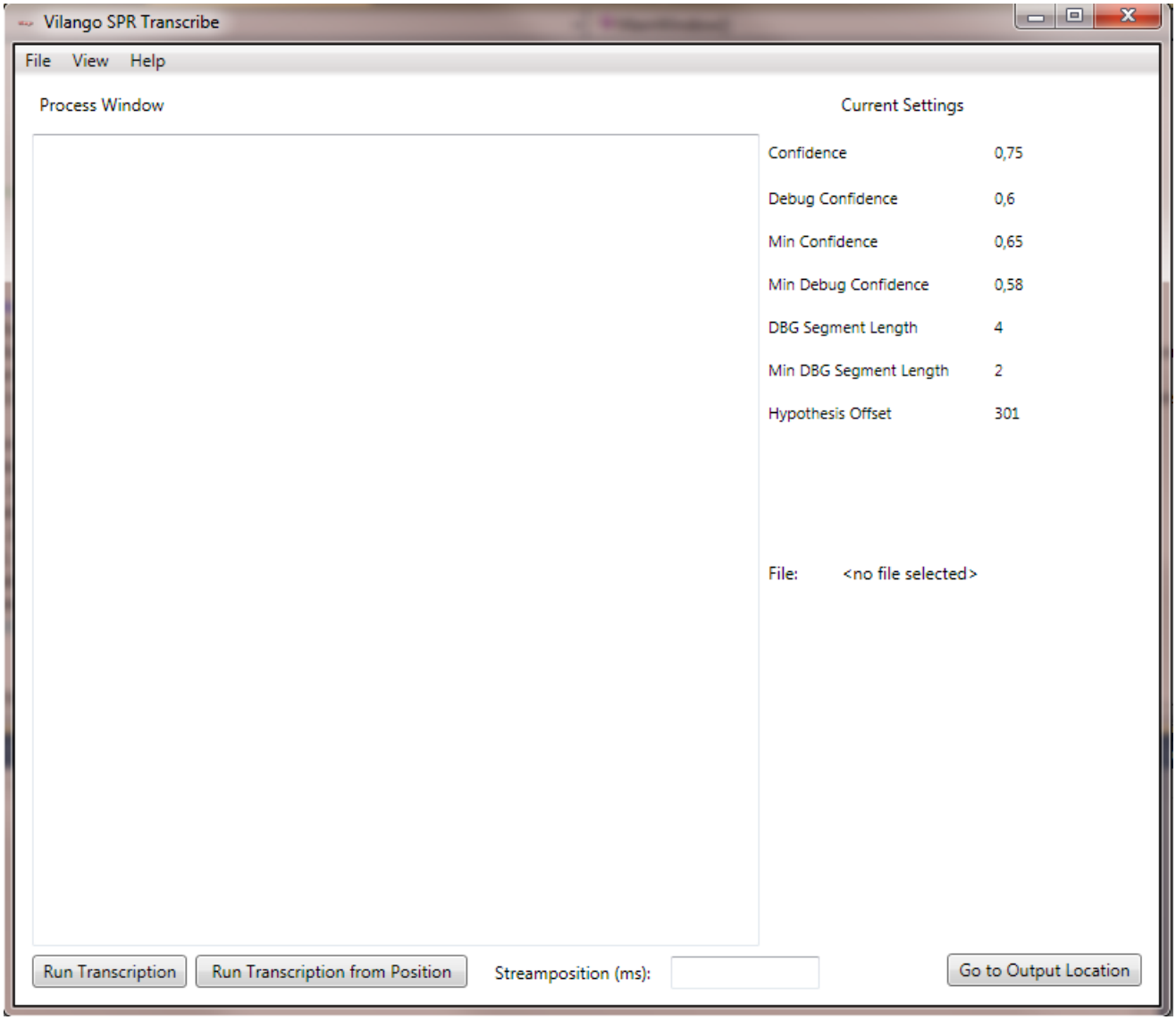


Abbildung 5.4: Vilango SpeechApp GUI.

Starten, Starten von Position und Resultate–Order öffnen: Die GUI ermöglicht dem Benutzer den Prozess von einer bestimmten Stelle in der Audiodatei zu starten, oder zum Ausgabeordner der XML–Dateien zu wechseln. Standardmäßig erfolgt das Starten vom Beginn der Audiodatei.

Benutzte Interfaces

Die GUI benutzt nur ein Interface: *ITranscriptionControllClass*. Die zwei wichtigsten Interfaces, die eine asynchrone Kopplung des Prozesses implementieren, werden im nächsten Unterkapitel *Interfaces* diskutiert.

5.3.2 Konfiguration

Die Konfiguration gliedert sich in zwei Segmente. Die interne Konfiguration, und die externe Konfiguration aus der *Settings.conf* Datei. Die interne Konfiguration ist temporär im Programm zur Laufzeit gespeichert. Die externe Konfiguration enthält die Standardkonfiguration und alternative Konfigurationsschemas, die vom Benutzer definiert werden können. Die Konfigurationen sind dabei in Sektionen und Einstellungen gegliedert. Die externe Konfiguration ist im XML–Format definiert.

5.3.3 TranskriptionControl

Die Steuerung des Transkription–Prozesses übernimmt die *TranscriptionControl*–Komponente. Alle Ereignisse des Erkenners werden an dieser Stelle behandelt. Zudem wird der Erkenner für jede Iteration initialisiert, die Daten analysiert und die Fehlerbehandlung durchgeführt.

5.3.4 EngineCoreComponent

Die *EngineCoreComponent* implementiert den *InProc–Recognizer* (siehe Microsoft InProcRecognizer [2010]. Die Komponente ist über das *IEngineCoreComponent*–Interface ansprechbar. Über das Interface ist eine Konfiguration der wichtigsten Einstellungen möglich. Das betrifft besonders den Dateipfad zur Audiodatei, und den Text der gesucht werden soll.

5.3.5 AudioManager–Komponente

Diese Komponente implementiert alle Funktionen, die für die Struktur und Organisation der Analysedaten notwendig sind. Weiteres wird ein Auswahlalgorithmus implementiert, der überladen werden kann, sodass die Implementierung von alternativen Algorithmen ermöglicht wird. Dies war eine Kundenanforderung. In Kombination mit der freien Wahl und Struktur der Konfigurationen, kann eine Kombination von Algorithmus, und seinen Einstellungen definiert werden.

5.4 Interfaces

Das Unterkapitel Interfaces stellt die zwei wichtigsten Interfaces und ihre Struktur vor. Beide Interfaces werden im Kapitel 6 detaillierter beschrieben. Dieses Kapitel dient als technische

Dokumentation der Architektur.

5.4.1 IEngineCoreComponent–Interface

Dieses Interface ist für die asynchrone Kopplung zwischen Erkenner und Transkription–Kontrolle verantwortlich, und ist in Quellcodeabbildung 5.1 dargestellt.

```
1  event VilangoSpeechApp.Engine.EngineCoreComponent.
       SapiEndStreamEventHandler SpeechEndStream;
2
3          event VilangoSpeechApp.Engine.EngineCoreComponent.
               SapiSpeechRecognizedEventHandler SpeechRecognized;
4
5          event VilangoSpeechApp.Engine.EngineCoreComponent.
               SapiHypothesisEventHandler HypothesisEvent;
6
7          void SetWavInput(String in_);
8
9          String GetInputWav();
10
11         void SetTextInput(String in_);
12
13         String GetLastInput();
14
15         String GetInput();
16
17         void CloseWavFile();
18
19         bool InitializeInProcRecognitionEngineAtStreamPosition(
               object position);
20
21         void StartInProcEngine();
```

Listing 5.1: Struktur von IEngineCoreComponent–Interface.

5.4.2 ITranscriptionControlClass–Interface

Dieses Interface steht der GUI zur Verfügung, und ist in 5.2 dargestellt.

```
1  event VilangoSpeechApp.Engine.TranscriptionControlClass.
       StdoutEventHandler StdOutProccessWindowEvent;
2
3          event VilangoSpeechApp.Engine.TranscriptionControlClass.
               StdoutMessageBox StdOutMessageBoxEvent;
4
5          event VilangoSpeechApp.Engine.TranscriptionControlClass.
               StdoutClearProcessWindow ClearWindow;
6
```

```
7           event VilangoSpeechApp.Engine.TranscriptionControlClass.
               FinishedTranscriptionEventHandler FinishedTranscription;
8
9           double GetBaseConfidence();
10
11          double GetBaseDebugConfidence();
12
13          double GetMinStandardConfidence();
14
15          double GetMinDebugConfidence();
16
17          long GetDebugCount();
18
19          long GetDebugMinCount();
20
21          long GetOffset();
22
23          string GetSavePath();
24
25          Boolean HandleInput(string filepath_, string filename_,
               string input_);
26
27          void SetSettings(double confidence, double debug_confidence
               , double min_confidence, double min_dbg_confidence,
               Int64 min_hypothesis_length, Int64 min_segment_length,
               Int64 offset);
28
29          void SetPath(string val);
30
31          void SaveSettings();
32
33          void CleanUpWave();
34
35          void CleanUpTemp();
36
37          void Run();
38
39          void RunFromPosition(int offset, string fromInputText);
```

Listing 5.2: Struktur von ITranscriptionControlClass–Interface.

5.5 Anforderungen an die Laufumgebung

Durch die Architektur und die Implementierung ergeben sich folgende Anforderungen an die Laufumgebung:

1. .NET 3.5 SP1. [Microsoft .NET, 2011]
2. 32/64–Bit Windows 7 System.

3. Windows 7 SDK [Microsoft SDK, 2011].

4. Installiertes Sprachpaket der Zielsprache, die Erkannt werden soll. Unterstützte Sprachen wurden bereits in Sektion 4.1 diskutiert.

5. An die Hardware der Laufumgebung werden dieselben Anforderungen gestellt, die durch Microsoft [2011] an die Laufumgebung für Windows 7 gestellt werden.

Kapitel 7 erörtert die Gründe, wieso eine höhere Rechnerleistung zu besseren Ergebnissen führt.

Kapitel 6

Implementierung und die wichtigsten Designdetails

Um einen besseren Einblick in die Implementation und Funktionsweise zu bekommen, werden an dieser Stelle selektive Details genauer beschrieben. Die verwendeten Technologien und Bibliotheken werden kurz vorgestellt.

6.1 Verwendete Technologien und Programmbibliotheken

Neben dem *.NET Framework* werden keine weiteren Programmbibliotheken verwendet. Es werden auch keine *3rd Party Tools*, oder andere Hilfsmittel benutzt. Im Kapitel 5 wird die Arbeitsumgebung vorgestellt.

6.2 Interface Kopplung durch Events

Das *.NET Framework* bietet eine besondere Form der *Observer Design Pattern*, genannt *Event Pattern* oder auch *Event–based Asynchronous Pattern* (siehe Gamma et al. [1994]; Purdy und Richter [2002]). Das *Observer Design Pattern* ist in Abbildung 6.1 dargestellt.

In dem Fall der *Vilango SpeechApp* geschieht die Kopplung zusätzlich über ein Interface, dass die konkrete Implementierung der Ereignisse kapselt. Dadurch sind Ereignisse und die Subjekte besser austauschbar.

Codebeispiel 6.1 zeigt ein konkretes Beispiel der Implementierung. Dadurch gibt es keine *Busy–waits*, und die Kommunikation mit dem asynchronen COM–Objekt (COM – "Component Object Model") verursacht keine Race Condition. Zu weiteren technischen Details über Microsoft COM wird auf MicrosoftCOM [2011] verwiesen.

```
1 //In IEngineCoreComponent.cs: Definition der Eventhandler:
2
3 event VilangoSpeechApp.Engine.EngineCoreComponent.
     SapiEndStreamEventHandler SpeechEndStream;
4
```

```
5  event VilangoSpeechApp.Engine.EngineCoreComponent.
       SapiSpeechRecognizedEventHandler SpeechRecognized;
6
7  event       VilangoSpeechApp.Engine.EngineCoreComponent.
       SapiHypothesisEventHandler HypothesisEvent;
8
9  //In EngineCoreComponent.cs: Implementation des Interfaces und der
       Delegates:
10
11 public delegate void SapiSpeechRecognizedEventHandler(int
       StreamNumber, object StreamPosition, SpeechRecognitionType
       RecognitionType, ISpeechRecoResult Result);
12
13 public delegate void SapiHypothesisEventHandler(int StreamNumber,
       object StreamPosition, ISpeechRecoResult Result);
14
15 public delegate void SapiEndStreamEventHandler();
16
17 public event SapiEndStreamEventHandler SpeechEndStream;
18
19 public event SapiSpeechRecognizedEventHandler SpeechRecognized;
20
21 public event SapiHypothesisEventHandler HypothesisEvent;
22
23 event SapiEndStreamEventHandler IEngineCoreComponent.
       SpeechEndStream
24         {
25             add
26             {
27                 lock (lock_)
28                 {
29                     SpeechEndStream += value;
30                 }
31             }
32             remove
33             {
34                 lock (lock_)
35                 {
36                     SpeechEndStream -= value;
37                 }
38             }
39         }
40
41 event SapiSpeechRecognizedEventHandler IEngineCoreComponent.
       SpeechRecognized
42         {
43             add
44             {
45                 lock (lock_)
46                 {
```

```
47                      SpeechRecognized += value;
48                  }
49              }
50              remove
51              {
52                  lock (lock_)
53                  {
54                      SpeechRecognized -= value;
55                  }
56              }
57          }
58
59  event SapiHypothesisEventHandler IEngineCoreComponent.
        HypothesisEvent
60          {
61              add
62              {
63                  lock (lock_)
64                  {
65                      HypothesisEvent += value;
66                  }
67              }
68              remove
69              {
70                  lock (lock_)
71                  {
72                      HypothesisEvent -= value;
73                  }
74              }
75          }
```

Listing 6.1: Implementation des *Event-based Asynchronous Pattern*.

6.3 Transcription Engine

Die gesamte Logik der Ereignisbehandlung ist in dieser Komponente definiert. Wie im Unterkapitel *Interface Kopplung durch Events* erklärt, werden Erkenner–Ereignisse an die Beobachter weitergereicht.

Die wichtigste Methode der Komponente ist die *sapi_SpeechEndStream*–Methode. Sie stellt den Dreh– und Angelpunkt des Erkennungsprozesses dar. Der aktuelle Zustand der Erkennung wird festgestellt, Resultate abgerufen, und entschieden welche Daten der nächste Durchlauf hat. Diese Methode entscheidet auch, ob ein weiterer Durchlauf nötig ist. Die Abbruchbedingung ist definiert durch zwei Faktoren:

1. Gibt es neue Resultate: Wenn ja, weiteren Durchlauf starten. Wenn nicht, dann wird überprüft, ob ein Durchlauf mit denselben Eingabedaten zu oft gescheitert ist. Im letzteren Fall werden die bereits gesammelten Daten ausgegeben, mit der Information welche Daten nicht erkannt werden konnten.

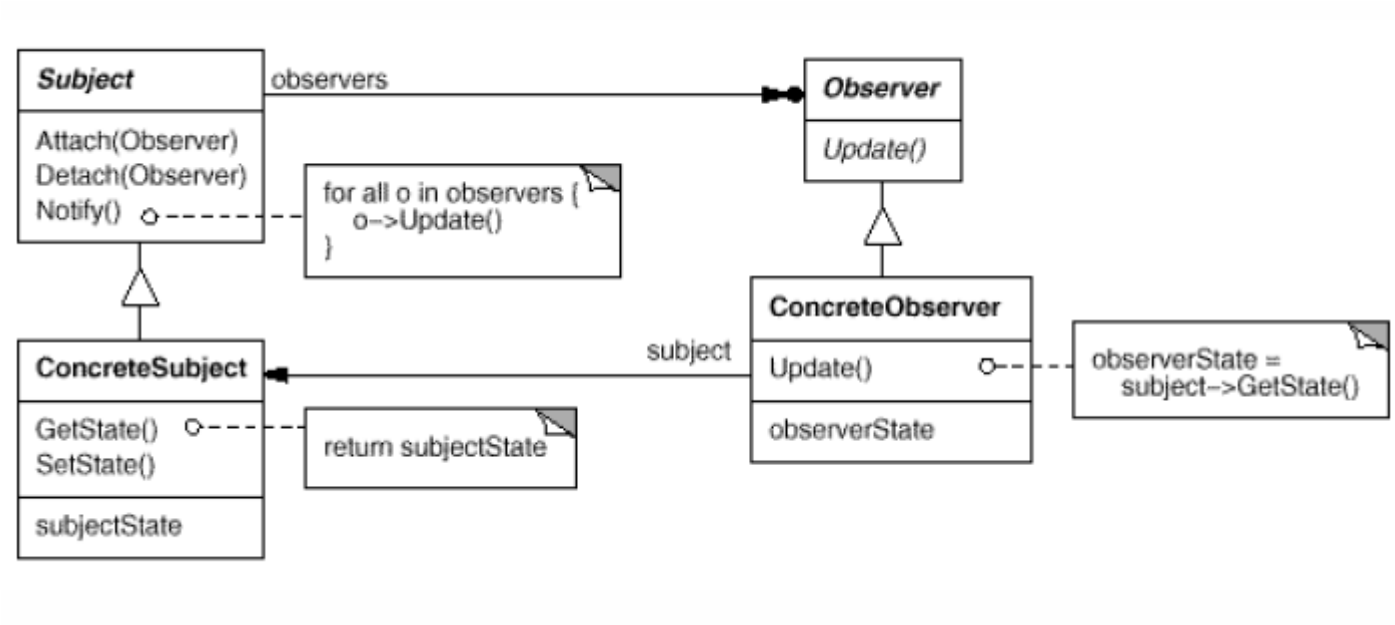


Abbildung 6.1: UML–Diagramm des Observer Design Pattern. Quelle: Gamma et al. [1994].

2. Der letzte Durchlauf hat die Erkennung komplettiert. Das heißt, dass alle Eingabedaten vom Benutzer erfolgreich erkannt wurden. Es erfolgt die Ausgabe, und das Programm wird in den Ausgangszustand versetzt. Eine neue Eingabe von Daten ist möglich.

6.4 Recognition Engine

Der benutzte Spracherkenner ist die *Speech Application Engine* von Microsoft. Der Erkenner bedient sich dabei der installierten Sprachpakete des Benutzercomputers. Diese enthalten Wörterbücher und Phonem–Profile der eingestellten Sprache. In den Kapitel 2 und 4, wurde der Lösungsansatz und die zugrunde liegende Technik erörtert.

Die wichtigsten Schritte sind im Beispiel 6.2 angeführt. Der Erkenner ist ein COM–Objekt. Zeile 6 erzeugt eine Instanz des aktuell aktiven Erkenners von Windows. Der Befehl in Zeile 11 *CreateRecoContext()* erschafft einen eigenen Erkenner-Kontext (siehe Kapitel 2) der gewählten Sprache. Diese ist, wie bereits erwähnt, der aktuell gewählte Spracherkenner von Windows. Zu diesem Kontext wird in der Zeile 12 eine Grammatik erstellt. Diese Grammatik besteht nur aus dem Eingabetext. Die Grammatik verwaltet auf welche Phrasen und Wörter sich der Erkenner beschränkt. *SRATopLevel* und *SRADynamic* sind Attribute die festlegen, dass diese Grammatikregel die Eintrittsregel für die Grammatik ist, und dass die Grammatik während der Laufzeit geändert werden kann. Zeile 14 legt den Zustand am Anfang der Erkennung fest. In Zeile 10 wurde eine Audiodatei als Eingabe festgelegt. Das ist nur im *InProcRecognizer* möglich, der *SharedRecognizer* erlaubt keine Dateieingabe. Der Spracherkenner ist im Zustand *idle*, da keine Grammatik aktiv ist, die er anwenden könnte. In der Methode *StartInProcEngine* wird der Zustand der definierten Grammatik auf *SGDSActive* gesetzt, und der Erkenner startet. Zuvor wurden in den Zeilen 18, 19 und 20 die Schnittstellen der Komponente als Ausgaben des Spracherkenners definiert.

```
1
2  public bool InitializeInProcRecognitionEngineAtStreamPosition(
       object position)
3          {
4              try
5              {
6                  SpInprocRecognizer isp_recognizer_ = new
                       SpInprocRecognizer();
7                  input_wav_ = new SpFileStream();
8                  input_wav_.Open(input_wav_path_,
                       SpeechStreamFileMode.SSFMOpenForRead, false);
9                  input_wav_.Seek(position,
                       SpeechStreamSeekPositionType.
                       SSSPTRelativeToStart);
10                 isp_recognizer_.AudioInputStream = input_wav_;
11                 recognizer_context_ = isp_recognizer_.
                       CreateRecoContext();
12                 recognizer_grammar_ = recognizer_context_.
                       CreateGrammar(0);
13                 recognizer__grammar_rule_ = recognizer_grammar_.
                       Rules.Add("base", SpeechRuleAttributes.
                       SRATopLevel | SpeechRuleAttributes.SRADynamic,
                       0);
14                 recognizer__grammar_rule_.InitialState.
                       AddWordTransition(null, input_, " ",
                       SpeechGrammarWordType.SGLexical, "", 0, "", 1f);
15                 recognizer_grammar_.Rules.Commit();
16
17                 //done setup now setup events
18                 ((SpInProcRecoContext)recognizer_context_).
                       Recognition += new
                       _ISpeechRecoContextEvents_RecognitionEventHandler
                       (SPR_Recognized);
19                 ((SpInProcRecoContext)recognizer_context_).
                       Hypothesis += new
                       _ISpeechRecoContextEvents_HypothesisEventHandler
                       (SPR_Hypothesis);
20                 ((SpInProcRecoContext)recognizer_context_).
                       EndStream += new
                       _ISpeechRecoContextEvents_EndStreamEventHandler(
                       SPR_EndStream);
21
22                 return true;
23             }
24             catch (Exception exception)
25             {
26                 throw new SpeechAppException(exception.Message,
                       exception);
27             }
28         }
```

```
29
30 public void StartInProcEngine()
31         {
32             try
33             {
34                 recognizer_grammar_.CmdSetRuleState("base",
                       SpeechRuleState.SGDSActive);
35             }
36
37             catch (Exception exception)
38             {
39                 throw new SpeechAppException(exception.Message,
                       exception);
40             }
41         }
```

Listing 6.2: Initialisierung und Starten des *InProcRecognizer*. Im Original ist die Initialisierung der Grammatik separat. Der logische Ablauf wurde wegen der besseren Übersicht zusammengelegt.

Das Beispiel 6.3 illustriert, wie die *TranscriptionControllClass* sich als Beobachter registriert. *sapi_SpeechRecognized* und die anderen zwei Methoden sind *Delegates* (delegierte Methoden), die zur Behandlung der Ereignisse aufgerufen werden.

```
1
2 void SetupEvents()
3         {
4             RecognitionEngine.SpeechRecognized += new
                  EngineCoreComponent.SapiSpeechRecognizedEventHandler
                  (sapi_SpeechRecognized);
5             RecognitionEngine.HypothesisEvent += new
                  EngineCoreComponent.SapiHypothesisEventHandler(
                  sapi_SpeechHypothesized);
6             RecognitionEngine.SpeechEndStream += new
                  EngineCoreComponent.SapiEndStreamEventHandler(
                  sapi_SpeechEndStream);
7         }
```

Listing 6.3: Registrieren bei der Schnittstelle für die *RecognitionCoreComponent*.

6.5 Auswahlalgorithmus

Der Auswahlalgorithmus ist in der Methode *GetBestHypothesisFromData* definiert.

Der Algorithmus bedient sich der Parameter, die im *configuration*–Block von Beispiel 6.4 zu sehen sind.

- BaseConfidence: Das ist das Vertrauen, das nötig ist, damit eine Hypothese als Resultat in Frage kommt.

- BaseDebugConfidence: Dies ist das Vertrauen, das nötig ist, wenn keine zufriedenstellende Resultate mit dem Standardvertrauen gefunden werden konnten.
- DebugCount: Mindestmenge an Worten einer Hypothese.
- MinStandardConfidence: Niedrigster Wert den eine Hypothese erreichen darf.
- MinDebugConfidence: Niedrigster Wert den eine Hypothese erreichen darf, wenn die *BaseDebugConfidence* gilt.
- DebugMinCount: Mindestmenge an Worten in der Hypothese wenn das Ergebnis von der *BaseDebugConfidence* abhängig ist.
- Offset: Der Offset legt fest wie weit die möglichen Hypothesen zeitlich voneinander Abweichen dürfen. Die Referenzhypothese wird gewählt aufgrund der Nähe zum letzten erkannten Wort. Da die Erkennung in einer festen Reihenfolge passieren muss (da der Eingabetext ein Transkript ist, sollten die Worte auch genau in der Reihenfolge erkannt werden), sollte der Abstand vom letzten erkannten Wort und dem ersten Wort der Hypothese möglichst gering sein. Damit nun mehrere aufeinanderfolgende gleiche Sätze voneinander unterschieden werden, ist der Abstand, der zwischen den Hypothesen erlaubt ist, möglichst gering zu wählen. Dadurch werden weiter entfernte, mit hohem Vertrauen erkannte Sätze, ausgeschlossen. Dreihundert Millisekunden ist ein empirisch gewählter Wert. Er entspricht einem Wort einer kleineren Länge als fünf. Das ist eine Einstellung für Englisch aufgrund der durchschnittlichen Wortlänge der Sprache (siehe: Salomon [2005]). Diese Einstellung ist auch für Deutsch und Französisch wählbar.

6.5.1 Konfiguration

Der Codeausschnitt 6.4 zeigt die Struktur der externen Parameterverwaltung. Das Löschen oder Weglassen der Parameter führt zu einem Ausnahmefehler. Diese Parameter sind nicht optional, sie können aber erweitert werden. Die Parameter im *configuration*–Block sind unbedingt notwendig. Diese können jedoch ergänzt werden. Parameter dürfen hinzugefügt werden, für den Fall, dass eine Änderung oder Erweiterung des implementierten Algorithmus notwendig wird.

Eigene Konfigurationsdefinitionen, wie in Zeile 13 bis 22 können auch angelegt werden. Für Informationen über die interne Verwendung sei auf die Testdokumentation verwiesen (siehe 7).

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <settings>
3    <configuration>
4      <BaseConfidence>0,75</BaseConfidence>
5      <BaseDebugConfidence>0,6</BaseDebugConfidence>
6      <DebugCount>4</DebugCount>
7      <MinStandardConfidence>0,65</MinStandardConfidence>
8      <MinDebugConfidence>0,58</MinDebugConfidence>
9      <DebugMinCount>2</DebugMinCount>
10     <Offset>300</Offset>
11     <save_path_>\Output\</save_path_>
12   </configuration>
13   <sampleConfigurationFrench>
14     <BaseConfidence>0,80</BaseConfidence>
```

```
15      <BaseDebugConfidence>0,51</BaseDebugConfidence>
16      <DebugCount>4</DebugCount>
17      <MinStandardConfidence>0,6</MinStandardConfidence>
18      <MinDebugConfidence>0,52</MinDebugConfidence>
19      <DebugMinCount>5</DebugMinCount>
20      <Offset>100</Offset>
21      <additionalParameter>value</additionalParameter>
22      <save_path_>\OutputForFrench\</save_path_>
23    </sampleConfigurationFrench>
24  </settings>
```

Listing 6.4: Externes Konfigurationsschema.

6.5.2 Erweiterbarkeit

Durch die Abkopplung des Auswahlalgorithmus von der Programmlogik, zum Steuern und Lenken des Spracherkenners, ist es möglich die Implementierung der *AudioManagerStrategy* in einer konkreten Strategie zu realisieren. Dadurch kann der Algorithmus, der entscheidet welche Hypothese selektiert wird, zur Laufzeit geändert werden. Da zusätzlich die Konfiguration getrennt ist in interne Einstellungen für jeden implementierten Algorithmus, und externe Einstellungen für alle Algorithmen, ist eine Erweiterung der benutzten Parameter, sowohl als auch des Algorithmus, möglich.

Kapitel 7

Testdokumentation und Codemetriken

“ If it cannot be expressed in figures, it is not science; it is opinion. ”

[Robert A. Heinlein, 1907-1988]

Die Dokumentation der angewendeten Testverfahren und der Struktur des Testens gliedert sich in eine Präsentation der Ergebnisse von ausgewerteten Codemetriken, die angewendeten Funktionstests, die Systemtests und in das Testen der Robustheit. Die Testabdeckung des Codes durch die funktionalen Tests wird am Ende dieses Kapitels präsentiert. Die Signifikanz des Softwaretesten sind ein integraler Bestandteil der Softwareentwicklung. Die Analyse, oder eine detaillierte Untersuchung von Testpraktiken, ist nicht die Aufgabe dieser Masterarbeit. Für tiefer greifende Fragen, zu den einzelnen Testarten, wird auf die einschlägige Fachliteratur in Levinson [2011]; Lewis [2005]; Patton [2005] und ins Besondere auf Myers et al. [2004] verwiesen, die als Grundlage für die benutzten Verfahren in diesem Kapitel dienten.

7.1 Codemetriken und Analyse der Codemetriken

Nach Zhao et al. [1998] sagen Codemetriken nicht viel über die Qualität eines Softwareproduktes aus. Sie sind allerdings ein Indikator, um festzustellen, wie schnell und einfach ein Fehler gefunden werden kann. Zhao et al. [1998] unterscheidet zwischen Codemetriken und Designmetriken. Vistual Studio Code Metrics [2010] gibt einen Überblick über die in Visual Studio automatisch ausgewerteten Codemetriken. Weitere Codemetriken und ihre Anwendung sind in Software Engineering Standards Committee, IEEE Society [1998]; Scheidewind [1992] beschrieben.

7.1.1 Logische Codezeilen – Logical Lines of Code (LLOC)

Es gibt verschiedene Arten die Codezeilen (“Lines of Code” – LOC) zu zählen (siehe Wikipedia [2011]). Die LLOC zählt nur die Codezeilen, die tatsächlich Funktionalität implementieren (LLOC: Logical Lines of Codes).

Die LLOC der Vilango SpeechApp betragen 981 LLOC. Nach Klaeren [2011] schafft ein Programmierer zwischen 7–17 Zeilen Code pro Tag. Dabei ist zu beachten, dass die gesamte

Arbeitszeit immer das Planen, Designen, Implementieren und Testen beinhaltet. Die LLOC sind demnach ein Faktor, der herangezogen werden kann, um den geleisteten Aufwand zu messen.

Nach Code Analysis Team Blog [2011] ist ein Betrachten der LOC als Indikator für Qualität ein schlechter Ansatz, da die objektorientierten Indikatoren immer im Kontext zu den Codezeilen gesehen werden müssen.

7.1.2 Instandhaltung–Index – Maintainablity Index (MI)

Der Instandhaltung–Index wird von Code Analysis Team Blog [2011] in Visual Studio definiert. Die Methode um den MI in *Visual Studio* zu berechnen, ist definiert durch Welker [2001]; Bray et al. [1997]. Formel 7.1 definiert die Berechnung des Index. Dieselbe Formel verwendet das Code Analysis Team Blog [2011] in *Visual Studio* Codemetriken.

Der MI definiert auf einer Skala von 0–100 wie leicht es ist, die untersuchte Software zu betreuen. Dabei wird die Skala in einen grünen, einen gelben und einen roten Bereich geteilt. Höhere Werte sind besser. Grün entspricht einem höheren Wert. Tabelle 7.1 beschreibt die verschiedenen Wertebereiche. Die MI der Vilango SpeechApp sind in Tabelle 7.2 zusammengefasst.

MI	Bereich
0–9	Rot
10–19	Gelb
20–100	Rot

Tabelle 7.1: Definition der MI Werte in Visual Studio. Quelle: Code Analysis Team Blog [2011].

171 - 5.2 * ln(aveV) - 0.23 * aveV(g') - 16.2 * ln (aveLOC) - 50 * sin (sqrt(2.4 * perCM))

The coefficients are derived from actual usage (see Usage Considerations, pg. 232). The terms are defined as follows:

aveV = average Halstead Volume V per module (see Halstead Complexity Measures, pg. 209)

aveV(g') = average extended cyclomatic complexity per module (see Cyclomatic Complexity, pg. 145)

aveLOC = the average count of lines of code (LOC) per module; and, optionally

perCM = average percent of lines of comments per module

Abbildung 7.1: Formel für den Instandhaltung–Index (MI). Quelle: Bray et al. [1997].

Diese Auswertung impliziert, dass die Vialngo SpeechApp eine gute Instandhaltung verspricht.

7.1.3 Klassen Kopplung – Coupling Between Objecs (CBO)

Die Klassen–Kopplung wurde ursprünglich von Chidamber und Kemerer [1994] definiert. Sie beschreibt wie stark die Kohäsion zwischen den Objekten einer Implementierung ist. Ein nied-

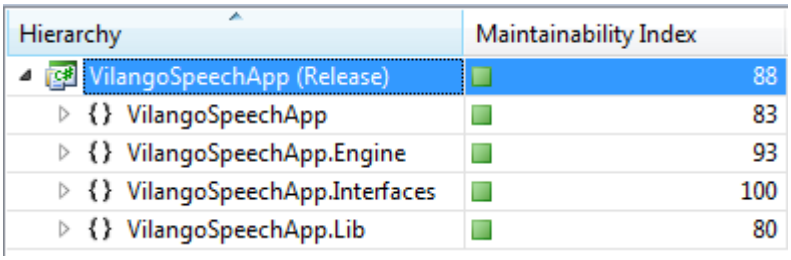


Abbildung 7.2: MI Werte der Vilango SpeechApp. Die MI–Werte sind Links im Bild.

riger Wert impliziert eine hohe Kohäsion und bessere Qualität. Das Code Analysis Team Blog [2011] beschreibt auf ihrem Blog eine Kopplung von Neun als optimal. *Visual Studio* gibt eine Warnung ab, wenn die CBO 30 pro Methode und 80 pro Klasse überschreitet. Zur Berechnung werden die einzelnen Klassen und ihre Abhängigkeit von anderen Klassen (einzigartiger Art) gemessen. Die Kopplung kann unter anderem durch Variablen, Methoden und weiteren Faktoren geschehen. Die genaue Berechnung ist auf Code Analysis Team Blog [2011] zu finden. Die CBO–Werte der Vilango SpeechApp sind in Tabelle 7.2 zusammengefasst. Die Werte werden pro Methode angegeben. Aus Platzgründen und der großen Anzahl der Werte sind wichtige Referenzwerte angegeben. Tabelle 7.2 fasst alle Werte der Vialango SpeechApp zusammen.

Wertdefinition	CBO
Maximale Kopplung einer Methode	14
Durchschnittliche CBO pro Methode	2.31
Durchschnittliche CBO pro Klasse	45.64

Tabelle 7.2: CBO–Werte der Vilango SpeechApp.

Eine hohe CBO ist ein Merkmal von schlechter objektorientierte Implementierung. Diese führt zu erhöhter Fehleranfälligkeit. Die Tabelle lässt den Schluss zu, dass die Vialngo SpeechApp objektorientiert ist. Die maximale Kopplung entsteht durch die starke Vernetzung der Grammatik, des Kontextes und der Dateioperationen beim Initialisieren des Spracherkenners. Die Methode ist im Kapitel 6 ausführlich beschrieben [Chidamber und Kemerer, 1994; Code Analysis Team Blog, 2011; Subramanyam und Krishnan, 2003].

7.1.4 McCabe–Metrik – Cyclomatic Complexity (CC)

Die McCabe–Metrik, oder auch *Cyclomatic Complexity* (CC), gibt an wie viele Entscheidungen in einem Codeabschnitt getroffen werden. Dadurch lässt sich ein Schluss ziehen, wie leicht oder wie schwer es wäre, diesen Abschnitt zu testen, oder den Fehler zu finden. Code Analysis Team Blog [2011]; Watson und McCabe [1996] gibt eine CC von Zehn als “magische Nummer” an, die sich als guter Wert bewährt hat. In Tabelle 7.3 sind die Werte der Vilango SpeechApp für die CC–Werte aufgeschlüsselt.

Der hohe Wert für die maximale CC wird durch die Methode, die den Auswahlalgorithmus implementiert verursacht. Es ist nicht gelungen aufgrund der vielen Auswahlparameter (sechs Parameter) eine übersichtliche und einfache Reduktion des Algorithmus durchführen. Da jeder Vergleich zu einer Erhöhung des CC um eins führt, und alle sechs Parameter untereinander verglichen werden müssen, fällt der CC dieser Methode sehr hoch aus.

Wertdefinition	CC
Maximale CC einer Methode	32
Durchschnittliche CC pro Methode	1.91
Durchschnittliche CC pro Klasse	37.71

Tabelle 7.3: CC–Werte der Vilango SpeechApp.

7.2 Funktionstests

Dieses Unterkapitel dokumentiert die Funktionstests, die im Rahmen des Erstellens der Software durchgeführt wurden. Das Gros der Tests stellen die *Unit Tests* dar. Dies ist aufgrund der Entwicklungsmethode, die in Kapitel 3 beschrieben ist, zustande gekommen. Die Funktionstests gliedern sich in *Unit Tests* und Abnahmetests [Myers et al., 2004].

7.2.1 NUnit – Unit Tests/Integrationstests

Die Fachliteratur beschreibt *Unit Tests* sehr ausführlich. *Unit Tests*, oder auch Modultests genannt, sind selektive Tests von Codeabschnitten. Aufgrund der durchgeführten Arbeitstechnik der Agilen Softwareentwicklung und des Test-geriebenen Entwickelns, sind viele erstellte Tests durch Verfeinerungen und Refactoring entweder zusammengelegt, oder überflüssig geworden. Die Tests wurden zuerst von kleinen funktionalen Tests zu Methodentests und in weiterer Folge Modultests zusammengefasst (Refacoring und das Konsolidieren von Tests, siehe: Beck [2002]). Die verbliebenen Tests decken, wie das Unterkapitel zur Testabdeckung mit NCover zeigen wird, das gesamte Programm und die Anforderungen, so wie sie in Kapitel 4 dokumentiert sind, ab. Die Tabellen 7.4, 7.5, und 7.6 führen alle Tests des Testkataloges auf [Beck, 2002; Beck und Andres, 2004; Astels, 2003; Link und Fröhlich, 2002; Levinson, 2011; Myers et al., 2004; Madeyski, 2010; Shore und Warden, 2007].

7.2.2 Abnahmetests

Abnahmetests stellen sicher, dass die definierten Anforderungen tatsächlich umgesetzt sind. Durch die Definition verschiedener Eingabe– und Benutzungsszenarien wird die Funktionalität innerhalb dieser Parameter sichergestellt [Myers et al., 2004].

Die Durchführung der Abnahmetests beinhaltet manuell durchgeführte Testfälle. Das Programm definiert folgende Testfälle für die Abnahme:

1. Durchführen einer ganzen oder partiellen Erkennung für Englisch.
2. Durchführen einer ganzen oder partiellen Erkennung für Deutsch.
3. Durchführen einer ganzen oder partiellen Erkennung für Französisch.
4. Durchführen einer Fehlerbehandlung aufgrund einer leeren Datei.
5. Durchführen einer Fehlerbehandlung aufgrund einer fehlerhaften Datei.
6. Durchführen einer Fehlerbehandlung aufgrund einer falsch formatierten Datei.

7. Durchführen einer ganzen oder partiellen Erkennung ab einer definierten Stelle in der Audiodatei und dem Text.

7.3 Systemtests

Nach Myers et al. [2004] sind Systemtests die am häufigsten miss–interpretierten Tests. Systemtests sollen laut Myers et al. [2004] das Programm und seine Implementation im Bezug auf ein technisch nicht definiertes Ziel hin testen. Es gibt keine standardisierte Methode um Testfälle für – zum Beispiel – Benutzerfreundlichkeit zu erstellen. Es kann aber die Aussage getroffen werden, ob ein Programm für die Zielgruppe benutzerfreundlich ist oder nicht [Myers et al., 2004; Andrews, 2011].

7.3.1 GUI

Die GUI ist möglichst einfach gehalten. Der überwiegende Teil der Arbeit soll nicht vom Benutzer, sondern von dem Programm erledigt werden. Der gesamte Arbeitsprozess ist in drei Schritten durchzuführen.

1. **Schritt 1:** Öffnen der Datei und Texteingabe.
2. **Schritt 2:** Drücken des Startknopfes.
3. **Schritt 3:** Öffnen der Ergebnisse.

Um die Benutzerfreundlichkeit zu testen wurden zwei Testpersonen eingesetzt. Das Programm soll in einem Softwareunternehmen eingesetzt werden, welches gleichzeitig mit Schauspielern für die Aufnahmen zusammenarbeitet. Das Feedback der Tester wurde im Programm umgesetzt. Die Profile der Tester waren:

1. **Testperson 1:** Weiblich, 22 Jahre alt, kein technischer Hintergrund, keine große Erfahrung mit Computern.
2. **Testperson 2:** Männlich, 28 Jahre alt, technischer Hintergrund, große Erfahrung mit Computern.

7.4 Stresstest, Robustheit und Testabdeckung

Das letzte Unterkapitel der Testdokumentation dokumentiert das Verhalten des Programmes unter Last, die Robustheit bezüglich Fehlern, und die erreichte Testabdeckung.

7.4.1 Stresstest

Da der Microsoft InProcRecognizer [2010] nur eine Instanz pro Prozess erlaubt, war die Anwendung von mehreren Threads und mehreren Instanzen des Erkenners nicht möglich.

Es wurde aber getestet, wie sich das Programm verhält, wenn die Last des Prozessors während der Erkennung hoch ist. Das Ergebnis der Untersuchung war schwer quantifizierbar.

Deswegen konnte keine genaue Aussage getroffen werden. Es kann aber gesagt werden, dass ein Erkennen unter hoher Last zu schlechteren Ergebnissen führt.

7.4.2 Testabdeckung mit NCover

Die Testabdeckung wurde mit NCover durchgeführt. Als Testabdeckung wird die Überprüfung der Programmlogik durch Modultests definiert. Die Abdeckung ergibt sich als Wert von ausführbaren Codezeilen zu tatsächlich ausgeführten Zeilen. Formel 7.7 stellt den Zusammenhang formal dar [Myers et al., 2004; Beck, 2002; Mutant Desing Ltd., 2011].

Ergebnisse der Analyse der Testabdeckung durch NCover

Die Fachliteratur beschäftigt sich häufig mit dem Wert der Testabdeckung. Nach Beck [2002]; Astels [2003]; Code Analysis Team Blog [2011]; Fowler et al. [1999] ist diese Frage folgend zu beantworten: Sowohl Beck [2002]; Astels [2003]; Fowler et al. [1999] sagen einheitlich, dass eine rigoros umgesetzte TDD–Methode zu 100%, oder beinahe 100% Testabdeckung führt. Es stellt sich auch die Frage, ob eine Abdeckung von *Setter–Getter*–Methoden für Variable notwendig ist. Dazu gibt es abweichende Meinungen [Code Analysis Team Blog, 2011].

Bei der Analyse der Testabdeckung wurden folgende Einschränkungen getroffen, die von *NCover* nicht automatisiert untersucht werden konnten:

- Es wurden *Getter–Setter*–Methoden nicht in die Berechnung mit einbezogen.
- Es wurden Pfade, die zwar getestet, aber von *NCover* nicht erkannt wurden, ausgenommen. Das Testen der Ereignisse wurde über *Mock–Objekte* realisiert. Diese decken einen großen Teil nicht erkannter Testpfade in isolierten Umgebungen ab. *NCover* erkennt nicht, dass ein serialisiertes Objekt dem Originalobjekt äquivalent ist. Diese Ergebnisse wurden manuell ergänzt.
- Es wurden GUI–Elemente, die nicht durch TDD entwickelt werden können, ausgeschlossen. *NCover* bezieht den *Namespace* der GUI–Elemente mit in die Berechnung ein. Diese Methoden sind aber für den Entwickler nicht erreichbar, noch können diese Implementiert werden (Windows–API–Methoden).
- Da der Microsoft InProcRecognizer [2010] *non–shared* und asynchron ist, konnte kein befriedigender Modultest geschrieben werden, der die Funktionalität unter Beweis stellt. Durch einen ausführlichen *Mock–Test* wurden allerdings alle Pfade der Ereignisse, die durch den Spracherkenner erzeugt werden, simuliert und getestet.

Abbildung 7.3 zeigt die erreichte Testabdeckung **nachdem** die Einschränkungen aufgrund des *Mock–Objektes* vorgenommen wurden. Im Anhang A unter “DVD–Material” werden beide Versionen aufgelistet.

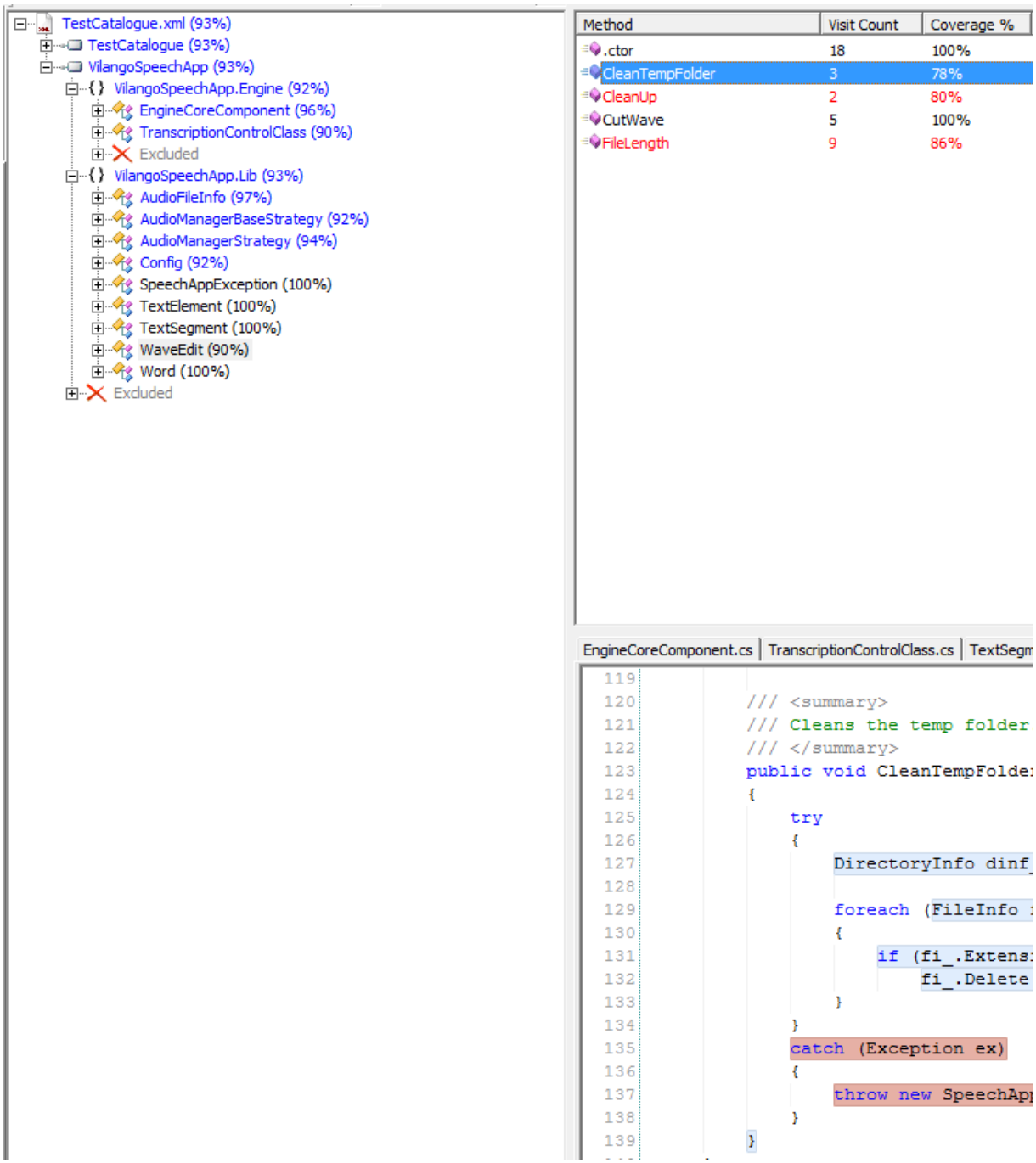


Abbildung 7.3: Erreichte Testabdeckung mit *NCover* ([Mutant Desing Ltd., 2011]).

InputTests

1. TestHandleInput
2. TestInvalidFileInput
3. TestTextParser
4. TestWrongInputParams

TestAudioFileInfoClass

1. TestConstructor
2. TestGetAudioDetails

TestAdudioManagerClass

1. TestAddingRcognizedSegments
2. TestAudiomanagerConstructors
3. TestChangingSettings
4. TestCompleteSimulatedRecognitionFromAcutalData
5. TestDecisionAlgorithmForSegmentSelection
6. TestGetRemainingTextAfterRecognition
7. TestInternalConfigSeggingsLoadFromFile
8. TestOffsetAllRecognizedSegmentsByValue
9. TestStringToSegmentConversions
10. TestXMLMarshallingAndOutput

TestTheSettingsClassAndSettingsFile

1. TestConstructors
2. TestAddAndCreateSetting
3. TestCreateSectionByCreation
4. TestGetSetting
5. TestGetSettings
6. TestSettingsChange
7. TestSettingsSaveToFile

Tabelle 7.4: Liste der Modultests, die Teil des Testkataloges zur Vilango SpeechApp sind.

TestTheTextSegmentClass

1. TestAddWords

TestTranscriptionControlClass

1. TestGetRecognizer
2. TestHandleInputAndRun
3. TestHandleInputAndRunFromPosition
4. TestHandleInputBadFile
5. TestRecognitionEngineSetup
6. TestResettingInternalSettings
7. TestRunMethod
8. TestSetOutputPath

TestTheEvents

1. TestEngineEvents
2. TestTranscriptionControlEvents

Tabelle 7.5: Liste der Modultests, die Teil des Testkataloges zur Vilango SpeechApp sind (fortgesetzt).

TestRecognizerEngine

1. TestConstructor
2. TestStartInProcEngine
3. TestInitialisiationOfEngine
4. TestCloseFile

TestWaveEditClass

1. TestCleanupOfTempFolderAtStart
2. TestConstructor
3. TestCuttingWaveMethod
4. TestGetFileLength

TestWordClass

1. TestPrivateMembers
2. TestSetTextMethod
3. TestStartTextMethod
4. TestWordConstructorLong
5. TestWordCOnstructorLongFailing
6. TestWordConstructorShort

Tabelle 7.6: Liste der Modultests, die Teil des Testkataloges zur Vilango SpeechApp sind (letzte).

$$Testabdeckung[\%] = \frac{TestedLLOC}{TotalLLOC} \tag{7.1}$$

Tabelle 7.7: Formel zur Berechnung der Testabdeckung. [Code Analysis Team Blog, 2011].

Kapitel 8

Schlußbemerkung und Ausblick

“ Learning is like rowing upstream: not to advance is to drop back. ”

[Chinese Proverb]

8.1 Schlussbemerkung

Während der Entwicklung der *Vilango SpeechApp* wurde, wie bereits diskutiert, Agile Softwareentwicklung angewendet. Ins besondere wurde der TDD–Ansatz forciert (*Test–Driven–Development*). TDD bietet große Vorteile im Bezug auf Flexibilität und schnell erstellter, robust funktionierender Software. Aber es wurden bei der Entwicklung der *Vilango SpeechApp* ebenso die Grenzen und Schwierigkeiten bei der Test–getriebenen Entwicklung sichtbar.

Den TF–Ansatz (*Test–First*) auf COM–Objekte ([MicrosoftCOM, 2011]) anzuwenden war eine große Herausforderung. Die traditionelle Literatur beschäftigt sich nicht mit COM–Objekten und TDD. Das *Mock–Objekt* scheint die Lösung darzustellen, aber schafft in diesem Fall große Probleme. Der Spracherkenner ist nicht–*shared*, asynchron, und Testumgebungen können das Objekt und die geworfenen Ereignisse nicht erkennen. Die Lösung, um dennoch den TF–Ansatz anzuwenden, war die Konstruktion einer Architektur, die mit serialisierten Ereignissen eine Ausgabe des Spracherkenners simuliert.

Es stellt sich die Frage, ob der geleistete Aufwand um den TF–Ansatz in diesem besonderen Fall anzuwenden, gerechtfertigt war. Nichts desto trotz hat sich TDD sehr gut bewährt. Nicht nur als Methode die hohe Qualität fördert, sondern auch aus psychologischer Sicht. Denn ganz besonders wenn mit COM–Objekten, oder generell Objekten die nicht im Einflussbereich des Entwicklers stehen, gearbeitet wird, ist es wichtig von der eigenen Implementierung überzeugt zu sein. TDD dürfte in der Zukunft noch bedeutender werden bei der Entwicklung von Software.

Während der Arbeit mit C# ergaben sich die größten Probleme durch die undurchsichtige Implementierung verwendeter Objekte aus der .NET–Bibliothek. Während der Programmierer bei der eigenen Implementierung stets weiß, wie die Implementierung umgesetzt wurde, so ist das im Fall von verwendeten *3rd–party* Bibliotheken nicht der Fall. Es ist eine große Zeitersparnis, aber auch ein Risiko, da Kontrolle abgegeben wird. TDD hilft Fehler früh zu erkennen, vor allem falsche Annahmen zu beseitigen, und schafft dadurch einen kleinen Raum, in dem Fehler passieren können.

In der Regel konnten kleine und große Fehler schnell gefunden, und ebenso schnell beseitigt werden.

8.1.1 Mögliche Anwendungsgebiete

Obwohl das Programm nur zur Transkription eingesetzt wird, so wäre es für weitere Einsatzgebiete tauglich. Zum Beispiel wäre eine Anwendung zur automatischen Bestellannahme in der Gastronomie denkbar. Es kommt auch ein Einsatz als Computer–Sprachsteuerung in Frage.

8.1.2 Stärken und Schwächen

Die Stärken liegen in der Robustheit, einer hohen Testabdeckung, und leichten Handhabung. Die Systemanforderungen sind dieselben Anforderungen, die Windows 7 stellt. Ein großer Vorteil ist, die leichte und flexible Erweiterbarkeit. Neue Funktionalität kann mit wenig Umstand ergänzt werden.

Die Schwächen sind eine fehlende Stapelverarbeitung und fehlende Möglichkeiten für das Übernehmen von externen XML–Schemas. Beide Funktionen können ohne große Schwierigkeiten implementiert werden.

Anhang A

Material

A.1 DVD–Material

Die DVD beinhaltet:

- Sourcecode, Installationsprogramm
- Testkatalog und Testdatensuite. (Benötigt *Visual Studio*) [Visual Studio, 2010].
- Codemetrik Analysedatei.
- Datei mit der Testabdeckung. Einmal modifiziert, einmal im Originalzustand.
- Latex–Sourcecode der Masterarbeit.

Literaturverzeichnis

Agile Alliance [2001]. *Manifesto for Agile Software Development*. http://agilemanifesto.org/. (Zitiert auf Seiten iii und 14.)

Allen, James [1987]. *Natural Language Understanding*. Erste Auflage. The Benjamin/Cummings Publishing Company. ISBN 0805303308. (Zitiert auf Seite 33.)

Amtrup, Jan Willers [1999]. *Incremental Speech Translation*. Erste Auflage. Springer Verlag. ISBN 3540667539. (Zitiert auf Seite 33.)

Andrews, Keith [2011]. *Human–Computer–Interfaces Lecture Notes*. http://courses.iicm.tugraz.at/hci/hci.pdf. (Zitiert auf Seite 61.)

Astels, David [2003]. *Test Driven Development: A Practical Guide*. Erste Auflage. Prentice Hall. ISBN 0131016490. (Zitiert auf Seiten 1, 15, 17, 19, 20, 21, 22, 23, 24, 60 und 62.)

Beck, Kent [2002]. *Test-Driven Development By Example*. Erste Auflage. Addison Wesley, Pearson Education. ISBN 0321146530. (Zitiert auf Seiten 1, 15, 17, 19, 20, 22, 23, 24, 29, 60 und 62.)

Beck, Kent und Cynthia Andres [2004]. *Extreme Programming Explained*. Zweite Auflage. Addison Wesley, Pearson Education. ISBN 0321278658. (Zitiert auf Seiten 1, 9, 10, 12, 13, 24 und 60.)

Bender, James und Jeff McWherter [2011]. *Professional Test Driven Development with C#*. Erste Auflage. Wiley Publishing, Inc. ISBN 047064320X. (Zitiert auf Seiten 1, 12, 15, 17, 19, 23 und 24.)

Birkenbihl, Verena F. [2004]. *T-Sprachen lernen, gehirn-gerecht Birkenbihl-Methode, Auszug*. *Birkenbihl Seminar-Handout*, 1(1), Seiten 194–205. http://www.birkenbihl.de/PDF/AuszugNeuesStrohImKopf.pdf. (Zitiert auf Seiten 1, 2 und 3.)

Bray, Michael, Maj David Luginbuhl, Kimberly Brune, William Mills, David A. Fischer, Robert Rosenstein, John Foreman, und Darleen Sadoski [1997]. *Carnegie Mellon – C4 Software Technology Reference Guide – A Prototype*. http://www.sei.cmu.edu/reports/97hb001.pdf. (Zitiert auf Seiten iii und 58.)

Carnegie Mellon University [2011]. *CMU Sphinx Open Source Toolkit For Speech Recognition*. http://www.speech.cs.cmu.edu/sphinx/. (Zitiert auf Seite 7.)

Carstensen, Kai-Uwe, Christian Ebert, Cornelia Endriss, Susanne Jekat, Ralf Klabunde, und Hagen Langer [2001]. *Computerlinguistik und Sprachtechnologie*. Erste Auflage. Spektrum Akademischer Verlag. ISBN 3827410274. (Zitiert auf Seiten 5 und 33.)

Chidamber, Shyam R. und Chris F. Kemerer [1994]. *A Metrics Suite for Object Oriented Design*. *IEEE Transactions on Software Engineering*, 20(6), Seiten 476–493. http://www.pitt.edu/~ckemerer/CK%20research%20papers/MetricForOOD_ChidamberKemerer94.pdf. (Zitiert auf Seiten 58 und 59.)

Chon, Mike [2010]. *Agile Softwareentwicklung: Mit Scrum zum Erfolg!* Erste Auflage. Addison-Wesley. ISBN 3827329876. (Zitiert auf Seiten 10 und 17.)

Code Analysis Team Blog [2011]. *Code Analysis Team Blog*. http://blogs.msdn.com/b/codeanalysis/. (Zitiert auf Seiten v, 58, 59, 62 und 66.)

Fowler, Martin, Kent Beck, John Brant, William Opdyke, und Don Roberts [1999]. *Refactoring: Improving the Design of Existing Code*. Erste Auflage. Addison Wesley. ISBN 0201485672. (Zitiert auf Seiten 19, 20, 21 und 62.)

Gales, Mark und Steve Young [2007]. *The Application of Hidden Markov Models in Speech Recognition*. *Foundations and Trends in Signal Processing*, 1(3), Seiten 195–304. doi:10.1561/2000000004. http://mi.eng.cam.ac.uk/~sjy/papers/gayo07.pdf. (Zitiert auf Seiten iii, 5, 6, 7 und 34.)

Gamma, Erich, Richard Helm, Ralph Johnson, und John Vlissides [2011]. *Gang Of Four*. http://c2.com/cgi/wiki?GangOfFour. (Zitiert auf Seite 12.)

Gamma, Erich, Richard Helm, und Ralph E. Johnson [1994]. *Design Patterns. Elements of Reusable Object-Oriented Software.* Erste Auflage. Addison-Wesley Longman. ISBN 0201633612. (Zitiert auf Seiten iii, 12, 49 und 52.)

Gamma, Erich und Bill Venners [2005]. *Erich Gamma on Flexibility and Reuse*. http://www.artima.com/lejava/articles/reuse2.html. (Zitiert auf Seiten 12 und 13.)

Hosom, John-Paul [2011]. *Automatic Speech Recognition with Hidden Markov Models*. http://www.cslu.ogi.edu/people/hosom/cs552/. (Zitiert auf Seite 5.)

Hosom, John-Paul, Ron Cole, und Mark Fanty [1999]. *Speech Recognition Using Neural Networks*. http://www.cslu.ogi.edu/tutordemos/nnet_recog/recog.html. (Zitiert auf Seiten 5, 6 und 7.)

HTK Toolkit [1989]. *Cambridge University Engineering Department*. http://htk.eng.cam.ac.uk/. (Zitiert auf Seite 7.)

Jeffries, Ron [2004]. *Extreme Programming Adventures in C#*. Erste Auflage. Microsoft Press. ISBN 0735619492. (Zitiert auf Seiten 2, 13, 15, 22, 23 und 24.)

Juang und Rabiner [1991]. *Hidden Markov Models for Speech Recognition*. *Technometrics*, 33(3), Seiten 251–272. http://www.jstor.org/discover/10.2307/1268779?uid=3737528&uid=2&uid=4&sid=47698953758577. (Zitiert auf Seite 34.)

Jurafsky, Daniel und James H. Martin [2000]. *Speech and Language Processing*. Erste Auflage. Prentice-Hall. ISBN 0130950696. (Zitiert auf Seiten 5, 6, 7 und 33.)

Klaeren, Herbert [2011]. *Skriptum Softwaretechnik*. http://pu.inf.uni-tuebingen.de/users/klaeren/sw.pdf. (Zitiert auf Seite 57.)

Lawrence und Rabiner [1989]. *A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition*. *Proceedings of the IEEE*, 77(2), Seiten 287–317. http://www.cs.cornell.edu/Courses/cs4758/2012sp/materials/hmm_paper_rabiner.pdf. (Zitiert auf Seite 5.)

Lechner, Martin [2010]. *Adaption of Agile Software Development Methods in Practice*. Dissertation, Graz University of Technology, Austria. (Zitiert auf Seite 9.)

Levinson, Jeff [2011]. *Software Testing with Visual Studio 2010*. Erste Auflage. Addison–Wesley. ISBN 0321734483. (Zitiert auf Seiten 57 und 60.)

Levinson, S. E., L. R. Rabiner, und M. M. Sondhi [1983]. *An Introduction to the Application of the Theory of Probabilistic Functions of a Markov Process to Automatic Speech Recognition*. *The Bell System Technical Journal*, 62(4), Seiten 1035–1074. http://www.alcatel-lucent.com/bstj/vol62-1983/articles/bstj62-4-1035.pdf. (Zitiert auf Seite 5.)

Lewis, William E. [2005]. *Software Testing and Continous Quality Improvement*. Zweite Auflage. Auerbach Publications. ISBN 0849325242. (Zitiert auf Seite 57.)

Link, Johannes und Peter Fröhlich [2002]. *Unit Tests mit Java, der Test-First-Ansatz*. Erste Auflage. dpunkt-Verlag. ISBN 3898641503. (Zitiert auf Seiten 1 und 60.)

Madeyski, Lech [2010]. *Test-Driven Development*. Erste Auflage. Springer-Verlag. ISBN 3642042872. (Zitiert auf Seiten iii, 1, 9, 10, 11, 15, 17, 18, 19 und 60.)

Markowitz, Judith und Bill Scholz [2010]. *Advances in Speech Recognition*. Erste Auflage. Springer. ISBN 1441959505. (Zitiert auf Seiten 1 und 33.)

Microsoft [2011]. *Windows 7 System Requirements*. http://windows.microsoft.com/systemrequirements. (Zitiert auf Seite 47.)

Microsoft InProcRecognizer [2010]. *SpRecognizer Interface*. http://msdn.microsoft.com/en-us/library/ee413260%28v=vs.85%29.aspx. (Zitiert auf Seiten 34, 36, 44, 61 und 62.)

Microsoft .NET [2011]. *.NET Framework 3.5*. http://msdn.microsoft.com/en-us/library/w0x726c2%28v=vs.90%29.aspx. (Zitiert auf Seiten 1 und 46.)

Microsoft SAPI [2011]. *Microsoft Speech API 5.4*. http://msdn.microsoft.com/en-us/library/ee125663%28v=vs.85%29.aspx. (Zitiert auf Seiten 1, 8, 33 und 34.)

Microsoft SDK [2011]. *Windows SDK*. http://msdn.microsoft.com/en-us/library/ms717422%28v=vs.90%29.aspx. (Zitiert auf Seiten 1, 8 und 47.)

MicrosoftCOM [2011]. *Component Object Model (COM)*. http://msdn.microsoft.com/en-us/library/windows/desktop/ms680573%28v=vs.85%29.aspx. (Zitiert auf Seiten 49 und 67.)

Mutant Desing Ltd. [2011]. *TestDriven.net*. http://www.testdriven.net/. (Zitiert auf Seiten iii, 39, 62 und 63.)

Myers, Glenford J., Tom Badgett, und Corey Sandler [2004]. *The Art of Software Testing*. Zweite Auflage. Wiley. ISBN 0471469122. (Zitiert auf Seiten 57, 60, 61 und 62.)

Nilsson, Mikael und Marcus Ejnarsson [2002]. *Speech Recognition using Hidden Markov Model*. Diplomarbeit, Blekinge Instutute of Technology. (Zitiert auf Seiten 4, 5, 6 und 7.)

NUnit.org [2009]. *NUnit Testsuite for .NET*. http://nunit.org/. (Zitiert auf Seiten iii, 25 und 39.)

Patton, Ron [2005]. *Software Testing*. Zweite Auflage. Sams Publishing. ISBN 0672327988. (Zitiert auf Seite 57.)

Paul, D.B [1990]. *Speech Recognition Using Hidden Markov Models*. *The Lincoln Laboratory Journal*, 3(1), Seiten 41–62. http://www.ll.mit.edu/publications/journal/journalarchives03-1.html. (Zitiert auf Seiten 5, 6, 7 und 34.)

Purdy, Doug und Jeffry Richter [2002]. *Exploring the Observer Design Pattern*. http://msdn.microsoft.com/en-us/library/ee817669.aspx. (Zitiert auf Seite 49.)

Salomon, David [2005]. *Foundations of Computer Security*. Erste Auflage. Springer London. ISBN 1846281938. (Zitiert auf Seite 55.)

Sandcasle Help File Builder Team [2011]. *Sandcasle Help File Builder*. http://shfb.codeplex.com. (Zitiert auf Seite 39.)

SAPI Audio Format [2010]. *Speech Audio Format Type*. http://msdn.microsoft.com/en-us/library/ee125189%28v=VS.85%29.aspx. (Zitiert auf Seite 35.)

Scheidewind, Norman F. [1992]. *Methodology For Validating Software Metrics*. http://www.cs.ucdavis.edu/~devanbu/teaching/289/Schedule_files/00135774.pdf. (Zitiert auf Seite 57.)

Schindler, Christian [2010]. *Review of Agile Software Development Methods in Practice*. Dissertation, Graz University of Technology, Austria. (Zitiert auf Seite 9.)

Shore, James und Shane Warden [2007]. *The Art of Agile Development*. Erste Auflage. O'Reilly Media. ISBN 0596527675. (Zitiert auf Seiten iii, 10, 12, 13, 14, 17 und 60.)

Software Engineering Standards Committee, IEEE Society [1998]. *IEEE Standard for a Software Quality Metrics Methodology*. http://cdn.bitbucket.org/cuatrorios/calidad-de-software/downloads/IEEE%201061%20%281998%29.pdf. (Zitiert auf Seite 57.)

Subramanyam, Ramanath und M. S. Krishnan [2003]. *Empirical Analysis of CK Metrics for Object–Oriented Design Complexity: Implications for Software Defects. IEEE Transactions on Software Engineering*, 29(4), Seiten 297–310. http://moosehead.cis.umassd.edu/cis580/readings/OO_Design_Complexity_Metrics.pdf. (Zitiert auf Seite 59.)

Team Audacity [2011]. *Audacity: Freier Audioeditor und Rekorder*. http://audacity.sourceforge.net/?lang=de. (Zitiert auf Seiten iii und 3.)

Vistual Studio Code Metrics [2010]. *Code Metrics Values*. http://msdn.microsoft.com/en-us/library/bb385914.aspx. (Zitiert auf Seite 57.)

Visual Studio [2010]. *Visual Studio 2010 Ultimate*. http://www.microsoft.com/visualstudio/en-us/products/2010-editions/ultimate/overview. (Zitiert auf Seiten iii, 39, 42 und 69.)

VoxForge [2011]. *VoxForge Open Source Speech Corpus*. http://voxforge.org. (Zitiert auf Seite 7.)

W3C [2011]. *Extensible Markup Language (XML)*. http://www.w3.org/XML/. (Zitiert auf Seite 1.)

Watson, A. H. und T. J. McCabe [1996]. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. NIST Special Publication 500–235*, 500(235), Seite 123. http://www.mccabe.com/pdf/mccabe-nist235r.pdf. (Zitiert auf Seite 59.)

Welker, Kurt D. [2001]. *The Software Maintainability Index Revisited*. http://staff.unak.is/andy/MScMaintenance0809/Lectures/Add/MIRevisited2001.pdf. (Zitiert auf Seite 58.)

Wikipedia [2011]. *Source lines of code*. http://en.wikipedia.org/wiki/Source_lines_of_code. (Zitiert auf Seite 57.)

Wolf, Henning und Wolf-Gideon Bleek [2010]. *Agile Softwareentwicklung*. Zweite Auflage. Dpunkt Verlag. ISBN 3898647013. (Zitiert auf Seiten 10 und 13.)

YAGNI [2006]. *You Ain't Gonna Need It*. http://c2.com/xp/YouArentGonnaNeedIt.html. (Zitiert auf Seite 22.)

Zhao, M., C. Wholin, N. Ohlsson, und M. Xie [1998]. *A Comparison between Software Design and Code Metrics for the Prediction of Software Fault Content*. *Information and Software Technology*, 40(14), Seiten 801–809. http://www.wohlin.eu/Articles/IST98-2.pdf. (Zitiert auf Seite 57.)