Universität des Saarlandes
Fachbereich 6.2 Informatik
66041 Saarbrücken

Diplomarbeit

# Erzeugung von Sprecherklassifikationsmodulen für multiple Plattformen

Michael Feld

2. März 2006

**1. Gutachter:**
Prof. Dr. Dr. h.c. mult. Wolfgang Wahlster

**2. Gutachter:**
Prof. Dr. Dietrich Klakow

**Wissenschaftlicher Betreuer:**
Dr.-Ing. Christian Müller

## Hilfsmittelerklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Diplomarbeit eigenständig und ohne fremde Hilfe verfasst habe. Ich habe dazu keine weiteren als die angeführten Hilfsmittel benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet.

Saarbrücken, den 2. März 2006

(Michael Feld)

## Danksagung

An erster Stelle möchte ich Herrn Prof. Wahlster herzlich danken, dass ich dieses interessante Thema hier am Lehrstuhl für Künstliche Intelligenz bearbeiten durfte. Mein ganz besonderer Dank gilt Herrn Dr. Christian Müller für die großartige Betreuung dieser Arbeit. Während der gesamten Zeit stand er immer für Fragen und Erklärungen zur Verfügung. Durch ihn habe ich das Thema der Sprecherklassifikation entdeckt und unter seiner Leitung an mehreren spannenden Projekten mitgearbeitet. Weiter danke ich Herrn Rainer Wasinger für die Unterstützung bei der Arbeit mit dem *ShopAssist*-Quellcode. Schließlich möchte ich auch meinen Eltern Roswitha und Josef Feld für das geduldige Korrekturlesen dieser Arbeit danken.

# Inhaltsverzeichnis

# 1 Einleitung

## 1.1 Motivation

Durch Fortschritte in den Bereichen der Informatik und Computerlinguistik ergeben sich neue Technologien, die Einsatz in der Praxis finden. Hierzu zählt die Sprecherklassifikation, deren Ziel ist es ist, allein auf Basis einer sprachlichen Äußerung Rückschlüsse auf Geschlecht, Alter, Laune, Unsicherheit, Nervosität oder weitere Merkmale des Sprechers zuzulassen. Das von Müller (2005) entwickelte AGENDER-Verfahren erlaubt es bereits heute, Alter und Geschlecht einer Person mit großer Genauigkeit zu klassifizieren. Daraus ergibt sich eine Reihe von möglichen Anwendungen.

### 1.1.1 Beispielszenarien für Sprecherklassifikation

#### Benutzeradaption auf mobilen Geräten

Geräte wie Mobiltelefone und *Personal Digital Assistants* (PDAs) stehen vor der Herausforderung, auf der einen Seite dem Benutzer Zugriff auf eine immer wachsende Anzahl an Funktionen bieten und auf der anderen Seite ein sehr großes Spektrum an Benutzern mit unterschiedlichen Anforderungen abdecken zu müssen. Können beispielsweise jüngere Benutzer mit einer großen Anzahl an Informationen und Funktionen, die vom Gerät auf einem Display angezeigt werden, noch gut umgehen, so bereitet dies älteren Menschen häufig Probleme, nicht zuletzt aufgrund möglicher Verschlechterungen der Sehkraft. Auch für Techniklaien stellen digitale Geräte eine besondere Herausforderung dar, die nur durch intelligente und intuitive Gestaltung der Geräte und Programme sinnvoll bewältigt werden kann und sollte, da ansonsten erhebliche Nachteile für die genannten Bevölkerungskreise entstehen können (vgl. Wahlster, 2000). Als wichtiges Hilfsmittel bei dieser Aufgabenstellung gilt die Benutzeradaption, d.h. die Anpassung des Gerätes an den Benutzer, gesteuert durch Software. Dies reicht von der Veränderung der Bildschirmdarstellung und Audioausgabe bis zur Aufbereitung eines individuellen Informationsangebots (s. Abb. 1.1). So lässt sich beispielsweise das Wissen, ob sich der Benutzer in einer lauten oder leisen Umgebung befindet, bei einer ganzen Reihe von mobilen Geräten dazu einsetzen, das Niveau der Tonausgabe (z.B. Hörerlautstärke und Klingelton des Handys) entsprechend zu regulieren.

Ähnliche Möglichkeiten ergeben sich durch die Sprecherklassifikation. Das Wissen

über das Alter des Benutzers lässt sich bei allen Geräten mit Bildschirm dazu einsetzen, die Lesbarkeit zu verbessern. Des Weiteren sind Alter und Geschlecht zwei gut geeignete Merkmale für *Content Customization* (vgl. Turpeinen, 2000), also um dem Benutzer möglichst personalisierte Dienstleistungen zu bieten. Bei den heute verbreiteten PDAs besteht hier zwar ein berechtigter Grund für den Einwand, dass diese Information durch einmalige explizite Eingabe schneller und zuverlässiger erhalten werden kann als durch ein Verfahren im Bereich der Sprachverarbeitung. Mit dem Einzug von portablen Geräten und digitaler Technologie im Alltag gibt es aber auch eine wachsende Zahl von Anwendungen, die mit wechselnden Benutzern zu tun hat. So wäre es z.B. als Serviceleistung an einem Flughafen denkbar, dass der Reisende am Terminal ein mobiles Gerät erhält, welches eine Funktionsweise wie das in Müller, Großmann-Hutter, Jameson, Rummer und Wittig (2001) beschriebene Flughafen-Navigationssystem besitzt. Dieses versorgt den Reisenden mit einer Wegbeschreibung zum Gate, Angaben zu Geschäften im Flughafengebäude und weiteren relevanten Informationen. Das Gerät wird dann beim Boarding wieder abgegeben.

Auch servergestützte Applikationen können von den in der Sprache enthaltenen Informationen profitieren, da hier oft gar keine Informationen über den Benutzer bekannt sind und eine explizite Eingabe keine praktikable Lösung darstellt. Ein sprachgesteuerter digitaler Einkaufsassistent (ähnlich dem in Abschnitt 4.1 beschriebenen M3I MOBILE SHOPASSIST), der automatisch beim Betreten des Ladens auf das mobile Gerät des Besuchers geladen wird, könnte in einem Bereich des Displays Sonderangebote präsentieren, die vom erkannten Geschlecht des Benutzers abhängen: Frauen würden z.B. bevorzugt Modeartikel präsentiert, während Männern Produkte aus der Elektroabteilung angeboten würden.

### Benutzeradaption an öffentlichen Terminals

An Terminals wie beispielsweise den Fahrkartenschalter am Bahnhof oder einen Geldautomaten werden ähnliche Anforderungen gestellt wie die oben bei den mobilen Geräten genannten: Sie müssen einer großen Zahl sehr verschiedener Benutzer Informationen anbieten, die auf optisch und akustisch geeignete Art präsentiert und ggf. personalisiert werden.

Der Kontext ist hier in der Regel weniger relevant, da Terminals an einen bestimmten Ort gebunden sind. Dafür wechseln die Benutzer aber meist noch häufiger als bei mobilen Geräten, so dass basierend auf dem Inhalt der Eingabe alleine manchmal der Benutzerwechsel für das System nicht einmal feststellbar ist. Auch hier können Zusatzinformationen über den Benutzer aus der Sprache wertvolle Hinweise geben.

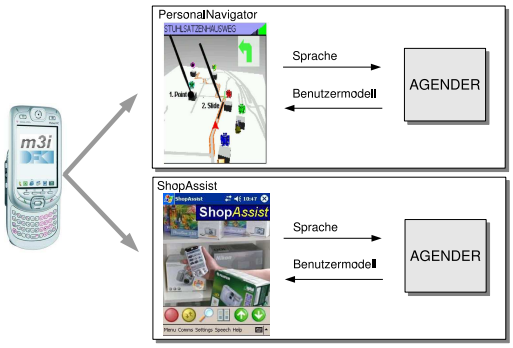



Abbildung 1.1: Benutzeradaption auf mobilen Geräten (vgl. Müller, 2005, S. 8)

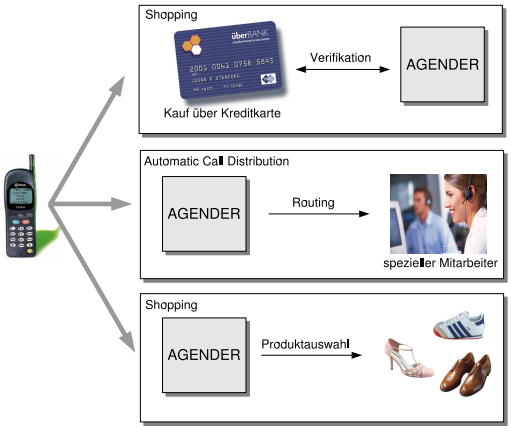



Abbildung 1.2: Sprecherklassifikation bei telefonbasierten Diensten (vgl. Müller, 2005, S. 8)

### Telefonbasierte Dienste

Auch klassische telefonbasierte Dienste wie Auskünfte und Service-Hotlines können von dem Hintergrundwissen über Alter und Geschlecht profitieren (s. Abb. 1.2). Ein mögliches Einsatzgebiet ist die *Automatic Call Distribution* (ACD), bei der versucht wird, den Anrufer aufgrund einer bereits ermittelten Einschätzung über Alter und Geschlecht an bestimmte Mitarbeiter oder Mitarbeitergruppen weiterzuleiten, die über zusätzliche Informationen oder Schulung im Umgang mit der betreffenden Personengruppe verfügen.

Bei vollständig sprachgeführten Dialogsystemen, z.B. zur Problemdiagnose und -lösung, ist es auch möglich, die Daten zur Adaption von Präsentation (Sprechgeschwindigkeit, Lautstärke u.a.) und Inhalt einzusetzen, wie zuvor schon unter „Benutzeradaption auf mobilen Geräten“ beschrieben. Eine weitere Möglichkeit besteht darin, die Information zur Verifizierung anderer Daten zu verwenden, beispielsweise um bei einer Zahlungstransaktion per Kreditkarte den Karteninhaber anhand des Geschlechts und Alters zu überprüfen. Bei fehlgeschlagener Prüfung kann der Anruf dann an einen Mitarbeiter zur weiteren Bearbeitung weitergeleitet werden.

Das Interesse der Telekommunikationsgesellschaften an den in diesem Abschnitt beschriebenen Einsatzmöglichkeiten hat in der jüngsten Zeit bedeutend zugenommen. Die in dieser Arbeit beschriebene Methode zur Erzeugung von Sprecherklassifikationsmodulen kommt derzeit bereits (unter ständiger Weiterentwicklung) in einem Projekt im Auftrag eines deutschen Telekommunikationsunternehmens zum Einsatz (vgl. insb. Ab-

schnitt 4.2).

### 1.1.2 Fazit

Neben den genannten drei Beispielen sind noch zahlreiche weitere Anwendungsmöglichkeiten denkbar. Es wird jedoch bereits deutlich, dass die Verfügbarkeit eines auf Sprache basierenden Verfahrens zur Bestimmung von Alter, Geschlecht und weiteren Eigenschaften des Benutzers einen hohen praktischen Nutzen aufweist. In dieser Arbeit soll daher eine Architektur vorgestellt werden, die es ermöglicht, Sprecherklassifikation für Anwendungen auf verschiedenen Plattformen unter Berücksichtigung der spezifischen Anforderungen verfügbar zu machen.

Grundlage der Architektur ist der von Müller (2005) vorgestellte AGENDER-Ansatz, welcher ein Verfahren zur Sprecherklassifikation beschreibt. Der Name setzt sich aus „Age“ (engl. für „Alter“) und „Gender“ (engl. für „Geschlecht“) zusammen, da Kernkomponenten des Verfahrens in der Tat Alter und Geschlecht sind, und weitere Informationen wie der Sprecherkontext (d.h. laute oder leise Umgebung) als supplementär angesehen werden. AGENDER ist Teil des Projektes M3I[1], welches in einer Gesamtarchitektur auch mögliche Anwendungen von AGENDER, Entwicklungs- und Analysewerkzeuge sowie Bibliothekskomponenten einschließt. M3I wiederum ist in das Rahmenprojekt COLLATE[2] eingegliedert, welches vom BMBF[3] gefördert und am DFKI[4] durchgeführt wird. Die einzelnen Bestandteile von M3I und ihre Zusammenhänge sind ausführlich in Müller (2005) beschrieben. Die zum Verständnis der Sprecherklassifikation und der in dieser Arbeit beschriebenen Architektur notwendigen Grundlagen werden ebenfalls in den folgenden Kapiteln behandelt. Dabei wird sich die vorliegende Arbeit auf die Merkmale Alter und Geschlecht des Benutzers beschränken, da diese für die aktuellen Projekte die größte Relevanz besitzen und auch in ihrer Entwicklung am weitesten fortgeschritten sind. Es ist jedoch durchaus möglich, alle Betrachtungen auf weitere Sprechermerkmale zu erweitern.

### 1.1.3 Bereiche mit Verbesserungspotential

In Müller (2005, S. 203ff) wird eine Client/Server-Architektur für AGENDER beschrieben, wobei für den Prototyp des Servers eine *Java*-basierte Anwendung entwickelt wurde. Diese kann entweder über eine Netzwerkverbindung mit einem Client kommunizieren und Klassifikationsanforderungen von ihm erhalten, oder auch in einem Stand-Alone-Modus ausgeführt werden. Bei jener Version steht die Demonstration der Sprachver-

[1] *A Mobile, Multi-modal and Modular Interface*, `http://www.dfki.de/~cmueller/`
[2] ***Co**mputational **L**inguistics and **La**nguage **Te**chnology for real-life applications*
[3] Bundesministerium für Bildung und Forschung
[4] Deutsches Forschungszentrum für Künstliche Intelligenz

arbeitung durch AGENDER und die Evaluierung verschiedener Klassifikationsmethoden im Vordergrund. Um jedoch den an eine industriell einsetzbare Architektur gestellten Anforderungen gerecht werden zu können, wurden diejenigen Bereiche identifiziert, die möglicherweise die nun veränderten Forderungen nicht uneingeschränkt erfüllen können oder in denen sich eine Überarbeitung des Systems vorteilhaft auswirken könnte.

### Geschwindigkeit

Die Optimierung der Laufzeit-Performanz war nicht Bestandteil des Entwurfs des ersten Prototypen des M3I-Servers. Durch die Wahl der Sprache *Java* als Grundlage für die Implementierung konnte zwar die Entwicklung flexibel gestaltet und zügig vorangetrieben werden, es wurde damit jedoch auch darauf verzichtet, den Code hardwarenah sowie compilerbedingt optimiert und damit schnell ausführen zu können, wie das beispielsweise bei einer Implementierung in der Programmiersprache *C* der Fall gewesen wäre. Stattdessen wird der M3I-Server in einer virtuellen Maschine ausgeführt, deren nachteiliger Einfluss auf die Geschwindigkeit deutlich zum Tragen kommt. Des Weiteren kann eine Verarbeitung in Echtzeit nie erreicht werden, da die virtuelle Maschine stetig Aufgaben im Hintergrund ausführt, allen voran die Speicherbereinigung (*Garbage Collection*[5]), die zu nicht vorhersagbaren Zeitpunkten die eigentliche Verarbeitung kurzfristig verlangsamen oder sogar anhalten kann.

Unabhängig von der Wahl der Sprache wurden aber auch beim Code selbst viele Gelegenheiten zur Optimierung ausgelassen, da sie für den Prototyp ungeeignet oder nicht relevant waren. Genannt sei hier die Wahl einer *Blackboard-Architektur* (vgl. Abschnitt 3.3.3) für die interne Kommunikation im Server. Diese bietet zwar eine große Flexibilität, die für die Entwicklung auch nötig war, jedoch auf Kosten einer schnellen Verarbeitung. Von der Industrie aber werden schnelle Reaktionszeiten gewünscht und sind bei manchen Projekten sogar wesentliche Voraussetzung.

### Integrierbarkeit

Der M3I-Server ist eine eigenständige Anwendung (*Executable*), die über eine grafische Oberfläche (GUI, *Graphical User Interface*) gesteuert und visualisiert werden kann. Um mit AGENDER eine Äußerung klassifizieren zu können, muss zuerst der Server gestartet und dann die Äußerung übergeben werden. Nach dem Start kann ein Client auch über eine TCP/IP-Verbindung eine Klassifikation einleiten.

Bei diesem Ansatz stellt die Netzwerkschnittstelle die einzige Möglichkeit zur Steuerung der Klassifikation durch eine externe Anwendung dar. Eine Programmierschnittstelle für Anwendungen (API, *Application Programming Interface*) oder die Möglichkeit zur Funktionsintegration (vgl. Hergula, 2003) von AGENDER ist nicht vorhanden. Eine

[5] Freigabe von Speicher, welcher von der Anwendung nicht mehr benötigt wird

M3I-Anwendung auf einem mobilen Gerät könnte beispielsweise über ein Funknetzwerk mit dem Server verbunden sein und dessen Dienste nutzen. Da eine Funkverbindung aber ein inhärentes Stabilitätsproblem birgt und zeitweise ausfallen kann, wäre es wünschenswert, in diesem Fall die Klassifikation auf dem mobilen Gerät fortzusetzen – unter Umständen mit eingeschränkter Qualität. Dazu müsste die Klassifikation aber auf dem Gerät transparent gestartet und beendet werden können (einmal abgesehen von weiteren Fragen wie der der Portierbarkeit, vgl. „Plattformunabhängigkeit / Portierbarkeit“ weiter unten).

### Modularität

Verschiedene Anwendungen erfordern möglicherweise vollkommen unterschiedliche Konfigurationen des AGENDER-Systems. So könnte es der Fall sein, dass eine Anwendung zur Anpassung der Schriftgröße an den Benutzer einfach zwischen den Benutergruppen „jung“ und „alt“ unterscheidet, wobei „alt“ Personen über 65 Jahre bezeichnet. Eine andere Anwendung benötigt aber unter Umständen eine feinere Einteilung in drei oder mehr Altersklassen. In einem dritten Fall könnte es sich herausstellen, dass für die konkrete Datenbasis ein bestimmter Klassifikationsalgorithmus besser funktioniert als ein anderer. Da bei der bestehenden Architektur solche Umstellungen nur mit einer Änderung im Quellcode des Servers zu bewerkstelligen sind, treten hier große Probleme bei der parallelen Entwicklung von mehreren Projekten mit unterschiedlichen Anforderungen auf, da die Codebasis sich vervielfacht und alle Änderungen manuell abgeglichen werden müssen. Dabei unterscheiden sich tatsächlich nur die Konfiguration und die Klassifizierungsmodule. Anstatt für jeden Anwendungsfall eine eigene Version des M3I-Servers zu verwenden ist es offenbar praktischer, nur eine Konfigurationsdatei anzulegen und mit den darin enthaltenen Informationen dann ein auf den konkreten Fall zugeschnittenes *Modul* (im Sinne von Parnas, 1972) zu erzeugen.

### Plattformunabhängigkeit / Portierbarkeit

Ein weiteres Problem des M3I-Servers liegt in seiner mangelnden Portierbarkeit. Die Wahl von *Java* zur Implementierung der Applikation legt zwar auf den ersten Blick eine gute Portierbarkeit nahe, und tatsächlich war dies auch ein wichtiger Grund für die Wahl der Sprache. Die Praxis hat aber gezeigt, dass gerade für die Plattformen, für die eine eigenständige AGENDER-Komponente von großem Interesse wäre, wie z.B. *Windows CE* bzw. *Windows Mobile*, welches auf den neueren *PocketPC*-Geräten läuft, keine Unterstützung für *Java* vorhanden ist oder diese mit nicht vertretbaren Performanzeinbußen verbunden ist. Eine Implementierung in *C/C++* ist zwar nie wirklich plattformunabhängig, dafür kann aber häufig mit mäßigem Aufwand eine optimal an die Zielplattform angepasste Version erzeugt werden, da es einen entsprechenden Compiler für fast alle

Plattformen gibt.

Neben der Implementierungssprache besitzt der Server zudem auch noch einige konzeptuelle Gegebenheiten, die ihn zunächst ungeeignet machen für eine Portierung. So werden u.a. externe Programme für die Vorverarbeitung der Audiodateien und zur Merkmalsextraktion verwendet, die auf anderen Plattformen nicht zur Verfügung stehen. Als problematisch erweisen sich in diesem Fall auch vom Server initiierte *SSH*-Verbindungen[6], die für die Verteilung der Verarbeitungsaufträge an mehrere Rechner in einem Cluster verwendet werden.

### Skalierbarkeit

Der Server wurde entwickelt mit einer relativ festen Demonstrations-Konfiguration im Blick, nämlich mit einem durchschnittlichen PC zur Ausführung und Steuerung des Programms, einem Cluster aus zehn Rechnern zur verteilten Durchführung der rechenintensiven Vorgänge, sowie einem *PocketPC* als Client.

Für industrielle Anwendungen genügt diese Dimensionierung häufig nicht. In einem größeren Callcenter müssen evtl. mehr als hundert Äußerungen pro Sekunde verarbeitet werden. Dies erfordert nicht nur eine schnelle Implementierung seitens des Servers, sondern auch weit reichende Überlegungen im Hinblick auf Parallelisierbarkeit der Anfragen und Speicherverwaltung. Auf der anderen Seite soll es aber auch möglich sein, die Sprecherklassifikation auf Plattformen mit wesentlich geringerer Leistungsfähigkeit (z.B. einem *Smartphone*) durchzuführen, wobei evtl. je nach Anwendung ein Kompromiss zwischen Geschwindigkeit und Klassifikationsgenauigkeit eingegangen werden muss.

Die genannten Beschränkungen des M3I-Prototyps rechtfertigen den Entwurf eines neuen Ansatzes, um AGENDER als eingebettete Komponente für verschiedene Anwendungen und auf multiplen Plattformen verfügbar zu machen. Dabei soll es sich um ein abgeschlossenes Konzept handeln, das nicht nur die Frage der Einbindung der Sprecherklassifikation in Anwendungen behandelt, sondern den gesamten Prozess einschließlich der Analyse der Sprachkorpora, dem Training von Klassifizierern und der Evaluierung neuer Klassifikationsalgorithmen bis zur Erzeugung eines verteilbaren Klassifikationsmoduls für eine gewünschte Zielplattform betrachtet. Darüber hinaus wird durch die konkrete Mitwirkung an Projekten, welche diese Module nutzen, eine Rückmeldung über Möglichkeiten zur Verbesserung von Klassifizierer und Anwendung geschaffen, die sich schon mehrfach als sehr nutzbringend erwiesen hat. Zur besseren Unterscheidung vom früheren M3I wird dieses neue Konzept als SBC (*Multiple Platform Speaker Classification*) bezeichnet, obgleich es sich um eine konsequente Weiterentwicklung der bisherigen Architektur handelt.

[6] *Secure Shell*: Verbindung zur Remotesteuerung von Rechnern über eine Konsole.

## 1.2 Gliederung

Diese Arbeit enthält eine implementierungsnahe Beschreibung des SBC-Konzepts. Sie ist ergänzend zu den in Müller (2005) entwickelten konzeptuellen Grundlagen gedacht. Auch ohne Studie der genannten Quelle kann sie aber als Dokumentation von SBC herangezogen werden.

Im folgenden Kapitel werden zunächst die Grundlagen der Sprecherklassifikation mit AGENDER und das verwendete Zwei-Ebenen-Modell erläutert. Das Verständnis der Funktionsweise ist wichtig, um später die Implementierung verstehen und die Gründe für diverse die Architektur betreffende Entscheidungen nachvollziehen zu können. Das Kapitel soll auch das nötige Hintergrundwissen aus dem Bereich der Phonetik, Mustererkennung und Stochastik vermitteln, da hier die meisten der in AGENDER verwendeten Methoden angesiedelt sind.

Aufbauend darauf wird im dritten Kapitel die SBC-Architektur im Detail vorgestellt. Dabei wird auf die Umsetzung der allgemeinen Anforderungen sowie die Wahl und Implementierung spezieller Verfahren eingegangen. Für die beiden Hauptkomponenten der Architektur zur Ausführung der Sprecherklassifikation und dem Entwurf von Sprecherklassifikationsmodulen wird eine umfassende Beschreibung des Aufbaus und der Funktionsweise gegeben.

In Kapitel 4 werden dann zwei prominente Beispiele zur Verwendung der zuvor entwickelten Klassifikationsmodule präsentiert. Dabei wird nicht nur der Nutzen der Technologie veranschaulicht, sondern auch die Integration in verschiedene Szenarien aus technischer Sicht demonstriert.

Kapitel 5 enthält einige statistische Daten zu SBC und AGENDER, welche einen Überblick über den aktuellen Stand des Verfahrens im Hinblick auf Klassifikationsgenauigkeit und Geschwindigkeit geben sollen. Im Vordergrund stehen hierbei mobile Plattformen aufgrund ihrer besonderen Ressourcenansprüche.

Die letzten beiden Kapitel schließen die Arbeit mit einer kurzen Zusammenfassung und einem Ausblick ab.

## 1.3 Wissenschaftliche Fragestellungen

Im Folgenden werden die zentralen Problemstellungen zusammengefasst, die Bestandteil der vorliegenden Arbeit sind und in den weiteren Kapiteln ausführlich behandelt werden.

**Welche Anforderungen werden an ein System gestellt, welches flexible Sprecherklassifikation mit AGENDER für industrielle Anwendungen verfügbar machen soll?**

In Abschnitt 3.1 wird ein Katalog von Vorgaben formuliert, welche die SBC-Architektur erfüllen muss. Weiterhin werden optionale Merkmale aufgeführt, die zu einer möglichst leistungsfähigen und vielseitig einsetzbaren Komponente beitragen.

**Welche Bestandteile beinhaltet die Architektur und wie arbeiten diese zusammen?**

Die Umsetzung der beschriebenen Architektur erfordert eine Reihe von Komponenten, die in unterschiedlichen Szenarien und Umgebungen zum Einsatz kommen. Zwischen ihnen findet eine multimodale Kommunikation statt, die in Abschnitt 3.2 grob skizziert und in den weiteren Abschnitten eingehend behandelt wird. Der Aufbau der Komponenten ist vor allem in Abschnitt 3.7 und 3.9 sehr ausführlich dokumentiert.

**Wie lassen sich vorhandene Technologien und Werkzeuge für SBC nutzen oder verbessern?**

Da das SBC-Projekt auf bereits existierenden Lösungen wie M3I basiert, ist die Nutzung dieser Werkzeuge und der bereits erzielten Ergebnisse von besonderem Interesse. Dies kann die Entwicklung beschleunigen, die Qualität steigern und zudem die Kompatibilität zu bestehenden Systemen und vorhandenem Datenmaterial verbessern.

**Wie lässt sich SBC in Anwendungen integrieren und zur Entwicklung von Klassifikationsmodellen nutzen?**

Durch die praxisbezogene Ausrichtung der Arbeit ist dieser Aspekt von großer Bedeutung. In Abschnitt 3.8 werden die Schnittstellen vorgestellt, die entworfen wurden, um eine einfache aber dennoch flexible Einbettung des Moduls in Applikationen zu ermöglichen. Die notwendigen Schritte zur Entwicklung von Klassifizierern können in Abschnitt 3.9 nachgelesen werden.

## 1.4 Verwandte Arbeiten

Im Bereich der maschinellen Verarbeitung von Sprache lassen sich zahlreiche Arbeiten finden. Da im Kern der vorliegenden Arbeit der Entwurf und die Implementierung einer Architektur stehen, sollen in diesem Abschnitt auch nur Quellen mit ebensolcher Fragestellung betrachtet werden. Dafür wird allerdings die Sicht nicht auf Sprecherklassifikation oder Spracherkennung eingeschränkt, sondern es werden bewusst ähnliche Problemstellungen in anderen Anwendungsgebieten vorgestellt, welche jedoch alle eine Mustererkennung oder andere Machine Learning-Verfahren implementieren.

### Multiagentensystem für Handschrifterkennung

Heutte, Paquet, Nosary und Hernoux (2000) beschreiben in ihrer Arbeit eine Erweiterung zu traditionellen Handschrifterkennungsmethoden. Diese betrachten neben den *omni-writer*-Methoden, deren Ziel es ist, sich auf möglichst jede Handschrift einzustellen, indem der Erkennung die gemeinsamen Merkmale zugrunde gelegt werden, auch die *mono-writer*-Methoden. Letztere versuchen auch Eigenheiten der Schrift einzelner Schreiber zu erkennen, um insgesamt eine höhere Erkennungsrate zu erreichen. So genannte *Schreiber-Invarianten*, welche typische Muster für die Schrift einer bestimmten Person umfassen, können zur Disambiguierung von erkannten Inhalten benutzt werden. So könnte beispielsweise „$\ell$“ sowohl als $e$, als auch als $l$ erkannt werden. Ist jedoch bekannt, dass die betreffende Person ein $e$ immer als „$e$“ schreibt, so ist die Erkennung in diesem Fall eindeutig. Der Erkennungsvorgang selbst ist ein Mustererkennungsproblem.

Heutte et al. (2000, S. 414) teilen die an der Handschrifterkennung beteiligten Strategien in drei Kontextebenen ein: *grafisch*, *symbolisch* und *lexikalisch*. Sie kommen zu dem Schluss, dass diese Vorgänge nicht sequenziell sind, sondern voneinander abhängen und in einer unbestimmten Reihenfolge ausgeführt werden können, da das Ergebnis eines Vorgangs die qualitative Performanz eines anderen Vorgangs positiv beeinflussen kann. Für die Wahl der Architektur bedeutet dies unter anderem, dass eine *Pipeline* (vgl. Abschnitt 3.3.3) hier nicht angewandt werden kann. Der Blick der Autoren wurde zunächst auf eine *Blackboard*-Architektur gelenkt, bei der sich allerdings die Wahl der Wissensquellen als problematisch herausstellte. Den Autoren schienen letzten Endes *Multiagentensysteme* am besten für die Lösung dieses Problems geeignet. Im Zentrum des Ansatzes aus der Verteilten Künstlichen Intelligenz stehen dabei autonome Einheiten (*Agenten*), die mit anderen Agenten oder festgelegten Gruppen von Einheiten kommunizieren können und somit unabhängig voneinander zur Lösung eines Gesamtproblems beitragen. Die in Heutte et al. (2000, S. 417) vorgeschlagene Implementierung der Architektur trägt die Bezeichnung *EMAC* und ist wie das SBC-Klassifikationsmodul (vgl. Abschnitt 3.3.1) in *C++* geschrieben. Um das Datenaufkommen bei der Kommunikation zwischen den Agenten zu reduzieren, werden spezielle *Broker*-Einheiten verwendet, die stellvertretend für eine feste Gruppe von Agenten Wissen verwalten, d.h. als gemeinsamer Speicher dienen. Als Plattform wird *PVM* eingesetzt.

### Framework zur Anrufweiterleitung in Callcentern

Aus der Zusammenarbeit von *IBM* und der *Carnegie Mellon University* ist das in Wu, Lubensky, Huerta, Li und Kuo (2003) beschriebene Framework entstanden, welches sich wie die in Abschnitt 4.2 entwickelte Anwendung der Problemstellung einer automatischen Anrufweiterleitung in einem Callcenter annimmt. Anders als bei AGENDER stellen nicht Sprechermerkmale die Grundlage für die Klassifizierung eines Anrufs dar, sondern

der Inhalt einer natürlichsprachigen Äußerung, die als Antwort auf eine vorgegebene Eingabeaufforderung gegeben wird. Dies erfordert in jedem Fall die Einbeziehung einer Komponente zur Spracherkennung in die Architektur.

Das Framework sieht folgende Architektur vor: Die Telefonieplattform wird durch *IBM WebSphere Voice Response* bereitgestellt. Dieses verwaltet die Verbindungen zu unterschiedlichen Endgeräten wie Mobiltelefonen und Breitband-Internet. Ein *Interactive Voice Response* (IVR) Server steuert den Dialog und kann mittels einer speziellen Schnittstelle (*WebSphere Voice App Access*, WVAA) auf *WebSphere* zugreifen. Zur Spracherkennung kann auf dem gleichen Server *IBM ViaVoice* ausgeführt werden und ggf. auch eine *IBM*-Lösung für die Sprachausgabe. Die noch fehlende Komponente sind die Klassifizierer. Diese können nach Wu et al. (2003) beliebig implementiert sein und müssen lediglich eine bestimmte Schnittstelle unterstützen. Bei der Evaluierung wurde ein in *C*, als auch ein in *TCL* (*Tool Command Language*) geschriebener Klassifizierer verwendet.

Besondere Bedeutung hat für Autoren eine gute Skalierbarkeit der Systems. Deutliche Parallelen zu SBC sind bei der Modularität der Architektur betreffend die Auswahl von Klassifizierern zu erkennen (vgl. Abschnitt 3.1.3). Vom prinzipiellen Aufbau sind auch große Ähnlichkeiten zu dem in Abb. 4.4 auf Seite 133 dargestellten Modell sichtbar. Wesentlicher Unterschied sind die jeweiligen Software-Implementierungen und die Klassifizierungsgrundlage. Daneben ist die Tatsache zu berücksichtigen, dass SBC nicht auf Telefonie-Applikationen beschränkt ist.

Ein ähnlicher Ansatz, welcher die linguistischen Aspekte der Sprache betrachtet, wird in Kneissler, Kienappel und Klakow (2003) verfolgt. Hierbei stehen die *Information Retrieval* Methoden im Vordergrund, die verwendet werden, um einen Anrufer basierend auf einer Äußerung an den am ehesten zutreffenden Dienst weiterzuleiten.

### Eingebettete Klassifizierung in Smart Cameras

*Smart Cameras* sind „intelligente“ Videokameras, welche Aufnahmeoptik mit digitaler Videoverarbeitung und einer Kommunikationsebene verbinden und alle Vorgänge eingebettet in einem Gerät durchführen. Bramberger, Brunner, Rinner und Schwabach (2004) beschreiben ein Szenario, in dem solche Kameras für die Verkehrsüberwachung eingesetzt werden. Sie können beispielsweise dabei helfen, Unfälle zu erkennen, Geisterfahrer zu identifizieren oder den Verkehrsfluss zu überwachen. Die angesprochene Arbeit beschreibt einen Prototypen, der durch Verwendung entsprechender Mustererkennungsmethoden dazu in der Lage ist, stationäre Fahrzeuge zu erkennen.

In Bramberger, Rinner und Schwabach (2004) wird eine passende Hard- und Softwarearchitektur zu diesen *Smart Cameras* vorgestellt und implementiert. Bei der Hardwareplattform handelt es sich um Digitale Signalverarbeitungsprozessoren (DSPs) der

Marke *MIPS* mit ca. 600 MHz, von denen jede Kamera 2 bis 4 Stück enthält. Daneben werden noch Sensoren und eine Kommunikationseinheit benötigt, welche Bilder und Daten über Ethernet, WLAN oder GPRS übertragen kann. Dies kann etwa mit der Architektur von serverbasierter Klassifikation auf mobilen Geräten verglichen werden (vgl. Abschnitt 4.1.1). Auch hier besteht die Möglichkeit, die Daten in einem Kontrollzentrum oder lokal auf der Kamera mit Hilfe eines der DSPs zu verarbeiten, wobei in modernen Systemen mit zunehmender Komplexität die zuletzt genannte Variante meist die geeignetere ist.

Im Unterschied zu SBC erfolgt die Verarbeitung bei *Smart Cameras* immer in Echtzeit. Die Klassifikation von Fahrzeugen stellt eine von vielen möglichen Aufgaben dar, welche die Kamera erfüllen kann. Ein weiteres Beispiel ist die Videokompression. Für jede solche Aufgabe wird - ähnlich wie bei SBC (vgl. Abschnitt 3.3.4) - ein Software-Modul erzeugt. Das Laden des Moduls auf die Kamera kann dynamisch erfolgen, da sich die Rolle einer Kamera während des Betriebs ändern kann. Zu jeder Zeit kann ein DSP jedoch nur ein Modul ausführen, welches aus einer einzigen Executable besteht, die statisch mit dem Betriebssystem verknüpft ist (vgl. Abschnitt 3.5 Seite 59). Zum Zugriff auf Betriebssystem-Funktionen und zur Kommunikation mit Arbeitsstationen dienen Frameworks, welche im Speicher der DSPs installiert sind. Sie implementieren unter anderem einen *Ressourcen-Manager*, der deutlich über die von SBC bereitgestellte Ressourcenverwaltung hinausgeht und echt adaptiv ist, d.h. bei Engpässen wie wenig freiem Arbeitsspeicher entsprechende Maßnahmen ergreift. Dies ist aber auch notwendig, da *Smart Cameras* als Echtzeit-Anwendungen besonders strikten Anforderungen an die Reaktionszeiten unterliegen.

Bramberger, Rinner und Schwabach (2005) erweitert die dargestellte Architektur um einen Mechanismus, um verteilte ressourcenadaptive Verkehrsüberwachung zu ermöglichen. Hierbei weist ein so genanntes *Überwachungscluster* den über ein Netzwerk verbundenen *Smart Cameras* mit Hilfe eines in Bramberger et al. (2005) beschriebenen Algorithmus dynamisch Aufgaben zu, z.B. je nach Verkehrsaufkommen. Besagte Algorithmen wurden zuerst in *Java* implementiert, dann aber aus Performanzgründen nach *C++* übertragen, was somit auch die für SBC getroffene Entscheidung bestätigt (vgl. Abschnitt 3.6).

### Verteilte eingebettete Architektur für den Einsatz im Gesundheitswesen

Eine proaktive Lösung zur gesundheitlichen Vorsorge insbesondere für ältere Menschen stellen Sung und Pentland (2004) mit *LiveNet* vor. Dieses System soll dazu dienen, den Gesundheitszustand und Lebensstil eines Menschen zu überwachen, die gewonnenen Daten zu speichern und wiederzugeben und auf intelligente Art und Weise Warnungen und Hilfestellungen bei potenziellen Risiken zu geben.

Auf der Akquisitionsseite stehen *Sensoren*, welche am Körper getragen werden verschiedene Biowerte messen, z.B. durch Beschleunigungsmesser, EKG/EMG, Pulsmessgerät, galvanische Hautreaktion, Mikrofon und Thermometer. Diese Sensoren sind über einen Sensorhub mit einem PDA verbunden.

Für die Implementierung von *LiveNet* wurde das *MITThril Framework* (vgl. DeVaul, Sung, Gips und Pentland, 2003) eingesetzt. Dieses enthält neben anderen Komponenten eine API und eine Kontext-Engine. Die so genannte *Enchantment API* ist als *Whiteboard*-Architektur implementiert und bietet der Anwendung die nötige Infrastruktur für Datenaustausch und -verarbeitung. Bei dem Whiteboard handelt es sich um eine Weiterentwicklung der Blackboard-Architektur, bei der Whiteboard-Clients Daten an einen Whiteboard-Server senden. Gegenüber einem Blackboard weist es eine Reihe zusätzlicher Merkmale auf, z.B. Nachrichten-Abonnements (*subscribing*), Whiteboard-Sperren (*locking*), symbolische Verknüpfungen über Servergrenzen hinweg, Streaming und ein effizientes Signalsystem. *Enchantmant* fungiert als Datenbank und kann zu jeder Zeit den aktuellen Zustand einer Person oder Personengruppe erfassen.

Die *MITThril* Kontext-Engine aus DeVaul et al. (2003, Abschnitt 4.3) führt folgende vier Schritte durch (vgl. auch Abschnitt 2.2.1): Sensordatenaufnahme, Merkmalsextraktion, Modellierung (⇒ Klassifizierung) und Inferenz (⇒ Nachverarbeitung). Bei der Modellierung können analog zu AGENDER verschiedene Verfahren genutzt werden.

# 2 Grundlagen der Sprecherklassifikation

Die Entwicklung von AGENDER beruht auf interdisziplinärer Forschung in mehreren Bereichen, unter anderem der Phonetik, der Mustererkennung und der Wahrscheinlichkeitstheorie. Dieses Kapitel enthält keine Einführungen in die genannten Fachgebiete, sondern soll vielmehr die für diese Arbeit relevanten Informationen aus den jeweiligen Bereichen vermitteln.

## 2.1 Phonetische Grundlagen

Die Wissenschaft der *Phonetik* befasst sich mit den physikalischen Aspekten der Lautbildung durch die menschliche Stimme (vgl. Ladefoged, 2001). Innerhalb dieser Disziplin können wiederum mehrere Teilgebiete unterschieden werden. Der akustischen Phonetik kommt bezüglich der Sprecherklassifikation die größte Bedeutung zu. Dort geht es um die Analyse von Schallwellen, die Beschreibung ihrer physikalischen Natur, ihre Entstehung, Darstellung, Veränderung durch verschiedene Umgebungseinflüsse und Wirkung auf einen „Empfänger", z.B. das menschliche Gehör.

Die Phonetik ist von der *Phonologie* zu unterscheiden, die sich eher mit den linguistischen Aspekten der Sprache befasst, wie Lauten innerhalb von Worten (vgl. Durand und Laks, 2002), und die für diese Arbeit nicht relevant ist.

### 2.1.1 Digitalisierung

*Schall* ist aus physikalischer Sichtweise eine Teilchenbewegung in einem Medium wie z.B. Luft, die immer wellenförmig verläuft. Dabei erfolgt die Ausbreitung mit der festen Schallgeschwindigkeit $c$. *Sprachschall* kann als eigene Form des Schalls betrachtet werden. Für die Sprachverarbeitung ist diese Unterscheidung wichtig, da die meisten Verfahren nur für die Verarbeitung der menschlichen Sprache ausgelegt sind und bei Geräuschen und nicht-vokalen Äußerungen versagen und arbiträre Ergebnisse liefern. Häufig werden daher zur Identifikation und zum Herausfiltern von Geräuschen gesonderte Algorithmen benötigt.

Für die computergestützte Sprachverarbeitung muss die Sprache in einer digitalen Repräsentation vorliegen, d.h. der Rechner arbeitet mit einer binär codierten Information, die in Form einer Audiodatei oder im Arbeitsspeicher vorliegt. Die Überführung

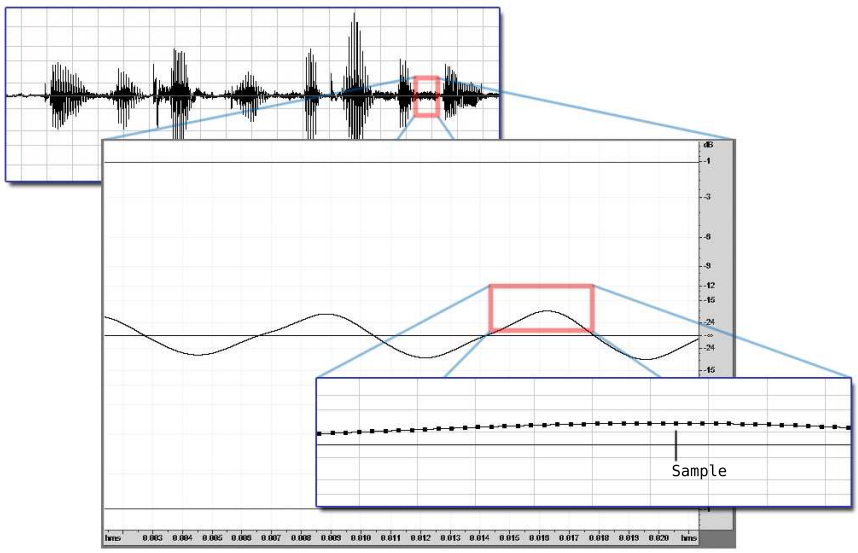



Abbildung 2.1: Samples in einem digitalen Audiosignal

des Sprachschalls in die digitale Form erfolgt in der Regel in zwei Schritten: Zuerst wird die Stimme mit dem Mikrofon aufgezeichnet und dabei in ein analoges Signal umgewandelt, das aus elektrischen Impulswellen besteht. Wenn die Qualität von Mikrofon und elektrischem Leiter gut sind, so bleibt das akustische Signal bis dahin ohne merkliche Veränderung der spezifischen Charakteristika erhalten (siehe auch Plichta, 2002, Abschnitt 1). Für den zweiten Schritt gilt das im Allgemeinen nicht. Hierbei wird die kontinuierliche Darstellung des Signals als Welle in eine maschinenkonforme binäre Darstellung überführt. Dieser Vorgang wird als *Quantisierung* oder *Sampling* bezeichnet (vgl. Orfanidis, 1996, Kapitel 1). Vereinfacht ausgedrückt wird dabei das analoge Signal an festen Stellen abgetastet, und der jeweils an diesen Stellen gemessene Wert wird als Zahl mit endlicher Genauigkeit (d.h. fester Anzahl von Nachkommastellen), ein so genanntes *Sample*, abgespeichert (s. Abb. 2.1). Dabei tritt ein möglicherweise nicht unerheblicher Verlust an Informationen an zwei Stellen auf: einerseits bei der Anzahl der Samples, für die ein Wert ausgelesen wird (bei dem kontinuierlichen Signal kann zwischen jedem Paar von Punkten ein weiterer Wert ausgelesen werden) und andererseits bei der Genauigkeit des gespeicherten Wertes, welcher in der Natur quasi unendlich viele Nachkommastellen besitzt.

Die Zahl der aufgezeichneten Samples pro Sekunde wird als *Abtastfrequenz* oder *Samplingrate* bezeichnet und in der Einheit *Hertz* (Hz) ausgedrückt. 44.100 Hz entspricht etwa der Qualität einer handelsüblichen Audio-CD. Das *Sampling-Theorem* (vgl. Plichta und Kornbluh, 2001) zeigt eine Möglichkeit auf, eine verlustfreie Digitalisierung zu erreichen. Dies ist nach besagtem Theorem nämlich genau dann der Fall, wenn die Ab-

tastfrequenz mehr als das Doppelte der höchsten im Signal auftretenden Frequenz (der *Nyquist-Frequenz*) beträgt. Für die Sprecherklassifikation bedeutet dies konkret, dass eine Auflösung von 16 kHz theoretisch ausreicht, um den von der menschlichen Stimme bei gesprochenen Äußerungen abgedeckten Frequenzbereich von 75 bis 500 Hz (vgl. Kent und Read, 2002 und Ladefoged, 2001) ohne Verlust aufzeichnen zu können. In der Praxis ist aber zu berücksichtigen, dass es bereits durch minimale Umgebungsgeräusche zu Artefakten kommen kann, welche auch die Klassifikation beeinträchtigen können.

Bei der Angabe der Genauigkeit der Samples wird von der *Abtasttiefe* oder *Bitrate* gesprochen, die besagt, wie viele Bits ein Sample im Speicher belegt. Die meisten Aufnahmen finden mit 16 Bit statt, besonders hochwertige Aufnahmen können durchaus auch 24 Bit aufweisen. Grundsätzlich ergibt eine höhere Abtasttiefe eine breitere Klangdynamik, bei Aufnahmen mit mehr als 24 Bit kann diese zusätzliche Tiefe vom menschlichen Gehör allerdings kaum noch wahrgenommen werden.

Es gibt noch weitere Faktoren, welche bestimmen, wie Sprache digital gespeichert wird. Dazu gehört die Anzahl der Kanäle (z.B. einer bei Mono- und zwei bei Stereo-Aufnahmen) oder die Komprimierung des Ergebnisses durch mitunter verlusthaltige Algorithmen wie Noll (1997). Meist handelt es sich bei dem gewählten Audioformat um einen Kompromiss aus Speicherverbrauch bzw. Verarbeitungsgeschwindigkeit (vgl. Kapitel 5) und der Präzision der Daten (und damit auch aller darauf aufsetzenden Verarbeitungsmethoden). Nähere Anhaltspunkte zur Auswahl eines geeigneten Audioformats in Verbindung mit AGENDER werden in Feld (2005, S. 3f.) gegeben.

### 2.1.2 Sprache als Unterscheidungsmerkmal von Alter und Geschlecht

Die Sprache eines Menschen liefert eine ganze Reihe von Anhaltspunkten über das Sprecheralter und -geschlecht. Die Ergebnisse der Literaturstudien von Müller (2005, S. 43ff) im Bereich der Phonetik und der medizinischen Phoniatrie zu diesem Thema stellen sich wie folgt dar:

Bei **Kindern** unter 8 Jahren lässt sich kaum ein Unterschied zwischen den Geschlechtern feststellen. Dies wird auch dadurch bestätigt, dass der anatomische Aufbau der Stimmorgane bis zu diesem Zeitpunkt fast gleich ist. Erst bei **Jugendlichen** ab der Pubertät erfolgt diesbezüglich eine unterschiedliche Entwicklung bei Mädchen und Jungen, und der hörbare Unterschied in der Stimme verstärkt sich zunehmend; bei Mädchen sinkt die Tonlage durchschnittlich um etwa ein Drittel, bei Jungen sogar um eine ganze Oktave (was sich dann im Stimmbruch manifestiert). Die Sprechgeschwindigkeit, die neben akustischen Merkmalen einen weiteren wichtigen Anhaltspunkt zur Klassifikation darstellt, nimmt bei Kindern und Jugendlichen mit dem Alter leicht zu, lässt allerdings keine geschlechtsspezifischen Unterschiede erkennen.

Bei **jüngeren Erwachsenen** kann das Geschlecht auf Basis der Stimme in der Regel

sehr gut unterschieden werden. Dies hängt damit zusammen, dass der Kehlkopf bei Männern größer ist als der von Frauen, und dass die Stimmlippen im Kehlkopf bei Frauen und Männern ebenfalls verschiedene Größen aufweisen, was Unterschiede in der Grundfrequenz (vgl. Abschnitt 2.1.3) bewirkt.

Mit zunehmendem Alter wirken sich anatomische Veränderungen auch auf die Sprachproduktion aus. Dies reicht von Alterungsprozessen beim Kehlkopf über Veränderung des Nervensystems bis zu Muskelrückgang im Gesichtsbereich. In jedem Fall ist aber zu beachten, dass diese Prozesse sich sehr stark von einer Person zur anderen unterscheiden können, so dass es schwierig ist, feste Altersgrenzen zu definieren. Müller (2005) definiert die Gruppe der **Senioren** ab einem Alter von 65 Jahren, da etwa zu dieser Zeit wesentliche Alterungsvorgänge im Bereich des Kehlkopfs einsetzen, die sich auf die Stimme auswirken. Messbar ist dabei zum einen eine Veränderung des Stimmumfangs, bei der das gesamte Spektrum sich bei Frauen nach unten verschiebt und bei Männern im oberen Bereich stark reduziert wird. Weitere geeignete akustische Maße sind zunehmende Instabilitäten in der Tonlage und Stimmintensität, Tremor-Effekte und spektrales Rauschen (vgl. Abschnitt 2.1.3). Bei der Mehrzahl der genannten Effekte ist ein Unterschied zwischen den Geschlechtern zu erkennen. Darüber hinaus sinkt erwartungsgemäß die Artikulationsgeschwindigkeit in der Regel ab. Es gibt auch bedeutende Unterschiede auf linguistischer Ebene, z.B. die Zunahme von Selbstkorrekturen und Füllworten, jedoch liegt deren nähere Betrachtung vorerst nicht im Blickfeld von AGENDER.

### 2.1.3 Wesentliche Größen sprachlicher Äußerungen

Sobald die Sprache in der digitalen Form vorliegt, können aus ihr diverse Werte berechnet werden, die dann für weitere Analysen und Vergleiche zur Verfügung stehen. Dieser Vorgang wird als *Merkmalsextraktion* bezeichnet. Eine der wichtigsten Methoden bei der Berechnung dieser Werte stellt die *diskrete Fourier-Transformation* dar, mit der das komplexe Signal in einzelne Frequenzbänder aufgeteilt werden kann. Ein performanter Algorithmus zur Implementierung der diskreten Fourier-Transformation ist mit dem FFT (*Fast Fourier Transform*) gegeben.

Zur praktischen Analyse großer Mengen von Sprachbeispielen, wie sie zusammengefasst in verschiedenen Sprachkorpora in der Forschung verwendet werden, dienen Werkzeuge wie etwa die Anwendung *Praat*, die 1992 von Boersma und Weenink vom *Department of Phonetics* an der Universität von Amsterdam geschrieben wurde und seitdem ständig weiterentwickelt wird (vgl. Boersma, 2001). Sie beinhaltet umfangreiche Funktionen und Algorithmen zur Arbeit mit Sprachdateien sowie Methoden zur Visualisierung der Ergebnisse. *Praat* erlaubt auch das Schreiben eigener Skriptdateien, um neue Funktionen zu definieren oder Sprachproben automatisiert bzw. stapelweise abzuarbeiten, was es insbesondere für die Analyse umfangreicher Korpora qualifiziert.

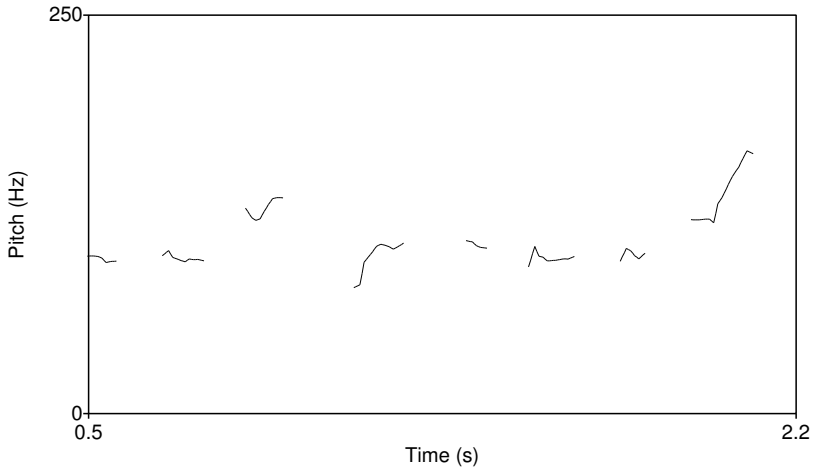



Abbildung 2.2: Grundfrequenzkontur eines Sprachsignals

Diese Vorzüge sowie der hohe wissenschaftliche Anspruch des Programms waren auch ausschlaggebend dafür, dass *Praat* bei der Entwicklung von AGENDER eingesetzt wurde.

Die Auswahl der Merkmale für AGENDER erfolgte basierend auf den in Abschnitt 2.1.2 genannten Beobachtungen und Hypothesen zu den Charakteristika der menschlichen Stimme bei den verschiedenen Geschlechtern und Altersstufen, sowie weit reichender Analysen mittels Merkmalsextraktion. Die Merkmale können in zwei Gruppen eingeteilt werden: Merkmale, die die Charakteristika der Stimme betreffen und solche, die das Sprechverhalten betreffen.

Zu den Stimmmerkmalen zählen:

**Grundfrequenz (F0).** Dieses Merkmal beschreibt die Tonlage (engl. „pitch“) von Schall und wird in Hz angegeben. Der Wert ergibt sich aus der Periodendauer bzw. Frequenz der Schallwellen, welche in der Natur kontinuierlich sind, und kann daher nur schwer an diskreten Stellen erfasst werden. Mittels eines geeigneten Verfahrens (vgl. Abschnitt 3.7.2) lässt sich für ein Sprachsignal eine *Grundfrequenzkontur* erstellen (s. Abb. 2.2), welche F0 in festen Abständen berechnet und alle Werte dazwischen durch Interpolation approximiert. Besonders aussagekräftig sind Mittelwert, Median und Minimum der Grundfrequenz sowie ihre Standardabweichung als Maß für den Frequenztremor.

**Jitter.** Mikrovariationen der Grundfrequenz (d.h. Abweichungen in kleinen zeitlichen Umgebungen) werden durch den *Jitter*-Wert beschrieben. Die Größe des betrachteten Zeitfensters kann unterschiedliche Stimmaspekte betonen und führt zu ver-

schiedenen Erkenntnissen.

**Shimmer.** Der Parameter *Shimmer* ist sehr eng mit dem Jitter verwandt. Es handelt sich um das Gegenstück zur Frequenz im Bereich der Amplitude, d.h. es werden Mikrovariationen der Stimmintensität gemessen. Auch hier werden gerne verschieden große Umgebungen eingesetzt.

**Harmonicity-to-Noise-Ratio (HNR).** Dieser Wert drückt den Anteil von Rauschen im Verhältnis zur Stimme bzw. harmonischer Energie aus. Dies kann auch als Grad der akustischen Periodizität interpretiert werden.

In die Kategorie der Sprechverhaltensmerkmale fallen:

**Artikulationsgeschwindigkeit (AR).** Dieses Merkmal beschreibt die Sprechgeschwindigkeit für eine Äußerung. Als geeignete Grundlage für dieses Maß wurde die so genannte *Energierate* oder *En-Rate* (vgl. Morgan, Fosler und Mirghafori, 1997) identifiziert, welche das erste Spektralmoment für eine Niederpassfilterung des Signals bei 16 Hz berechnet.

**Sprechpausen.** Über ein Verfahren, das auf dem *Syllable Rate Speech Activity Detector* (SRSAD, vgl. Smith, Townsend, Nelson und Richman, 1999) basiert, wird die Äußerung in Stimmsegmente und Pausen unterteilt. Es lassen sich daraus mehrere Maße ableiten, u.a. die aufsummierte Pausendauer und Pausenanzahl sowie die durchschnittliche Pausendauer und die Anzahl der Pausen pro Sekunde.

## 2.2 Grundlagen der Klassifizierung

Die Klassifikation von Alter und Geschlecht eines Benutzers, wie sie durch AGENDER durchgeführt wird, ist eine Form der *automatischen Klassifizierung*. Hierbei wird allgemein für eine gesuchte Eigenschaft eines Objekts eine feste Menge von Klassen definiert. Einem konkreten Objekt aus einer bestimmten Domäne kann für die betreffende Eigenschaft dann nach festgelegten Kriterien eine der Klassen zugeordnet werden. Beispielsweise werden für die Eigenschaft *Geschlecht* des Objekts *Sprecher* die Klassen *weiblich* und *männlich* festgelegt.

### 2.2.1 Mustererkennung

Das bei der automatischen Klassifizierung in AGENDER zum Einsatz kommende Verfahren ist die *Mustererkennung*, die in der Künstlichen Intelligenz als Teilgebiet der Informatik breite Anwendung findet, beispielsweise bei der optischen Zeichenerkennung, der automatischen Spracherkennung oder zum Filtern von eMails auf Spam.

Die Musterkennung kann nach Duda, Hart und Stork (2000) in fünf Teilschritte wie in Abb. 2.3 gezeigt aufgegliedert werden. Zur Veranschaulichung stellen die Autoren folgendes Beispiel dar: In einer Fischverpackungsfabrik treffen Lachse und Seebarsche auf einem Förderband ein, wo sie für die weitere Verarbeitung automatisch sortiert werden sollen. Ein Computersystem soll also für jeden Fisch entscheiden, ob er der Klasse *Lachs* oder *Seebarsch* zuzuordnen ist. Im ersten Schritt erfolgt die *Akquisition von Sensordaten* und Umwandlung dieser Daten in ein digitales Format, welches vom Computer verarbeitet werden kann. In der Fischfabrik könnte dazu an geeigneter Stelle eine Kamera installiert werden, die einen Ausschnitt des Förderbands aufnimmt und dem Computersystem als Videodatenstrom zuleitet.

Anschließend wird die *Segmentierung* durchgeführt. Dabei handelt es sich um einen Vorverarbeitungsprozess, welcher der Filterung und Aufbereitung der Daten für die nachfolgenden Prozesse dient. Am Beispiel der Fischfabrik bedeutet dies die Auswahl einzelner Standbilder aus dem kontinuierlichen Videobild und die Erkennung der Umrisse der Fische durch Methoden der Bildverarbeitung, also z.B. die Entscheidung darüber, ob ein bestimmter Bildbereich noch zu diesem oder bereits zum nächsten Fisch gehört.

Nach der Segmentierung erfolgt die *Merkmalsextraktion*, d.h. das Auslesen oder Berechnen von Werten, die für die Klassifizierung verwendet werden. Dieser Schritt und die eigentliche Klassifizierung sind nicht immer eindeutig trennbar, da ein optimaler Merkmalsextraktor die Klassifikation zu einem trivialen Problem macht und umgekehrt ein komplexer Klassifizierer bereits mit den segmentierten Daten die gewünschten Ergebnisse liefern kann. Dennoch ist die Trennung nützlich, um zwischen für die Differenzierung geeigneten Merkmalen und nicht relevanten Charakteristika zu unterscheiden. Dafür ist oft eine umfangreiche theoretische und statistische Problemanalyse notwendig,

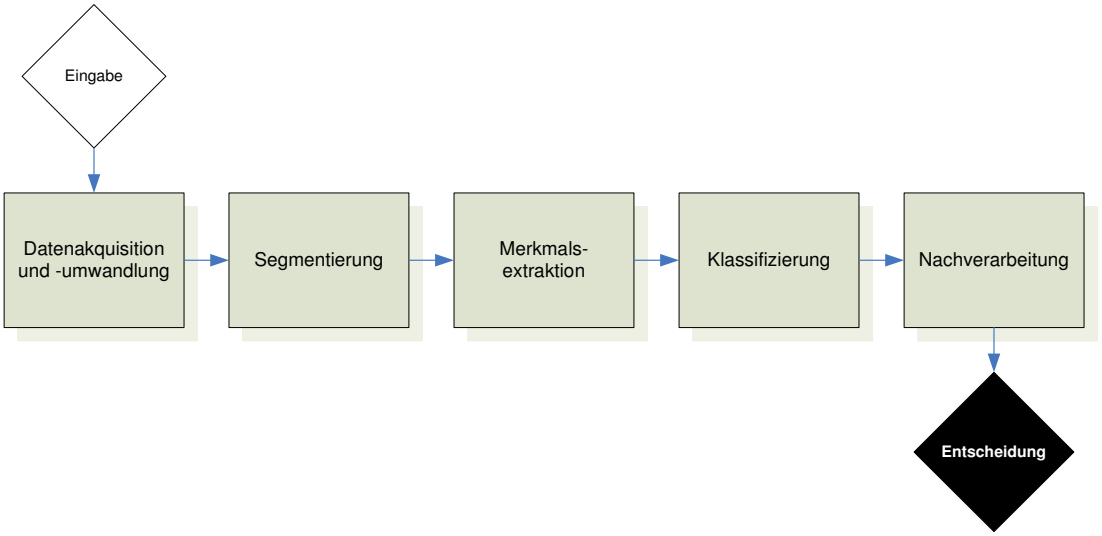



Abbildung 2.3: Typischer Ablauf einer Mustererkennung nach Duda et al. (2000, S. 10)

nicht zuletzt um stark voneinander abhängige (korrelierende) Merkmale auszuschließen. Im Fall der Fischklassifikation wären Merkmale wie Länge und Farbe charakteristische Merkmale, wohingegen die Lage und Orientierung auf dem Förderband ungeeignet sind.

Im nächsten Schritt, der *Klassifizierung*, werden die extrahierten Merkmale, der so genannte *Merkmalsvektor*, durch einen Algorithmus ausgewertet, welcher als Ergebnis diejenige Klasse zurückliefert, der die Eingabedaten am ehesten entsprechen. In den meisten Fällen ist eine exakte Aussage nicht möglich, da die Wertebereiche einzelner Merkmale (*Merkmalsräume*) oft nicht klar getrennt sind und sich überschneiden oder weil nicht alle Merkmale einwandfrei bestimmt werden konnten.

Als letzter Schritt kann eine *Nachverarbeitung* durchgeführt werden. Dies ist z.B. dann notwendig, wenn die Ergebnisse mehrerer Klassifizierer zu einem Gesamtergebnis kombiniert werden sollen, oder zur Berücksichtigung der Kosten einer Fehlklassifikation (vgl. Abschnitt 2.2.4). Wenn beispielsweise zwei von drei Klassifizierern einen Fisch als *Seebarsch* einordnen und einer als *Lachs*, so würde im Falle einer Mehrheitsentscheidung das Ergebnis *Seebarsch* lauten. Liegt jedoch in diesem Fall der einzelne Klassifizierer mit *Seebarsch* richtig, so ist dies aus Sicht des Käufers tragischer, als wenn umgekehrt der Lachs als Seebarsch verkauft würde.

Der Entwurf eines Mustererkennungssystems besteht nach Duda et al. (2000, S. 10) aus mehreren Teilschritten: Datensammlung, Auswahl geeigneter Merkmale, Auswahl eines Modells, Training des Klassifizierers und Evaluierung des Klassifizierers. Die Durchführung der ersten beiden Schritte wurde im Wesentlichen in Abschnitt 2.1 dargelegt. Einige der Kriterien für die Auswahl eines Modells, d.h. eines geeigneten Klassifikati-

onsalgorithmus, sowie häufig verwendete Modelle, werden in Abschnitt 2.2.2f. gegeben.

Nachdem die relevanten Merkmale bestimmt wurden, wird das *Training* durchgeführt. Dabei handelt es sich um einen Lernvorgang, bei dem mittels eines bestimmten Algorithmus eine mathematische oder statistische Beziehung zwischen den Eingabemerkmalen und den Klassen hergestellt wird. Bei diesem Vorgang erhält die Trainingsprozedur als Eingabe eine Datenbasis mit möglichst vielen Datensätzen sowie die zugehörige richtige Klassenzuordnung. Das Ergebnis des Trainings ist der eigentliche Klassifizierer. Dabei handelt es sich um eine Datenstruktur, mit Hilfe derer der zugehörige Klassifikationsalgorithmus die eigentliche Klassifizierung durchführen kann.

Die Leistung eines Mustererkennungssystems kann durch die Klassifizierung anhand von Testobjekten ermittelt werden. Dies wird als *Evaluierung* bezeichnet und nach dem Training neuer Klassifizierer durchgeführt (vgl. Abschnitt 3.9.5).

Im verbleibenden Teil dieser Arbeit bezieht sich der Begriff der Klassifizierung immer auf eine Mustererkennung.

### 2.2.2 Kriterien der Klassifizierungsalgorithmen

Die Umsetzung eines Klassifikationsverfahrens kann auf ganz verschiedene Weise erfolgen. Es gibt eine Reihe verbreiteter Modelle z.B. aus der Statistik, die jeweils ihre spezifischen Vor- und Nachteile aufweisen. Zu jedem Algorithmus gehört eine Implementierung für das Training sowie für die eigentliche Klassifizierung.

Die wichtigsten Unterscheidungskriterien der Algorithmen sind:

**Klassifikationsgenauigkeit.** Die Genauigkeit eines Klassifizierungsverfahrens wird primär an der so genannten *True Positive Rate* gemessen, d.h. wie viele Äußerungen korrekt klassifiziert werden. Hier kann meist keine generelle Aussage über die Algorithmen gemacht werden, sondern die Eignung des jeweiligen Modells hängt von der Beschaffenheit der Daten beim konkreten Klassifikationsproblem ab. So kann ein einfaches Modell versagen, indem es komplexe Entscheidungsgrenzen nicht erkennt. Auf der anderen Seite kann ein komplexes Modell evtl. zu starr an den Trainingsdaten orientiert sein und neue Proben nicht richtig einordnen (*Overfitting-Problem*, vgl. Dietterich, 1995). Die Genauigkeit eines Klassifizierungsverfahrens wird typischerweise durch eine Kreuzvalidierung (vgl. Abschnitt 3.9.5) ermittelt.

Es ist zu beachten, dass neben dem Algorithmus auch Qualität und Umfang der Datenbasis eine wichtige Rolle bezüglich der Klassifizierungsgenauigkeit spielen (vgl. Abschnitt 2.3.2).

**Laufzeit / Komplexität.** Für viele Anwendungen spielt die Geschwindigkeit, mit der einzelne Äußerungen klassifiziert werden, eine große Rolle. Um Skalierbarkeit zu gewährleisten, sollte die komputationelle Komplexität (d.h. amortisierte Laufzeit)

des Algorithmus betrachtet werden. Wichtiger noch sind aber Laufzeit-Analysen (*Benchmarks*) mit vorliegenden Implementierungen eines Algorithmus unter realen Betriebsumgebungen und bei einem Datenaufkommen, dass der geschätzten durchschnittlichen Auslastung im praktischen Einsatz entspricht.

**Trainingsdauer.** Der Lernvorgang eines Klassifizierungsalgorithmus kann bei manchen Verfahren – offensichtlich auch in Abhängigkeit vom Umfang der Trainingsdaten – eine nicht unerhebliche Zeit beanspruchen und wird damit auch zu einem möglichen Auswahlkriterium. Dieser Aspekt spielt jedoch in der Regel nur eine untergeordnete Rolle, da das Training nur einmalig für jede zu klassifizierende Eigenschaft durchgeführt werden muss, und danach nur bei Änderung der Klassen oder des Verfahrens.

**Speicherverbrauch.** Die beim Training erstellte und zur Klassifizierung benötigte Datenstruktur belegt eine gewisse Menge Speicherplatz auf der Festplatte (und ggf. auch im Arbeitsspeicher), die vom Umfang der Trainingsdaten abhängen kann. Das Szenario der Klassifizierung auf einem mobilen Gerät mit beschränkten Ressourcen lässt schnell ersichtlich werden, weshalb auch der Speicherverbrauch ein wichtiges Argument für oder wider einen Algorithmus sein kann.

### 2.2.3 Häufig verwendete Klassifizierungsalgorithmen

Die folgenden Algorithmen stellen eine Auswahl der am weitesten verbreiteten Verfahren dar, die durch Müller (2005) im Rahmen von AGENDER eingehend evaluiert wurden.

#### Bayesscher Klassifizierer (Naive Bayes)

Dieser Algorithmus basiert auf der Gauß'schen Wahrscheinlichkeitsdichte, welche die Verteilung einer Zufallsgrößen $\mathbf{x}$ (in diesem Fall ein Merkmalsvektor) über einem Wertebereich (hier der Merkmalsraum) angibt. Um die Merkmalswerte besser vergleichen zu können, werden diese zuvor normiert. Die Klassifizierung erfolgt anhand der *multivariaten Gauß'schen Normalverteilung*

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{d/2}\,|\Sigma|^{1/2}} \quad exp\left[-\frac{1}{2}\,(\mathbf{x}-\mu)^t\,\Sigma^{-1}\,(\mathbf{x}-\mu)\right], \tag{2.1}$$

durch die für jede Klasse eine Wahrscheinlichkeit $p_i$ berechnet wird. Die Mittelwertvektor $\mu$, die Kovarianzmatrix $\Sigma$ sowie ihre Determinante $|\Sigma|$ und Inverse $\Sigma^{-1}$ werden beim Training über der gesamten Datenbasis bestimmt. Die Variable $d$ bezeichnet die Anzahl der Merkmale im Merkmalsvektor $\mathbf{x}$. Als Ergebnis liefert der Bayessche Klassifizierer die Klasse mit der höchsten Wahrscheinlichkeit $p$.

Trotz der einfachen Implementierung lassen sich mit dieser Klassifizierungsmethode oft schon gute Ergebnisse erzielen. Das Training geht sehr schnell vonstatten, und auch die gute Laufzeit und der minimale Speicherverbrauch sprechen für den Klassifizierer.

### Gaussian Mixture Model

Dieses Modell, welches nahe mit dem Bayesschen Klassifizierer verwandt ist, bestimmt die zugrunde liegende klassenspezifische Wahrscheinlichkeitsdichte, auf Basis derer ein *Likelihood-Ratio-Klassifizierer* dann einen Merkmalsvektor einer Klasse zuordnet. Im Gegensatz zu Formel 2.1 werden die Wahrscheinlichkeiten nun mit merkmalsspezifischen Gewichten $w_j$ versehen:

$$p(\mathbf{x}|\mu) = \sum_{j=1}^{M} w_j p_j(x) \tag{2.2}$$

Um die optimalen Gewichte $w_j$ zu ermitteln wird beim Training der *Expectation-Maximization-Algorithmus* (vgl. Moon, 1996) angewandt.

Die erhöhte Genauigkeit gegenüber dem Naive Bayes-Algorithmus geht zu Lasten einer etwas höheren Trainingsdauer, ansonsten aber kann er die positiven Leistungseigenschaften beibehalten.

### Nearest-Neighbor

Der Nearest-Neigbor-Algorithmus versucht aus der Menge aller Trainingsdatensätze bestimmte Prototypen zu identifizieren oder neu zu definieren, für die die meisten in ihrer näheren Umgebung (in Bezug auf den euklidischen Abstand im $d$-dimensionalen Raum für $d$ = *Anzahl der Merkmale*) angesiedelten Datensätze zur gleichen Klasse gehören. Diese Klasse sowie die Entscheidungsregion wird dem Prototypen zugewiesen. Bei der Klassifikation wird dann für einen Merkmalsvektor die Klasse des am nächsten liegenden Prototypen zurückgeliefert.

Der Algorithmus kann erweitert werden, indem die am häufigsten vertretene Klasse unter den $k$ nächsten Nachbarn berechnet wird. Man spricht dann vom *k-Nearest-Neighbor-Algorithmus.*

Die Genauigkeit des Klassifizierers variiert stark je nach zugrunde liegenden Daten. Aus Sicht der Performanz bei Klassifizierung als auch Training sind bei Verwendung dieses Verfahrens keine Nachteile erkennbar, allerdings ist der Speicherbedarf höher als bei den meisten anderen Verfahren.

### Entscheidungsbäume

Dieser Algorithmus besitzt in der Ausführung eine sehr einfache Form. Er besteht aus mehreren Blöcken, die jeweils eine Entscheidung bzgl. des Werts eines Merkmals darstellen:

```
Block_k:
  Wenn Merkmal_i > [Wert] Dann
    Gehe Zu Block_m   (oder: Klassifiziere als Klasse_a)
  Sonst
    Gehe Zu Block_n   (oder: Klassifiziere als Klasse_b)
```

Da es genau einen Startblock (Wurzel) gibt und der Algorithmus immer terminiert (d.h. es gibt keine Rekursion bzw. Zyklen), kann das Modell auch als Baum wie in Abb. 2.4 dargestellt werden.

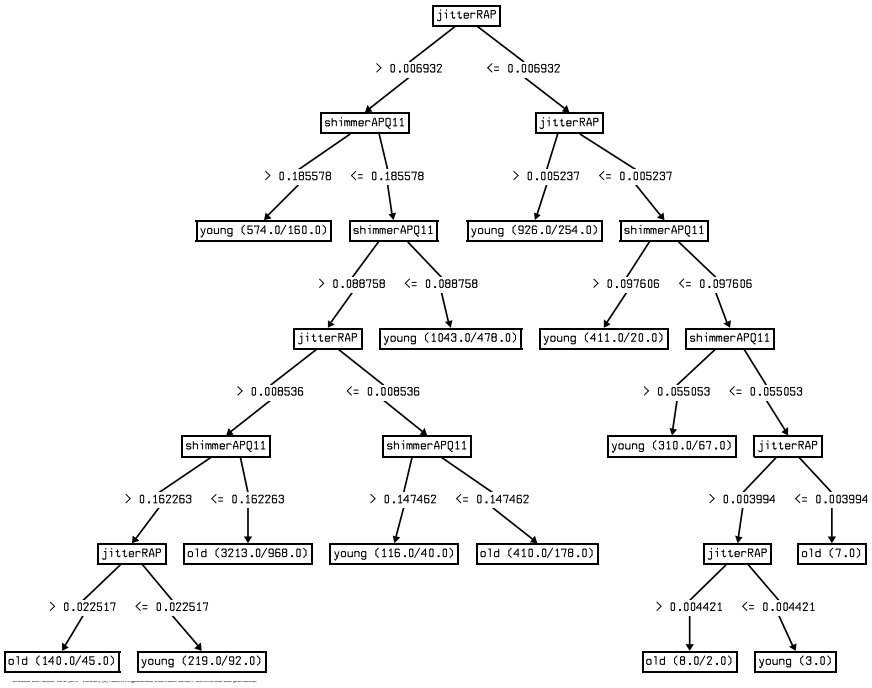



Abbildung 2.4: Grafische Darstellung eines Entscheidungsbaums für einen Altersklassifizierer

Die Entscheidungsbäume bestechen durch eine sehr hohe Klassifizierungsgeschwindigkeit, die auf die einfache Struktur zurückzuführen ist. Zudem kann ein solcher Baum in ein sehr kompaktes Format (vgl. Abschnitt 4.1.5) gebracht werden, wodurch er sich für mobile Plattformen als besonders geeignet erweist.

#### Künstliche Neuronale Netze

Die Idee der neuronalen Netze ist durch die Biologie inspiriert. Sie entstand bei dem Versuch, die Verarbeitungsmuster der Nervenzellen des Gehirns für Computer nachzuempfinden. Für die künstlichen Neuronen, die gegenüber den Zellen des Gehirns stark vereinfacht sind, gibt es unterschiedliche Implementierungen. Für AGENDER wurde eine Variante mit *mehrlagigen Perzeptronen* eingesetzt. Diese verwenden die Funktion

$$f(x) = \langle w, x \rangle + b \tag{2.3}$$

wobei $x$ der Merkmalsvektor, $w$ ein Vektor mit Gewichten, $\langle \cdot, \cdot \rangle$ die Punkt-Multiplikation von Vektoren und $b$ ein Schwellenwert ist. Wird der Schwellenwert erreicht, so gibt der Knoten 1 aus, andernfalls 0. Aus mehreren solcher Knoten entsteht ein Netz, indem die Ausgänge einiger Knoten mit den Eingängen anderer Knoten verbunden werden.

Das Training erfolgt bei dieser Art von neuronalem Netz über *Rückwärtspropagierung* (vgl. Rumelhart, Hinton und Williams, 1986) und dauert länger als bei allen anderen genannten Klassifizierungsmethoden. Dafür erzielte dieses Modell jedoch auf der AGENDER-Datenbasis die besten Ergebnisse (vgl. Kapitel 5).

Für eine detaillierte Behandlung der genannten und weiterer Methoden wie z.B. *Support Vector Machines* wird auf Müller (2005, S. 129ff) verwiesen.

*Java*-Implementierungen für eine Vielzahl von Algorithmen findet man in dem frei verfügbaren *WEKA*[1]-Paket aus Witten und Frank (2005), welches auch intensiv für die Implementierung des M3I-Servers genutzt wurde. Neben den Klassifizierern selbst sind unter anderem Werkzeuge zur Evaluierung und Visualisierung enthalten. Die Klassen lassen sich entweder direkt in den Code einbinden oder über die enthaltene GUI-Applikation nutzen.

### 2.2.4 Zweistufige Verarbeitung

Die direkte Weitergabe der Klassifikationsergebnisse würde zu mehreren Problemen bei der Interpretation der Daten führen. Aus diesem Grund wurde von Müller (2005) eine zweite Verarbeitungsebene entwickelt, welche die Durchführung aller im Folgenden aufgeführter Nachverarbeitungsschritte auf einheitliche Weise ermöglicht.

**Klassifikationsinhärente Unsicherheiten.** Bei der Klassifizierung können generell keine sicheren Resultate erwartet werden, da es sich ansonsten um ein triviales Problem handeln würde. Bei der Sprecherklassifikation besteht nicht zuletzt im Einzelfall immer die Möglichkeit starker Abweichungen von Sprachmerkmalen vom

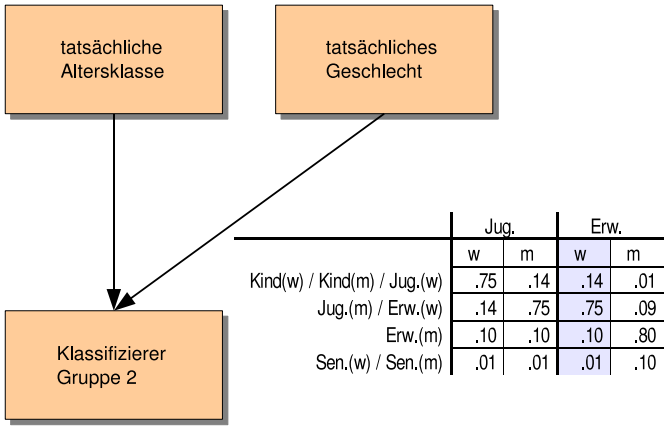
[1] *Waikato Environment for Knowledge Analysis*

Erwartungswert. Ein Klassifizierer sagt die Ergebnisklasse daher immer nur mit einer gewissen Wahrscheinlichkeit $p_i < 1$ voraus. Da der exakte Wert von $p_i$ im Allgemeinen nicht bekannt ist, kann eine Annäherung durch die Gesamtqualität des Klassifizierers erfolgen, welche durch die bei einer Evaluierung ermittelte klassenspezifische True Positive Rate ausgedrückt wird. Diese Wahrscheinlichkeit bestimmt, wie stark das Ergebnis des Klassifizierers gewertet wird. Wenn beispielsweise ein Altersklassifizierer für Kinder, Jugendliche, Erwachsene und Senioren in der Evaluierung nur 70% der Kinder korrekt klassifiziert, so wird das Ergebnis *Kind* dieses Klassifizierers immer mit dieser Wahrscheinlichkeit von 0,7 annotiert. Da AGENDER bei jedem Klassifizierungsvorgang mehrere Klassifizierer für die gleiche Eigenschaft verwenden kann, wird $p_i = 0{,}7$ tatsächlich bestimmen, wie die Aussage dieses einen Klassifizierers anteilig am Gesamtergebnis gewichtet wird.

**Expertenwissen und Kosten der Fehlklassifikation**. Am Beispiel der Fischfabrik in Abschnitt 2.2.1 konnte verdeutlicht werden, dass bei einer Fehlklassifikation Kosten entstehen können, die abhängig von der Klasse verschieden sind. Werden diese Kosten in die Nachverarbeitung integriert, so spricht man von der Einbeziehung von *domänenspezifischem Top-Down-Wissen* oder *Expertenwissen*. Ähnlich verhält es sich, wenn in einem bestimmten Anwendungsszenario die Ausgangswahrscheinlichkeiten (*A-priori-Wahrscheinlichkeiten*) für bestimmte Klassen angepasst werden sollen. Beispielsweise ist die Wahrscheinlichkeit, dass es sich bei dem Benutzer eines Bankautomaten um ein Kind handelt eher gering, so dass diese Klasse mit einer geringeren A-priori-Wahrscheinlichkeit bedacht werden könnte.

**Fusion mehrerer Klassifikationsergebnisse** .

Als dritte Aufgabe der Nachverarbeitung besteht die Notwendigkeit, mehrere Ergebnisse aus dem Teilschritt der Klassifizierung zu einem Gesamtergebnis zusammenzuführen. Hierbei unterscheidet Müller (2005, S. 191f.) zwischen statischer und dynamischer Fusion. Die statische Fusion beschreibt die Kombination der Ergebnisse mehrerer Klassifizierer, ggf. unter Berücksichtigung einer Gewichtung wie zuvor beschrieben. Ziel der dynamischen Fusion ist es, die Ergebnisse mehrerer separat durchgeführter Mustererkennungsdurchläufe so zusammenzufassen, dass jeweils immer auch die früheren Ergebnisse beachtet werden. Dies reduziert den Einfluss von Fehlklassifikationen, z.B. durch Umgebungsgeräusche oder vorübergehende Stimmvariationen, und führt zu einer temporalen Konvergenz der Sprecherklasse.




| | Jug. | | Erw. | |
|---|---|---|---|---|
| | w | m | w | m |
| Kind(w) / Kind(m) / Jug.(w) | .75 | .14 | .14 | .01 |
| Jug.(m) / Erw.(w) | .14 | .75 | .75 | .09 |
| Erw.(m) | .10 | .10 | .10 | .80 |
| Sen.(w) / Sen.(m) | .01 | .01 | .01 | .10 |

Abbildung 2.5: Beispiel für ein Bayessches Netz mit der Modellierung von Unsicherheiten (vgl. Müller, 2005, S. 188)

### Dynamische Bayessche Netze

Alle zuvor genannten Probleme lassen sich durch *dynamische Bayessche Netze* lösen (vgl. Müller, 2005, S. 181ff). Dabei handelt es sich um ein Konstrukt aus der Stochastik, mit welchem die Wechselwirkung zwischen einer beliebigen Anzahl von Beobachtungen und (bedingten) Wahrscheinlichkeiten modelliert werden kann. Abb. 2.5 zeigt ein einfaches Bayessches Netz für AGENDER mit einem einzelnen Klassifizierer.

Der dargestellte Klassifizierer unterscheidet die vier Klassen *Kinder und Weibliche Jugendliche*, *Männliche Jugendliche und Weibliche Erwachsene*, *Männliche Erwachsene* und *Senioren*[2]. Der Knoten, durch den der Klassifizierer im Netz repräsentiert wird, stellt eine Beobachtung dar, nämlich das Klassifizierungsergebnis. Durch automatische Inferenz im Bayesschen Netz kann dann für die Knoten *Alter* und *Geschlecht* berechnet werden, wie hoch die Wahrscheinlichkeit der jeweiligen Eigenschaftsklassen unter Berücksichtigung dieser Beobachtung liegt. Um die in den vorangegangenen Abschnitten beschriebenen Informationen über die Qualität eines Klassifizierers einfließen zu lassen, verwendet man so genannte *Conditional Probability Tables* (CPTs, s. Abb. 2.5). Jede Zeile der Tabelle beschreibt die Wahrscheinlichkeit für alle Eigenschaftsklassen, die in den Spalten angeordnet sind, für den Fall, dass die sich am linken Zeilenrand befindliche Klasse ermittelt wurde. Auf diese Weise wird also nicht nur wie zuvor beschrieben die korrekte Klasse spezifisch gewichtet, sondern auch für die restlichen Klassen kann eine Wahrscheinlichkeit ausgedrückt werden. Die Zahlen in den CPTs ergeben sich in diesem Fall aus der Evaluierung des Klassifizierers: Für Äußerungen weiblicher Erwachsener

[2]Solche auf den ersten Blick unsymmetrisch erscheinende Gruppierungen ergeben sich durch die Analyse der Beziehung zwischen einzelnen Sprachmerkmalen und Klassen, vgl. Abschnitt 2.3.2.

Sprecher enthält die Spalte *Erw. w.* die prozentualen Klassifizierungsergebnisse. Im dargestellten Fall wurden 14% als *Kinder und Weibliche Jugendliche* klassifiziert, 75% als *Männliche Jugendliche und Weibliche Erwachsene* usw.

Weitere Klassifizierer werden durch zusätzliche Knoten dargestellt. Interdependenzen zwischen Alter und Geschlecht und externe Kontextfaktoren werden durch die Netzstruktur automatisch berücksichtigt, indem die CPTs entsprechend gewählt werden.

Ein Bayessches Netz wird als *dynamisch* bezeichnet, wenn es den zeitlichen Aspekt berücksichtigt. Dazu werden mehrere Netze, jeweils eines für jede *Zeitscheibe*, zu einem einzigen dynamischen Netz zusammengefasst und mit bestimmten Zeitscheiben-Übergängen versehen, die sich aber genau wie alle anderen Kanten im Netz verhalten.

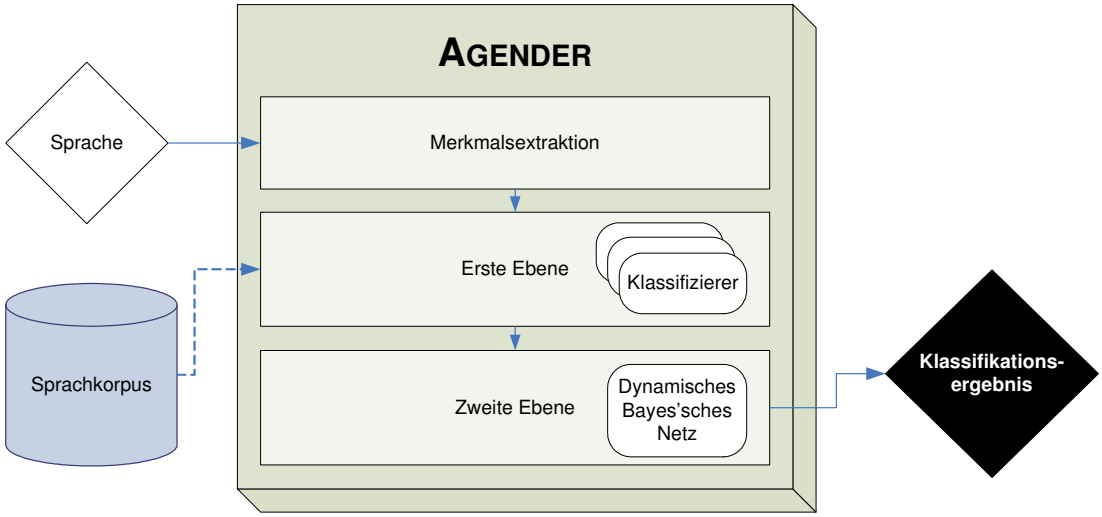



Abbildung 2.6: Schematischer Aufbau von AGENDER

## 2.3 Aufbau und Funktionsweise von Agender

### 2.3.1 Aufbau

Die Arbeit mit AGENDER kann in zwei Phasen eingeteilt werden: Eine *Entwurfsphase* und eine *Ausführungsphase*. In der Entwurfsphase werden die Daten gesammelt, mit deren Hilfe AGENDER im Hinblick auf ein bestimmtes Sprecherklassifikationsproblem optimiert werden kann. Genannt sei hier der Grad der Granulation bei den Altersklassen oder das Training mit bestimmten Personengruppen oder Kontexten als Möglichkeiten zur Spezialisierung. Die Ausführungsphase besteht aus der Klassifizierung einer konkreten Sprachprobe. Beide genannten Aspekte sind in Abb. 2.6 dargestellt.

### 2.3.2 Entwurfsphase

Die theoretischen Untersuchungen der Sprachmerkmale (vgl. Abschnitt 2.1.2) legen eine Klassifizierung mittels einer Mustererkennung, wie in Abschnitt 2.2.1 beschrieben, nahe. Vor dem eigentlichen Klassifizierungsprozess jedoch müssen einige Entscheidungen über die einzusetzenden Klassifizierer getroffen werden. Ein Klassifizierer definiert sich in AGENDER aus der Implementierung eines Mustererkennungsalgorithmus, einem Satz von Sprachmerkmalen, die als Eingabe verwendet werden, und einer Menge von Klassen, von denen eine das Ergebnis des Vorgangs darstellt. Die wohl wichtigste Aufgabe der Entwurfsphase besteht darin, diese Auswahl zu treffen und so viele Klassifizierer zu erstellen wie nötig.
Das Design der Klassifizierer hängt dabei von mehreren Faktoren ab:

*Was sind die Anforderungen der Anwendung?*
*Existieren geeignete Sprachmerkmale zur Umsetzung?*
*Ist der Sprachkorpus zum Training geeignet?*

Die Anforderungen der Anwendung sind in der Regel extern vorgegeben. Die in der Einleitung erwähnten Szenarien zur Adaption der Benutzeroberfläche zur Unterstützung älterer Menschen beispielsweise kommen mit der Unterscheidung der beiden Klassen *jung* und *alt* aus. Im Gegensatz dazu erfordert eine telefoniebasierte Anwendung unter Umständen drei Altersklassen und die Unterscheidung nach Geschlecht, also insgesamt $3 * 2 = 6$ Klassen. Eine Reduzierung der Anzahl an Klassen beim Entwurf der Klassifizierer kann wesentlich effektiver sein als eine nachträgliche Fusion von Klassen auf Applikationsebene, da mit der Anzahl der Klassen im Allgemeinen[3] auch die Unsicherheit des Klassifizierers zunimmt, sofern zusätzliche Klassen nicht durch Teilmengenbildung in vorhandenen Klassen entstehen.

Nicht jede angestrebte Einteilung in Klassen ist realisierbar. Eine Unterscheidung mittels Klassifizierung ist nur dann möglich, wenn es entsprechende Sprachmerkmale gibt, welche eine solche Differenzierung zulassen. Die in Abschnitt 2.1.2 beschriebenen Forschungsergebnisse lassen derzeit eine Einteilung in maximal vier Altersklassen zu, wobei die Klassifikationsgenauigkeit stark variiert (vgl. Kapitel 5). Das für diese Arbeit verwendete Referenzmodell arbeitet in der Regel mit folgenden acht Klassen, wobei an dieser Stelle zur besseren Übersichtlichkeit Abkürzungen eingeführt werden: **KW** (Kinder, weiblich), **KM** (Kinder, männlich), **JW** (Jugendliche, weiblich), **JM** (Jugendliche, männlich), **EW** (Erwachsene, weiblich), **EM** (Erwachsene, männlich), **SW** (Senioren, weiblich) und **SM** (Senioren, männlich). Da aber die Wahrscheinlichkeit hoch ist, dass in Zukunft noch weitere Merkmale zur feineren Granulierung der Altersklassen identifiziert, oder weitere Sprecher- und Kontexteigenschaften modelliert werden können, ist es nicht sinnvoll, AGENDER auf die genannten acht Klassen zu fixieren.

Zuletzt muss auch ein Sprachkorpus zum Training der Klassifizierer bereitgestellt werden, welcher eine ausreichende Anzahl an Sprachproben aus jeder Klasse enthält. Wenn z.B. ein Korpus 1000 Stimmen von Erwachsenen, aber nur 10 von Kindern enthält, so wird der auf dieser Datenbasis trainierte Klassifizierer mit großer Wahrscheinlichkeit nur einen sehr geringen Teil der Stimmen von Kindern korrekt klassifizieren und bei Erwachsenen fast keine Fehler machen (wobei letztendlich auch der verwendete Algorithmus eine Rolle spielt). Für besondere Anwendungsszenarien kann auch die Verwendung eines speziellen Sprachkorpus in Betracht gezogen werden: Ein Klassifizierer für eine Telefonieapplikation sollte demnach mit Sprachproben trainiert werden, welche ebenfalls vom Telefon aufgenommen wurden bzw. die gleichen akustischen Eigenschaften wie eine Telefonverbindung besitzen.

Eine Technik, die sich bei AGENDER für den Entwurf der Klassifizierer als nützlich

[3] Auf künstliche Daten und triviale Probleme trifft dies evtl. nicht zu.

erwiesen hat, ist die *Gruppierung* von Klassen (vgl. Müller, 2005, S. 123ff). Nicht jeder Klassifizierer muss alle Klassen unterscheiden, die als für das Problem relevant festgelegt wurden. Stattdessen ist es sinnvoll, die Untersuchung aus Sicht der Sprachmerkmale zu betreiben: So bietet die Artikulationsgeschwindigkeit keine Anhaltspunkte, die eine Aussage über das Geschlecht zulassen. Wird dieses Merkmal für einen Klassifizierer verwendet, welcher nur oder unter anderem das Geschlecht unterscheidet, so werden aufgrund der zwangsläufig bei großen Datenmengen vorhandenen Variationen möglicherweise falsche Regeln abgeleitet, also ein nicht bestehender Zusammenhang zwischen Artikulationsgeschwindigkeit und Geschlecht hergestellt. Daher ist es günstig, bei jedem Klassifizierer nur die Klassen zu unterscheiden, die nachweislich durch die Eingabemerkmale gestützt werden. Klassen, zwischen denen keine oder nur geringe Unterschiede – ausgehend von den Sprachmerkmalen – bestehen, werden zu einer Gruppe zusammengefasst. Der Klassifizierer aus Abb. 2.5 auf Seite 28 unterscheidet die Gruppen KwKmJw, JmEw, Em und SwSm.

Sobald die Konzeption der Klassifizierer abgeschlossen ist, kann das Training erfolgen. Für Agender wurden verschiedene Korpora mit insgesamt 38202 Äußerungen von 1164 Sprechern verwendet. Tatsächlich können allerdings nicht alle Äußerungen genutzt werden, sondern nur so viele, dass für jede der Klassen die gleiche Anzahl von Äußerungen bereitgestellt werden kann (*Balancierung*). Ferner ist bei der Arbeit mit Äußerungen aus verschiedenen Quellen darauf zu achten, dass alle Audiodateien vor dem Training auf ein gemeinsames Format gebracht werden müssen, einschließlich Normierung der Intensität. Die Ausführung der Korpusanalyse erfolgt bei Agender über die webbasierte Anwendung m3i Cat[4] (siehe Abb. 2.7 und Müller, 2005, S. 229ff).

An das Training schließt sich in der Regel eine Evaluierung an. Bei diesem Vorgang wird der fertige Klassifizierer mit einer Anzahl von Äußerungen getestet. Das Ergebnis wird in Form einer Konfusionsmatrix angegeben (vgl. Abschnitt 3.9.5). Evaluiert werden an dieser Stelle immer nur einzelne Klassifizierer; es handelt sich nicht um eine Ausführung von Agender.

Um nun die Ergebnisse der verschiedenen Klassifizierer zu kombinieren und, wie in Abschnitt 2.2.4 dargelegt, die Unsicherheiten bei der Klassifizierung zu behandeln, muss schließlich noch ein dynamisches Bayessches Netz passend zu den verwendeten Klassifizierern entworfen werden. Für das Referenzsystem wurden jeweils Netze verwendet, die in der ersten Zeitscheibe einen Knoten für jeden Klassifizierer sowie einen Knoten pro Eigenschaft des Sprechers besitzen. Die CPTs dieser Netze sind in hohem Maße abhängig von der Anwendung.

[4] m3i Corpus Analysis Tool

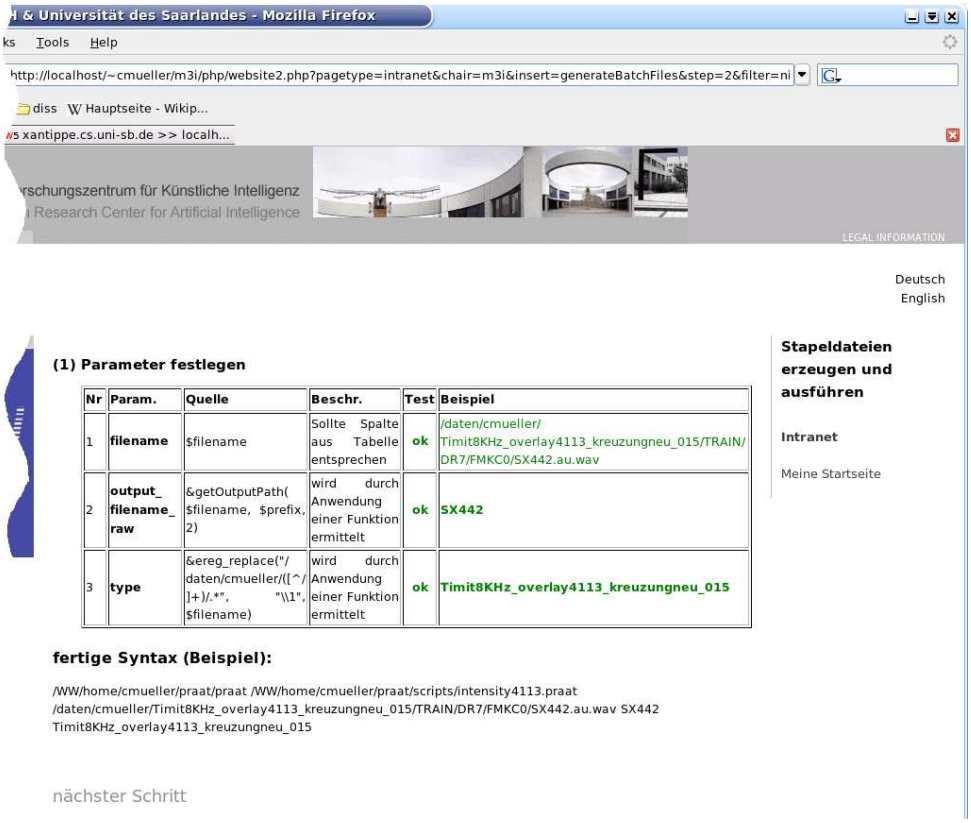



Abbildung 2.7: Korpusanalysewerkzeug M3I CAT im Webbrowser (vgl. Müller, 2005, S. 236)

### 2.3.3 Ausführungsphase

Um mit AGENDER Äußerungen im laufenden Betrieb klassifizieren zu können, wird als Eingabe eine digitale Audiodatei bereitgestellt. In einem ersten Vorverarbeitungsschritt wird diese ggf. in das benötigte Format umgewandelt (z.B. durch Resampling oder Dekodierung). Zurzeit verwendet AGENDER intern das Format *PCM*[5] *8 kHz, 16 bit mono*, da hier die größte Kompatibilität mit den Trainingskorpora gewährleistet ist.

Anschließend erfolgt die Merkmalsextraktion (vgl. Abschnitt 3.7.2). Bei diesem Vorgang werden die benötigten Stimmmerkmale und Sprechverhaltensmerkmale aus den Audiodaten berechnet und temporär gespeichert. Danach erfolgt die eigentliche Klassifizierung.

Die Klassifizierung ist bei AGENDER auf zwei Ebenen verteilt: Auf der *ersten Ebene* wird die Mustererkennung (vgl. Abschnitt 2.2.1) mit den in der Entwurfsphase erstellten

[5] *Pulse Code Modulation*, Standardverfahren zur Repräsentation digitaler Audiodaten

Klassifizierern durchgeführt. Dabei kann jeder Klassifizierer unterschiedliche Merkmalsvektoren anfordern und Ergebnisse zurückliefern, welche nicht unmittelbar den gesuchten Sprechereigenschaften zugeordnet werden können. Diese Zuordnung wie auch die Kombination der Resultate der verschiedenen Klassifizierer erfolgt auf der *zweiten Ebene* durch das ebenfalls in der Entwurfsphase generierte dynamische Bayessche Netz (vgl. Abschnitt 2.2.4). Die Ausgabeknoten dieses Netzes enthalten dann die Wahrscheinlichkeiten für alle Eigenschaftsklassen des Klassifizierungsproblems.

Neben diesen Wahrscheinlichkeiten, die an sich schon sehr interessant für bestimmte Anwendungen sein können, wird das Endergebnis dargestellt durch die Bezeichnung der Klasse, welche die höchste Wahrscheinlichkeit im Bayesschen Netz erhält. Damit ist die Klassifizierung beendet.

# 3 Eine Architektur für die Sprecherklassifikation auf multiplen Plattformen

Hauptgegenstand dieses Kapitels ist die Beschreibung der im Rahmen dieser Arbeit entwickelten SBC-Architektur im Detail. Zuerst sollen die wichtigsten Ziele der Lösung in Form einer Aufstellung von Anforderungen festgehalten werden. Danach wird ein Überblick über den Gesamtaufbau von SBC gegeben. Eine Architekturskizze soll die Zusammenhänge zwischen den einzelnen Komponenten verdeutlichen, welche dann in den darauf folgenden Abschnitten eingehend behandelt werden. Zuvor noch werden die beiden Ansätze M3I und SBC verglichen und der Übergang zu Letzterem erörtert. Im Anschluss daran werden die wichtigsten Ideen und Konzepte vorgestellt, mit denen die Erfüllung der genannten Anforderungen gelingen soll. Dabei wird insbesondere auch Bezug genommen auf die in Abschnitt 1.1 aufgezählten Limitierungen der M3I-Architektur. Die Vorstellung der Komponenten von SBC beginnt mit dem Klassifikationsmodul und geht dann über zur Entwicklungsplattform. Für das Verständnis dieser Abschnitte sind die Informationen in Kapitel 2 von besonderer Relevanz. Im letzten Abschnitt dieses Kapitels wird dann noch das Gesamtkonzept näher beschrieben, welches schon in Abschnitt 1.1 kurz angesprochen wurde.

## 3.1 Anforderungen

Aufbauend auf der Notwendigkeit einer universell einsetzbaren Architektur zur Sprecherklassifikation lassen sich einige Anforderungen an die Implementierung formulieren. Neben den grundsätzlichen Forderungen gibt es auch mehrere Eigenschaften, die darüber hinaus wünschenswert sind.

### 3.1.1 Sprecherklassifizierung mit hoher Genauigkeit

Primäres Ziel von SBC ist die Klassifizierung des Benutzers auf Basis von Sprache. Von großer Bedeutung hierbei ist es, dass die Klassifizierungsgenauigkeit möglichst hoch ist, oder anders ausgedrückt, dass so wenige Fehlklassifikationen auftreten wie möglich. AGENDER trifft keine Festlegung über den zu verwendenden Klassifizierer, sondern

gestattet es, je nach Problem flexibel die Klassifizierer und die Implementierung der zweiten Ebene zu gestalten. Beides sind Faktoren, die sehr stark über die Qualität des Klassifizierungsergebnisses entscheiden. Dazu kommen noch eine Reihe weiterer Umstände, wie z.B. die Größe und Eignung des Trainingskorpus, die Wahl des Audioformats und die Implementierung der Merkmalsextraktion.

Bei den meisten der genannten Einflussfaktoren auf die Qualität besteht jedoch ein Trade-Off zwischen der maximal möglichen Klassifikationsgenauigkeit und weiteren Zielen wie hohe Geschwindigkeit und geringer Ressourcenverbrauch. Folglich ist es Aufgabe der Architektur, die Flexibilität bei der Wahl der Methoden zu gewährleisten, so dass unter Berücksichtigung der Prioritäten des jeweiligen Anwendungskontexts klassifiziert werden kann.

### 3.1.2 Hohe Performanz

In den meisten Szenarien ist eine gute Performanz eine wichtige Anforderung, oft gleichgestellt mit der Klassifikationsgenauigkeit. Dies wird besonders am Beispiel des Callcenters deutlich: Je schneller das Ergebnis über den Anrufer vorliegt, desto früher kann das Durchstellen erfolgen. Reaktionszeiten im Bereich von deutlich unter einer Sekunde sind hier erstrebenswert.

Um eine gute Performanz zu erhalten, ist auf Seiten der Implementierung die Wahl der Programmiersprache sehr wichtig. Sprachen, welche direkt in Maschinencode übersetzt werden und viele Optimierungen der Geschwindigkeit erfahren (z.B. *C/C++*), besitzen hier einen großen Vorteil gegenüber Sprachen, welche in einem Interpreter (z.B. *Basic*) oder einer Virtuellen Maschine (z.B. *Java* oder *C#*) ausgeführt werden.

Die Sprache alleine ist jedoch nur ein Faktor bei der Betrachtung der Geschwindigkeit. Eine ebenfalls wichtige Rolle spielt der Code selbst. Die verwendeten Algorithmen sollten möglichst geringe Laufzeiten besitzen, was meist durch eine niedrige komputationelle Komplexität erreicht werden kann.

Von Vorteil kann auch eine kompakte Gestaltung des Klassifikationsmoduls sein. Generell ist es performanter, wenn die gesamte Funktionalität sich in einem Modul befindet, als wenn verschiedene externe Komponenten aufgerufen werden müssen, möglicherweise gar über Prozessgrenzen hinweg. Dies kann natürlich nur dann gewährleistet werden, wenn alle Bestandteile des Moduls im Quellcode verfügbar sind.

Wenn die Anwendung in der Praxis betrachtet wird, tritt ein weiterer Performanz-Aspekt in den Vordergrund, und zwar die Verfügbarkeit (engl. „availability“): Dies bedeutet, dass bei einer neuen Anforderung zur Klassifizierung einer Äußerung die Zeitspanne zwischen dem Stellen der Anfrage durch die aufrufende Anwendung und dem Beginn der Verarbeitung möglichst gering ist, und dass die Verarbeitung nicht durch andere Vorgänge unterbrochen wird. Um eine möglichst gute Verfügbarkeit zu erreichen,

ist zum einen die Konzeption des Moduls im Hinblick auf parallele Anforderungen von Bedeutung (vgl. Abschnitt 3.1.7). Bei der Koordination zweier gleichzeitig eintreffenden Anfragen kommt es meist zu einem Koordinierungsprozess, bei dem eine der Anforderungen warten muss (*locking*). Diese Locking-Phase (auch als *Kritischer Abschnitt* bezeichnet) sollte jedoch auf ein Minimum beschränkt werden, um schnelle Antwortzeiten zu garantieren. Weitere Beispiele für Faktoren, die eine schlechte Verfügbarkeit bewirken, sind Hintergrundprozesse des Betriebssystems oder der Virtuellen Maschine, wie z.B. eine automatische *Garbage Collection*.

### 3.1.3 Modularität

AGENDER zeichnet sich unter anderem dadurch aus, dass es nicht an ein bestimmtes Sprechermodell gebunden ist, sondern mit einer freien Kombination aus beliebigen Klassifikationsalgorithmen auf der ersten Ebene und Bayesschen Netzen auf der zweiten Ebene umgehen kann. Dies ist eine Eigenschaft, die es in der Praxis sehr vielseitig anwendbar macht, da es je nach Anforderungen des Einsatzbereichs so angepasst werden kann, dass es diesen Anforderungen gerecht wird.

Diese Flexibilität soll für SBC beibehalten werden, woraus sich die Notwendigkeit ergibt, dass sowohl die Klassifizierer als auch das Bayessche Netz frei gewählt und je nach Anwendung in das Klassifizierungsmodul eingesetzt werden können. Dies könnte zwar erreicht werden, indem die Klassifizierer in Dateien eines bestimmten Formats gespeichert und bei der Ausführung geladen werden; die Herausforderung besteht jedoch darin, trotz dieser Modularität die Kompaktheit und Performanz zu bewahren, was bedeutet, dass sowohl die Klassifizierer, als auch das Bayessche Netz in kompilierter Form vorliegen und im gleichen Prozess wie das Modul ausgeführt werden sollten. Ein solcher Ansatz erfordert ein besonderes Konzept, da die üblichen Verfahren zum Speichern von Daten hier nicht geeignet sind. Eine Lösung zur Integration der Klassifizierer im Klassifikationsmodul wird in Abschnitt 3.3.4 beschrieben.

In jedem Fall besteht die Notwendigkeit, ein Format zu definieren – sei es portierbarer Code oder ein benutzerdefiniertes interpretierbares Dateiformat – durch das die Klassifizierer und das Bayessche Netz in einer für das Klassifikationsmodul erschließbaren Form repräsentiert werden können. Besonders vorteilhaft wäre es, wenn diese Repräsentation aus der des M3I-Servers bzw. einer Entwicklungsplattform abgeleitet werden könnte, um so den Klassifizierer nur auf einer Plattform trainieren zu müssen, nämlich der Entwicklungsplattform. Die in Abschnitt 3.1.9 vorgestellte SBC DEVELOPMENT PLATFORM erfüllt diesen Zweck.

### 3.1.4 Unterstützung multipler Plattformen

Bedarf an eingebetteter Sprecherklassifikation besteht bei einer Vielzahl von Anwendungen, die auf unterschiedlichen Plattformen ausgeführt werden. In der Regel existiert eine feste Softwareumgebung, in welche die Komponente integriert werden soll (vgl. Abschnitt 3.1.6). Es kann nicht davon ausgegangen werden, dass die Rahmenarchitektur den Systemvoraussetzungen der Klassifikationskomponente entspricht oder an selbige angepasst wird, da dies im ungünstigen Fall mit einer Portierung oder Neuimplementierung verbunden wäre. Also muss das Modul so konzipiert sein, dass es sich auf einem möglichst breiten Spektrum von Plattformen installieren lässt.

Bei dieser Fragestellung wird oft der Begriff der (vollständigen) *Plattformunabhängigkeit* in die Diskussion gebracht. Darunter versteht man, dass die gleiche Anwendung ohne Veränderung in mehreren Umgebungen ausgeführt werden kann. Allerdings setzt dies meist voraus, dass auf jeder Zielplattform ein generisches Framework installiert ist, welches die Ausführung ermöglicht. Dabei handelt es sich im Allgemeinen um eine virtuelle Maschine. Dies ist bereits das erste Problem bei dieser Variante: Die Plattformunabhängigkeit des Moduls würde erhebliche Performanzeinbußen bei der Ausführung in der virtuellen Maschine mit sich bringen, weshalb diese Lösung für SBC keine realistische Wahlmöglichkeit darstellt. Hinzu kommt noch die Tatsache, dass die Vorteile einer echten Plattformunabhängigkeit nicht zur Geltung kämen, da auch in diesem Fall für jede Plattform ein spezifisches Modul erzeugt wird, welches die benötigten Klassifizierer enthält.

Als Alternative zur vollständigen Plattformunabhängigkeit besteht die Möglichkeit, für jede Plattform eine eigene Version zu erzeugen. Dies hat den unbestrittenen Vorteil, dass jede Version auf Assembler-Ebene hoch optimiert ist für die jeweilige Plattform und daher schneller als ein plattformunabhängiges Produkt. Darüber hinaus besteht keine Abhängigkeit von einer Laufzeitumgebung wie einer virtuellen Maschine, so dass auch weniger verbreitete Plattformen angesteuert werden können, und die Komplexität zudem noch deutlich geringer ist. Der Erfolg dieser Variante hängt davon ab, ob eine gute Infrastruktur zum Erstellen der plattformspezifischen Versionen existiert, z.B. eine Abstraktionsebene für Plattforminterna (*Platform Abstraction Layer*, PAL). Wenn die passenden Werkzeuge zur Verfügung stehen ist dieser Weg sehr viel versprechend.

Aufgrund der Projekte, die maßgeblich für die Entwicklung von SBC verantwortlich waren, kommen noch zwei konkrete Plattformanforderungen hinzu: Der SHOP-ASSIST (vgl. Abschnitt 4.1) wird auf dem *PocketPC* ausgeführt, weshalb das SBC-Klassifikationsmodul unter *Windows Mobile 5.0* lauffähig sein muss. Ein entsprechendes Framework wird mit dem *Windows Mobile 5.0 SDK* von *Microsoft* zur Verfügung gestellt. Die Telefonie-Server-Anwendung (vgl. Abschnitt 4.2) läuft in einer *Windows 2000 Server*-Umgebung, welche ebenfalls unterstützt werden muss. Die zum Kompilieren er-

forderlichen Dateien sind im *Microsoft Platform SDK* enthalten.

### 3.1.5 Ressourcenadaptivität

SBC kann sowohl auf Server-Plattformen mit großzügiger Hardware-Ausstattung, als auch auf mobilen Clients mit wenig Speicher und geringer Rechenleistung zum Einsatz kommen. Das Klassifizierungsmodul muss aus diesem Grund möglichst ressourcensparend, im Optimalfall sogar ressourcenadaptiv implementiert sein. Ressourcenadaptivität würde es gestatten, dass für Plattformen mit geringer Leistungsfähigkeit entsprechend wenige Ressourcen in Anspruch genommen werden, bei mehr zur Verfügung stehenden Kapazitäten diese allerdings auch zur Leistungssteigerung voll ausgeschöpft werden können, z.B. durch Implementierung von Caches bei vorhandenen Arbeitsspeicherreserven.

Daneben hängt der tatsächliche Ressourcenverbrauch auch von den verwendeten Klassifizierern und der Größe der Trainingsdatenbank ab. Ein Nearest-Neighbor-Klassifizierer verbraucht beispielsweise mehr Speicher als die meisten anderen Algorithmen, und ein Entscheidungsbaum kommt im Gegensatz zu anderen Methoden ohne Fließkomma-Rechnungen aus. Solche Eigenschaften sind in der Entwurfsphase abzuwägen, damit ein für das Einsatzszenario passender Kompromiss gefunden werden kann.

### 3.1.6 Integrierbarkeit

Das SBC-Klassifikationsmodul ist keine allein stehende Anwendung, sondern stellt die Sprecherklassifikation als Dienst einem Konsumenten wie anderen Anwendungen, Applets oder Komponenten zur Verfügung. Die Art dieser Anwendung und die Programmiersprache, in der sie geschrieben ist, spielen dabei keine Rolle, entscheidend ist nur die Schnittstelle zur Integration. Das SBC-Modul stellt damit eine Komponente oder Bibliothek dar.

Die Schnittstelle sollte so universell wie möglich sein, ohne jedoch zusätzlichen Aufwand durch redundante Konvertierung von Daten zu verursachen. Außerdem ist eine einfache Bedienung seitens des Aufrufers wünschenswert, um eine schnelle und fehlerfreie Einbindung sicherzustellen.

Die Integrierbarkeit geht aber noch über die genannten Aspekte hinaus. Auch von Seiten der Ein- und Ausgabedaten sollte eine hohe Kompatibilität gewährleistet sein, z.B. was die Unterstützung von Audioformaten betrifft.

### 3.1.7 Skalierbarkeit

Skalierbarkeit ist ein weiterer Aspekt, der je nach Anwendung von sehr großer Bedeutung sein kann. Letzteres ist bei dem Telefonie-Projekt der Fall. Hierbei läuft die Sprecherklassifikation auf einem Hochleistungs-Server, welcher ein Datenaufkommen von bis zu

mehreren hundert Äußerungen zur gleichen Zeit bewältigen muss. Es gibt einige bekannte Entwurfsmuster im Software Engineering, die sich dieser Aufgabenstellung annehmen und Möglichkeiten zur effizienten Implementierung von paralleler Verarbeitung anbieten, z.B. *Clustering* oder *Pipelining* (vgl. Abschnitt 3.3.3). Eine solche Parallelverarbeitung ist unumgänglich, wenn große Verarbeitungsmengen bei kleinen Latenzen realisiert werden sollen, und sie macht sich besonders bezahlt, wenn mehrere Prozessoren zur Lastverteilung gegeben sind. Dennoch muss aber gewährleistet sein, dass die Architektur auch bei nur einem (exklusiven) Nutzer die optimale Performanz liefert.

### 3.1.8 Robustheit

An die Komponenten von serverseitigen Diensten werden besondere Anforderungen bezüglich der Robustheit gestellt. Die Komponente sollte im durchgehenden Betrieb laufen können, ohne Instabilitäten zu verursachen. Bei einer Desktop-Implementierung profitieren die meisten Anwendungen von der Tatsache, dass der Rechner oder die Anwendung nach einiger Zeit neu gestartet wird, was automatisch bewirkt, dass durch den Neustart potenzielle Absturzursachen eliminiert werden. Ein Server wird aber sehr viel seltener neu gestartet (im Optimalfall nie), so dass hier die Gefahr eines Absturzes größer ist. Durch Befolgung zentraler Programmierrichtlinien wie Speicherbereinigung oder Vermeidung von überflüssigem Code kann die Stabilität erhöht werden. Vor der Weitergabe kann das Programm dann nochmals mittels verschiedener Debugging-Werkzeuge getestet und verifiziert werden. Genannt seien hier Diagnoseprogramme für *Speicherlecks* (engl. „memory leaks“) oder *Unit-Tests*, mit welchen zuvor definierte Testfälle durchlaufen werden.

Zusätzlich zu robustem Code sollte das Modul auch über *Tracing-Funktionalität* verfügen. Dabei handelt es sich um Möglichkeiten, zu Testzwecken wichtige Nachrichten und Zustandsmeldungen des Moduls in einer Protokolldatei zu speichern.

### 3.1.9 Unterstützung der Entwicklung

Die bis jetzt genannten Anforderungen beziehen sich in der Hauptsache auf den Teil der Software, welcher für die Ausführung der Anwendung, die Sprecherklassifikation nutzt, benötigt wird. Fester Bestandteil von SBC ist allerdings auch die Anpassung dieser Klassifikationskomponente an die Anwendung durch Zusammenstellung der ersten und zweiten Ebene von AGENDER. Zu diesem Zweck soll auch eine *Entwicklungsplattform* bereitgestellt werden, die alle benötigten Werkzeuge zum Entwickeln, Trainieren und Evaluieren der Klassifizierer enthält. Dieses Programm kann unabhängig vom Klassifikationsmodul sein und muss keine herausragenden Eigenschaften im Hinblick auf Portierbarkeit oder Performanz besitzen, da sie nur in der Entwicklungsumgebung genutzt wird.

## 3.2 SBC-Architektur im Überblick

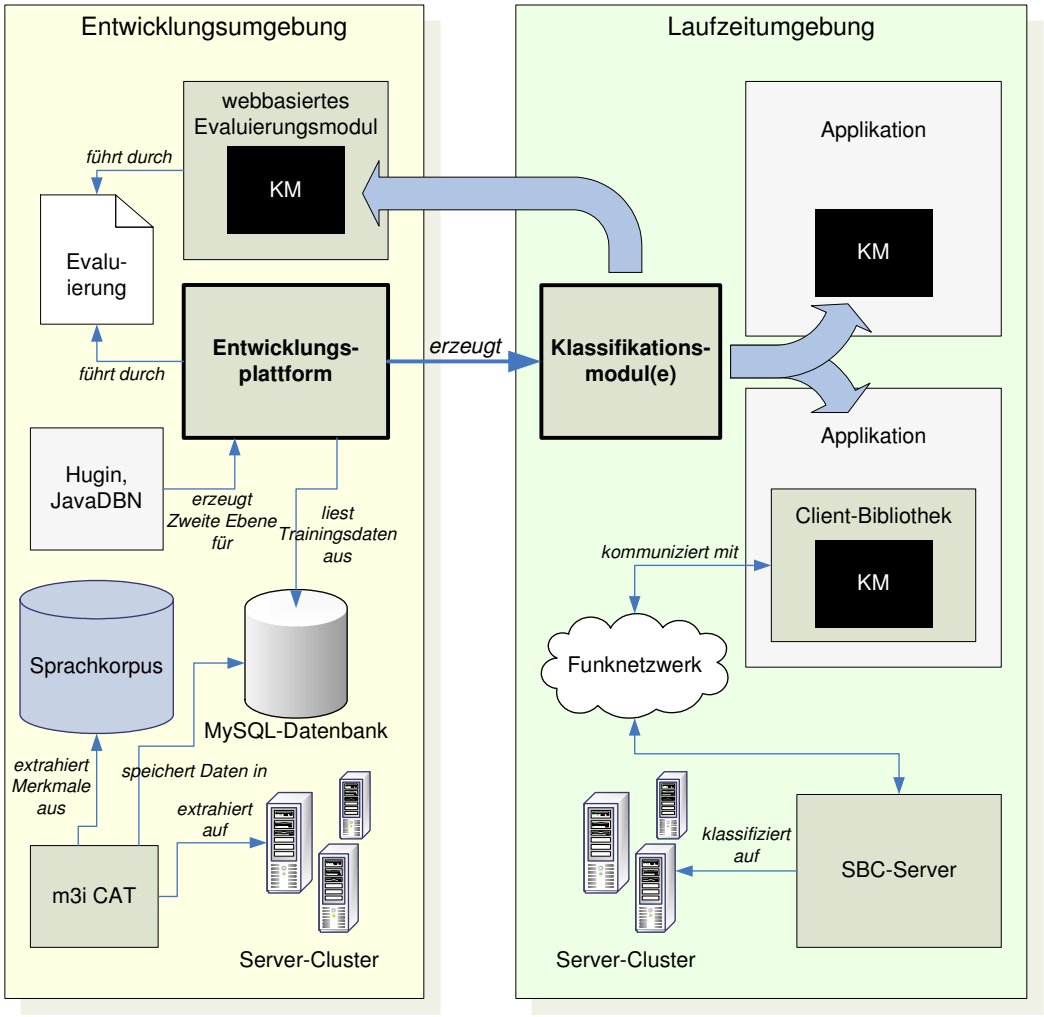



Abbildung 3.1: Bestandteile der SBC-Architektur

Abb. 3.1 stellt alle Teile der SBC-Architektur in ihrer Rolle innerhalb des konzeptionellen Rahmens dar und zeigt ihre Zusammenwirkung auf.

Die beiden primären Säulen der Konzeption sind das **Klassifikationsmodul** und die **Entwicklungsplattform**. Beide werden für den Einsatz von AGENDER benötigt: die Entwicklungsplattform in der Entwurfsphase und das Modul für die Ausführungsphase. Das Modul wird in die eigentliche Anwendung integriert, wobei es häufig notwendig oder hilfreich ist, eine weitere Abstraktionsebene zwischenzuschalten, z.B. um plattformspezifische Besonderheiten zu kapseln, oder wenn aufgrund verschiedener Pro-

grammiersprachen keine direkte Einbindung erfolgen kann. Speziell für den *PocketPC* wurde eine Bibliothek entwickelt, mit Hilfe derer kompatible *PocketPC*-Anwendungen sehr leicht die Dienste von AGENDER, sowohl in der eingebetteten Version als auch über einen externen Server, in Anspruch nehmen können. Es handelt sich dabei um die **SBC-Client-Bibliothek**, die in Kapitel 4 vorgestellt wird.

Für eine umfassendere Beschäftigung mit der Materie der Sprecherklassifikation werden außer der Entwicklungsplattform noch Werkzeuge zum Verwalten und Analysieren des Stimmenkorpus benötigt. Hier können für SBC die gleichen Werkzeuge genutzt werden, die auch schon bei Müller (2005) für M3I zum Einsatz kommen: Auf dem gesamten Sprachkorpus wird eine automatisierte Merkmalsextraktion über *Praat*-Skripte durchgeführt, welche auf einem **Server-Cluster** läuft. Die Ergebnisse werden in einer **MySQL-Datenbank** gespeichert. Mithilfe des Werkzeugs **M3I CAT** können anschließend verschiedene Analysen auf dem Datenbestand durchgeführt werden, z.B. Visualisierung der Daten, Gauß'sche Verteilungsdichte, Korrelation und weitere. Die dadurch gewonnenen Erkenntnisse werden bei der Auswahl der Merkmale für die Klassifizierer genutzt, welche dann – basierend auf der gleichen Datenbank – in der Entwicklungsumgebung trainiert werden. Nach Abschluss des Trainings erfolgt die Evaluierung und ggf. Visualisierung der Ergebnisse durch M3I CAT. Sind die Resultate zufrieden stellend, kann die zweite Ebene konfiguriert werden. Dies erfolgt zurzeit noch durch eine externe Applikation (*Hugin Expert*). Das Bayessche Netz wird dann automatisch mit einem von Brandherm entwickelten Compiler, welcher auf dem in Brandherm und Jameson (2004) vorgestellten Verfahren beruht, in eine performante Implementierung überführt (vgl. Abschnitt 3.7.6).

Ebenfalls aus M3I übernommen wurde die servergestützte Klassifikation, die nach wie vor ihre Verwendung in einigen Szenarien findet, beispielsweise als leistungsstärkere Alternative zur Klassifikation auf dem mobilen Gerät, sofern die Verbindung zu einem Server besteht. Als **SBC-Server** kann entweder der Prototyp des M3I-Servers zum Einsatz kommen oder – wie auf dem Client – eine plattformspezifische Version des Klassifikationsmoduls. Der M3I-Server bietet derzeit den Vorteil, dass die Verarbeitung auch auf einem Cluster erfolgen kann, was je nach Auslastung und Hardware-Umgebung unter Umständen effizienter ist.

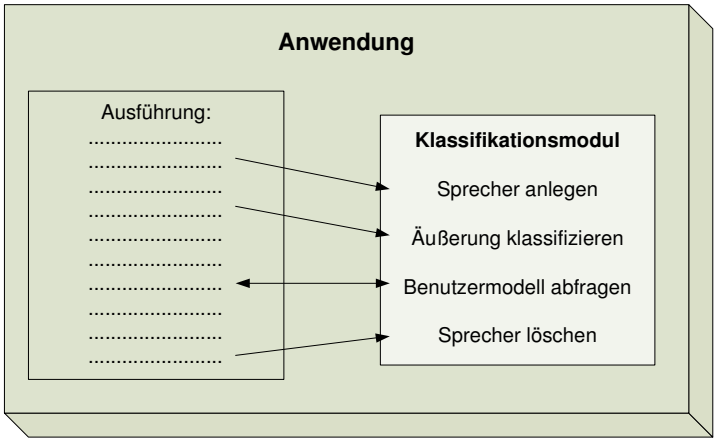



Abbildung 3.2: Verwendung des Klassifikationsmoduls durch eine Applikation (eingebettete Klassifikation)

## 3.3 Zentrale Konzepte

Die Entwicklung der SBC-Architektur zur Sprecherklassifikation machte den Entwurf einiger grundlegend neuer Konzepte im Rahmen dieser Arbeit erforderlich, um die in Abschnitt 1.1 beschriebenen Probleme von M3I beim Einsatz in der Praxis auf verschiedenen Plattformen lösen und den in Abschnitt 3.1 genannten Anforderungen entsprechen zu können.

### 3.3.1 Eingebettetes Klassifikationsmodul

Die Sprecherklassifikation soll auf einer Reihe von Zielplattformen ausgeführt werden können. Dazu stellt SBC eine Komponente bereit, welche die Ausführungsphase von AGENDER durchführt. Diese Komponente besitzt weder eine grafische Oberfläche, noch kann sie über eine Konsole gesteuert werden. Sie wird auch nicht gestartet oder beendet. Sie ist vollständig in die Anwendung oder Plattform integriert, d.h. sie ist sowohl vom Programmcode her, als auch von Seiten der Ausführung in die Host-Anwendung eingebettet. Die somit anwendertransparent genutzte prozessinterne Komponente stellt der Applikation einen Dienst zur Verfügung, nämlich die Sprecherklassifikation. Das Modul ist ferner eine in sich abgeschlossene Einheit, die nicht von weiteren Komponenten abhängt. Aus diesen Gründen wird diese Komponente als SBC EMBEDDED CLASSIFICATION MODULE oder einfach *(eingebettetes) Klassifikationsmodul* bezeichnet. Dieser Sachverhalt ist in Abb. 3.2 dargestellt.

In der genannten Abbildung wird auch die Aufgabe des Moduls deutlich: Die Anwendung stellt eine Anfrage zur Klassifizierung einer Äußerung. Um mehrere Äußerungen

zeitgleich zu verarbeiten, aber dennoch mehrere Äußerungen des gleichen Sprechers korrekt zuordnen zu können, damit der Effekt der Erhöhung der Wahrscheinlichkeit des Ergebnisses durch statistische Konvergenz genutzt wird, ist ein Mechanismus erforderlich, welcher die Verwaltung mehrerer Sprecher erlaubt. Bevor die eigentliche Verarbeitung einer Äußerung erfolgt, wird ein neuer Sprecher angelegt. Der Verweis auf diesen Sprecher kann dann von der aufrufenden Anwendung dazu genutzt werden, die darauf folgenden Äußerungen diesem oder einem anderen Sprecher zuzuordnen. Wird nun eine solche Äußerung einem Sprecher zugeordnet und dem Modul zur Klassifizierung übergeben, so läuft dieser Vorgang vollständig innerhalb des Moduls ab, bis das Ergebnis feststeht. An dieser Stelle sei auf große Unterschiede zum M3I-Server hingewiesen, welcher die Verarbeitung über verschiedene externe Prozesse, zum Teil sogar andere Rechner, abwickelt. Der reduzierte Overhead und die verringerten Systemanforderungen kommen der Verarbeitungsgeschwindigkeit zugute.

Sobald die Verarbeitung beendet ist, kann die Anwendung die Ergebnisse abrufen. Dies geschieht in einem gesonderten Aufruf, da womöglich die Resultate nicht nach jeder Äußerung benötigt werden, sondern nur an bestimmten Stellen im Anwendungsablauf. Beispielsweise könnten bei einem ACD-System dem Anrufer zuerst eine Reihe von Fragen gestellt werden, und nach Abschluss dieser Fragen die Weiterleitung, basierend auf den aktuellen Einschätzungen von Alter und Geschlecht, erfolgen. Durch den separaten Aufruf müssen weniger Daten übertragen werden, was die Leistung verbessert. Zudem sind Informationsakquisition und Informationsapplikation dadurch unabhängig voneinander. Die Übermittlung der Ergebnisse erfolgt in einer generischen Wörterbuch-Datenstruktur, wobei für die Schlüssel eine Kombination aus Eigenschaftsname und Ergebnistyp verwendet wird (vgl. Abschnitt 3.8.2). Man kann in diesem Zusammenhang auch von einem kompakten Benutzermodell sprechen, welches durch das Klassifikationsmodul verwaltet und durch die Anwendung abgefragt wird.

Es können so viele Äußerungen eines Sprechers wie nötig klassifiziert werden. Das dynamische Bayessche Netz wird automatisch um die neue Zeitscheibe erweitert. Der zusätzlich benötigte Speicherplatz pro Äußerung ist dabei minimal. Wenn die Daten eines Sprechers nicht mehr benötigt werden, müssen diese gelöscht werden, um den Speicherplatz freizugeben. Wann dieser Zeitpunkt erreicht ist, entscheidet die Anwendung. Eine *PocketPC*-Applikation wird in der Regel während der gesamten Laufzeit mit nur einem Sprecher arbeiten. Allerdings kann es sinnvoll sein, dem Benutzer eine Funktion zum expliziten Sprecherwechsel zur Verfügung zu stellen. In diesem Fall würde dann das aktuelle Benutzermodell gelöscht und ein neuer Sprecher ohne Vorwissen angelegt. Weitere Informationen zur Kommunikation zwischen dem Modul und der Host-Anwendung finden sich in Abschnitt 3.8.

Eine echte Plattformunabhängigkeit des Klassifikationsmoduls ist nicht erstrebenswert. Die vorliegende Implementierung in *C++* besitzt jedoch alle Voraussetzungen,

um für unterschiedliche Plattformen kompiliert werden zu können. Erfolgreiche Tests unter *Windows 2000* und *PocketPC 2002* wurden bereits absolviert. Wie genau die Plattformtransparenz erreicht wird, kann in Abschnitt 3.5 nachgelesen werden.

Das Klassifikationsmodul ist in jeglicher Hinsicht auf möglichst hohe Performanz ausgelegt. Außerdem kommt es mit nur geringen Ressourcen aus, so dass es auch auf einem mobilen Gerät erfolgreich eingesetzt werden kann. Näheres hierzu wird in Abschnitt 3.6 vermittelt.

Details bezüglich der Implementierung des Moduls werden in Abschnitt 3.7 erörtert. Dort ist auch zu erfahren, wie der AGENDER-Ansatz und seine einzelnen Ebenen konkret für SBC umgesetzt wurden.

### 3.3.2 SBC-Entwicklungsplattform

Die zweite wichtige Komponente von SBC neben dem Klassifikationsmodul ist die Entwicklungsplattform, welche auch die Bezeichnung SBC DEVELOPMENT PLATFORM trägt. Ziel dieser Plattform war es, ein Werkzeug zum Konstruieren und Testen der Klassifizierer in der Entwurfsphase bereitzustellen, wie unter Abschnitt 3.1.9 ausgeführt. Aus praktischen Gründen wurde dafür der vorhandene M3I-Server als Codebasis genutzt und entsprechend modifiziert bzw. weiterentwickelt. Aufgabe des Programms ist nun nicht mehr die Live-Klassifikation, allerdings auch nicht ausschließlich das Training der Klassifizierer. Vielmehr wird die Anwendung auch zur Weiterentwicklung von SBC und der Klassifikationstechnologie selbst verwendet, was kein Problem darstellt, da die entsprechende Architektur bereits mit dem M3I-Server gegeben ist. Der Aufbau und die Funktionsweise der Entwicklungsplattform werden eingehend in Abschnitt 3.9 behandelt.

### 3.3.3 Pipeline-Verarbeitung

Eines der Kernkonzepte von SBC ist das der Pipeline-Verarbeitung. Dieser Ansatz wurde eingeführt, um eine möglichst schnelle und effiziente Abarbeitung von Äußerungen im Klassifikationsmodul zu gewährleisten.

Der M3I-Server verwendet als interne Kommunikationsschnittstelle ein *Blackboard* (s. Abb. 3.3), wobei es sich um eine Art Pool von Objekten handelt. Dazu gehören dann Dienste (*Services*) wie z.B. einzelne Klassifizierer, die nach einer einmaligen Registrierung beim Blackboard die vorhandenen Objekte abrufen und neue Objekte hinzufügen dürfen. Bei jedem neuen Objekt, z.B. einer zu klassifizierenden Audiodatei oder einem Klassifizierer-Ergebnis, werden alle registrierten Dienste benachrichtigt. Diese Architektur erlaubt zwar ein einfaches Hinzufügen von neuen Diensten und Objekten, jedoch entsteht ein nicht unerheblicher Kommunikationsaufwand, da die Dienste nur über das Blackboard miteinander in Kontakt treten können und jeder Dienst bei jedem Objekt

dieses zuerst untersuchen muss, bevor er es ablehnen oder annehmen kann. Aus diesem Grund ist sie für die eingebettete Variante nicht sinnvoll.

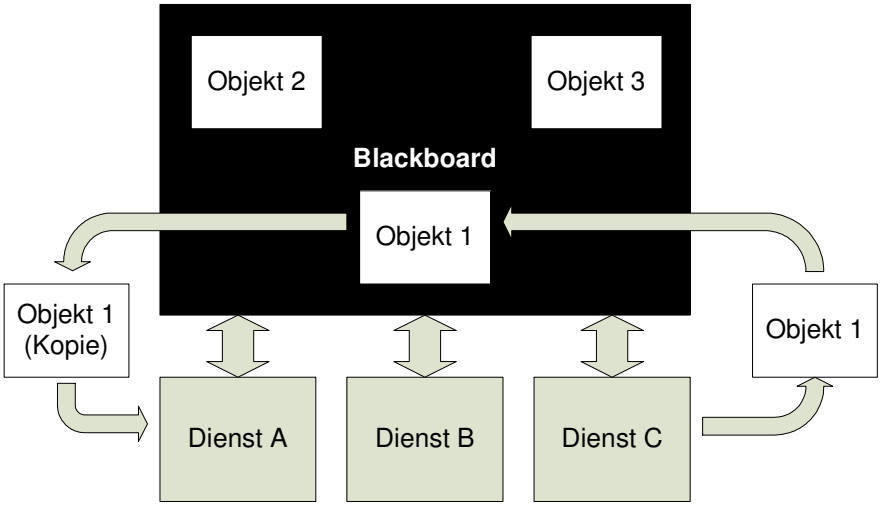



Abbildung 3.3: *Blackboard*-Architektur

Eine Alternative besteht darin, auf die Kommunikationsstruktur komplett zu verzichten und die Dienste direkt und gezielt miteinander kommunizieren zu lassen (s. Abb. 3.4). Dadurch würde sich eine Einsparung bei der Kommunikation ergeben, da nur noch diejenigen Dienste angesprochen werden müssen, welche tatsächlich angefordert sind. Dann wird jedoch eine allgemeine Schnittstelle zwischen den Diensten benötigt, über die ein Objekt in generischer Form ausgetauscht werden kann. Dies ist jedoch ebenfalls nicht effizient, da spezielle Datenstrukturen beim Übertragen in allgemeinen Container-Objekten oft kopiert oder zumindest mit zusätzlichen Daten annotiert werden müssen.

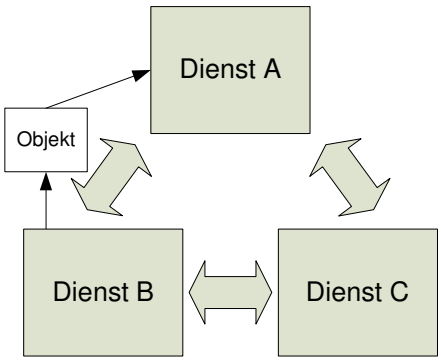



Abbildung 3.4: Direkte Kommunikation zwischen den Diensten

Dieses Problem kann mittels einer Architektur mit einer *Controller-Instanz* eliminiert werden (s. Abb. 3.5). In diesem Szenario gibt es einen Steuerungsdienst, der nur für die Weiterleitung der Daten an die eigentlichen Arbeitsdienste zuständig ist. Der Steuerungsdienst wird für ein konkretes Szenario geschrieben oder angepasst und kennt daher die Schnittstellen der anderen Dienste, d.h. er kann die Daten im jeweils passenden Format annehmen und weiterleiten. Dies ist bereits eine sehr effiziente Variante, die sich auch gut in SBC hätte einsetzen lassen.

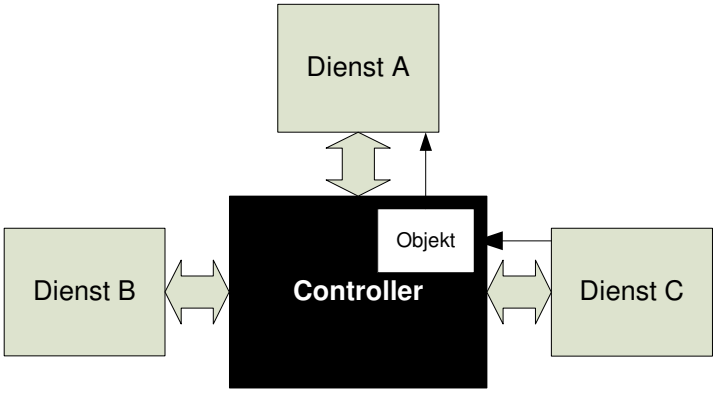



Abbildung 3.5: *Controller*-Architektur

Durch den Aufbau von AGENDER bietet sich jedoch noch eine weitere Alternative, welches einen Spezialfall des zuletzt genannten Ansatzes darstellt: Die Hauptprozesse, die für die Klassifizierung einer einzelnen Äußerung ausgeführt werden müssen, laufen alle sequenziell in einer festen Reihenfolge ab. Diese Verarbeitung kann man sich wie eine vertikale Röhre vorstellen, in welche oben die Daten eingegeben werden, diese dann nach unten durch die Röhre wandern und als Endprodukt diese wieder verlassen. Man spricht wegen diesem Vergleich auch von einer *Pipeline-Architektur* (s. Abb. 3.6). Charakteristisch dafür sind in jedem Fall die zwei externen Zugriffspunkte (Ein- und Ausgang) sowie die feste, sequenzielle Verarbeitungsrichtung. Die Implementierung der gesamten Verarbeitung als Pipeline stellt somit eine weitere Optimierung der Geschwindigkeit dar.

Es gibt einen Fall, bei dem ein Einwand gegen diese Methode vorgebracht werden könnte, und zwar bei der Klassifizierung. Wenn jeder Klassifizierer als eigener Dienst implementiert wird wie bei dem M3I-Server, dann wäre tatsächlich auch eine parallele Verarbeitung möglich. Hier hat sich aber gezeigt, dass die Laufzeit eines solchen Klassifizierungsdienstes vergleichsweise gering ist im Verhältnis zu dem Aufwand, der nötig wäre, um einen neuen Verarbeitungs-Thread zu erzeugen und die Daten zu kopieren.

Nichtsdestoweniger ist das Pipeline-Modell sogar sehr gut zur Parallelverarbeitung geeignet, da die Pipeline selbst durch die einfache Gestaltung so gut wie keinen Speicher

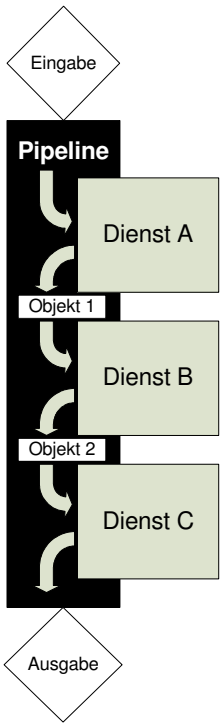



Abbildung 3.6: *Pipeline*-Architektur

benötigt und daher problemlos für jede Äußerung eine eigene Pipeline erstellt werden kann. Da die Dienste keine dienst-lokalen Daten zur aktuellen Äußerung speichern, sondern nur den thread-spezifischen Kontext verwenden[1], können sie zentral verwaltet und von verschiedenen Pipelines gemeinsam genutzt werden.

### 3.3.4 Erzeugen von Klassifikationsmodulen

In Abschnitt 3.1.3 wurde die Problematik eines modularen Aufbaus des Klassifikationsmoduls angesprochen. Der Teil des Klassifikationsmoduls, welcher die Schnittstelle, Pipeline und Merkmalsextraktion enthält und für jede Anwendung einer bestimmten Zielplattform identisch ist, soll als *Basiscode* bezeichnet werden. Dieser Basiscode kann zu einer fertigen Einheit kompiliert und in eine Anwendung integriert werden. Was dann noch fehlt sind die *kontextspezifischen Daten* wie Klassifizierer und die zweite Ebene. Da diese ohnehin den einzigen Teil des Moduls darstellen, welcher sich zwischen verschiedenen Versionen und Varianten verändert (einmal abgesehen von grundlegenden Weiterentwicklungen des Basiscodes), scheint es auf den ersten Blick plausibel, diese

[1] Dies bedeutet, dass Daten nicht über den Definitionsbereich einer Prozedur hinaus gültig sind, mit der Ausnahme des Rückgabewerts.

Daten in Form von zusätzlichen Konfigurationsdateien zu der Anwendung hinzuzufügen. Dadurch ergäbe sich auch die Möglichkeit, diese ohne Neuerstellung der Anwendung austauschen zu können.

Hierbei tritt aber ein erheblicher Nachteil zu Tage: Die Klassifizierer usw. müssten in einer binären oder textuellen Form gespeichert werden, welche zur Laufzeit geladen (d.h. geparst und in eine interne Datenstruktur überführt) und interpretiert werden muss. Da aber gerade diese Komponenten den mitunter rechenintensivsten Teil der Verarbeitung darstellen, ergeben sich deutliche Performanznachteile gegenüber kompiliertem Code. Darüber hinaus wäre das Modul weit weniger kompakt, da es nun aus mehreren Dateien bestehen würde.

Es besteht noch eine weitere theoretische Möglichkeit, mit kompilierten Klassifizierern bei hoher Modularität umzugehen: Es wäre auch denkbar, in den Datendateien Quellcode zu speichern und diesen beim Starten des Klassifikationsmoduls zu Maschinencode zu kompilieren. Dies würde die Performanzprobleme der zuvor genannten Variante beheben. Aber auch diese Lösung ist nur eingeschränkt brauchbar. So wäre es erforderlich, dass auf der Zielplattform ein Compiler für diese Plattform zur Verfügung steht, was gerade bei mobilen Betriebssystemen wie Windows CE normalerweise nicht der Fall ist. Auch könnten durch Fehlschlagen der Kompilierung beim Start des Moduls Fehler auftreten, welche andernfalls schon beim Erzeugen des Moduls festgestellt worden wären. Ein weiterer Punkt ist die deutlich längere Startzeit, da gerade auf Plattformen mit geringerer Leistungsfähigkeit eine Kompilierung jede Menge Zeit beanspruchen kann. Zuletzt ändert sich auch nichts an der Tatsache, dass das Modul durch die Verwendung mehrerer Dateien nicht so kompakt ist, wie es wünschenswert wäre.

Aus den genannten Gründen wurde die Speicherung von Klassifizierern in externen Dateien für SBC verworfen. Die Idee, wie sowohl eine maximale Performanz als auch eine hohe Modularität erreicht werden kann, sieht eine Integration der Kompilierung des Klassifikationsmoduls in die Entwurfsphase vor. Diese besteht nun grob aus drei Teilen:

1. Entwurf der ersten Ebene (Klassifizierer)
2. Entwurf der zweiten Ebene (Bayessches Netz)
3. Erzeugung des Moduls (*Build-Vorgang*)

Hinzugekommen ist hier der dritte Teil. Bei dem Build-Vorgang wird eine einzelne, kompakte Datei für das Modul erzeugt, welche sowohl den Basiscode als auch die kontextspezifischen Daten in kompilierter Form enthält. Dies garantiert bereits eine optimale Geschwindigkeit. Um nun auch die Modularität zu gewährleisten, verfügt SBC über eine Infrastruktur zum Erzeugen und Integrieren von Klassifizierer-Quellcode in

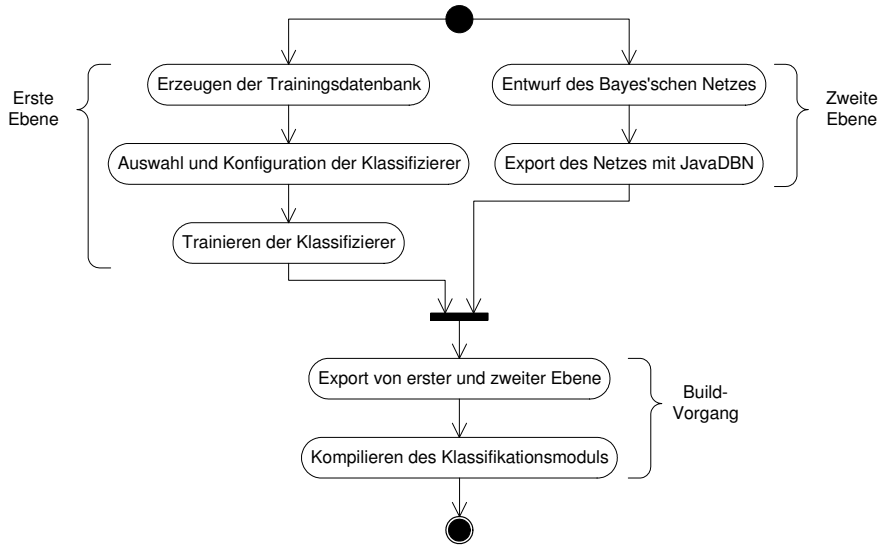



Abbildung 3.7: Aktivitäten in der Entwurfsphase einer SBC-Applikation

den Erstellungsprozess des Klassifikationsmoduls. Den entscheidenden Beitrag dazu leistet die SBC DEVELOPMENT PLATFORM: Sie erlaubt es, ausgewählte Klassifizierer und Netze zu exportieren. Dabei werden die fertig trainierten Klassifizierer über eine Schablone in Quellcode überführt und automatisch zusammen mit einer Konfigurationsdatei und dem dynamischen Bayesschen Netz in die richtigen Verzeichnisse des Projekts für das eingebettete Modul kopiert (vgl. Abschnitt 3.9.7). Wird in einem zweiten Schritt nun dieses Projekt erstellt, so enthält die Ausgabedatei ein auf den Kontext spezialisiertes Klassifikationsmodul, das in die Anwendung eingebunden werden kann. Die gesamte Entwurfsphase ist noch einmal in Abb. 3.7 zu sehen.

Dem Leser ist an dieser Stelle möglicherweise eine zusätzliche Schwierigkeit bei dem beschriebenen Build-Vorgang aufgefallen: Die Unterschiede zwischen den Betriebssystemen und evtl. auch Programmiersprachen von Entwicklungsumgebung und Klassifikationsmodul erfordern es nämlich, dass die Klassifizierer für jede Plattform gesondert implementiert werden; jedoch auch ohne die angesprochenen Unterschiede gäbe es gute Gründe für eine getrennte Implementierung (vgl. Abschnitt 3.1.4). Damit nun eine Evaluierung eines Klassifizierers in der Entwicklungsumgebung überhaupt eine Aussagekraft bzgl. des bei der Klassifikation in der Praxis eingesetzten Moduls besitzt, muss die Äquivalenz der Implementierungen verifiziert werden. In manchen Fällen ist keine hundertprozentige Übereinstimmung zu erreichen, da beispielsweise die Genauigkeit der Fließkommaberechnung je nach zugrunde liegender Hardware unterschiedlich sein kann. In diesem Fall kann die Evaluierung auf der Entwicklungsplattform immer noch als Approximation genutzt werden; verlässliche Zahlen erfordern aber eine Evaluierung

direkt über das Klassifikationsmodul. Diese Möglichkeit besteht bei SBC und ist in Abschnitt 3.10 beschrieben.

## 3.4 Von der Client/Server-Architektur zum Klassifikationsmodul

Der Prototyp des M3I-Servers wurde primär zur Demonstration von AGENDER entwickelt. Die Implementierung erfolgte in der Sprache *Java*, um die Plattformunabhängigkeit des Systems hervorzuheben. Die Applikation selbst war ohne grafische Oberfläche ausgestattet, jedoch konnte der aktuelle Zustand über einen ebenfalls im Rahmen des Projekts M3I entwickelten *GUI-Server* visualisiert werden. Diese GUI konnte per *Java RMI*[2] eine Verbindung zum Server herstellen, und alle beim Blackboard registrierten Dienste in hierarchischer Ordnung anzeigen. Zu diesen Diensten gehörten die Klassifizierer der ersten Ebene, die zweite Ebene, Merkmalsextraktion, Vorverarbeitung, Live-Klassifikation, Aufnahme, Benutzerverwaltung und andere. Als Klassifizierer wurden zum Teil eigene *Java*-Implementierungen eingesetzt, und zum Teil Komponenten aus dem *WEKA*-Paket. Die Aufnahme erfolgte vorerst direkt über das an den Server angeschlossene Mikrofon, wobei aber schon mehrere verschiedene Sprecher unterstützt wurden. Die aufgezeichnete Datei wurde als Objekt auf das Blackboard geschrieben, und bei eingeschalteter Live-Klassifikation von dieser an die weiteren Dienste weitergeleitet, welche für die eigentliche Implementierung von AGENDER zuständig waren. Für die einzelnen Klassifizierer und für die zweite Ebene wurden jeweils *Java Swing*-Panels bereitgestellt, welche eine genaue Darstellung des Zustandes und – im Falle der zweiten Ebene – des Klassifikationsergebnisses in der GUI ermöglichten (s. Abb. 3.8).

Der nächste Schritt bestand darin, die Aufnahme nicht nur von einem lokal angeschlossenen Mikrofon aus durchführen zu können, sondern von einem entfernten Rechner. Dazu wurde ein über Funknetzwerk (WLAN) mit dem Server verbundener *PocketPC* vorgesehen, um eines der typischen Anwendungsszenarien für AGENDER zu modellieren. Um möglichst auch bestehende und extern entwickelte Anwendungen mit der neuen Funktionalität ausstatten zu können, wurde eine Schnittstelle für den Client geschaffen, welche über das TCP/IP-Protokoll eine Verbindung zum Server aufbauen und Klassifizierungsanfragen absetzen kann. Eine Anwendung, welche diese M3I-Client-Bibliothek verwendet, muss lediglich zur Verfügung stehende Audiodaten an die Schnittstelle weiterleiten und kann bei Bedarf das aktuelle Benutzerprofil (Alter und Geschlecht) und den aktuellen Kontext abfragen. Diese Bibliothek wurde in *C++* geschrieben, der einzigen zu diesem Zeitpunkt auf dem *PocketPC* unterstützten Sprache. Auf der Seite des Servers erforderte die Erweiterung um mobile Clients lediglich eine Socket-Verwaltung und die Implementierung des Netzwerk-Protokolls.

Eines der Modellszenarien für den mobilen Client ist ein digitaler Einkaufsassistent, welcher in Abschnitt 4.1 zusammen mit der Implementierung der Client-Bibliothek

[2] *Remote Method Invocation.* Erlaubt es einem Server, Objekte für Clients zur Verfügung zu stellen.

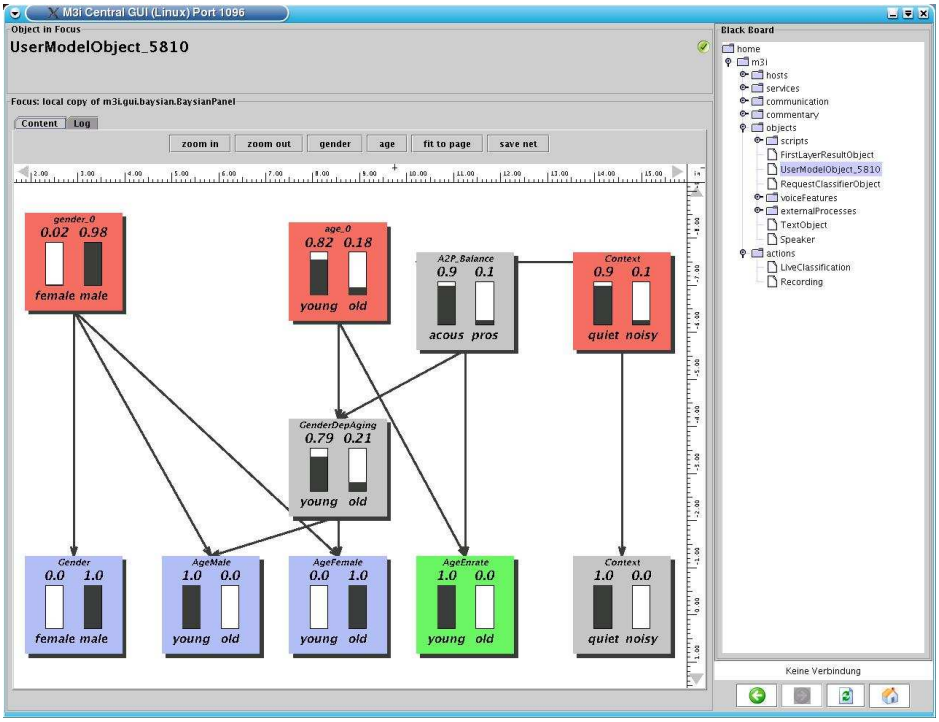



Abbildung 3.8: m3i-Server GUI mit Darstellung des Klassifikationsergebnisses auf der zweiten Ebene

genauer vorgestellt wird. Eine weitere Beispielanwendung ist der M3I *Personal Navigator* (vgl. Wasinger, Stahl und Krüger, 2003). Darüber hinaus kommt eine für das *.NET Compact Framework* geschriebene Demonstrationsanwendung hinzu, welche die Client/Server-Kommunikation über die M3I-Client-Bibliothek unterstützt und auf der *CeBit '05* vorgeführt wurde. Bei dieser Anwendung kann über das Mikrofon des mobilen Geräts eine Äußerung aufgezeichnet werden (man hält dazu die Aufnahmetaste des *PocketPCs* gedrückt), welche anschließend zur Klassifizierung an den Server gesendet wird. Liegt das Ergebnis vor, so werden die Daten zu Alter, Geschlecht und Kontext in Balkendiagrammen präsentiert (s. Abb. 3.9). Die ebenfalls abgebildete Schaltfläche *New Speaker* dient dazu, einen Sprecherwechsel anzukündigen: Die nächste Äußerung wird dann der ersten Zeitscheibe in einem neuen dynamischen Bayesschen Netz zugeordnet.

Dieses Szenario war für die ersten Versuche ausreichend, jedoch waren weitere Konzepte für die ersatzweise Klassifizierung auf dem Client notwendig. Letztere ist erforderlich, da die Verbindung zum Server bei einem mobilen Gerät in der Praxis naturgemäß über eine Funkverbindung erfolgt und diese leider eine gewisse Instabilität mit sich bringt: Sie kann durch Interferenzen von anderen Geräten oder physikalischen Hindernissen ge-


Abbildung 3.9: Testapplikation für AGENDER auf dem *PocketPC*

stört sein, kann zu Stoßzeiten womöglich überlastet sein oder der Benutzer kann sich außerhalb der Sendereichweite begeben. Es kann davon ausgegangen werden, dass eine solche Unterbrechung der Verbindung sowohl verhältnismäßig häufig auftritt, als auch dass sie temporär, d.h. vorübergehend ist. Zwei Dinge wären daher wünschenswert: Zum einen sollte die Client-Bibliothek großzügig auf den Abbruch der Verbindung reagieren, diese – soweit möglich – transparent wieder herstellen und ggf. die Klassifizierung erneut versuchen. Bei nicht wiederherstellbarer Verbindung sollte die Klassifizierung auf dem mobilen Gerät erfolgen, unter Berücksichtigung dessen beschränkter Ressourcen und Inkaufnahme einer verminderten Ergebnisqualität. Darüber hinaus soll die Anwendung über den eingetretenen Zustand informiert werden, um eigene, evtl. besser geeignete Maßnahmen ergreifen zu können.

Die automatische Wiederherstellung der Verbindung wurde in einer späteren Version der M3I-Client-Bibliothek implementiert. Für die Klassifizierung auf dem mobilen Gerät wurde der Prototyp eines „internen Servers“ geschaffen, welcher eine relativ performante, aber äußerst unflexible und nur bedingt modulare Struktur besaß. Im Rahmen dieser Entwicklung wurde bereits die Merkmalsextraktion, welche ansonsten der Server ausgeführt hätte, auf den *PocketPC* portiert (vgl. Feld, 2005). Die Klassifizierer waren auf einen *C4.5-Entscheidungsbaum* (vgl. Quinlan, 1993) beschränkt und wurden

bei der ersten Verbindung mit dem Server von diesem angefordert und in einem kompakten Textformat übertragen. Dieses Format erforderte zwar Parsing und musste bei der Klassifizierung interpretiert werden, anstatt dass es in Maschinencode ausgeführt werden konnte (vgl. Abschnitt 3.3.4); dennoch war die Laufzeit sehr gering, sofern keine parallele Verarbeitung mehrerer Benutzer stattfand. Das größere Problem stellten jedoch die fortbestehende Abhängigkeit vom Server und die statische Architektur dar. Außerdem wurde bei der ersten Version des internen Servers komplett auf ein Bayessches Netz verzichtet und das Ergebnis nur durch Mittelwertbildung berechnet.

Etwa an dieser Stelle, d.h. mit der Vorstellung des beschriebenen Prototyps eines internen Servers, wurde parallel die Arbeit an einem weiteren Szenario für AGENDER begonnen, nämlich einer schnellen, serverbasierten Lösung zur Klassifizierung von Anrufern. Es wurde erkannt, dass trotz der verschiedenen Plattformen sehr viele Gemeinsamkeiten zwischen den Anforderungen der beiden Szenarien bestanden. Daher wurde der Ansatz des internen Servers nach und nach verworfen, und durch die neue SBC-Architektur ersetzt. Trotz dieser Umorientierung konnten die mit dem internen Server gesammelten Erfahrungen und auch einige Teile des Codes für SBC genutzt werden.

## 3.5 Plattformspezifischer Entwurf

Gemäß den in Abschnitt 3.1.4 formulierten Anforderungen ist es notwendig, dass das Klassifikationsmodul unter verschiedenen Plattformen ausgeführt werden kann. Dazu muss schon vor der Wahl dieser Plattformen eine flexible Infrastruktur bereitgestellt werden, welche alle Möglichkeiten abdeckt.

Die Wahl der Programmiersprache für das Klassifikationsmodul fiel auf *C++*, welches eine plattformtransparente Entwicklung begünstigt. Dies liegt vor allem an drei Umständen: Die Sprache ist für fast jede erdenkliche Plattform verfügbar, d.h. es gibt einen Compiler, welcher Code für diese Plattform erzeugen kann. Anbieter von neuen Betriebssystemen stellen in der Regel zuallererst sicher, dass *C++* für die Entwicklung von Anwendungen eingesetzt werden kann. Daneben besteht ein kompiliertes *C++*-Programm aus Maschinencode, der keine weitere Unterstützung in Form einer Laufzeitumgebung auf der Zielplattform benötigt. Es kann zwar sein, dass für den Zugriff auf Betriebssystem-Funktionen so genannte Header-Dateien mit Deklarationen dieser Funktionen benötigt werden; diese sind dann aber Teil der Unterstützung von *C++* auf der Plattform und werden in der Regel in Form eines *Software Development Kit*[3] (SDK) vom Hersteller angeboten. Der dritte Grund besteht darin, dass die Sprache selbst die geeigneten Mittel anbietet, um den gleichen Code für mehrere Plattformen verwalten zu können. Dabei handelt es sich hauptsächlich um die bedingte Kompilierung, die nachfolgend erläutert wird. Gegenüber *C* erlaubt *C++* primär eine besser strukturierte Entwicklung; aus Sicht der Anwendung wäre aber *C* genauso gut geeignet gewesen.

Die wichtigsten Konzepte zur plattformspezifischen Programmierung stellen *Erstellungs-Konfigurationen* und *bedingte Kompilierung* dar. Erstellungs-Konfigurationen werden als Teil der Entwicklungsumgebung oder Build-Skripte verwendet, um je nach Zielplattform bestimmte Einstellungen vorzunehmen. Beispielsweise könnte es die Konfigurationen *Win32* (für 32-bit *Windows*), *WinCE* (für *Windows CE*) und *Linux* geben, welche jeweils den zu verwendenden Compiler festlegen, plattformspezifische Einstellungen beinhalten und Präprozessor-Konstanten definieren. Bei den Präprozessor-Konstanten handelt es sich um einfache Wertzuweisungen wie `MOBILE_DEVICE=1`. Diese werden zusammen mit der bedingten Kompilierung eingesetzt, um bestimmte Codeblöcke je nach Zielplattform zu kompilieren. Listing 1 zeigt ein Beispiel für einen solchen bedingten Codeblock. Der erste Teil wird dann übersetzt, wenn die Zielplattform *Windows* ist, der zweite Teil bei *Windows CE*. Man spricht bei den mit einer Raute (#) beginnenden Zeilen von *Präprozessor-Direktiven*, da sie von einem Präprozessor verarbeitet werden, noch bevor der Compiler seine Arbeit verrichtet. Durch diese Konstruktion kann leicht mit plattformspezifischen

[3] Ein SDK wird oft zusammen mit einem Produkt vertrieben, um Entwicklern die zur Programmierung bzw. Erweiterung des Produkts nötigen Werkzeuge zur Verfügung zu stellen.

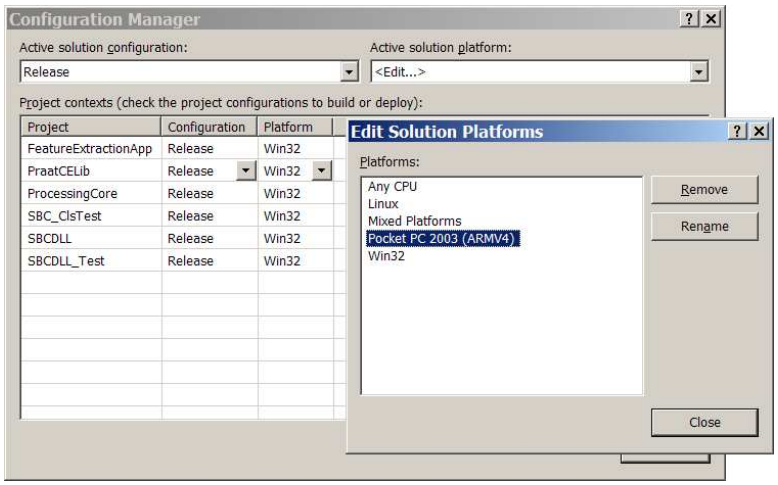



Abbildung 3.10: Verwaltung plattformspezifischer Konfigurationen in *Visual Studio 2005*

Unterschieden umgegangen werden, ohne mehrere getrennte Versionen des Codes verwalten zu müssen. Neue Plattformen lassen sich an den betroffenen Stellen auch leicht einfügen.

In Listing 1 wird ein typisches Problem bei der Unterstützung modelliert: Eine Funktion (im vorliegenden Beispiel `time`) steht auf einer bestimmten Plattform nicht zur Verfügung. In diesem Fall muss ein entsprechender Ersatz gefunden oder nachgebildet werden. Die Präprozessor-Direktive `#define` erlaubt es, alle Aufrufe an die nicht vorhandene Funktion im restlichen Programm an eine andere Funktion umzuleiten, ohne dass dabei Performanzeinbußen entstehen.

Für die Verwaltung des plattformübergreifenden Projekts wurde *Visual Studio 2005* gewählt. Dieses bietet eine gute Unterstützung für mehrere Erstellungs-Konfigurationen und bedingte Kompilierung. Besonders vorteilhaft ist die bereits integrierte und in Ab-

**Listing 1** Präprozessor-Direktiven zur plattformspezifischen Kompilierung

```
#ifdef WIN32
    #define time clock;
#endif
#ifdef POCKET_PC
    #define time GetTickCount;
#endif
```

| **Plattform** | Win32-Desktop | Win32-Server | Linux-Server | PocketPC (Windows Mobile 5.0) | Smartphone (Windows Mobile 5.0) |
|---|---|---|---|---|---|
| **Compiler** | cl | cl | gcc | cl for ARM | cl for ARM |
| **SDK** | Windows Platform SDK | Windows Platform SDK | Linux Kernel Sources | Windows Mobile 5.0 SDK for Pocket PC | Windows Mobile 5.0 SDK for Smartphone |
| **OS-Besonderheiten** | - | - | - | Reduzierter Funktionsumfang | Noch stärker reduzierter Funktionsumfang |
| **HW-Besonderheiten** | - | Evtl. Multiprozessor-Architektur | Evtl. Multiprozessor-Architektur | Geringere Leistung, wenig Speicher | Sehr geringe Leistung, wenig Speicher |
| **Bereits integriert in SBC?** | ja | ja | nein | ja | ja |
| **Sprecher** | 1 | Mehrere 100 | Mehrere 100 | 1 | 1 |

Tabelle 3.1: Übersicht verschiedener Plattformen für Sprecherklassifikation

schnitt 3.1.4 geforderte Unterstützung für *Windows* Desktop-Betriebssysteme und *Windows Mobile 5.0*. Weitere Zielplattformen für die Cross-Kompilierung lassen sich in den Erstellungsvorgang integrieren. So könnte mittels des im *Cygwin*-Paket (vgl. Racine, 2000) enthaltenen *Linux*-Compilers *gcc* auch über die gleiche Umgebung eine *Linux*-Variante des Klassifikationsmoduls generiert werden. Abb. 3.10 zeigt die Verwaltung plattformspezifischer Konfigurationen für das Klassifikationsmodul in *Visual Studio*.

Tabelle 3.1 enthält eine kurze Zusammenstellung der wichtigsten Plattformen für SBC und eine Auswahl an Attributen. Die Angaben für hardwaremäßige Besonderheiten der Plattform sind typische Werte; je nach individueller Ausstattung des Rechners können diese auch abweichen.

Einer der Gründe für die schwierige Portierbarkeit des M3I-Servers sind die zahlenreichen externen Anwendungen, die durch Befehlszeilenaufrufe in den Verarbeitungsprozess eingebunden wurden. Hierzu zählen folgende Programme:

**socks:** Vorverarbeitung einschließlich Konvertierung des Audioformats

**praat:** Extraktion der meisten Sprachmerkmale

**enrate:** Berechnung der *En-Rate*

**srsad:** Berechnung der *Syllable Rate Speech Activity*

Diese Programme werden zum Teil auch auf einem entfernten Rechner über eine *SSH*-Verbindung gestartet. Leider sind weder die Werkzeuge selbst, noch die *SSH*-Kommunikation für die Portierung auf verschiedene Plattformen geeignet, die im Voraus nicht einmal immer bekannt sind. Dies bedeutet, dass alle benötigten Verfahren als Quellcode in das Klassifikationsmodul integriert werden müssen, um überhaupt kompiliert werden und auch von dessen Multiplattform-Infrastruktur profitieren zu können. Ob dies durch Portierung und Integration vorhandener Tools oder durch komplette Neuimplementierung erfolgt, hängt von dem jeweiligen Kosten-Nutzen-Verhältnis ab. Im Rahmen von SBC wurden beide Wege beschritten.

Die *SSH*-Verbindungen entfallen bei dem Klassifikationsmodul vollständig. Um in Szenarien mit hohem Datenaufkommen dennoch skalierbar zu bleiben, kann entweder die Parallelverarbeitung genutzt werden oder eine Cluster-Lastaufteilung auf einer dem Modul vorgeschalteten Ebene vorgenommen werden.

Das Klassifikationsmodul wird beim Erstellen in eine Bibliotheksdatei (.lib) kompiliert. Diese Datei enthält den Maschinencode (auch als *Objektcode* bezeichnet) und kann in den Erstellungsvorgang der Host-Anwendung eingebunden werden. Dieser Vorgang wird als *Linken* bezeichnet, da hierbei eine „Verknüpfung“ von den Funktionsaufrufen zu den Stellen im Code, an denen die aufgerufene Funktion definiert ist, hergestellt wird. Die Bibliotheksdateien verschmelzen dabei mit dem Anwendungscode zu einer einzigen ausführbaren Datei, wobei die gleiche Kompaktheit und Geschwindigkeit erreicht wird, als wäre die Bibliothek schon im Quellcode Teil der Anwendung gewesen und mit dieser zusammen kompiliert worden.

Bei dem beschriebenen Vorgang handelt es sich um *statisches* Linken. Dieses kann unabhängig von der Zielplattform immer durchgeführt werden, da als Ausgabe eine konventionelle ausführbare Datei erzeugt wird. Einige Betriebssysteme unterstützen jedoch noch weitere Dateitypen, welche zum Zugriff auf die Klassifizierungsdienste geeignet sein können, allen voran *gemeinsam genutzte Bibliotheken*. Dabei wird die Bibliothek nicht mit der Anwendung zu einer Datei verschmolzen, sondern bleibt als separate Datei bestehen. Dies hat mehrere Vorteile: Die Bibliothek muss so nur einmal auf dem Zielsystem installiert werden, und kann dann von mehreren Anwendungen genutzt werden. Es ist also eine Speicherersparnis gegeben. Auch kann die Anwendung das Laden der Bibliothek verzögern, bis diese tatsächlich benötigt wird. Das bietet sich an, wenn die AGENDER-Funktionalität nicht bei jeder Programmausführung eingesetzt wird. Ein weiterer Vorteil besteht darin, dass die Anwendung das Modul zur Laufzeit auswählen kann. Dies erlaubt es beispielsweise, mehrere Klassifikationsmodule zu erzeugen und zur Laufzeit das zu verwendende Modul auszuwählen.

Gemeinsam genutzte Bibliotheken besitzen aber auch Nachteile. Wie schon erwähnt

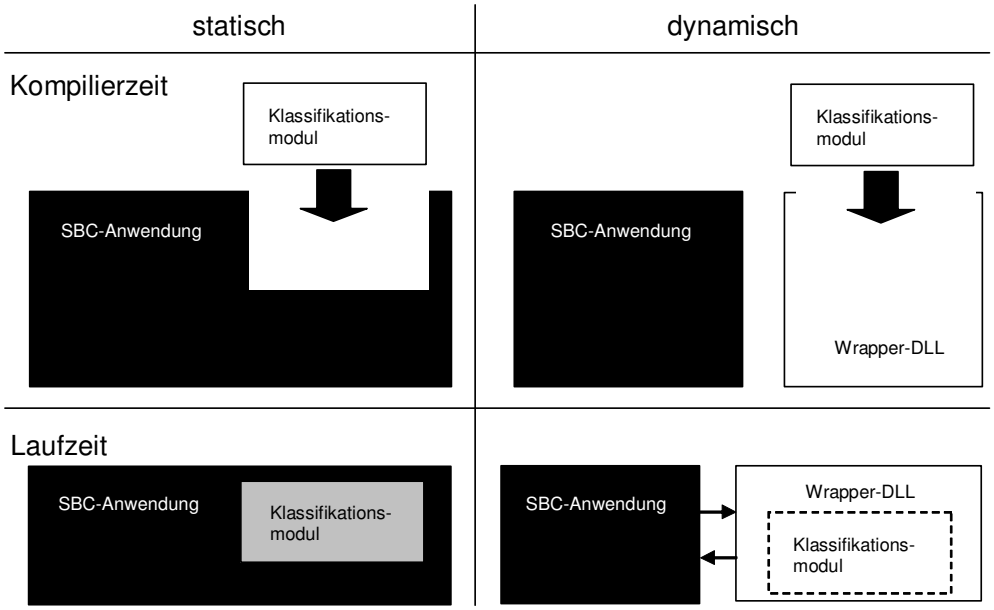



Abbildung 3.11: Vergleich zwischen statischem und dynamischem Einbinden des Klassifikationsmoduls in eine Anwendung

wurde, sind sie ein betriebssystem-spezifisches Merkmal und müssen daher für jede Plattform gesondert entwickelt werden. Unter *Windows*-Plattformen wird diese Funktionalität von *DLL*-Dateien (*Dynamic Link Library*) bereitgestellt, und unter *Linux* dienen *Shared Object*-Dateien dem gleichen Zweck. Jedoch muss nicht nur die Aufrufschnittstelle an die jeweilige Plattform angepasst werden, sondern es gibt auch subtile Unterschiede in der Implementierung: So werden unter *Windows* globale Variablen von allen Konsumenten einer Bibliothek gemeinsam genutzt, was teilweise sehr nützlich sein kann, aber auch eine potenzielle Fehlerquelle darstellt; unter *Linux* werden Variablen getrennt für jeden Prozess verwaltet. Ein weiterer Nachteil besteht in dem aufwändigeren Ladevorgang: Durch einen besonderen Aufruf (z.B. `LoadLibrary` unter *Windows*) wird das Betriebssystem aufgefordert, die Bibliothek zu laden und das Linken der Funktionsaufrufe im Arbeitsspeicher durchzuführen. Da es nun zur Laufzeit ausgeführt wird, wird es als *dynamisches* Linken bezeichnet. Nicht zuletzt ist die gesamte Anwendung auch weniger kompakt: Wird es versäumt, eine Bibliothek mitzuliefern, oder verlässt sich die Anwendung auf das Vorhandensein einer Bibliothek aufgrund einer weiteren Installation, so kommt es leicht zu Fehlern bei der Ausführung, die vom Benutzer schwer zurückzuverfolgen sind.

Für den Fall, dass eine Entscheidung zugunsten einer gemeinsam genutzten Bibliothek

gefällt wird, sieht SBC den Entwurf einer eigenen *Wrapper-Bibliothek* vor: Diese bietet nach außen hin die benötigte Schnittstelle an und leitet intern alle Aufrufe an das statisch eingebundene Klassifikationsmodul weiter. Ein ähnlicher Ansatz wurde bei der SBC-Client-Bibliothek verfolgt (vgl. Abschnitt 4.1.2), nur dass hierbei noch eine Reihe weiterer Funktionen in die Bibliothek integriert wurden, wie z.B. die Kommunikation mit einem SBC-Server über Netzwerk. Eine grafische Gegenüberstellung von statischer Einbindung in die Anwendung und gemeinsam genutzter Bibliothek wurde in Abb. 3.11 vorgenommen.

| **Szenario** | **Anforderung an** | |
|---|---|---|
| | Performanz | Ressourcensparsamkeit |
| Benutzeradaption auf einem *PocketPC* | *Gering.* Im Vordergrund steht die Applikation, die Adaption ist eher Nebensache. | *Hoch.* Das mobile Gerät besitzt wenige Ressourcen. Eine Belastung bewirkt langsamere Ausführung der Anwendung. |
| Adaption an einem öffentlichen Terminal | *Durchschnittlich.* Die Adaption ist zwar nicht zwingend erforderlich, vermittelt dem Kunden aber zweckmäßige Zusatzleistungen. | *Gering.* Ressourcen sind ausreichend vorhanden, durch exklusive Verarbeitung entsteht kein Engpass. |
| ACD in einem Callcenter | *Hoch.* Die vom Anrufer spürbare Verzögerung bei der Weiterleitung sollte so gering wie möglich sein. | *Explizit gering.* Es sollen bewusst alle verfügbaren Ressourcen genutzt werden, um die Performanz zu steigern. |

Tabelle 3.2: Vergleich verschiedener Anwendungsszenarien im Hinblick auf Performanz- und Ressourcenanforderungen

## 3.6 Performanz- und Ressourcenaspekte

Ausführungsgeschwindigkeit und ressourcenschonende Arbeitsweise sind beides Gesichtspunkte, die für einen Prototypen wie den bestehenden M3I-Server keine besondere Bedeutung haben. Dies ändert sich jedoch schlagartig, wenn die Systeme von der Testumgebung in ein tatsächliches Produktionsszenario übergehen. Dabei sind nicht alle Szenarien gleich: Tabelle 3.2 vergleicht drei mögliche Anwendungsszenarien im Hinblick auf ihre Anforderungen.

Eine sehr wichtige Rolle für die Geschwindigkeit spielt die Wahl der Programmiersprache. Hier wurde bei SBC eine entscheidende Veränderung gegenüber dem M3I-Server vorgenommen: Das Klassifikationsmodul wurde komplett in *C++* neu geschrieben, da diese Sprache bekannt ist für ihre Geschwindigkeitsvorteile. Dies rührt zum einen daher, dass sie gegenüber *Java* nicht in einer virtuellen Maschine, sondern direkt vom Prozessor ausgeführt wird. Zum anderen gibt es für *C++* eine Reihe bewährter Compiler, welche hoch optimierten Code erzeugen. Die Folge dieser Entscheidung ist neben dem – wie zuvor erwähnt nur theoretischen – Verlust der Plattformunabhängigkeit auch ein impliziter Verlust an Robustheit, da *C++* durch die systemnahe Implementierung mehr potenzielle Gefahrenquellen einführt und einiges mehr an Aufmerksamkeit bei der Programmierung erfordert. Dieser Kompromiss erscheint jedoch durchaus gerechtfertigt.

Der Compiler alleine ist ein mächtiges Werkzeug für die Optimierung von Code im

Hinblick auf Geschwindigkeit oder Größe, es gibt aber noch weitere Faktoren. So sind bei der Wahl sprachspezifischer Konstrukte und Plattformaufrufe jeweils auch die sich ergebenden Geschwindigkeitsaspekte zu beachten, da es hier manchmal feine Unterschiede gibt, die sich bei umfangreicher Anwendung potenzieren können. Beispiele hierfür sind die Fragen, ob Zeichenfolgen in einer Klasse gekapselt oder als Buchstaben-Datenfeld (*Character Array*) verwaltet werden sollten, und ob Fehler mittels eines `Throw ... Catch`-Konstrukts oder des Rückgabewerts `False` gemeldet werden sollten. Es handelt sich meist um Entscheidungen zwischen besserer Strukturiertheit und geringerer Fehleranfälligkeit gegenüber höherer Geschwindigkeit.

Aber nicht alle Geschwindigkeitsoptimierungen auf Codeebene müssen Resultat einer Entscheidung sein: Einige erfordern lediglich die Befolgung grundlegender Programmierprinzipien und eine gewisse Kenntnis der Sprache bzw. des Compilers. So wurde bei SBC auf das häufige Kopieren von Daten (z.B. die zu klassifizierenden digitalen Audiodaten) verzichtet, da dies immer ein zeitintensiver Vorgang ist. Weiterhin wurden Schleifen und Bedingungen so angeordnet, dass unnötige Verarbeitungsschritte vermieden werden. Es ließen sich viele weitere Beispiele für Optimierungen auf der Implementierungsebene finden. Wie nicht anders zu erwarten ist, spielen diese Bestrebungen dagegen bei zielgerichteter Programmierung, wie es bei der Entwicklung des ersten Prototyps des M3I-Servers der Fall war, wiederum keine große Rolle.

Bei der Beschreibung des plattformspezifischen Entwurfs wurde bereits erwähnt, dass auf Aufrufe externer Prozesse wie *Praat* beim Klassifikationsmodul komplett verzichtet wurde. Stattdessen wurden diese Funktionseinheiten direkt in den Quellcode übernommen oder per statischem Linken eingebunden, was jeweils eine deutliche Geschwindigkeitsverbesserung gegenüber dem M3I-Server darstellt. Beim Aufrufen eines ausführbaren Programms, dem die benötigten Informationen über die Befehlszeile mitgeliefert werden, entsteht nämlich ein beträchtlicher Verwaltungsaufwand für das Betriebssystem, der sich fast immer in einer spürbaren Verzögerung manifestiert. Werden nun bestimmte Befehle sehr oft aufgerufen (wie bei einer Stapelverarbeitung), so vervielfacht sich dieser Effekt.

Weitere Geschwindigkeitsvorteile sind durch das Pipeline-Design gegeben (vgl. Abschnitt 3.3.3). Dies bedeutet, dass der Aufwand für die Kommunikation sehr gering ist. Zudem kann eine Pipeline pro Sprecher erstellt werden, welche dann von verschiedenen Threads ausgeführt werden. Dies erlaubt es servergestützten Anwendungen, mehrere Äußerungen parallel zu verarbeiten und dadurch zusätzliche Performanzgewinne zu erhalten. Für ein mobiles Gerät ist keine Parallelverarbeitung erforderlich, da zu jeder Zeit nur ein einzelner Nutzer das Gerät bedienen und Spracheingaben tätigen kann. Ohnehin sind die CPUs mobiler Geräte weniger gut für parallele Verarbeitung geeignet, als es bei Desktops oder Servern der Fall ist.

Für die Umgangsweise mit Ressourcen gelten zum Teil ähnliche Verfahrensweisen wie

bei der Performanz. Denn auch hier muss bei jeder Implementierung im Detail geprüft werden, ob bestimmte Operationen unter Umständen beispielsweise mehr Speicher verwenden als benötigt und daher optimiert werden könnten, oder ob unnötig viele Dateien geöffnet werden. Viele dieser Optimierungen betreffen indirekt auch die Geschwindigkeit. Man kann auch die Leistung des Prozessors selbst als Ressource sehen, die zwar nicht verwaltet werden muss, aber nur in begrenztem Maße zur Verfügung steht.

Um das Klassifikationsmodul robust zu halten, ist es notwendig, dass jede Ressource nach Abschluss der Verarbeitung konsequent wieder freigegeben wird. Als Ressourcen, welche eine Verwaltung erfordern, gelten primär Speicherreservierungen, geöffnete Dateien und Betriebssystem-Handles[4]. Nicht freigegebene Ressourcen führen zu Speicherlecks, die teilweise schwer zu finden sind. Leider sind solche kleinen Fehler auch bei sorgfältiger Arbeit in der Sprache *C++* kaum völlig zu vermeiden. Auch bei der Arbeit mit SBC mussten bei einem Prototypen Speicherlecks gefunden und geschlossen werden. Es gibt verschiedene Werkzeuge, welche diese Arbeit unterstützen, wie z.B. *Purify* (vgl. Hastings und Joyce, 1991).

Die Feststellung, dass eine möglichst ressourcensparende Implementierung optimal ist, trifft so nicht zu. Ab einer bestimmten Grenze bei der Reduzierung der verwendeten Ressourcen entwickelt sich nämlich die Effizienz der Komponente stark rückläufig, so dass kein vernünftiger Kompromiss mehr vorliegt. Beispielsweise kann beim Parsen (oder Einlesen) einer Datei bestimmt werden, in welchen Blöcken der Speichertransfer erfolgt. Werden die Blöcke größer gewählt, so ist mehr temporärer Speicher erforderlich. Wird die Größe der Blöcke gesenkt, im Extremfall auf ein Byte, so werden mehr Schritte und damit mehr Prozessorzyklen benötigt.

Das beschriebene Beispiel verdeutlicht auch, dass es sogar wünschenswert sein kann, Ressourcen bewusst zu nutzen, um eine zusätzliche Erhöhung der Verarbeitungsgeschwindigkeit zu erreichen. Das wohl bekannteste Beispiel ist *Caching*. Dabei wird ein Speicherbereich reserviert und dazu verwendet, einmal akquirierte Informationen und Objekte zwischenzuspeichern, um sie später nicht neu berechnen oder laden zu müssen. Je größer der Cache gewählt wird, desto höher ist der Geschwindigkeitsgewinn, bis zu dem Punkt, an dem alle Informationen im Cache enthalten sind. Die Effizienz des Caches hängt von einer Reihe von Faktoren ab, z.B. der Anzahl der Zugriffe auf Objekte, der Größe der Objekte, der Art und Größe des Speichers von Seiten der Hardware usw. Intelligente Cache-Verfahren verwenden komplexe Heuristiken zur Auswahl der Objekte, die in den Cache aufgenommen werden. Die in dieser Arbeit vorgestellte SBC-Architektur verwendet Caching, um die Klassifizierer zwischenzuspeichern. Ein Erstellen dieser Objekte erfordert einen gewissen Initialisierungsaufwand, der aber dank der verwendeten Technik nur einmal pro Anwendungsstart ausgeführt werden muss. In diesem Fall bringt der Cache in quasi jedem Bereich Vorteile, da die Klassifizierer ohne-

[4]Numerische Bezeichner für Betriebssystem-Ressourcen, z.B. Dateien.

hin geladen werden müssen, wenn sie benötigt werden, und es sich dabei um eine feste, geringe Anzahl von Objekten handelt.

Weitere Fälle sind denkbar, in denen eine unterschiedliche Vorgehensweise je nach Zielplattform und -anwendung angebracht ist. Im Optimalfall sollte das Klassifikationsmodul sich echt ressourcenadaptiv verhalten und genau die Ressourcen nutzen, die auf der Plattform, unter der es ausgeführt wird, frei zur Verfügung stehen. Dies ist in der aktuellen Fassung des Moduls noch nicht implementiert, kann jedoch für spätere Versionen angedacht werden. Es ist allerdings auch zu erwähnen, dass derzeit noch keine größeren Mengen von Ressourcen eingesetzt werden, für die sich ein ressourcenadaptives Caching in besonderem Maße eignen würde. Eine Einflussnahme des Anwendungsarchitekten auf die zur Klassifizierung verwendeten Ressourcen ist jedoch jetzt schon sehr wohl möglich: Die Klassifizierungsalgorithmen sind wie in Abschnitt 2.2.2f. beschrieben sehr unterschiedlich, was ihre Komplexität und Anforderungen anbelangt. Diese Tatsache sollte dazu genutzt werden, um die jeweils am ehesten geeigneten Verfahren auszuwählen. So ist beispielsweise ein Künstliches Neuronales Netz für ressourcenschwache Plattformen generell schlechter geeignet als ein Naives Bayes Algorithmus. Einige Ressourcenaspekte des Klassifizierers lassen sich möglicherweise auch direkt durch kleinere Code-Optimierungen am Klassifizierer anpassen.

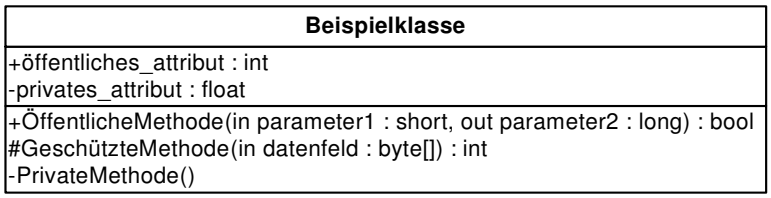



Abbildung 3.12: Beispiel für ein UML-Klassendiagramm

## 3.7 Aufbau des Klassifikationsmoduls

In den bisherigen Kapiteln wurden hauptsächlich allgemeine Konzepte und Ideen beschrieben, die bei SBC zum Einsatz kommen. Nun soll der Aufbau des Klassifikationsmoduls systematisch behandelt werden, in der Form wie es im Rahmen der vorliegenden Arbeit tatsächlich implementiert wurde.

Das gesamte Projekt ist intern in zwei Teile gegliedert: Einer für den SBC-spezifischen Code und ein weiterer für die Merkmalsextraktion. Diese Auslagerung der Merkmalsextraktion liegt darin begründet, dass es sich um die Portierung einer kompletten Anwendung zu einer Bibliothek handelt, die als Ganzes beibehalten werden sollte. Durch statisches Linken entsteht kein Geschwindigkeitsnachteil, es handelt sich lediglich um eine Trennung zur Entwicklungszeit. Weitere Komponenten könnten hinzukommen, wenn sie aus analogen Gründen eine eigene Bibliothek rechtfertigen, wie z.B. das am *International Computer Science Institute* in Berkeley entwickelte Werkzeug *mrate* zur Berechnung der Artikulationsgeschwindigkeit (vgl. Morgan und Fosler, 1998).

Der Grobaufbau des Moduls entspricht dem in Abb. 2.6 auf Seite 30 dargestellten Sachverhalt. Angelpunkt der Verarbeitung ist die Pipeline, welche den kompletten Klassifizierungsprozess durchführt und außerdem das aktuelle Benutzermodell speichert. Innerhalb der Pipeline erfolgt zuerst die Merkmalsextraktion. Die Merkmale werden dann an die Klassifizierer übergeben, welche nacheinander ausgeführt werden. Die jeweiligen Ergebnisklassen werden in das für die Pipeline exklusive dynamische Bayessche Netz eingetragen. Sobald alle Klassifizierer diesen Vorgang abgeschlossen haben, wird die Inferenz im Netz durchgeführt und die Eigenschaften des Sprechers ausgelesen. Dieses Benutzerprofil, einschließlich der jeweiligen Wahrscheinlichkeiten, kann dann zu jedem beliebigen Zeitpunkt von der Anwendung ausgelesen werden.

Im Folgenden sollen die einzelnen Bestandteile näher betrachtet werden. Zur Veranschaulichung der Klassen werden *Klassendiagramme* in UML[5]-Notation verwendet. Abb. 3.12 zeigt ein Beispieldiagramm. Klassen können *Felder* und *Methoden* besitzen (die jeweiligen UML-Fachbegriffe dafür lauten *Attribut* und *Operation*). Felder entspre-

[5] *Unified Modeling Language*: Standardisierte Sprache für die Modellierung von Software und anderen Systemen.

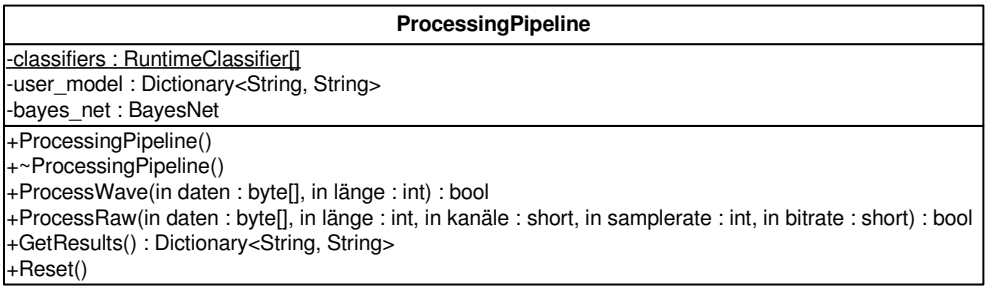



Abbildung 3.13: UML-Diagramm der Klasse `ProcessingPipeline`

chen lokalen Variablen eines Objekts der Klasse, und Methoden stellen aufrufbare Funktionen dar. Die dargestellte Klasse verfügt über zwei Felder und drei Methoden. Der Sichtbarkeitsbereich wird durch ein Präfix dargestellt, wobei ein + für öffentliche Member steht, # für geschützte Member (d.h. nur für die eigene Klasse und alle abgeleiteten Klassen sichtbar) und – für private Member. Methodenparameter stehen in runden Klammern. Für jeden Parameter ist der Datentyp und die Richtung angegeben, wobei *in* für eingehende Daten und *out* für ausgehende Daten steht. Datenfelder (engl. „arrays“) und Vorlagenklassen (engl. „templates“) verwenden die jeweilige *C++*-Notation. Der Einfachheit halber wurde in den UML-Diagrammen auf die Kennzeichnung von *C++*-Zeigern (engl. „pointers“) verzichtet.

### 3.7.1 Verarbeitungs-Pipeline

Die Verarbeitungs-Pipeline ist die einzige Komponente des Klassifikationsmoduls, welche „von außen“ her angesprochen wird. Sie stellt einerseits die Schnittstelle zwischen der Anwendung und dem Modul her, auf der anderen Seite aber ist sie auch für die interne Kommunikation und Ausführung zuständig.

Die Verarbeitungs-Pipeline wird durch die Klasse `ProcessingPipeline` repräsentiert, welche in Abb. 3.13 als UML-Diagramm dargestellt ist. Jede Pipeline kann nur für einen einzelnen Sprecher verwendet werden, allerdings können beliebig viele Äußerungen des gleichen Sprechers nacheinander verarbeitet werden. Bevor eine Äußerung klassifiziert werden kann, muss also die Anwendung zuerst ein neues Objekt vom Typ `ProcessingPipeline` erstellen.

Beim ersten Aufruf des Konstruktors[6] werden über die Konfigurationsdatei (vgl. Abschnitt 3.7.7) alle im Modul vorhandenen Klassifizierer geladen und in einem statischem Feld (`classifiers`) gespeichert. Die Tatsache, dass es sich um ein *statisches* Feld handelt impliziert, dass es von allen Objekten der Klasse gemeinsam genutzt wird. Konkret

[6]Funktion, welche automatisch aufgerufen wird, wenn eine Instanz der Klasse erzeugt wird.

bedeutet dies, dass die Klassifizierer nur einmal pro Anwendung geladen werden müssen und dann in einem Cache erhalten bleiben.

Die Pipeline verfügt zurzeit über zwei private Instanzvariablen. Die erste (`user_model`) speichert das aktuelle Benutzermodell. Dabei handelt es sich um ein Zeichenfolgenwörterbuch, d.h. eine Abbildung von Zeichenfolgen auf Zeichenfolgen. Den Definitionsbereich dieser Abbildung stellen so genannte *Profilpfade* dar, welche nach einer einheitlichen Konvention benannt werden (vgl. Abschnitt 3.8.2). Bei der Initialisierung ist diese Datenstruktur noch leer, d.h. es sind keinerlei Informationen über den Sprecher bekannt.

Die zweite lokale Variable (`bayes_net`) ist das dynamische Bayessche Netz, welches für den aktuellen Benutzer verwendet wird. Im Prinzip handelt es sich dabei auch um eine Repräsentation des Benutzermodells, jedoch aus einer etwas anderen Perspektive. Um die Daten aus dem Bayesschen Netz in das Zeichenfolgenwörterbruch zu übertragen, werden noch einige Konvertierungsfunktionen auf die rein numerische Darstellung im Netz angewendet. Das Objekt wird zusammen mit der Verarbeitungs-Pipeline in deren Konstruktor erstellt.

Nach der Erstellung der Pipeline kann die Klassifizierung durch eine von zwei alternativen öffentlichen Methoden angestoßen werden: `ProcessWave` und `ProcessRaw`. Der Unterschied besteht darin, dass die erste Funktion eine Audiodatei im WAVE-Dateiformat erhält, und die zweite „rohe“ PCM-Daten. Das WAVE-Format verfügt über einen Header mit Metainformationen, welche Abtastfrequenz, Abtasttiefe und Anzahl der Kanäle enthalten. Diese werden benötigt, um die eigentlichen Audiodaten interpretieren zu können. Da diese Informationen bei den rohen Daten fehlen, müssen sie vom Aufrufer als Parameter übergeben werden. Werden falsche Werte angegeben, so kann dies gravierende Auswirkungen auf die Klassifikation haben, da im Grunde genommen keine sinnvollen Daten mehr verarbeitet werden.

Die Verarbeitung innerhalb der genannten Funktionen kann als UML-Aktivitätsdiagramm darstellt werden (s. Abb. 3.14). Sie läuft wie folgt ab: Zuerst wird ein neuer Merkmalsextraktor erstellt. Bei dem Objekt an sich handelt es sich – vereinfacht gesagt – um eine Reihe von Variablen, die jeweils ein erkanntes Sprachmerkmal enthalten. Der Merkmalsextraktor wird dann mit den bekannten Audiodaten ausgeführt. Falls die Extraktion fehlschlägt – und dieser Fall tritt in der Praxis nicht nur bei fehlerhaften Daten auf, sondern z.B. auch bei bestimmten Artefakten – so wird die Klassifizierung abgebrochen und ein Fehler gemeldet.

Anschließend erfolgt der Aufruf aller Klassifizierer. An dieser Stelle sollte erwähnt werden, dass das Klassifikationsmodul auch die Ausführung ohne Bayessches Netz unterstützt. Dies ist beispielsweise dann praktisch, wenn lediglich die Effektivität neuer Klassifizierer getestet werden soll, ohne den Einfluss der zweiten Ebene auf das Ergebnis zu beachten. Der Vorgang ist leicht verschieden, je nachdem, ob das Netz verwendet

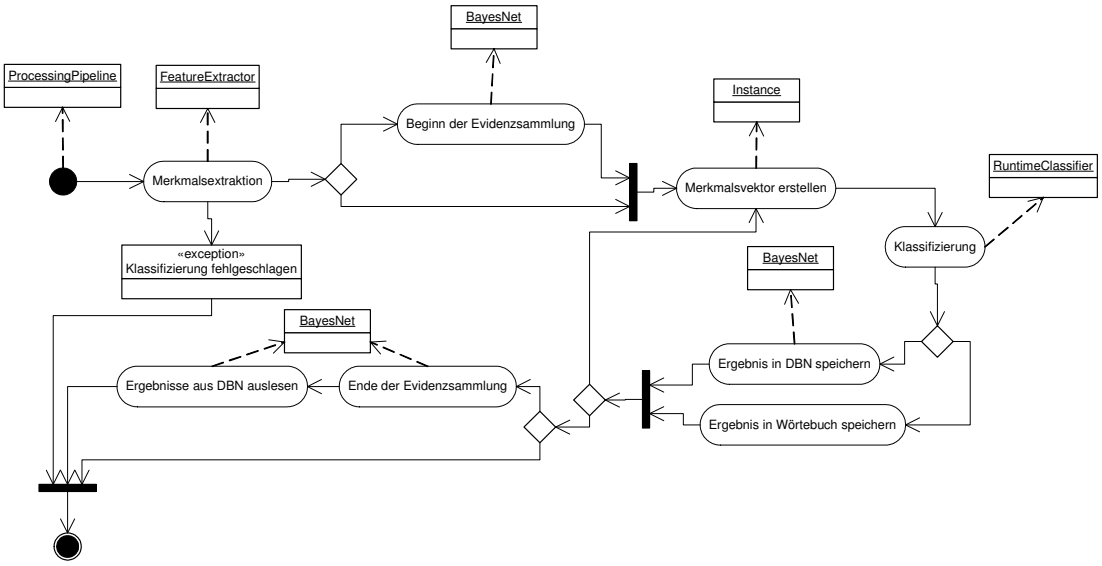



Abbildung 3.14: Aktivitätsdiagramm für die Funktionen `Process<...>` der Klasse `ProcessingPipeline`

wird oder nicht. Ist die zweite Ebene aktiviert, so wird zuerst die Evidenzsammlung für das Netz initialisiert. Danach wird für jeden Klassifizierer geprüft, ob alle benötigten Merkmale extrahiert werden konnten. Auch hier kann der Fall eintreten, dass durch Artefakte bestimmte Algorithmen zur Merkmalsextraktion fehlschlagen und daher nicht alle Merkmale vorliegen. Sollte dies der Fall sein, so wird nur der betreffende Klassifizierer übersprungen und nicht die komplette Verarbeitung. Sofern aber alle Merkmale vorhanden sind, wird ein Merkmalsvektor in Form der Hilfsklasse `Instances` (vgl. Abschnitt 3.7.5) erstellt und durch Aufruf von `Classify` an den Klassifizierer übergeben. Falls die Klassifizierung erfolgreich war, wird die Ergebnisklasse in den dem Klassifizierer zugeordneten Knoten des Bayesschen Netzes eingetragen. Nach dem Abarbeiten aller Klassifizierer wird die Evidenzsammlung beendet und die Inferenz, d.h. Berechnung der bedingten Wahrscheinlichkeiten, durchgeführt. Danach müssen die Ergebnisse noch aus dem Bayesschen Netz ausgelesen und in das lokale Zeichenfolgenwörterbuch `user_model` kopiert werden. Dieses kann von der Anwendung zu einem beliebigen Zeitpunkt über die Funktion `GetResults` abgefragt werden. Häufig geschieht dies unmittelbar nach dem Aufruf von `Process<...>`.

Ist die zweite Ebene deaktiviert, so läuft der Vorgang etwas anders ab. Jedem Klassifizierer ist zusätzlich noch ein Profilpfad zugeordnet, der angibt, welche Sprechereigenschaft er bestimmt. Dies kann z.B. das Sprecheralter sein. Ein Klassifizierer wird nur dann ausgeführt, wenn die durch ihn bestimmte Eigenschaft noch nicht innerhalb des

laufenden Aufrufs von einem anderen Klassifizierer bestimmt wurde. Liegt die Eigenschaft bereits vor, so wird der Klassifizierer übersprungen. Die Ergebnisse werden dann ohne Konvertierung direkt in das lokale Feld kopiert, was auch zur Folge hat, dass sich das Parsen der Ergebnisse mit und ohne zweite Ebene unterscheiden kann.

Wird nun nach Abschluss der Verarbeitung in der Pipeline eine der `Process<...>`-Funktionen erneut mit einer neuen Äußerung aufgerufen, so wird diese als zusätzliche Zeitscheibe in das dynamische Bayessche Netz eingefügt. Dadurch können die Ergebnisse der vorangegangenen Klassifizierungen in die neusten Berechnungen mit einbezogen werden. Ist dies nicht gewünscht, so kann die Anwendung durch Aufruf der Funktion `Reset` alle Zeitscheiben im Netz sowie das vorhandene Benutzerprofil löschen. Dabei handelt es sich um eine in Funktion und Performanz etwa gleichwertige Alternative zum Löschen und Neuanlegen der Pipeline. Lediglich der Klassifizierer-Cache wird nicht neu erstellt, falls das betroffene Objekt die letzte Instanz einer `ProcessingPipeline` ist.

Im jetzigen Entwurf der Verarbeitungs-Pipeline noch nicht enthalten ist die Vorverarbeitung der Sprachdatei. Primär handelt es sich dabei um die Konvertierung in das benötigte Format im Hinblick auf Abtastfrequenz, Abtasttiefe und Kanalanzahl. Die Merkmalsextraktion selbst kann zwar mit beliebigen Formaten arbeiten, jedoch sollten diese kompatibel zu den beim Training verwendeten Audioformaten sein. Da allerdings Training der Klassifizierer und Ausführung der Klassifikation im Rahmen des gleichen Anwendungsszenarios ausgeführt werden, sollte es kein Problem darstellen, in beiden Fällen die gleichen Formate zu verwenden. Was weitere Vorverarbeitungsschritte anbelangt, wie z.B. das Herausfiltern von Artefakten oder das Entfernen von Einsatzlatenz (Pausen zu Beginn der Äußerung), so wird zuerst noch einige Vorarbeit auf theoretischer Ebene benötigt, sowie eine testweise Implementierung im AGENDER-Prototypen. Mit Verfügbarkeit dieser Vorgaben kann dann auch die Implementierung im Klassifikationsmodul erfolgen.

Es liegt in der Verantwortung der Host-Anwendung, eine einmal erstellte `ProcessingPipeline` nach Verwendung wieder zu zerstören, um den Speicher freizugeben. Dies geschieht durch einen standardgemäßen Aufruf des Destruktors. Wird die letzte `ProcessingPipeline` zerstört, so wird auch der Klassifizierer-Cache geleert. Falls die Anwendung jeweils Pipelines nach Bedarf erstellt, kann es daher sinnvoll sein, eine zusätzliche Pipeline während der gesamten Anwendungslaufzeit mit zu verwalten, um das Leeren des Cache zu verhindern. Falls die Zahl der Sprecher dann zwischenzeitlich häufig auf Null zurückgeht, kann dadurch ein Performanzgewinn erzielt werden.

Um die Funktionsweise der Parallelverarbeitung zu verstehen soll nun kurz die Funktionsweise von Threads beschrieben werden. Ein *Thread* im Kontext von Betriebssystemen bezeichnet die Ausführung einer Funktion durch den Prozessor, wobei alle Instruktionen innerhalb der Funktion sequenziell ausgeführt werden, mehrere Threads aber parallel. Da ein einzelner Prozessor in der Praxis allerdings nur eine Anweisung zur gleichen Zeit

verarbeiten kann, wird die Parallelausführung simuliert, indem die Funktionen abwechselnd[7] ausgeführt werden. Man unterscheidet zwei Arten von Threads: Vordergrund-Threads (oder UI-Threads) und Hintergrund-Threads (oder Arbeitssthreads). Erstere sind für die Interaktion mit dem Nutzer zuständig, während Hintergrundthreads die rechenintensiven Vorgänge ausführen. Wenn zwei Threads auf die gleiche Variable zugreifen, kann es zu ungewollten Nebeneffekten kommen. Um dies zu vermeiden, müssen solche Zugriffe synchronisiert werden, z.B. über kritische Abschnitte.

Die Verarbeitungs-Pipeline unterstützt die parallele Klassifizierung durch mehrere Threads, sofern jeder Thread eine eigene Pipeline verwendet. Die Aufgabe der Erstellung dieser Threads liegt jedoch bei der Anwendung. Es hätte durchaus die Möglichkeit bestanden, die Thread-Verwaltung selbst in SBC zu integrieren. Dies schien jedoch nicht sinnvoll, da die meisten Anwendungen zur Wahrung ihrer Flexibilität eine vollständige Kontrolle über die laufenden Threads benötigen. Gerade serverbasierte Anwendungen steuern dies häufig über einen *Thread-Pool*, aus dem mittels Schablonen fertige Arbeitsthreads erstellt und ausgeführt werden. Folgende Vorgehensweise ist empfehlenswert, um optimale Skalierbarkeit und Performanz zu gewährleisten: Bei einem einzelnen exklusiven Nutzer sollte die Applikation für SBC einen Hintergrund-Thread anlegen und dort alle Aufrufe an die Verarbeitungs-Pipeline ausführen. Dadurch kann der Benutzer ungestört weiterarbeiten. Die Abfrage der aktuellen Daten kann von jedem Thread aus erfolgen. Für Serverszenarien mit mehreren Benutzern sollte für jeden Benutzer ein eigener Thread mit einer eigenen Pipeline erstellt werden. Ist die Zahl der gleichzeitig tätigen Benutzer sehr hoch oder die Rechenleistung nicht ausreichend, so kann die Anzahl der Threads auch begrenzt werden, um keine Ressourcenengpässe hervorzurufen. Falls dann das Maximum an Threads erreicht ist, werden Klassifizierungsanfragen von der Anwendung in eine Warteschlange gestellt und bei „Freiwerden“ eines Threads auf diesem ausgeführt. Es spielt dabei keine Rolle, ob der Thread zuvor auf einer anderen Pipeline klassifiziert hat (solange keine Pipeline gleichzeitig von zwei Threads verwendet wird). Dies entspricht dann vom Prinzip her einem Thread-Pool. Es wäre auch denkbar, dass besonders leistungsfähige Server mit mehreren Prozessoren die Klassifizierungs-Threads auch auf unterschiedliche Prozessoren verteilen könnten. Bei einem Datenvolumen, dem ein einzelner Rechner nicht mehr nachkommen kann, ist es auch möglich, eine Cluster-Verarbeitung zu implementieren. Dabei wird auf jedem Knoten des Clusters eine unabhängige Instanz des Klassifizierungsmoduls ausgeführt. Der Lastausgleich wird von der Anwendung durchgeführt, d.h. die Anwendung entscheidet, an welches Klassifikationsmodul ein neuer Benutzer weitergeleitet wird. Diese Zuordnung muss für nachfolgende Äußerungen des gleichen Benutzers beibehalten werden, um die Vorteile des dynamischen Bayesschen Netzes zu nutzen.

[7] Es gibt in der Praxis verschiedene Algorithmen zur Zeitscheibenverteilung, z.B. *Round Robin* (vgl. Jones, Roşu und Roşu, 1997).

### 3.7.2 Merkmalsextraktion

Die Merkmalsextraktion bezeichnet den Vorgang, durch welchen die zur Klassifizierung benötigten Stimm- und Sprecherverhaltensmerkmale aus einer binären Audiodatei erhalten werden (vgl. Abschnitt 2.1). In der vorliegenden Version beherrscht das Klassifikationsmodul noch nicht alle der in Abschnitt 2.1.3 aufgeführten Größen. Der Satz der Stimmmerkmale, welche nach derzeitigen Erkenntnissen von Müller (2005) die wichtigste Grundlage zur Bestimmung von Alter und Geschlecht darstellen, liegt bereits annähernd komplett vor. Dem gegenüber stehen zwar die noch nicht verfügbaren Sprechverhaltensmerkmale; diese dienen jedoch bis jetzt hauptsächlich der Unterstützung der restlichen Merkmale für Grenzfälle der Klassifizierung, oder wenn akustische Merkmale nicht verlässlich bestimmt werden können, z.B. aufgrund von lauten Hintergrundgeräuschen.

Die Merkmalsextraktion ist in der überwiegenden Mehrzahl der Konfigurationen der aus rechnerischer Sicht aufwändigste und damit zeitintensivste Teil der Verarbeitung einer Äußerung. Auch wenn man sich Klassifizierer vorstellen könnte, welche theoretisch beliebig komplex sind – in der Praxis ist die Klassifizierung selbst bei Sprache sehr viel schneller als die zugrunde liegende Datenanalyse. Daher ist es sehr wichtig, dass die verwendeten Algorithmen möglichst effizient sind.

Zu Beginn der Implementierung der Merkmalsextraktion wurden die Möglichkeiten zur Integration ausgelotet: Die Einbindung einer Bibliothek oder ausführbaren Datei kam nicht in Frage, da sie die Plattformtransparenz zerstört hätte – die Extraktionskomponente hätte für alle benötigten Plattformen ebenfalls vorliegen müssen. Auch hätte dies der Kompaktheit des Moduls geschadet, und überdies wäre eine ausführbare Datei sehr ineffizient gewesen. Da also diese Variante ausgeschlossen wurde, bleiben nur noch zwei Möglichkeiten: Die Einbindung von existierendem Quellcode oder die Neuimplementierung der benötigten Algorithmen. Die Kosten einer Neuimplementierung sind im Falle einer Merkmalsextraktion relativ hoch, da bei der Audioanalyse eine Vielzahl mathematischer Verfahren zum Einsatz kommt. Aber auch die Code-Integration bedarf einigen Aufwands, da der Quellcode zum einen von der ursprünglichen Zielplattform abstrahiert werden muss, um in das SBC-Framework zu passen, und zum anderen für jede Plattform zusammen mit dem Klassifikationsmodul portiert werden muss. Der einzelne Aufwand für die Portierung kann natürlich je nach Plattform sehr verschieden sein. Wenn die Plattformabstraktion des Codes gut gelingt, kann es sein, dass jeweils keine oder nur geringe Anpassungen notwendig sind.

Umgesetzt wurde letztlich nach Auseinandersetzung mit allen in Frage kommenden Möglichkeiten eine Portierung der Phonetik-Software *Praat* (vgl. Abschnitt 2.1.3). Neben der grundsätzlichen Eignung des Programms waren hauptsächlich folgende Gründe für diese Entscheidung verantwortlich.

Zunächst ist der *C*-Quellcode des Programms frei verfügbar. Dies schafft überhaupt

erst die Voraussetzung, die eine performante, plattformtransparente Integration möglich macht. Daneben enthält *Praat* eine Vielzahl an hochwertigen Algorithmen, welche auf jahrelanger Forschung basieren. Der Autor hat viele der von ihm entwickelten oder verbesserten Verfahren auch veröffentlicht. Für eine eigene Implementierung wäre innerhalb kurzer Zeit keine adäquate Forschung möglich gewesen, und die Konsultierung verschiedener Quellen hätte schnell zu Inkonsistenzen in der Methodik geführt. Aber nicht nur die Qualität der Algorithmen ist herausragend, sondern auch die Geschwindigkeit. So wurden beispielsweise effiziente Algorithmen aus der *GSL* (*GNU Scientific Library*, vgl. Galassi et al., 2003) zur Implementierung der *Fast Fourier Transform* verwendet und an zahlreichen Stellen Optimierungen durchgeführt. Auch hier hätte eine Neuimplementierung nur schwerlich Schritt halten können. Ferner wurde *Praat* schon einige Zeit erfolgreich im M3I-Server und bei der Korpusanalyse eingesetzt; somit lag es nahe, das Phonetikwerkzeug auch für das Klassifikationsmodul zu verwenden. Neben den guten Erfahrungen bei der Desktop-Version von *Praat* wird durch die Verwendung der gleichen Algorithmen sichergestellt, dass die auf einer Entwicklungsplattform trainierten Klassifizierer auch mit den Merkmalswerten des eingebetteten Moduls arbeiten können. Hierbei ist zu erwähnen, dass sich bei frühen Tests[8] herausstellte, dass plattformbedingte Unterschiede dennoch Abweichungen in den errechneten Werten bewirken können. In dem Fall gibt es prinzipiell zwei Möglichkeiten: Entweder man akzeptiert diesen Präzisionsverlust, sofern die Evaluierung immer noch eine akzeptable Gesamtgenauigkeit ergibt. Eine bessere Alternative bietet sich aber möglicherweise an, indem zum Training in der Entwicklungsplattform eine durch das auf der Zielplattform laufende Klassifikationsmodul erstellte Merkmalsdatenbank verwendet wird. Dazu muss eine Wrapper-Applikation geschrieben werden, welche eine Korpusanalyse auf der Zielplattform durchführen kann, ähnlich wie in Abschnitt 3.8.3 beschrieben.

*Praat* wurde im Rahmen von Feld (2005) als statische Bibliothek für den *PocketPC* portiert. Die in der genannten Arbeit präsentierte Version sollte ursprünglich nur in die M3I-Client-Bibliothek auf dem *PocketPC* integriert werden. Durch die Neuorientierung mit SBC hat sich eine etwas generellere Formulierung der Zielplattform ergeben. Dennoch konnten Großteile der Portierungsarbeit und Optimierung direkt übernommen werden, wie z.B. die Ausgliederung der grafischen Benutzeroberfläche. Prinzipiell lassen sich alle Merkmale auslesen, die auch mit der Desktop-Version des Werkzeugs zu erfassen sind, also insbesondere *Grundfrequenz*, *Jitter*, *Shimmer*, *Intensität* und *Harmonicity-to-Noise-Ratio*.

Im SBC-Klassifikationsmodul wird *Praat* gekapselt durch die Klasse `FeatureExtractor` (s. Abb. 3.15). Ein Objekt dieser Klasse wird erstellt, sobald Sprachmerkmale aus einer Datei extrahiert werden sollen. Das Objekt enthält für jedes Merkmal eine loka-

[8] Bei dem betroffenen Szenrio war die Entwicklungsplattform *Linux*, und das Klassifikationsmodul wurde unter *Windows* ausgeführt.

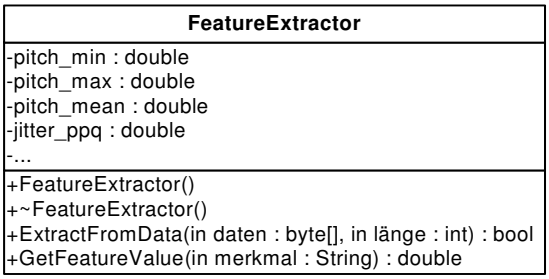



Abbildung 3.15: UML-Diagramm der Klasse `FeatureExtractor`

le Variable. Die Datentypen sollen der jeweiligen Merkmalsrepräsentation bestmöglich angepasst sein. Zurzeit verwenden alle Merkmale Fließkommazahlen mit doppelter Genauigkeit (`double`).

Im Konstruktor für den `FeatureExtractor` werden alle Merkmale auf einen Standardwert festgelegt, welcher für „nicht verfügbar“ steht. Kann ein Merkmal nicht erhalten werden, so bleibt die lokale Variable auf dem Standardwert eingestellt und der Aufrufer erkennt, dass das Merkmal nicht verfügbar ist. In diesem Fall würde ein Klassifizierer, der dieses Merkmal als Teil des Eingabevektors benötigt, für die betreffende Äußerung nicht ausgeführt werden.

Zur Durchführung der Merkmalsextraktion wird die Funktion `ExtractFromData` aufgerufen. Diese Funktion ruft intern hauptsächlich Routinen der *Praat*-Bibliothek auf, um die Daten zu bestimmen. Zuerst werden die Audiodaten von *Praat* gelesen und interpretiert. Liegen ungültige Daten vor, so bricht die Funktion mit einem Fehler ab (es wird `False` zurückgegeben). Anschließend wird die Grundfrequenzkontur berechnet. Hierbei wird eine Frequenzanalyse mit Hilfe einer Autokorrelationsmethode aus Boersma (1993) wie folgt durchgeführt: Für eine große Anzahl an Punkten innerhalb des Sprachsignals (standardmäßig im Abstand von je 0,01 Sekunden) wird ein kleiner Bereich einer festen Größe betrachtet, der um diesen Punkt zentriert ist. Die Größe des Bereichs hängt von der minimalen zu erkennenden Frequenz ab. Auf das Signal in diesem Bereich wird dann ein *Hanning-Window* angewendet. Dies ist eine Funktion, die den gleichen Definitionsbereich wie das Teilsignal besitzt und zu den Rändern hin gegen Null geht. Bei Multiplikation mit dem Audiosignal wird dieses an den Rändern abgeschwächt und im Zentrum hervorgehoben. Vom Ergebnis wird dann die normierte Autokorrelation berechnet, und diese wiederum durch die Autokorrelation der Hanning-Window-Funktion dividiert. Um die Pitch-Frequenz zu finden, müssen nun die lokalen Maxima gesucht werden, wobei das größte Maximum der beste Kandidat für die Frequenz am betrachteten Punkt ist. Da das Signal aber nicht als Funktion vorliegt, sondern vielmehr in Form von einzelnen Samples, können diese Maxima nur durch Interpolation der vorhandenen Werte bestimmt werden. Die Form der Kurve entspricht der einer $sin(x)/x$-Funktion,

daher wird diese zur Berechnung benutzt. Die Tiefe der Interpolation bestimmt dabei die Genauigkeit des Ergebnisses. Dieser Prozess beansprucht in den meisten Szenarien den größten Teil der gesamten Verarbeitungszeit. Dank einer in Feld (2005, S. 27) vorgeschlagenen Optimierung konnte die Geschwindigkeit jedoch um ca. Faktor 9 gesteigert werden, wobei geringe Präzisionsverluste in Kauf genommen werden müssen (weniger als 0,00001 Prozentpunkte Abweichung). Ist dies nicht akzeptabel, so kann die Anwendung auch auf die Optimierung verzichten. Benchmarks in Feld (2005, S. 26ff) haben jedoch gezeigt, dass auf leistungsschwachen Plattformen wie dem *PocketPC* die Verarbeitungszeit durch die Optimierung etwa von 60 auf 7 Sekunden[9] gesenkt werden konnte, also unbedingt durchgeführt werden sollte.

Die Grundfrequenzkontur wird zur Berechnung aller Grundfrequenz-Merkmale benötigt, sowie zum Erstellen des `PointProcess`-Objekts. Dieses Objekt enthält eine Reihe von Punkten auf der Zeitachse, die für akustische Wiederholungsintervalle stehen, und wird über einen Periodenerkennungsalgorithmus in der Nähe von Stellen mit großer Amplitude berechnet. Der `PointProcess` wiederum ist Grundlage zur Berechnung der Jitter- sowie der Shimmer-Werte. Im Fall des Shimmers wird die Amplitude nur an den im `PointProcess` gespeicherten Markierungen herangezogen. Als Letztes werden noch Objekte vom Typ `Harmonicity` und `Intensity` direkt aus den Audiodaten erstellt, welche – wie sich schon am Namen vermuten lässt – der Berechnung von HNR und Intensität dienen. Eine genauere Beschreibung der genannten *Praat*-Routinen kann in Feld (2005, S. 7ff) nachgelesen werden.

Nachdem die Werte berechnet und in den lokalen Variablen abgespeichert wurden, werden die *Praat*-Objekte wieder zerstört und der Speicher freigegeben. Um von außerhalb der Klasse auf die berechneten Werte zuzugreifen, kann die Funktion `GetFeatureValue` verwendet werden. Diese nimmt als Argument den Namen eines Sprachmerkmals und gibt dessen Wert im `double`-Format zurück. Diese Konvertierung ist – sofern der Merkmalstyp ein anderer als `double` ist – erforderlich, da alle Klassifizierer mit Merkmalsvektoren arbeiten, die als Datenfeld von `double`-Werten repräsentiert werden. Dieses Format kann bislang für alle in AGENDER auftretenden Merkmalswerte verwendet werden. Der Merkmalsname muss dem Wert aus einer statisch im Code enthaltenen Liste entsprechen, die jedoch in Zukunft recht einfach um weitere Merkmale ergänzt werden könnte, sofern dies nötig werden sollte. Die aktuelle Liste der unterstützten Merkmale ist in Tabelle 3.3 dargestellt. Die Namen folgen der Merkmalsbenennungskonvention von AGENDER. Für eine genauere Beschreibung der einzelnen Funktionen siehe Müller (2005, S. 62).

Eine Optimierung, die bei dem `FeatureExtractor` in Zukunft noch durchgeführt werden soll, ist die Beschränkung der Merkmalsextraktion auf diejenigen Merkmale, die auch tatsächlich von einem der Klassifizierer benötigt werden. In der Regel ist es so,

[9] *HP Ipaq H5450* mit einer Audiodatei von 2,3s, 8000 Hz, 16 bit, mono

| **Name** | **Merkmal** | **Name** | **Merkmal** |
|---|---|---|---|
| pitch_min | Tiefste Grundfrequenz | pitch_max | Höchste Grundfrequenz |
| pitch_diff | Grundfrequenzbereich (Maximum-Minimum) | pitch_stddev | Standardabweichung der Grundfrequenz |
| pitch_mean | Durchschnittliche Grundfrequenz | pitch_quant | Median (50%-Quantil) der Grundfrequenz |
| pitch_mas | Absolute mittlere Steigung der Grundfrequenz | pitch_swoj | Geschwindigkeit der F0-Veränderungen ohne Oktavensprünge |
| shim_apq3 | *Three-point Amplitude Perturbation Quotient* (APQ) in kleiner Umgebung | shim_apq5 | APQ in mittlerer Umgebung |
| shim_apq11 | APQ in großer Umgebung | shim_l | Lokaler Shimmer |
| shim_ldb | Lokaler Shimmer in dB | shim_ddp | *Difference of Differences of Periods* |
| harm_min | Minimum der Harmonizität | harm_max | Maximum der Harmonizität |
| harm_mean | Durchschnittliche Harmonizität | harm_stddev | Standardabweichung der Harmonizität |
| intens_min | Minimum der Intensität | intens_max | Maximum der Intensität |
| intens_mean | Durchschnittliche Intensität | intens_stddev | Standardabweichung der Intensität |
| jitt_l | Lokaler Shimmer | jitt_la | Absoluter lokaler Shimmer |
| jitt_ppq | *Five-point Period Perturbation Quotient* (PPQ5) | jitt_rap | *Relative Average Perturbation* |
| jitt_ddp | *Difference of Differences of Periods* | | |

Tabelle 3.3: Liste der bereits von `FeatureExtractor` unterstützten Merkmale

dass pro Merkmalsgruppe (z.B. Grundfrequenz, Jitter, Shimmer) einmalig ein etwas aufwändiger Analysevorgang durchgeführt werden muss, die abgeleiteten Größen (z.B. *jitter_ppq*, *jitter_rap*) dann aber fast unmittelbar ausgelesen werden können. Außerdem bauen einige Größen aufeinander auf: So ist für die Bestimmung des Jitter oder Shimmer zwangsläufig auch immer die Grundfrequenzkontur des Signals erforderlich. Aus den genannten Gründen ist die angesprochene Optimierung nicht ganz so aussichtsreich wie es auf den ersten Blick erscheint, soll aber nichtsdestoweniger umgesetzt werden.

### 3.7.3 Erste Ebene

Die erste Ebene des AGENDER-Ansatzes beschreibt die Mustererkennung. Diese wird durch die einzelnen Klassifizierer implementiert, welche wiederum jeweils durch ein einzelnes Objekt repräsentiert werden. Um eine größtmögliche Flexibilität zu gewährleisten, wird für jeden Klassifizierer eine eigene *C++*-Klasse angelegt, welche den beim Entwurf festgelegten Klassifizierungsalgorithmus beinhaltet. Prinzipiell wäre es zwar auch denkbar, für jeden Typ von Klassifizierer eine eigene Klasse oder Basisklasse zu schaffen und dort den gemeinsamen Code unterzubringen, z.B. eine Klasse `NeuralNetClassifier` für Künstliche Neuronale Netze. Die Vorteile eines solchen Designs würden jedoch nicht zur Geltung kommen: Weder bietet die zusätzliche Basisklasse Einsatzmöglichkeiten im Klassifikationsmodul in Form einer zusätzlichen Schnittstelle für alle NeuralNet-Klassifizierer, noch dient sie der Vereinfachung der Implementierung aus objektorientierter Sicht, da alle Klassifizierer-Klassen automatisch von der DEVELOPMENT PLATFORM generiert werden und danach nicht manuell bearbeitet werden. Umgekehrt gibt es einen entscheidenden Vorteil, wenn für jeden Klassifizierer eine eigene Klasse erzeugt wird: Es lassen sich klassifizierer-spezifische Optimierungen vor der Kompilierung durchführen. Beispielsweise muss für ein Datenfeld nicht getrennt die Obergrenze in einer Variablen gespeichert und zur Laufzeit ausgewertet werden, sondern die Obergrenze wird an jeder Stelle im Quellcode als Zahl eingesetzt. In diesem und ähnlichen Fällen können also allgemeine Konstrukte durch spezifischeren und damit häufig schnelleren Code ersetzt werden.

Obwohl es keine modellabhängigen Klassen gibt, existiert dennoch eine allgemeine Basisklasse für alle Klassifizierer. Sie trägt den Namen `RuntimeClassifier`, um die zweckbedingte Unterscheidung von den Klassifizierern der Entwicklungsplattform zu verdeutlichen. Diese besitzen im Gegensatz zu denen des Klassifikationsmoduls nämlich noch die Möglichkeit, durch Training die interne Datenstruktur zu erstellen oder zu erweitern, wohingegen ein `RuntimeClassifier` kein Training unterstützt und lediglich der Klassifizierung dient. Die Klasse ist in Abb. 3.16 dargestellt. Die Notwendigkeit dieser Basisklasse ergibt sich daraus, dass das Klassifikationsmodul die Klassifizierer nur über eine gemeinsame Schnittstelle bedienen kann. Für diesen Fall scheint die Vererbung in *C++* die geeignete Lösung zu sein: Die Klasse ist abstrakt, da es keine „Standard"-Implementierung einer Klassifizierung gibt, d.h. jeder Klassifizierer muss diese Methode beschreiben.

Alle Klassifizierer verfügen über eine Reihe von Feldern. An erster Stelle zu nennen ist die `ID`, welche beim Entwurf festgelegt wird und den Klassifizierer innerhalb des Moduls eindeutig kennzeichnet. Über sie wird das Ergebnis des Klassifizierers demjenigen Knoten im Bayesschen Netz zugeordnet, welcher die gleiche ID besitzt. Diese enthält in der Regel keinen Hinweis auf den zugrunde liegenden Algorithmus, so dass dieser nachträg-

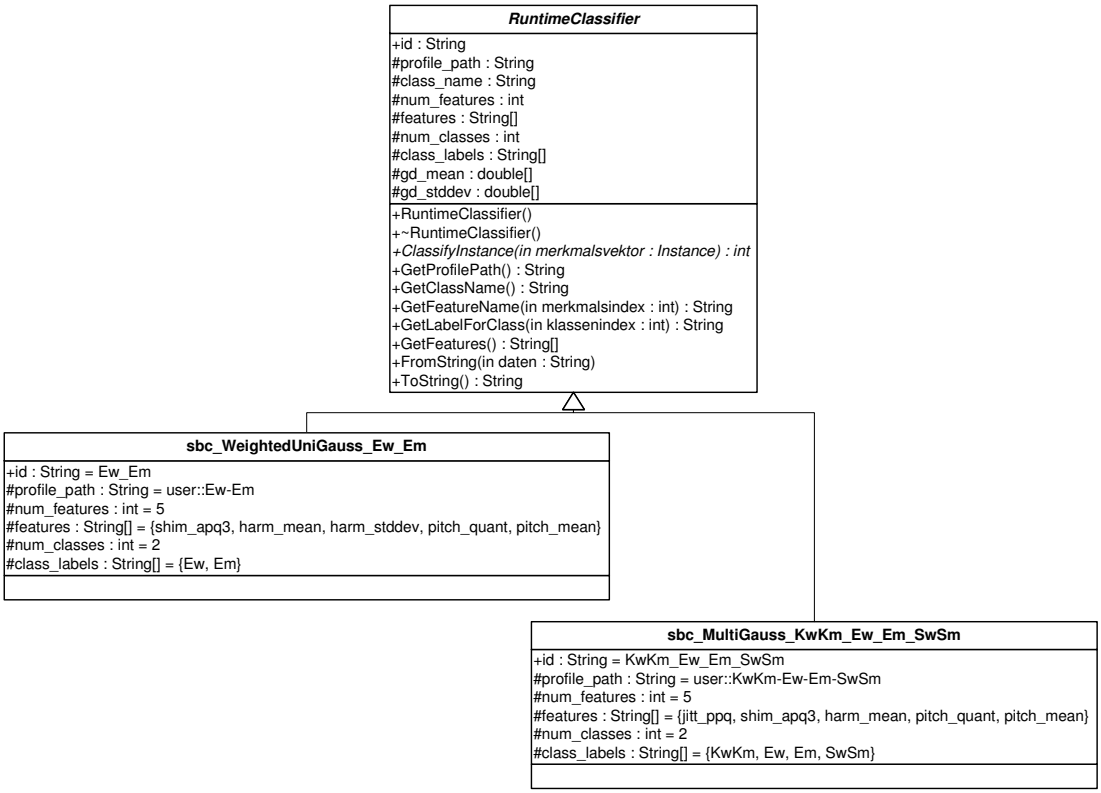



Abbildung 3.16: UML-Diagramm der Klasse `RuntimeClassifier` sowie zweier davon abgeleiteter Klassen

lich geändert werden kann, ohne dass sich weitere Auswirkungen auf die Konfiguration der zweiten Ebene des Moduls ergeben.

Daneben hat jeder Klassifizierer einen Profilpfad (`profile_path`) und eine Klasse (`class_name`). Der Profilpfad wird nur dann verwendet, wenn das Modul ohne Bayessches Netz konfiguriert wird, um das Ergebnis der Klassifizierer unmittelbar einer Benutzereigenschaft zuordnen zu können. Der Klassenname wird als Ersatz für die in *C++* nicht vorhandene Reflektion gespeichert. Er setzt sich aus dem Anwendungsnamen (vgl. Abschnitt 3.9.2), dem Algorithmus und der ID zusammen. Genutzt wird dieser Wert momentan lediglich zu Debugging-Zwecken, d.h. um die Protokolldatei informativer zu gestalten. Für beide Werte existiert noch eine so genannte Get-Accessor-Methode (`GetProfilePath` und `GetClassName`), deren einzige Aufgabe es ist, diese Felder nach außen hin schreibgeschützt bereitzustellen.

Zwei zentrale Attribute eines jeden Klassifizierers sind die von ihm benötigten Merkmale und die ausgegebenen Klassen (im Sinne der Klassifikation). Beide werden intern als Vektor behandelt und nur durch Indizes referenziert, da dies sehr viel effizienter ist als z.B. über Bezeichner. Ihre jeweilige Anzahl ist in `num_features` und `num_classes` gespeichert; in der Regel wird jedoch im automatisch erzeugten Code der eigentliche Wert anstelle der Variablen eingesetzt. Innerhalb der Instanz eines Moduls müssen die Indizes konsistent sein, was durch den Erzeugungsvorgang sichergestellt wird. Es wird jedoch nicht garantiert, dass die Indizes zwischen zwei getrennt erzeugten Instanzen eines Moduls mit gleicher Konfiguration identisch sind. Für den Fall, dass ein Eintrag nicht ausschließlich über den Index referenziert werden kann, werden auch die Bezeichnungen der Merkmale und Klassen in Feldern gespeichert (`features` und `class_labels`). Um die Bezeichnung eines bestimmten Merkmals, basierend auf dem Index, zu bestimmen, kann die Methode `GetFeatureName` verwendet werden; für Klassen geschieht dies analog durch `GetLabelForClass`. Ein Aufruf von `GetFeatures` liefert in einem Aufruf alle Merkmale zurück, indem eine Kopie des Feldes `features` erstellt wird. Die Merkmalsbezeichnungen dienen der Zuordnung zu den statischen Merkmalen des FeatureExctractors (s. Tabelle 3.3 auf Seite 76) beim Erstellen eines `Instance`-Objekts (vgl. Abschnitt 3.7.5) aus einem `FeatureExtractor`-Objekt. Beide Listen werden darüber hinaus zur Verbesserung der Lesbarkeit der Protokolldatei verwendet. Außerdem lassen sich dadurch auch Werkzeuge realisieren, die zur Evaluierung eines Klassifikationsmoduls oder zur Korpus-Analyse verwendet werden können.

Die eigentliche Klassifizierung erfolgt in der Methode `ClassifyInstance`[10]. Diese Methode besitzt in der Klasse `RuntimeClassifier` keinen Rumpf und wird daher als *abstrakt* oder *pure virtual* bezeichnet. Eine von `RuntimeClassifier` abgeleitete Klasse muss diese Methode überschreiben und den Rumpf implementieren, damit sie im Modul verwendet werden kann. Die Methode erhält als Parameter einen Merkmalsvektor

[10]Die Namensgebung wurde von *WEKA* übernommen.

in Form eines `Instance`-Objekts, welches nur die vom Klassifizierer benötigten Merkmale in der beim Export des Klassifizierers festgelegten Reihenfolge enthält, so dass über Indizes auf die Werte zugegriffen werden kann. Rückgabewert ist die durch die Klassifizierung ermittelte Ergebnisklasse, die ebenfalls über einen nullbasierten Index ausgedrückt wird. Die Methode kann auch $-1$ zurückliefern. Dies wird in dem Sinn interpretiert, als wenn der Klassifizierer keine Angabe zur Klasse machen kann. Ohne zweite Ebene fehlt dann der Wert für den zugehörigen Profilpfad. Wird das Bayessche Netz verwendet, so kommt eine nicht vorhandene Evidenz entsprechend in der Berechnung zum Ausdruck (vgl. Abschnitt 3.7.6).

Einige Klassifizierer arbeiten mit normierten Merkmalswerten. In der Regel verbessert dies die Aussagekraft der Werte beim Testen der Klassifizierungsverfahren, da sich zum Beispiel der Einfluss verschiedener Merkmale bzw. deren Abweichungen von einem Durchschnittswert im Hinblick auf die berechnete Ergebnisklasse besser nachvollziehen lässt. Einige Algorithmen funktionieren nur mit normierten Zahlen. Wird die Normierung verwendet, so muss bereits das Training der Klassifizierer mit normierten Werten durchgeführt werden. Da die Entwicklungsplattform Zugriff auf die vollständige Datenbasis hat, können die zur Normierung benötigten Größen *Mittelwert* und *Standardabweichung* leicht berechnet werden. Beides wird pro Merkmal und pro Klassifizierer bestimmt. Die Aufteilung nach Klassifizierern basiert auf der Beobachtung, dass Klassifizierer mit unterschiedlichen Klassen mitunter vollständig verschiedene Wertebereiche abdecken können, und dass sich eine exaktere Normierung durch Berücksichtigung dieser jeweiligen Bereiche umsetzen lässt. Das Klassifikationsmodul liest mit Hilfe der Merkmalsextraktion absolute Werte ein. Um diese normieren zu können, muss auch das Modul den Wertebereich der jeweiligen Merkmale kennen. Da dem Modul im Gegensatz zur Entwicklungsplattform weder der Sprachkorpus noch eine Datenbank mit Merkmalswerten aller Äußerungen zur Verfügung stehen, erhält jeder Klassifizierer die Datenfelder `gd_mean` und `gd_stddev`, die jeweils Mittelwert und Standardabweichung für alle Merkmale des Klassifizierers enthalten. Wie die restlichen Felder werden diese beim Export festgelegt.

Ein weiteres Konzept, welches durch `RuntimeClassifier` unterstützt wird, ist das der *Serialisierung*. Dabei handelt es sich allgemein um die Überführung eines sich im Arbeitsspeicher befindlichen Objekts oder einer Objekthierarchie in eine kompakte binäre oder textuelle Repräsentation, welche in einer Datei oder einem Datenstrom gespeichert werden kann. Hauptzweck der Serialisierung ist die persistente Speicherung von Objekten. Mit Verbreitung der verteilten Verarbeitung und *Web Services* wird Serialisierung aber auch häufig dazu verwendet, um Objekte über eine Netzwerkverbindung zugänglich zu machen. Genutzt wird dies unter anderem bei Technologien wie *Java RMI* oder *.NET Remoting*. Die Serialisierung eines Objekts kann grundsätzlich auf zwei Arten bereitgestellt werden: Entweder über eine automatische, vom Programmier-Framework

bereitgestellte Standard-Implementierung, welche über vordefinierte Verfahren lokale Felder des Objekts sowie alle verknüpften Objekte serialisiert, oder über eine benutzerdefinierte Funktion, welche die Daten in eine vorgegebene Datenstruktur schreibt und aus dieser wieder auslesen kann.

Für die SBC-Klassifizierer wurde eine angepasste[11] Form der Serialisierung entwickelt, bei der ein Klassifizierer aus einer codierten Zeichenfolge erstellt oder in eine solche umgewandelt werden kann. Dadurch ist es möglich, dass Klassifizierer mit einem Server oder zwischen Clients ausgetauscht werden. Die Methode `FromString` dient dazu, den Klassifizierer mit der spezifizierten Zeichenfolge neu zu initialisieren. Der verwendete Parsing-Vorgang ist unterschiedlich, je nachdem welcher Algorithmus verwendet wird. Nicht alle Modelle eigenen sich zur Serialisierung: Hier sind hauptsächlich die Faktoren Speicherverbrauch und Komplexität zu nennen. Im Rahmen von M3I und SBC wurden bereits Serialisierungsmethoden für die Entscheidungsbaum-Implementierung *C4.5* von *WEKA* entwickelt und getestet. Diese Klasse eignet sich aufgrund der kompakten Struktur besonders gut zur Serialisierung: Eine *C4.5*-Codezeichenfolge war bei den verwendeten Korpora in der Regel weniger als 1 KB groß und kann folglich problemlos über Funkverbindungen zu einem mobilen Gerät gesendet werden. Analog zu `FromString` gibt es eine Methode `ToString`, welche den betreffenden Klassifizierer in eine Zeichenfolge umwandelt. Dies kann dann nützlich sein, wenn das Modul in eine Server-Applikation integriert werden soll, welche Clients mit Klassifizierer-Updates versorgt. Die beschriebenen Methoden werden nicht von allen Klassifizierungsverfahren unterstützt. Wird der Austausch von Klassifizierern von einer Anwendung gewünscht, so müssen im Vorfeld passende Modelle gewählt oder neu entwickelt werden.

Als Einschränkung bei der Serialisierung ist zu beachten, dass die Klasse eines Klassifizierers – und damit die Implementierung des eigentlichen Algorithmus – bereits im Klassifikationsmodul enthalten sein muss. Bei den austauschbaren Daten handelt es im Wesentlichen um den Teil des Klassifizierers, welcher durch den Trainingsvorgang erstellt oder erweitert wird. Somit können durch die Serialisierung der Klassifizierer keine neuen Verfahren installiert werden, aber die Datenbasis lässt sich aktualisieren oder an einen neuen Kontext anpassen. Letzteres bietet sich besonders für mobile Geräte an, welche je nach Standort mit verschiedenen Dienstanbietern verbunden sein können. Der SBC-Server des Anbieters kann dann einen an die akustischen und soziografischen Gegebenheiten der Einrichtung angepassten Klassifizierer-Kontext auf das mobile Gerät laden.

[11] Der *C++*-Standard selbst definiert keine Serialisierung.

### 3.7.4 Verfügbare Klassifizierungsmethoden

Bisher wurden zwei Klassifizierungsmethoden für das eingebettete Modul implementiert. Dabei handelt es sich um den *WeightedUniGaussClassifier*, welcher im Wesentlichen einem Gaussian Mixture Model entspricht, und den *MultiGaussClassifier*, der einen klassischen Bayesschen Klassifizierer darstellt. Die beiden Methoden wurden als Schablonen für die SBC DEVELOPMENT PLATFORM angelegt. Bei jedem Build-Vorgang werden je nach Konfiguration der ersten Ebene aus diesen Schablonen Klassen erzeugt, ähnlich den in Abb. 3.16 dargestellten abgeleiteten Klassen `sbc_WeightedUniGauss_Ew_Em` und `sbc_MultiGauss_KwKm_Ew_Em_SwSm`.

Der *WeightedUniGaussClassifier* funktioniert folgendermaßen: In der Methode `ClassifyInstance` wird für jede Klasse eine gewichtete durchschnittliche Wahrscheinlichkeit $p_i$ berechnet. Die Klasse mit dem höchsten $p_i$ wird als Ergebnis zurückgeliefert. $p_i$ wird bestimmt, indem für jedes Merkmal eine *univariate Gauß'sche Normalverteilung* durchgeführt und das Ergebnis mit einem Gewicht $w_i$ multipliziert wird. Dazu benötigt der Klassifizierer lediglich die nach Klassen und Merkmalen getrennten Zahlen für Mittelwert und Standardabweichung der Trainingsdaten, wodurch er nur sehr wenig Speicher benötigt. Die Gewichte werden beim Exportvorgang von der Entwicklungsplattform bereitgestellt unter Verwendung eines beliebigen Algorithmus wie z.B. dem Expectation-Maximization-Algorithmus oder einer einfachen Evaluierung der True Positive Rate für das Merkmal.

Ähnlich aufgebaut ist der *MultiGaussClassifier*. Auch er bestimmt in `ClassifyInstance` die Klasse mit der höchsten Wahrscheinlichkeit $p_i$. In diesem Fall stellt $p_i$ allerdings die *multivariate Gauß'sche Normalverteilung* dar, die mit Hilfe von Formel 2.1 – ohne nachfolgende merkmalsspezifische Gewichtung – berechnet wird. Wie anhand der Formel zu erkennen ist, werden dafür die Mittelwerte, die inverse Matrix der Kovarianzmatrix und die Determinante der Kovarianzmatrix für alle Klassen benötigt.

Zukünftig sollen noch weitere Verfahren für SBC verfügbar gemacht werden, wie beispielsweise Entscheidungsbäume und Künstliche Neuronale Netze (siehe auch Kapitel 7).

### 3.7.5 Klassifizierer-Hilfsklassen

Bei der Klassifizierung werden Datenstrukturen benötigt, welche nicht im Sprachumfang von *C++* enthalten sind und daher Teil dieser Arbeit waren. Zwei solcher Hilfsstrukturen sollen hier kurz vorgestellt werden.

Die erste Datenstruktur (s. Abb. 3.17) trägt den Namen `Instance` und wurde von der gleichnamigen Klasse im *WEKA*-Paket abgeleitet. Sie kapselt einen Merkmalsvektor für die Klassifizierer. Mit der Methode `FromFEx` wird der Merkmalsvektor durch die Verarbeitungs-Pipeline, basierend auf einem `FeatureExtractor`-Objekt, für einen bestimmten Klassifizierer erstellt. Dazu prüft die Methode zunächst, ob der angegebene

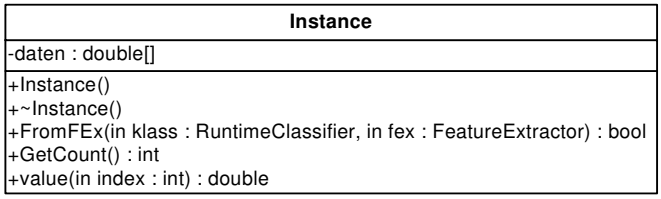



Abbildung 3.17: UML-Diagramm der Klasse `Instance`

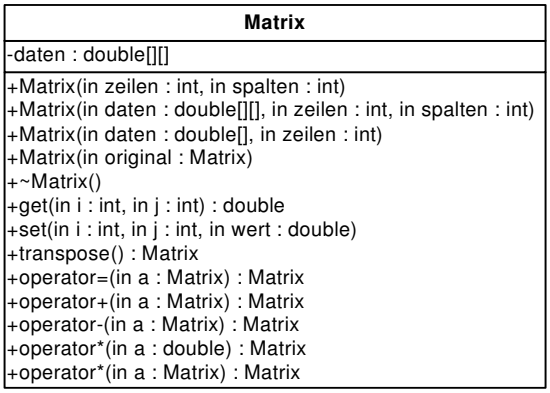



Abbildung 3.18: UML-Diagramm der Klasse `Matrix`

`FeatureExtractor` alle benötigten Merkmale enthält, indem die Merkmale des Klassifizierers mit `GetFeatures` ausgelesen und dann mit den durch die `GetFeatureValue`-Methode des FeatureExtractors zurückgegebenen Werten abgeglichen werden. Sind alle Werte vorhanden, so werden sie in ein internes Datenfeld kopiert. Andernfalls liefert die Methode `False` zurück und der Klassifizierer wird übersprungen (vgl. Abschnitt 3.7.1).

Bei der zweiten Datenstruktur (s. Abb. 3.18) handelt es sich um eine Matrix-Implementierung. Da einige Klassifizierungsverfahren wie der *MultiGaussClassifier* extensive Berechnungen mit Matrizen durchführen, ist eine effiziente Grundlage hier sinnvoll. Die entwickelte Klasse kann eine Matrix basierend auf einem ein- oder zweidimensionalen Datenfeld erstellen, mit einer leeren Matrix beginnen oder eine vorhandene Matrix kopieren. Neben den Auslesen und Setzen einzelner Zellen mit get/set werden bisher die am häufigsten verwendeten Operationen unterstützt: Matrix-Addition bzw. -Subtraktion, Matrix-Multiplikation, Skalarmultiplikation und Transponierung. Um die Klasse nicht zu „überladen“ wurde vorerst auf weitere Funktionen verzichtet. Diese können jedoch im Bedarfsfall leicht ergänzt werden.

Bei der Gestaltung von Hilfsklassen müssen grundsätzlich die Anforderungen des Gesamtprojekts beachtet werden, also insbesondere im Hinblick auf Performanz und Komplexität. Dies schlägt sich beispielsweise darin nieder, dass nur die wirklich benötigten

Methoden implementiert werden, anstatt – wie in einer typischen Klassenbibliothek – möglichst umfangreiche Funktionalität für alle denkbaren Anwendungen bereitzustellen. Auch ist keine Klassenhierarchie oder Benennungskonvention erforderlich, da auf die Hilfsklassen in der Regel nicht von außerhalb des Moduls zugegriffen wird.

### 3.7.6 Zweite Ebene

Für die Implementierung der zweiten Ebene, wie durch AGENDER beschrieben, ist ein dynamisches Bayessches Netz erforderlich. Ein solches Netz kann nach einer Reihe von Äußerungen schon bei einer geringen Anzahl von Klassifizierern eine beträchtliche Größe erreichen. Da bei jeder Klassifizierung ein neuer Satz von Evidenz hinzukommt und eine Verrechnung mit den bestehenden Daten erforderlich macht, bevor das Ergebnis der Klassifizierung feststeht, handelt es sich um einen zeitkritischen Vorgang. Es wurde also Wert darauf gelegt, dass eine performante Implementierung in das Klassifikationsmodul integriert wird, die jedoch gleichzeitig die Flexibilität bietet, um – analog zum Export-Vorgang bei den Klassifizierern – für jedes Modul eine eigene Netzstruktur verwenden zu können.

Zum Einsatz kommt dazu ein Compiler für dynamische Bayessche Netze von Brandherm mit der Bezeichnung *JavaDBN*, welcher aus Zeitscheibenschemata Quellcode in *C++* und *Java* erzeugen kann (vgl. Brandherm und Jameson, 2004). Die *C++*-Implementierung ist sehr effizient, z.B. durch Zusammenfassen der Daten mehrerer Zeitscheiben zu einer einzigen (*Rollup*, vgl. ebd.), und lässt sich unter Verwendung einer Wrapper-Klasse, die im Folgenden näher beschrieben wird, in das Klassifikationsmodul integrieren. Gegenwärtig muss der *C++*-Code für das Bayessche Netz noch vor der Erzeugung des Klassifikationsmoduls separat generiert und bereitgestellt werden. Für diese Arbeit wurden dazu die in der Applikation *Hugin Expert* erstellten Netze exportiert und von *JavaDBN* verarbeitet. Zukünftig soll auch dieser Teil des Workflows in den vollautomatischen Build-Vorgang mit einbezogen werden.

Jeder Entwurf eines Bayesschen Netzes für AGENDER wird mit einem Namen eindeutig gekennzeichnet, z.B. `secondLayer2005-12-14`. Der daraus erzeugte *C++*-Code enthält eine Klasse mit dem gleichen Namen. Der Code wird beim Build-Vorgang direkt in das Klassifikationsmodul integriert, wobei der Name der Klasse für das verwendete Netz in der Modul-Konfiguration gespeichert wird (vgl. Abschnitt 3.7.7). Der Zugriff auf die zweite Ebene erfolgt jeweils über die Klasse `BayesNet` (s. Abb. 3.19). Das lokale Feld `dbn_internal` enthält eine Referenz auf die eigentliche Datenstruktur und wird im Konstruktor erstellt. Die Wrapper-Klasse `BayesNet` bietet einerseits einen einheitlichen Zugriff auf die dynamische Schnittstelle des Bayesschen Netzes, andererseits soll sie auch sicherstellen, dass kein inkonsistenter Zustand erreicht wird, indem z.B. kein Rollup ausgeführt wird oder die Zustände von Knoten mit unbekannten Werten nicht

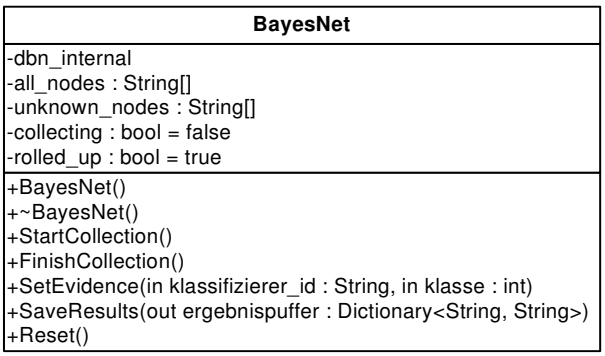



Abbildung 3.19: UML-Diagramm der Klasse `BayesNet`

aktualisiert werden.

Das Feld `all_nodes` wird einmalig festgelegt und enthält danach eine Liste aller Knoten des Netzes. Dies sollte bei einem konsistenten Entwurf des Moduls der Liste der IDs der im Modul enthaltenen Klassifizierer entsprechen. Die Variablen `collecting` und `rolled_up` werden in Sicherheitsklauseln verwendet, um die Ausführung von Methoden im falschen Zustand des Netzes zu verhindern. Sie enthalten den Wert `True`, wenn die Evidenzsammlung gestartet bzw. nachdem der Rollup durchgeführt wurde.

Die möglichen Zustände der Klasse `BayesNet` sind in Abb. 3.20 in Form eines *UML-Zustandsdiagramms* dargestellt. Nach der Konstruktion der Klasse ist `rolled_up` auf `True` eingestellt, da die Zeitscheibe 0 standardmäßig verfügbar ist. Die Sammlung von Evidenz, d.h. Ergebnissen von Klassifizierern, wird durch den Aufruf von `StartCollection` eingeleitet. Dies geschieht bei jeder Klassifizierungsanfrage in der Verarbeitungs-Pipeline, nachdem die Merkmalsextraktion erfolgt ist, sofern die zweite Ebene verwendet wird. Sind bereits Ergebnisse einer vorigen Äußerung im Netz gespeichert, so wird zuerst ein Rollup durchgeführt, d.h. es wird eine neue Zeitscheibe eingeführt. Das lokale Feld `unknown_nodes` wird anschließend mit der Liste aller Knoten initialisiert, d.h. es hat zu diesem Zeitpunkt identischen Inhalt wie das Feld `all_nodes`, von dem die Liste übertragen wird. Das Netz befindet sich nun im Zustand *collecting*, daher wird `collecting` der Wert `True` zugewiesen. In diesem Zustand wird für jedes neue Ergebnis eines Klassifizierers die Methode `SetEvidence` aufgerufen. Diese Methode legt – je nach Ergebnisklasse – den internen Status des Knotens im Netz für die aktuelle Zeitscheibe fest und entfernt den Knoten aus der Liste `unknown_nodes`. Wurden alle Klassifiziererergebnisse eingetragen, so wird durch `FinishCollection` die Evidenzsammlung beendet. Hierbei werden zuerst für alle fehlenden Ergebnisse – bestimmt anhand der Liste `unknown_nodes` – die zugehörigen Knoten auf den Status *unbekannt* eingestellt. Anschließend wird die Inferenz im Netz durchgeführt. Nach diesem Vorgang enthalten die Knoten, welche die Sprechereigenschaften modellieren, die gesuchten Ergebnisse. Die Variablen `collecting`

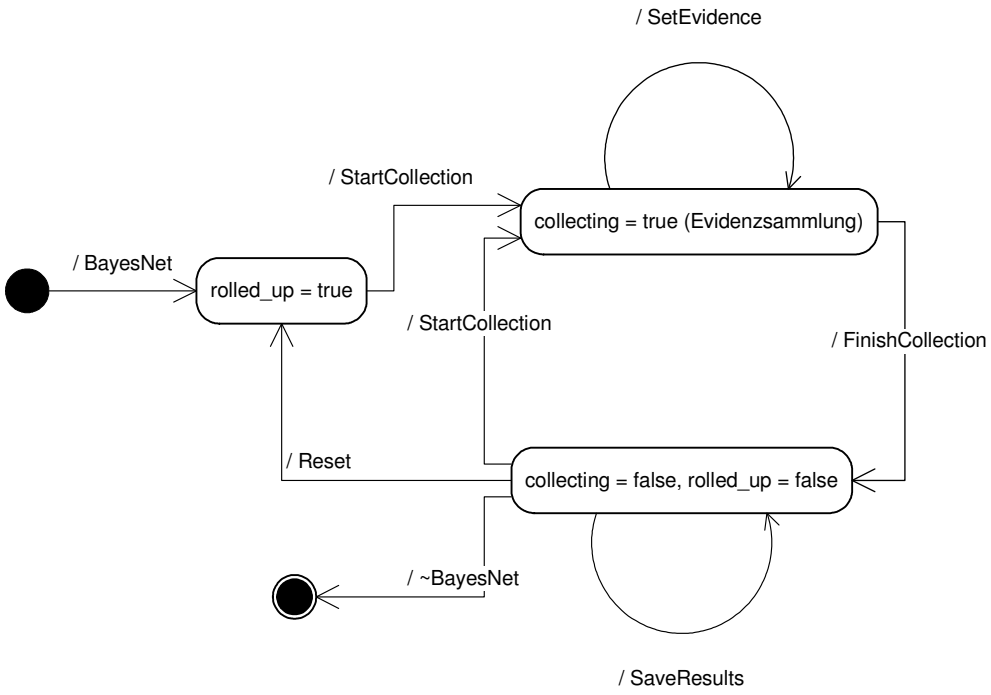



Abbildung 3.20: Zustandsdiagramm für die Klasse `BayesNet`

und `rolled_up` werden nun beide auf `False` gesetzt, so dass beim nächsten Aufruf von `StartCollection` ein Rollup durchgeführt wird.

Zum Auslesen der Ergebnisse aus dem Netz wird die Methode `SaveResults` bereitgestellt. Diese kann nur aufgerufen werden, wenn sich das Netz nicht im Zustand *collecting* befindet. Sie liest für jeden Knoten im Netz, der eine Eigenschaft des Sprechers wie Alter oder Geschlecht darstellt, die Wahrscheinlichkeiten aller möglichen Zustände aus, welche den Klassen der jeweiligen Eigenschaft entsprechen. Die Klasse mit der größten Wahrscheinlichkeit wird zusammen mit dem Profilpfad und der Angabe dieser Wahrscheinlichkeit in einem neuen Zeichenfolgenwörterbuch gespeichert.

Neben den beschriebenen Funktionen kann das dynamische Bayessche Netz auch komplett zurückgesetzt werden, indem die Methode `Reset` aufgerufen wird. Dieser Aufruf kommt z.B. bei einem Sprecherwechsel zum Einsatz.

### 3.7.7 Konfigurationsbeschreibung

Jedes Klassifikationsmodul enthält neben dem Basiscode (vgl. Abschnitt 3.3.4) einen dynamischen Anteil am Code, welcher beim Build-Vorgang erzeugt und in das Projekt des Klassifikationsmoduls eingebaut wird. Dieser Teil enthält die anwendungs- und kontextspezifischen Komponenten, nämlich die Klassifizierer und das Bayessche Netz, jedoch

nicht die Schnittstellenklassen `RuntimeClassifier` und `BayesNet`. Daneben existiert noch eine Quellcode-Datei mit der so genannten *Modul-Konfigurationsbeschreibung*, die Informationen darüber enthält, welche Klassen tatsächlich in der jeweils verwendeten Version des Moduls enthalten sind, sowie weitere Einstellungen, z.B. ob die zweite Ebene verwendet werden soll. Diese Datei wird als eine der ersten Header-Dateien vom Compiler verarbeitet, da sie einige Präprozessor-Direktiven enthält, die sich auf die Kompilierung der weiteren Dateien auswirkt, so wie die eben angesprochene Deaktivierung der zweiten Ebene.

*C++* bietet keine Möglichkeit, zur Laufzeit eine Liste der verfügbaren Klassen zu erhalten oder ein Objekt aus einem Klassennamen zu erstellen, was bei anderen Hochsprachen mittels *Reflektion* durchführbar ist. Um eine Instanz jedes verfügbaren Klassifizierers zu erstellen, wird eine andere Methode angewandt: Beim Build-Vorgang wird in der Konfigurationsbeschreibung die Konstante `MODULE_CLASSIFIERS` auf die Anzahl der Klassifizierer im Modul festgelegt und daneben eine globale Funktion `CreateModuleClassifier` angelegt, die als Argument einen Index annimmt und für jeden solchen Index einen anderen `RuntimeClassifier` zurückliefert. Somit kann also die genannte Funktion mit zunehmendem Index so oft wie in `MODULE_CLASSIFIERS` angegeben aufgerufen werden, um alle Klassifizierer zu erstellen.

Das Erstellen der internen Klasse für das dynamische Bayessche Netz ist weniger kompliziert: Die Konstante `BAYES_NET_CLASS` enthält den Klassennamen für das Netz und kann direkt zusammen mit dem Schlüsselwort `new` verwendet werden, um selbiges zu konstruieren. Dennoch sind auch hier weitere dynamische globale Funktionen erforderlich, um die Kommunikation mit dem Objekt zu ermöglichen; aufgrund dieser Eigenschaft werden sie auch als *Proxy-Funktionen* bezeichnet. Diese Funktionen werden von den Methoden der `BayesNet`-Klasse aufgerufen. Die Funktion `GetAllBayesNetNodes` füllt eine Liste mit den Namen aller Klassifizierer-Knoten, die im Netz enthalten sind. Die Namen der Knoten müssen den IDs der Klassifizierer der ersten Ebene entsprechen, damit die Ergebnisse zugeordnet werden können. Dies erfolgt durch die Funktion `SetBayesNetNode`, welche im Grunde genommen eine Abbildung

$$\textit{(Klassifizierer-ID, Klassenindex)} \rightarrow \textit{(Knotenname, Zustandsindex)}$$

darstellt. Alle Paare von Knotennamen und Zustandsindex werden jeweils durch eine eigene parameterlose Methode repräsentiert[12], die über die Proxy-Methode aufgerufen werden müssen, da jedes Bayessche Netz offensichtlich einen eigenen Satz von Methoden enthält. Die Funktion `SaveBayesNetResults`, die von der gleichnamigen Methode der Klasse `BayesNet` aufgerufen wird, führt ebenfalls ein Mapping durch: Hier werden

[12] Dieses Konzept ist vorgegeben durch den *JavaDBN*. Wenn das DBN der Applikation im Voraus bekannt ist, kann dadurch die Geschwindigkeit verbessert werden.

| Funktion | Beschreibung |
|---|---|
| `dlog` | Schreibt eine als Argument angegebene Zeichenfolge zusammen mit einem Zeitstempel als vollständige Zeile in die Protokolldatei. Beispiel: `dlog("Hallo Welt!");` |
| `dlogx` | Schreibt eine als Argument angegebene `String`-Variable in die Protokolldatei. |
| `dlogp` | Schreibt eine als Argument angegebene Zeichenfolge in die Protokolldatei. Eignet sich zur Kombination mit `dlogx` und `dlogn`. |
| `dlogn` | Schreibt einen Zeilenumbruch in die Protokolldatei. Gedacht zur Verwendung mit allen Tracing-Funktionen außer `dlog`. |
| `dlogi` | Schreibt eine als Argument angegebene `int`-Variable in die Protokolldatei. |
| `dlogf` | Schreibt eine als Argument angegebene `float`- oder `double`-Variable in die Protokolldatei. |

Tabelle 3.4: Tracing-Funktionen des Klassifikationsmoduls

die Zustände der Knoten für die Sprechereigenschaften auf Klassenbezeichnungen und die Knoten selbst auf Profilpfade abgebildet. Dies ist erforderlich, da weder die Eigenschaften wie *Alter* und *Geschlecht*, noch die Klassen wie z.B. *Kinder*, *Erwachsene* und *Senioren* statisch in das Modul einprogrammiert sind.

### 3.7.8 Tracing

Jede Anwendung oder Komponente sollte neben den zentralen Funktionen auch über einen *Tracing-Mechanismus* (auch als *Ablaufverfolgung* bezeichnet) verfügen. Dabei handelt es sich um einen Satz von Routinen, mit denen Nachrichten auf dem Bildschirm ausgegeben oder in eine Protokolldatei geschrieben werden, und welche zu Debugging-Zwecken verwendet werden können. In der Regel werden solche Nachrichten am Anfang und Ende einer Prozedur generiert, sowie vor wichtigen Operationen oder um eine Debug-Ausgabe (*Dump*) von wichtigen Daten zu erstellen. Mit Hilfe der Tracing-Ausgaben lässt sich der Ablauf eines Programms einfacher nachvollziehen und im Falle eines Fehlers dessen Ursprung leichter finden.

Da es sich bei dem Klassifikationsmodul um eine Komponente handelt, die keinen Zugriff auf Anzeigegeräte wie den Bildschirm besitzt und auch keine Konsolenausgabe erzeugt, werden alle Tracing-Ausgaben in eine Protokolldatei geschrieben. Diese kann nach Beendigung der Applikation und eingeschränkt auch während ihrer Laufzeit begutachtet werden. Alle verfügbaren Tracing-Funktionen sind in Tabelle 3.4 zusammen mit ihrer Arbeitsweise aufgelistet.

Da die Tracing-Funktionen zusätzliche Ausführungsschritte im Programmablauf be-

wirken, kann sich ihre Verwendung mitunter spürbar auf die Geschwindigkeit auswirken, besonders dann, wenn eine sehr häufig aufgerufene Funktion bei jedem Aufruf eine Trace-Meldung erzeugt. Aus solchen Überlegungen heraus wurde die Möglichkeit geschaffen, eine Version des Klassifikationsmoduls wahlweise mit oder ohne Tracing-Funktionalität zu kompilieren. Dabei ist zu beachten, dass es sich bei Versionen mit Tracing nicht automatisch um Entwicklungs- oder Testversionen handelt. Tracing kann auch in Release-Versionen vom Entwickler der Host-Applikation z.B. dazu verwendet werden, Fehler bei der Bedienung der Komponente oder in den Eingabedaten aufzudecken. Auch kann bei schwer auffindbaren Fehlern die Protokolldatei aus einem Real-World-Szenario an die Entwickler weitergeleitet werden, da die Fehler unter Laborbedingungen oft nicht reproduzierbar sind. Da es nicht praktikabel wäre, jeden Aufruf einer Tracing-Funktion mit einer Präprozessor-Klausel (`#ifdef ... #endif`) zu umschließen, wurde eine bei *C*-Programmierern beliebte Technik angewandt: Für jede der betroffenen Funktionen wurde ein so genanntes *Makro* definiert, welches bei aktiviertem Tracing ein Alias für die Funktion darstellt. Statt die Tracing-Funktionen direkt aufzurufen wird im Code der Name des Makros verwendet. Um eine Version ohne Tracing zu erzeugen, müssen dann nur die Makros so umdefiniert werden, dass sie den Wert `((void) 0)` haben. Dies bewirkt, dass alle Tracing-Aufrufe vom Compiler ignoriert werden[13].

Das SBC-Klassifikationsmodul hält im Tracing-Protokoll alle Zustände innerhalb der Verarbeitungs-Pipeline fest. Es lässt sich nachvollziehen, mit welchen Werten die Sprachmerkmale extrahiert wurden, welche Klassifizierer ausgeführt wurden und welche Ergebnisse von ihnen im Einzelnen klassifiziert wurden sowie welchen Inhalt die Knoten auf der zweiten Ebene aufwiesen. Darüber hinaus wurde ein zusätzlicher Test aller Merkmalswerte integriert: Dabei wird für alle extrahierten Werte in der Protokolldatei dokumentiert, ob sie im typischen Wertebereich für das Merkmal liegen, wobei dieser Bereich vom Mittelwert aus eine Abweichung nach oben und unten um ein (konfigurierbares) Vielfaches der Standardabweichung beträgt. Die Ausgabe erfolgt nach Klassen getrennt in tabellarischer Form, aus der hervorgeht, wie viele Merkmale für jede der Klassen innerhalb oder außerhalb des Bereichs liegen. Die Bedeutung dieser Daten liegt darin, dass sie schnell Aufschluss über die grundsätzliche Plausibilität der Werte geben, was besonders dann hilfreich ist, wenn ein Klassifizierer ein falsches Ergebnis geliefert hat. Anhand der Tracing-Ausgaben kann dann festgestellt werden, ob es sich um ein Problem des Klassifizierers handelt, oder ob die Werte tatsächlich außerhalb der typischen Werte für die korrekte Klasse liegen. Im Anfangsstadium des SBC-Projekts wurde diese Methode auch zum Testen der Klassifizierer eingesetzt. Inzwischen haben verbesserte Evaluierungsmethoden, die in die M3I CAT Software integriert und in Abb. 3.21

[13] Beim Kompilieren ersetzt `((void) 0)` den Funktionsaufruf. `((void) 0)` bezeichnet den Aufruf „keiner" Funktion und hat also keine Wirkung. Die Funktionsparameter werden im Zuge der Optimierung durch den Compiler ebenfalls nicht ausgewertet.

dargestellt sind, diese Aufgabe übernommen. Zur Fehlersuche „vor Ort“ bleibt Tracing dennoch die einzige Möglichkeit, und hier kann der Plausibilitäts-Test auch in Zukunft wertvolle Dienste leisten.

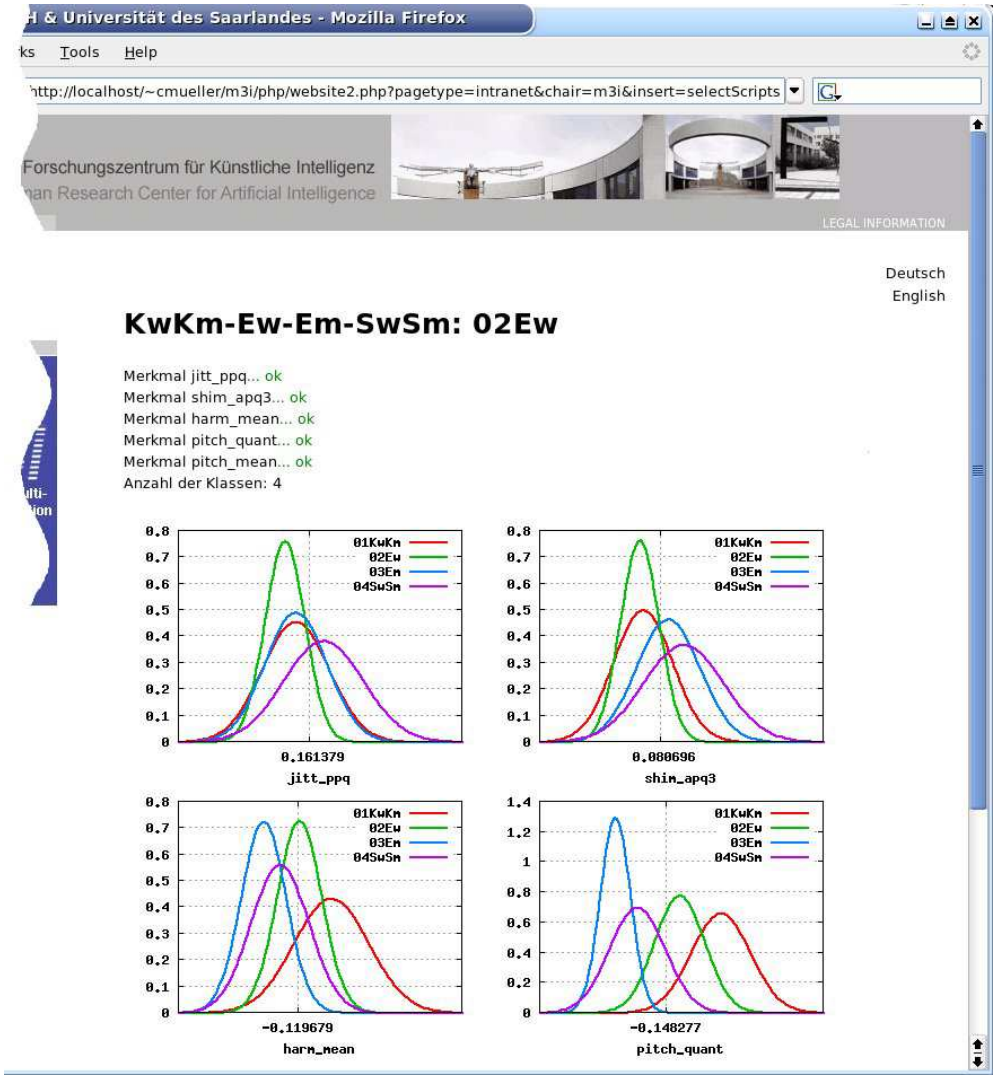



Abbildung 3.21: Plausibilitätsanalyse eines Klassifikationsergebnisses nach Merkmalen in M3I CAT. Jede Kurve stellt die Normalverteilung für eine einzelne Klasse dar. Die vertikale gepunktete Linie gibt den gemessenen Wert an.

## 3.8 Integration in Applikationen

Die Integration des Klassifikationsmoduls in eine Host-Anwendung erfolgt auf drei Ebenen: Auf konzeptueller Ebene, auf Kommunikationsebene und auf Plattformebene.

### 3.8.1 Konzeptebene

Auf konzeptueller Ebene werden allgemeine Überlegungen angestellt, wie die Dienste der Sprecherklassifikation in der Anwendung genutzt werden. Ein Programm, welches die Sprecherklassifikation zur Unterstützung der primären Aufgabe verwendet, z.B. zur Benutzeradaption, muss entscheiden, wann die Adaption durchgeführt werden soll, welche Teile der Spracheingabe für die Klassifikation genutzt werden. Auch die Frage, wie mit einem unsicheren Ergebnis umzugehen ist, d.h. mit sehr geringen Konfidenzwerten, ist von der Applikation zu beantworten. Für eine skalierbare serverbasierte Lösung wäre weiterhin zu klären, wie viele Sprecher parallel verarbeitet werden sollen, was auf die Frage der Anzahl von Threads und Verarbeitungs-Pipelines hinausläuft (vgl. Abschnitt 3.7.1). Die Liste der Beispiele lässt sich noch weiter fortführen. Alle genannten Überlegungen sind abhängig von der Architektur der Host-Anwendung und müssen daher von den Anwendungsarchitekten in Übereinstimmung mit den Entwicklern der Integrationsschicht seitens der Anwendung durchgeführt werden.

### 3.8.2 Kommunikationsebene

Die Kommunikationsebene betrifft sowohl die Entwickler der Anwendung, als auch den Entwurf des SBC-Klassifikationsmoduls. Von Seiten des Moduls muss eine Schnittstelle geschaffen werden, über die sich alle Funktionen ansprechen und Daten effizient und komfortabel austauschen lassen. Aufgabe der Anwendung ist dann die zweckmäßige Verwendung der Schnittstelle, die meist durch eine eigene Integrationsschicht oder Wrapper-Klasse gekapselt wird. Dies dient auch dazu, von der zugrunde liegenden Technologie zu abstrahieren, welche eher der Plattformebene zuzuordnen ist und ausgetauscht werden kann, ohne die Schnittstelle oder das Protokoll zu betreffen. Ebenfalls Bestandteil der Schnittstellenspezifikation ist das Format der übertragenen Daten wie z.B. des Audiosignals.

Ein Ziel beim Entwurf der Schnittstelle für das Klassifikationsmodul war die möglichst einfache Gestaltung der Funktionen. Umfangreiche Initialisierungsaufrufe und Handles sollten vermieden werden; stattdessen wird ein durchgängig objektorientierter Ansatz angeboten.

Zur Nutzung der Funktionen des Klassifikationsmoduls muss lediglich ein Verweis auf die Datei `ProcessingCore.h` über eine C-Inklusionsanweisung (`#include`) in den Quellcode der Anwendung eingefügt werden. Alle weiteren Dateien sind über Verweise

in `ProcessingCore.h` bereits automatisch eingebunden. Die Teile des SBC-Quellcodes, welche die Schnittstellenbeschreibung enthalten, werden für die syntaktische Analyse beim Kompilieren benötigt.

In einem typischen Szenario wird die Schnittstelle zur Anwendung vollständig durch die Klasse `ProcessingPipeline` gekapselt. Durch Erstellen einer Instanz dieser Klasse wird das Modul initialisiert und durch Zerstörung der letzten Instanz bereinigt. Sprecherklassifikation und Zugriff auf das aktuelle Benutzermodell wird durch die Methoden dieses Objekts bereitgestellt. Die Kernfunktionalität des Klassifikationsmoduls lässt sich bereits mit zwei Methoden erschließen: `ProcessRaw` und `GetResults`. Die weiteren Methoden dieses Objekts haben eher ergänzenden Charakter. Zwei mögliche Kommunikationsabläufe mit einer Anwendung sind in Abb. 3.22 und Abb. 3.23 als UML-Sequenzdiagramm dargestellt. In Abb. 3.22 ist die minimal erforderliche Schnittstellenkommunikation aufgezeigt, um eine Klassifizierung durchzuführen (vgl. auch Abb. 3.2). Abb. 3.23 hingegen zeigt drei fortgeschrittene Vorgehensweisen, die in einer Server-Applikation Anwendung finden könnten: Es werden mehrere Klassifizierungen simultan durchgeführt, ein Sprecherwechsel tritt auf, und das verzögerte Abfragen der Ergebnisse an festen Checkpoints in der Applikation wird demonstriert.

Die Audiodaten, welche über die Schnittstelle an das Modul übertragen werden, können eines der Dateiformate WAVE, *Audio Interchange File Format* (AIFF), *NeXT/Sun* oder *National Institute Of Standards And Technology* (NIST) besitzen. Dabei handelt es sich um die gängigsten Formate, welche die häufigsten Anwendungsszenarien abdecken sollen. Die jeweiligen Routinen zum Lesen der Formate sind Bestandteil der *Praat*-Bibliothek. Im Falle des Raw-PCM-Formats muss statt der Funktion `ProcessWave` wie unter Abschnitt 3.7.1 beschrieben `ProcessRaw` mit den jeweiligen Parametern aufgerufen werden; abgesehen davon ist die Vorgehensweise bei allen Formaten identisch.

Neben den Audiodaten werden auch die Klassifikationsergebnisse über die Schnittstelle übertragen, in diesem Fall vom Modul zur Anwendung. Als Container wird ein *Zeichenfolgenwörterbuch* eingesetzt, welches Schlüsseln[14] Werte zuweist. Dabei ist es wichtig, ein einheitliches Format zu verwenden, um die Kompatibilität auch mit zukünftigen Versionen des Klassifikationsmoduls zu gewährleisten, welche möglicherweise noch zusätzliche Informationen bereitstellen können. Als Grundregel bei der Verarbeitung des Wörterbuchs gilt daher, dass unbekannte Schlüssel ignoriert werden müssen, da sie Bestandteil einer solchen Weiterentwicklung sein können. Die aktuelle Version verwendet zwei Arten von Schlüssel-Wert-Paaren:

1. Klassenergebnisse der Form `<Profilpfad> = <Klasse>` und
2. Wahrscheinlichkeitsergebnisse der Form `<Profilpfad>_prob = <Wahrscheinlichkeit>` .

[14] Schlüssel sind per Definition eindeutig.

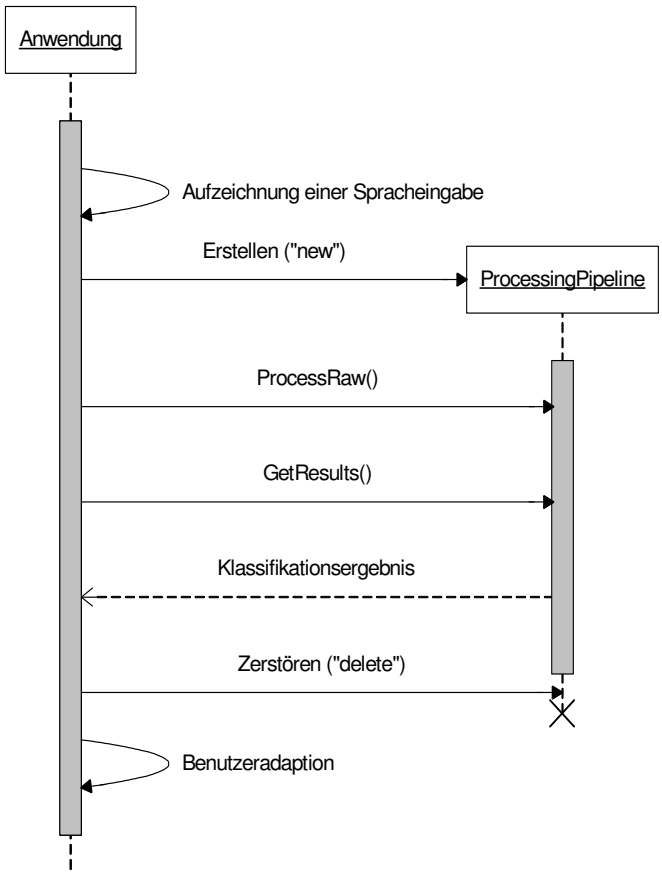



Abbildung 3.22: Sequenzdiagramm für eine einfache eingebettete Klassifikation mit dem SBC-Modul

Die Auswahl der Ergebnisse, sowie die Profilpfade und Klassennamen hängen von der Konfiguration des jeweils vorliegenden Moduls ab. Wahrscheinlichkeiten werden in internationaler Notation (mit Punkt als Dezimaltrennzeichen) angegeben.

Obgleich die Profilpfade vom Entwickler des Moduls frei gewählt werden können, spricht einiges dafür, nach Möglichkeit einem gemeinsamen Standard zu folgen. Einige Arbeiten im Bereich der Benutzermodellierung beschäftigen sich mit dieser Thematik und schlagen Konventionen zur Benennung von Benutzereigenschaften wie Alter und Geschlecht vor. So entwickelt Heckmann (2005) neben einer umfangreichen Ontologie[15] für Benutzermodelle im Sinne des *ubiquitous computing* (vgl. Weiser, 1991) auch Metadaten, die mit den Eigenschaften verbunden sind, z.B. über Dauer der Speicherung und Vertraulichkeit. Als Profilpfad würden in diesem Fall die so genannten *Ubis Identifier*

[15] *General User Model Ontology* (GUMO)

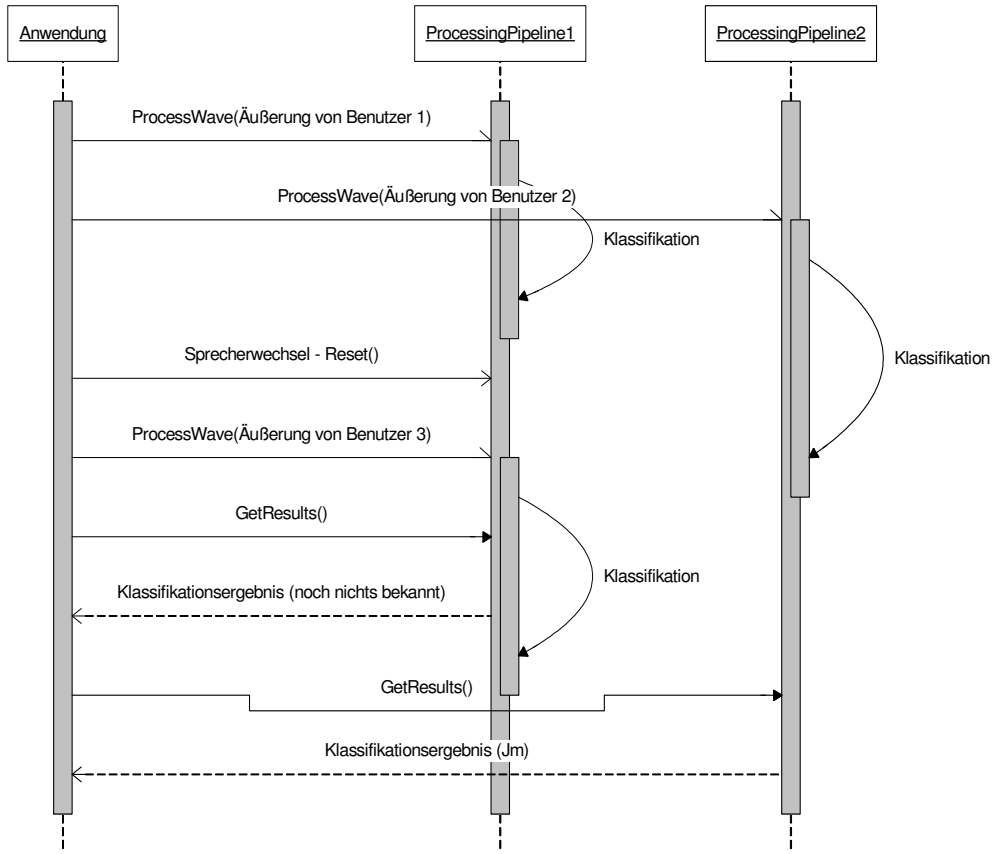



Abbildung 3.23: Sequenzdiagramm für die Verwendung des Klassifikationsmoduls in einem Server mit mehreren Benutzern

(vgl. Heckmann, Schwartz, Brandherm und von Wilamowitz-Moellendorff, 2005) dienen. Indem die Applikation dieser oder einer vergleichbaren Konvention folgt, ist eine gute Grundlage für die Austauschbarkeit der Informationen mit anderen Anwendungen oder Diensten wie *UbisWorld*[16] gegeben.

Beim Kompilieren der Anwendung ist die Bibliotheksdatei des Klassifikationsmoduls (standardmäßig `ProcessingCore.lib`) erforderlich. Deren Speicherort kann dem Compiler über einen Befehlszeilenparameter oder einen Verweis in der Entwicklungsumgebung mitgeteilt werden. In einigen Situationen kann der Workflow verbessert werden, indem das Erstellen der Anwendung in den Build-Vorgang des Moduls einbezogen wird. So lässt sich durch einen einzigen Befehl das Klassifikationsmodul mit den aktuellen Klassifizierern zusammen mit der Applikation erstellen und unmittelbar in diese integrieren.

[16] `http://www.u2m.org`

### 3.8.3 Plattformebene

Die Integration auf Plattformebene bestimmt, wie Anwendung und Modul aus Sicht des Betriebssystems miteinander verbunden sind. Bedingung ist, dass beide Programmteile effizient miteinander kommunizieren können. Nach den in Abschnitt 3.1.6 beschriebenen Gesichtspunkten kommt dafür nur eine Implementierung in Form einer Bibliothek in Frage. In Abschnitt 3.5 wurden bereits zwei Arten von Bibliotheken vorgestellt. SBC verwendet eine statische Bibliothek für das Klassifikationsmodul, was zur Konsequenz hat, dass als unmittelbare Integrationstechnologie nur die Einbindung in den *C++*-Quellcode der Anwendung in Frage kommt. Ist die Anwendung jedoch in einer anderen Sprache wie *Java* geschrieben, so lässt sich in den meisten Fällen eine weitere bidirektionale Schnittstelle zwischenschalten, welche auf der einen Seite eine direkte *C++*-Anbindung an das SBC-Klassifikationsmodul herstellt und auf der anderen Seite eine von der Anwendung unterstützte Schnittstelle anbietet. Dabei ist allerdings ein gewisser Geschwindigkeitsnachteil gegenüber einer homogenen *C++*-Implementierung kaum zu vermeiden, da die Daten zwischen verschiedenen Sprachen konvertiert werden müssen. Der Vorgang des Weiterleitens von Daten und Aufrufen wird in diesem Fall auch als *Marshalling* oder *Pickling* (vgl. Tack, Kornstaedt und Smolka, 2005) bezeichnet. Ein Beispiel für eine solche Implementierung ist in Abschnitt 4.2 zu finden.

## 3.9 Development Platform

Neben dem Klassifikationsmodul stellt die Entwicklungsplattform die zweite Säule der SBC-Architektur dar. Es handelt sich dabei um eine eigenständige Anwendung mit der Bezeichnung SBC DEVELOPMENT PLATFORM. Sie wird in der Entwicklungsphase der Anwendung verwendet, aber nicht zusammen mit dieser ausgeliefert. Ferner unterstützt sie auch die Weiterentwicklung der SBC-Architektur selbst, einschließlich AGENDER. Aus diesem Grund stehen auch andere Aspekte bei der Implementierung im Vordergrund als dies beim eingebetteten Modul der Fall ist. So ist der Performanzaspekt kein gewichtiger Faktor, wohingegen die gute Wartbarkeit und flexible Anpassungsmöglichkeiten an neue Verfahren eine größere Rolle spielen.

Für den Anwendungsentwickler besteht die Hauptfunktion der Umgebung darin, den Entwurf und die Konfiguration der Klassifikationsmodule je nach Szenario zu unterstützen. Dabei werden im Einzelnen Möglichkeiten zur Auswahl, Training, Evaluierung und Export von Klassifizierern bereitgestellt. Auf alle genannten Funktionen wird in den folgenden Abschnitten näher eingegangen.

Die DEVELOPMENT PLATFORM wurde unmittelbar aus dem M3I-Server (vgl. Müller, 2005, S. 207ff) heraus entwickelt, da dieser bereits einen Großteil der benötigten Funktionalität bietet. Teilweise handelt es sich dabei um eine Weiterentwicklung der bestehenden Codes, aber auch um grundlegende Optimierungen sowie Entfernung von nicht benötigter Funktionalität. Die Erweiterung bezieht sich hauptsächlich auf den Build-Vorgang, welcher es ermöglicht, aus einer Reihe von Konfigurationen für ein Klassifikationsmodul eine auszuwählen und automatisch alle modulspezifischen Dateien im benötigten Format zu exportieren. Die Trennung in Verarbeitungs-Einheit und GUI, welche in der M3I-Architektur über *Java RMI* miteinander in Verbindung treten, wurde aufgehoben und in einem Modul zusammengelegt. Die Live-Klassifikation, d.h. die Klassifizierung einer über ein Mikrofon aufgezeichneten Äußerung inkl. Merkmalsextraktion und Cluster-Verarbeitung, steht in der DEVELOPMENT PLATFORM nicht zur Verfügung. Stattdessen wurden neue Möglichkeiten zur Evaluierung von Klassifizierern geschaffen. Zudem ist keine Möglichkeit zur Kommunikation mit Clients über ein Netzwerk vorhandenen. Diese Aufgabe, einschließlich der serverbasierten Klassifikation von Äußerungen, übernimmt in der SBC-Architektur der SBC-Server (vgl. Abschnitt 4.1.3).

Die DEVELOPMENT PLATFORM ist wie ihr Vorgänger in *Java* programmiert. Ein Vorteil der DEVELOPMENT PLATFORM besteht allerdings darin, dass keine externen Programme oder Befehle aufgerufen werden müssen, so dass grundsätzlich alle Voraussetzungen für eine echte Plattformunabhängigkeit geschaffen sind. Die vorliegende Implementierung wurde allerdings bisher nur unter *Windows* getestet.

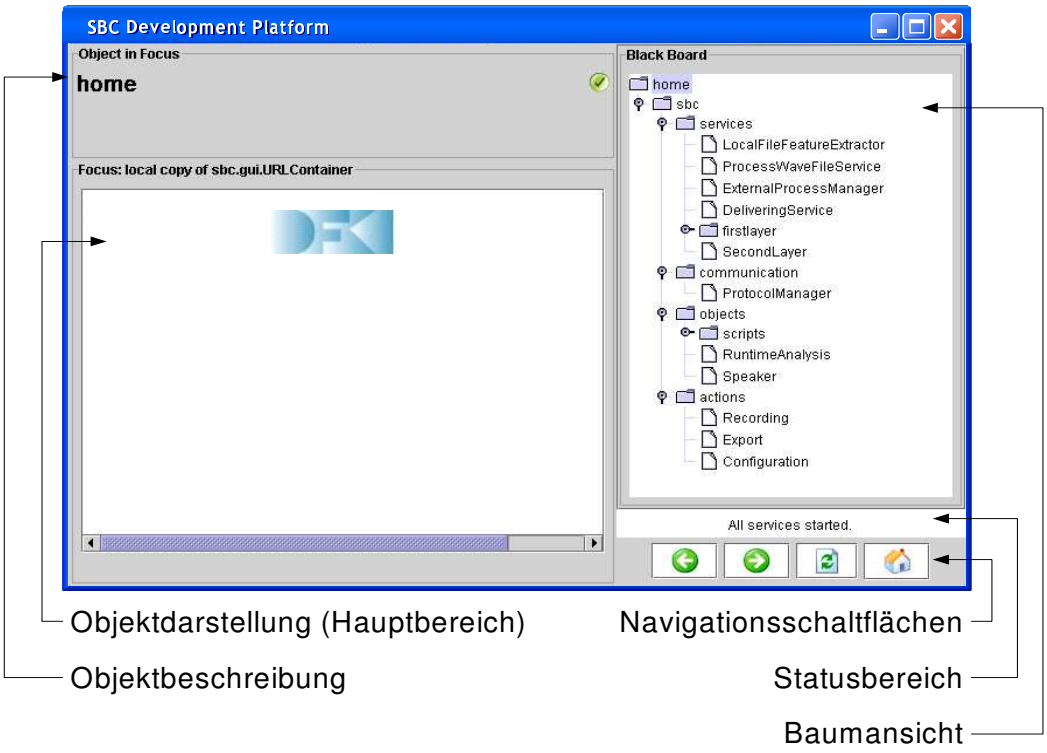



Abbildung 3.24: Bestandteile der Oberfläche von SBC DEVELOPMENT PLATFORM und M3I-Server

### 3.9.1 Allgemeine Bedienung

Für die Bedienung der DEVELOPMENT PLATFORM stehen zwei Modi zur Verfügung: Der *GUI-Modus* und der *Konsolenmodus*. Die Auswahl eines Modus erfolgt über die Befehlszeile: Wird das Argument `console` angegeben, so wird das Programm im Konsolenmodus gestartet, andernfalls im GUI-Modus.

Der grundlegende Aufbau der Benutzeroberfläche im GUI-Modus unterscheidet sich nicht von der des M3I-Servers, soll jedoch der Vollständigkeit halber kurz erläutert werden. Für eine ausführlichere Referenz wird auf Müller (2005, S. 205ff) verwiesen.

Abb. 3.24 stellt die gemeinsame Ansicht der Benutzeroberfläche vor. Die Anwendung speichert alle geladenen Dienste und andere Objekte des Blackboards in einer hierarchischen Organisationsstruktur. Diese ist als Baumansicht in einer Seitenleiste rechts im Fenster dargestellt. Die einzelnen Ordner sind Knoten im Baum, die durch Doppelklicken erweitert werden können. Die Dokument-Symbole repräsentieren die eigentlichen Objekte, welche durch Auswahl mit der Maus im Hauptbereich visualisiert werden können. Dabei kann sich die Oberfläche, d.h. Darstellung und Steuerelemente, je nach Objekttyp

| Befehl | Beschreibung |
|---|---|
| `help` | Gibt die Liste der verfügbaren Befehle auf der Konsole aus. |
| `start` | Startet alle in der Konfigurationsdatei festgelegten Blackboard-Dienste. |
| `services` | Listet die derzeit geladenen Dienste auf der Konsole auf. |
| `load <Datei>` | Lädt eine Konfigurationsdatei (vgl. Abschnitt 3.9.3). |
| `export` | Startet den Exportvorgang für die geladene Modulkonfiguration (vgl. Abschnitt 3.9.7). |
| `exit` | Beendet die DEVELOPMENT PLATFORM. |

Tabelle 3.5: Verfügbare Befehle im Konsolenmodus der SBC DEVELOPMENT PLATFORM

unterscheiden. Es können auch mehrere Ansichten für ein Objekt verfügbar sein, zwischen denen über Registerkarten am oberen Rand des Hauptbereichs gewechselt werden kann. Unten rechts befindet sich ein Statusfeld, in dem Meldungen angezeigt werden, und darunter Navigationsschaltflächen analog zu denen in einem Webbrowser.

Über die GUI können alle Funktionen des Programms auf einfache Weise angewählt werden. Zwei Nachteile bestehen jedoch darin, dass die GUI sich nicht auf Terminals ausführen lässt und nicht gut automatisiert werden kann. Letzteres ist aber notwendig für eine möglichst automatisierte Erzeugung eines Klassifikationsmoduls. Aus diesem Grund wurde der Konsolenmodus neu eingeführt. Dabei handelt es sich um eine Eingabeaufforderung, an der einfache Befehle eingegeben werden können. Statusmeldungen und Ergebnisse werden vom Programm ebenfalls auf der Konsole ausgegeben. Die Liste der verfügbaren Befehle und deren Beschreibung kann in Tabelle 3.5 nachgeschlagen werden. Dabei handelt es um eine Untermenge der über den GUI-Modus anwählbaren Funktionen. Es ist zu beachten, dass der Konsolenmodus keine Visualisierungsmöglichkeiten bietet und daher zum Testen von Datenbasis und Algorithmen ungeeignet ist.

### 3.9.2 Datensichtung

Um mit Klassifizierern arbeiten zu können, ist ein umfangreicher Bestand an Daten erforderlich. Die Ausgangsbasis bilden ein oder mehrere Sprachkorpora mit einer großen Anzahl Äußerungen verschiedener Sprecher. Um aus diesen Sprechproben die Datenbasis zu erhalten, welche für das Training verwendet werden kann, sind jedoch noch zwei weitere Schritte notwendig: zum einen die Merkmalsextraktion und zum anderen eine Ausbalancierung der Daten (vgl. Abschnitt 2.3.2). Die daraus resultierenden Zahlenwerte können in vielen Fällen wieder verwendet werden: So ist eine Neubalancierung nur erforderlich, wenn sich die Klassen ändern, nicht aber bei einer Wahl eines anderen Klassifizierungsalgorithmus. Noch bedeutsamer ist dies für die Merkmalsextraktion. Da

| | shim_l | shim_apq3 | harm_stddev | pitch_mean | KwKmJw-JmEw-SwSm-Em |
|---|---|---|---|---|---|
| 1 | 0.258534405966789 | 0.769960885071339 | 0.505873645689773 | 0.44897573678622 | 01KwKmJw |
| 2 | -0.72867227380404 | -0.375883050537573 | -0.798200564211738 | 1.20933197303257 | 01KwKmJw |
| 3 | -0.805351780972872 | -0.935653840362812 | 0.488698721449728 | 2.06659194932486 | 01KwKmJw |
| 4 | -0.570099076230833 | -0.272029823534812 | 0.356801474131493 | 0.242822006397792 | 01KwKmJw |
| 5 | -0.448971227526972 | -0.300843234222372 | 1.58714604285264 | 0.707561089439186 | 01KwKmJw |
| 6 | -1.61756510996717 | -1.03938089488815 | 3.29492925728978 | 0.589508833682869 | 01KwKmJw |
| 7 | -0.687951970394372 | -0.588379919768676 | 0.767445912935989 | 0.628694792439141 | 01KwKmJw |
| 8 | -0.817181174614706 | -0.823437556608365 | 2.10131518363828 | 1.26170769386509 | 01KwKmJw |
| 9 | 0.247002099696321 | -0.173922532692444 | 1.32955888680685 | 0.815000500575925 | 01KwKmJw |
| 10 | 2.14324929235328 | -0.0410984859017862 | -1.75891283545212 | 0.556493670496287 | 01KwKmJw |
| 11 | -1.39201278680241 | -1.09485072168313 | -0.587399992491828 | 0.880835508981269 | 01KwKmJw |

Abbildung 3.25: Ausschnitt aus der Tabelle `training_sbc_KwKmJw-JmEw-SwSm-Em`

der Vorgang selbst auf schnellen Rechnern und paralleler Verarbeitung je nach Umfang des Korpus einige Stunden[17] in Anspruch nehmen kann, sollten die einmal erlangten Daten auf jeden Fall in einer Datenbank gespeichert werden. Wird bei der Merkmalsextraktion eine Obermenge der benötigten Merkmale und Sprecher verwendet, so ist eine nochmalige Extraktion nur nötig, wenn ein verbesserter Korpus zur Verfügung steht, neue Merkmale entdeckt wurden oder der Extraktionsalgorithmus geändert wurde, wovon die beiden zuletzt genannten Möglichkeiten während der Anwendungsentwicklung normalerweise nicht auftreten.

Aus diesem Grund wurde in der M3I-Architektur die Merkmalsextraktion aus dem Server ausgenommen. Stattdessen wird auf eine *MySQL*-Datenbank zugegriffen, welche lediglich die Merkmalswerte und die Klassenbezeichnungen enthält. Dies hat auch den weiteren Vorteil, dass die tatsächliche Klasse einer Sprachäußerung nicht über einen unintuitiven Mechanismus aus einer zusätzlichen Metadatei oder aus dem Dateinamen der Aufnahme selbst extrahiert werden muss. Überdies lassen sich die Datenbanken auch besser mit anderen Entwicklungsrechnern austauschen, da sie nur einen Bruchteil des Speicherplatzes eines vollständigen Sprachkorpus umfassen. Dieses Konzept wurde auch für die SBC DEVELOPMENT PLATFORM beibehalten. Die zugrunde liegende Datenbank muss einem bestimmten Format entsprechen, was die Benennung und das Layout der Tabellen betrifft, um verwendet werden zu können. Ein Ausschnitt aus einer solchen Tabelle ist in Abb. 3.25 dargestellt.

Der Tabellenname muss der Syntax

`training_<Anwendung>_<Klasse>`

entsprechen. Dabei steht <*Anwendung*> für einen Bezeichner, welcher den Anwendungskontext eindeutig kennzeichnet, so dass mehrere Klassifikationsprobleme parallel und unabhängig voneinander betrieben werden können. <*Klasse*> steht für die Bezeichnung der Klasse, welche durch einen Klassifizierer, dem die Tabelle zugrunde liegt, bestimmt

[17] Im Fall von AGENDER dauert die Analyse aller Merkmale des gesamten Korpus ca. 5 Stunden unter Verwendung eines Clusters mit 10 Knoten.

wird. Diese kann mit der ID des Klassifizierers übereinstimmen. Die Tabelle enthält eine Spalte für jedes Merkmal, das vom Klassifizierer benötigt wird, sowie eine Spalte für die tatsächliche Klasse.

Als Werkzeug zur Datensichtung kann M3I CAT verwendet werden (vgl. Müller, 2005, S. 229ff). Dazu gehört die Merkmalsextraktion auf dem gesamten Sprachkorpus, welche intern durch spezielle Skripte umgesetzt wird, die von der Befehlszeilenversion von *Praat* ausgeführt und auf mehrere Rechner verteilt werden. Die resultierende Tabelle enthält alle bekannten Merkmale aus allen Sprechproben. Mit einer weiteren Funktion von M3I CAT lassen sich daraus problemspezifische Tabellen erstellen, welche nur die benötigten Merkmale enthalten und jeweils die gleiche Anzahl von Datensätzen für jede Klasse aufweisen. Diese Tabelle kann ohne weitere Anpassung von der SBC DEVELOPMENT PLATFORM verwendet werden.

Um den Zugriff auf die Datenbanken aus der DEVELOPMENT PLATFORM heraus zu ermöglichen, müssen Rechnername, Datenbank, Benutzername und Passwort (mit Leseberechtigung) einmalig in einer Konfigurationsdatei im XML-Format gespeichert werden. Diese Datei wird beim Starten der Umgebung gelesen. Sie enthält noch weitere Einstellungen zur Anpassung der Arbeitsweise des Programms, z.B. diverse Pfadangaben.

### 3.9.3 Konfiguration von Klassifikationsmodulen

Um der Anforderung an die Modularität der Architektur Rechnung zu tragen, unterstützt die Entwicklungsumgebung die Erstellung von unterschiedlichen Klassifikationsmodulen je nach Anwendung. Die Module entscheiden sich voneinander durch die in ihnen enthaltenen Klassifizierer und das Bayessche Netz, also durch die Implementierung der ersten und zweiten Ebene von AGENDER, sowie ggf. durch weitere Parameter. Dies stellt zusammen genommen die Konfiguration eines Moduls dar. Bevor ein Modul erstellt werden kann, muss die Konfiguration vollständig angelegt werden. Dies geschieht über eine *Modul-Konfigurationsdatei*, welche vom Server gelesen und verarbeitet wird. Mehrere Konfigurationen können parallel verwaltet und je nach Bedarf ausgewählt und erstellt werden.

Eine Konfigurationsdatei enthält alle notwendigen Einstellungen in einem einfachen, erweiterbaren Textformat, welches im Wesentlichen aus Zuweisungen der Form `<Name> = <Wert>` und speziellen Anweisungen besteht. Ein Beispiel für eine solche Datei ist in Listing 2 zu finden.

Über die allgemeine Anweisung `TableSet = <Bezeichner>` kann ein Klassifikationsproblem von Seiten der Datenbasis gegenüber anderen Problemen abgegrenzt werden. Der *Bezeichner*, welcher der Problemstellung zugewiesen wird, ist auch Bestandteil der Tabellennamen aller Klassifizierer in der Datenbank. Mithilfe dieser Variablen konnte

**Listing 2** Beispiel einer Modul-Konfigurationsdatei

```
TableSet=sbc

BayesNetClass=secondLayer2005-12-14
ResultNodes=Geschlecht1;Altersklasse1
ResultProfilePaths=user::gender;user::age
ResultStates=female,male;child,adult,senior

classifier KwKm_Ew_Em_SwSm;KwKm-Ew-Em-SwSm;VotingGauss;0.18,
           0.3,0.91,1.92,1.7
classifier KwKm_Ew_Em_SwSm;KwKm-Ew-Em-SwSm;VotingGauss
```

| **Bezeichner** | **Algorithmus** |
|---|---|
| MultiGauss | Bayesscher Klassifizierer |
| WeightedUniGauss | Gaussian Mixture Model |
| J48 | C4.5 Entscheidungsbaum |
| NaiveBayes | Naive Bayes |
| KNN | Nearest-Neighbor (k=3) |
| SVM | Support Vector Machines |
| NeuralNet | Künstliches Neuronales Netz |

Tabelle 3.6: Liste der in der DEVELOPMENT PLATFORM verfügbaren Klassifikationsverfahren

beispielsweise parallel zur Entwicklung konkreter Projekte auch AGENDER weiterentwickelt werden.

Die Konfiguration der ersten Ebene erfolgt über `classifier`-Anweisungen. Diese verwenden die folgende Syntax:

`classifier <`*`Klassifizierer-ID`*`>;<`*`Klasse`*`>;<`*`Algorithmus`*`>[;<`*`Parameter`*`>]`

Die Funktionsweise der *Klassifizierer-ID* entspricht der in Abschnitt 3.7.3 dargestellten Semantik und ist folglich nur intern von Bedeutung. Der *Klassenname* wird primär zur Zuordnung zur Klassenspalte in den Tabellen der Datenbank und zu Ausgabezwecken verwendet; intern verwenden die Klassifizierer keine Klassenbezeichnungen. <*Algorithmus*> gibt den Namen der Klassifikationsmethode aus einer vordefinierten Liste von Konstanten an. Tabelle 3.6 führt die auf der DEVELOPMENT PLATFORM verfügbaren Algorithmen auf. Die Mehrzahl davon stammt aus dem *WEKA*-Paket. Es ist zu beach-

ten, dass nicht alle Klassifizierer auch eine Implementierung für das eingebettete Modul bereitstellen, jedoch eignen sie sich dennoch für die theoretische Evaluierung. Als letztes Argument in der `classifier`-Anweisung können *Parameter* definiert werden, welche an den Klassifizierer übergeben werden. Dieses Argument ist optional und wird nicht von allen Klassifizierern verwendet. Es erlaubt beispielsweise für den *k-Nearest-Neighbor*-Algorithmus die Spezifizierung des Parameters $k$, oder bei dem *WeightedUniGaussClassifier* die Angabe der Gewichte (wie in Listing 2).

Der Wert `BayesNetClass = <Name>` bestimmt, welches Bayessche Netz verwendet wird. Dabei wird vorausgesetzt, dass eine mit dem in Abschnitt 3.7.6 beschriebenen Werkzeug erstellte Klasse, bestehend aus den Dateien `<Name>.h` und `<Name>.cpp`, in einem bestimmten Verzeichnis verfügbar ist. Wird <*Name*> nicht angegeben, so wird die zweite Ebene deaktiviert.

Wird die zweite Ebene verwendet, so müssen der DEVELOPMENT PLATFORM einige Eigenschaften des Netzes mitgeteilt werden: Über `ResultNodes = <Knoten1>;<Knoten2>;...` werden die Namen der Ergebnisknoten im Netz angegeben, welche für die zu klassifizierenden Sprechereigenschaften stehen. Der Name muss dabei lediglich der in *JavaDBN* verwendeten Bezeichnung entsprechen, um die Zuordnung herzustellen. Die Reihenfolge der Angabe ist beliebig. Durch `ResultProfilePaths = <Profilpfad1>:<Profilpfad2>;...` wird jeder der Sprechereigenschaften ein *Profilpfad* zugewiesen (vgl. Abschnitt 3.8.2). Dies ist der Schlüssel, unter dem ein Klassifikationsergebnis gespeichert und an die Anwendung übermittelt wird. Die Reihenfolge der Knoten muss der in `ResultNodes` verwendeten Anordnung entsprechen. Schließlich muss noch die Variable `ResultStates = <Klasse1Zustand1>,<Klasse1Zustand2>,...;<Klasse2Zustand1>,...;...` gesetzt werden. Sie dient dazu, für jede Eigenschaft die Zustände des Bayesschen Netzes, welche numerisch gespeichert sind, einer Zeichenfolge zuzuordnen, damit sie von der Anwendung sinnvoll interpretiert werden können. Es handelt sich bei der Zeichenfolge um ein zweidimensionales Datenfeld: Die erste Dimension bezeichnet die Knoten und darf in ihrer Reihenfolge nicht von `ResultNodes` abweichen. Die zweite Dimension bezeichnet die Klassen- bzw. Zustandsnamen und muss der Reihenfolge entsprechen, wie sie im Bayesschen Netz gespeichert sind.

Modul-Konfigurationsdateien werden in einem bestimmten Verzeichnis gesucht. In der Benutzeroberfläche der DEVELOPMENT PLATFORM kann unter dem Blackboard-Objekt *home* → *sbc* → *actions* → *Configuration* (s. Abb. 3.26) eine Liste der verfügbaren Dateien angezeigt werden. Nach der Auswahl einer Datei werden die Konfiguration durch Klicken auf die Schaltfläche *Load Module Configuration* (Modulkonfiguration laden) eingelesen und die Klassifizierer geladen.

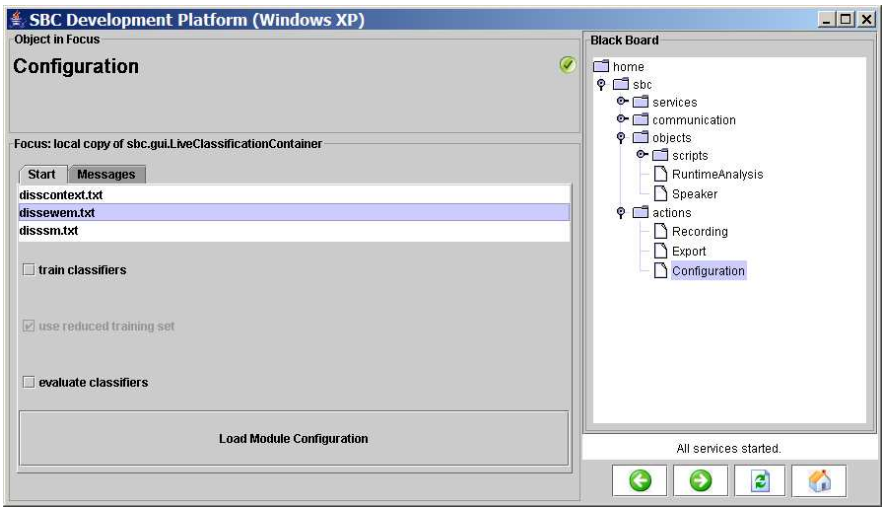



Abbildung 3.26: Laden einer Modul-Konfiguration in der DEVELOPMENT PLATFORM

### 3.9.4 Trainieren von Klassifizierern

Das Training der Klassifizierer erfolgt automatisch beim Laden der Modulkonfiguration. Dabei wird eine Verbindung zur Datenbank hergestellt und die benötigten Tabellen werden gelesen. Der Trainingsvorgang kann je nach Größe der Datenbank und verwendetem Algorithmus sehr viel Zeit beanspruchen. Daher wird ein fertig trainierter Klassifizierer in einer Klassifiziererdatei gespeichert, sofern er die Serialisierung durch *Java* unterstützt, was bei den meisten Klassifizierern ohne zusätzlichen Aufwand durch Implementierung der Schnittstelle `Serializable` erreicht werden kann. Wird darauf folgend die Modulkonfiguration neu geladen, so werden die Klassifizierer direkt aus den zuvor erstellten Dateien in den Speicher geladen, und der Trainingsvorgang entfällt. Werden jedoch Änderungen an der Konfiguration vorgenommen, so kann ein neuer Trainingsvorgang erforderlich sein, damit die Klassifizierer die aktualisierten Daten widerspiegeln. Um dies durchzuführen muss vor dem Laden der Modulkonfiguration das Kontrollkästchen *Train Classifiers* (Klassifizierer trainieren) aktiviert werden. Dadurch werden alle im Modul enthaltenen Klassifizierer neu trainiert, auch wenn bereits eine Klassifiziererdatei vorhanden ist. Alternativ kann auch die Klassifiziererdatei manuell vom Datenträger gelöscht werden, was auch ein selektives Trainieren ermöglicht.

Nach dem Training werden die Klassifizierer dem Blackboard und der Navigationshierarchie hinzugefügt. Diejenigen Klassifizierer, welche über eine grafische Ausgabemethode in der GUI verfügen, können dann visualisiert werden.

| | Jw | Jm | Km | Sw | Kw | Ew | Sm | Em |
|---|---|---|---|---|---|---|---|---|
| Jw | 559 | 268 | 49 | 28 | 1103 | 26 | 2 | 2 |
| Jm | 129 | 1207 | 81 | 101 | 409 | 21 | 66 | 23 |
| Km | 255 | 316 | 252 | 77 | 1105 | 23 | 4 | 5 |
| Sw | 3 | 51 | 23 | 1570 | 12 | 77 | 282 | 19 |
| Kw | 277 | 103 | 83 | 9 | 1550 | 11 | 3 | 1 |
| Ew | 4 | 11 | 0 | 141 | 5 | 1726 | 80 | 70 |
| Sm | 0 | 34 | 1 | 254 | 0 | 24 | 1694 | 30 |
| Em | 0 | 15 | 0 | 32 | 0 | 72 | 128 | 1790 |

Abbildung 3.27: Beispiel für eine Konfusionsmatrix

### 3.9.5 Automatisches Evaluieren von Klassifizierern

Bei der Evaluierung handelt es sich um eine statistische Methode zum Feststellen der Genauigkeit eines Klassifizierers. Dabei wird für eine Anzahl von Äußerungen, bei denen die tatsächliche Klasse bekannt ist, eine Klassifizierung durchgeführt und das Resultat mit dem tatsächlichen Ergebnis verglichen. Offensichtlich hängt die Aussagekraft des Ergebnisses unter anderem von der Menge der Daten ab, die zum Evaluieren verwendet werden. Je mehr Daten dies sind, desto eher entspricht das Ergebnis der tatsächlichen Genauigkeit.

Eine Darstellungsform für Evaluierungsergebnisse, die für beliebig umfangreiche und auch nicht notwendigerweise ausbalancierte Daten geeignet ist, ist die *Konfusionsmatrix* (Beispiel: Abb. 3.27). Dabei handelt es sich im Kern um eine $N \times N$-Matrix, wobei $N$ für die Anzahl der Klassen des Klassifizierers steht. Zur besseren Lesbarkeit werden in der Regel noch Beschriftungen in der ersten Zeile bzw. Spalte angegeben. Innerhalb der Matrix steht jede Spalte für eine durch den Klassifizierer vorhergesagte Klasse und jede Zeile für eine tatsächliche Klasse. Die Zahl in der Zelle $\langle i, j \rangle$ beschreibt die Anzahl der Instanzen aus der Klasse in Zeile $i$, welche vom Klassifizierer als die Klasse in Spalte $j$ bestimmt wurde. Daraus folgt, dass in den Zellen der Diagonalen die Anzahl der korrekt klassifizierten Instanzen steht und in allen anderen Zellen jeweils die falsch klassifizierten Daten. Da diese nach Klassen getrennt sind, lässt sich aus dieser Darstellung gut ablesen, welche Klassen häufig verwechselt werden. M3I CAT enthält Methoden, um eine Konfusionsmatrix in Diagrammform darzustellen, was eine weitere Verbesserung der Lesbarkeit für bestimmte Aspekte bedeutet.

Für die Evaluierung sollten möglichst andere Daten verwendet werden als zur Klassifizierung. Dies trifft besonders auf diejenigen Klassifizierungsmethoden zu, welche das Overfitting-Problem aufweisen. Als besonders deutliches Beispiel stelle man sich einen Klassifizierer vor, welcher die Eingabedaten „auswendig lernt“. Wird dieser mit den gleichen Daten evaluiert, so erreicht er naturgemäß eine gesamte True Positive Rate von 100%, wohingegen diese bei unbekannten Daten um das Zufallsniveau liegen würde. Andererseits ist es aber auch im Interesse des Entwicklers, möglichst viele Daten zum Training zu verwenden, anstatt sie ausschließlich für die Evaluierung zu reservieren. Als

Kompromiss bietet sich die $n$-fache *Kreuzvalidierung* an. Dabei werden zunächst alle verfügbaren Daten zufällig in $n$ gleich große, disjunkte Mengen aufgeteilt. Anschließend wird für jede dieser $n$ Mengen eine Evaluierung durchgeführt, wobei die jeweils nicht in der Menge enthaltenen Instanzen zum Training des Klassifizierers herangezogen werden und die Menge selbst zur Evaluierung verwendet wird. Die Resultate der einzelnen Evaluierungen werden zu einem Gesamtergebnis aufsummiert. Der auffälligste Nachteil der Kreuzvalidierung ist die deutlich längere Dauer der Evaluierung durch die mehrfachen Trainingsphasen.

Die SBC DEVELOPMENT PLATFORM bietet eine integrierte Funktion zur zehnfachen Kreuzvalidierung der Klassifizierer in einer Modul-Konfiguration. Um diese zu starten ist vor dem Laden des Moduls das Kontrollkästchen *Evaluate Classifiers* (Klassifizierer evaluieren) zu aktivieren. Die Evaluierung selbst ist Bestandteil von *WEKA*. Die Ergebnisse werden in einem festen Textformat auf der Konsole ausgegeben und können für spätere Analysen in eine Datei umgeleitet werden. Sie enthalten die Konfusionsmatrix und die True Positive Rate für jede Klasse.

Ein Problem, das sich durch die Trennung von Klassifikationsmodul und Entwicklungsplattform ergibt, ist das der Vergleichbarkeit der Daten. Jeder Klassifizierer muss gesondert in zwei Versionen geschrieben werden, wobei beide Implementierungen von der Berechnung her identisch sein müssen. Aber selbst wenn dies durch Verifizierung des Algorithmus bestätigt wurde, so können dennoch durch Unterschiede in der Sprache und Plattform abweichende Werte errechnet werden. Daher kann die tatsächliche Genauigkeit des Klassifizierers in der Anwendung von der durch die Evaluierung gemessenen Qualität abweichen. Dabei sollte es sich jedoch in aller Regel nur um sehr geringe Differenzen handeln, die sich nicht auf die Eignung eines Moduls auswirken. Um exakte Werte zu erhalten, besteht auch noch die Möglichkeit, eine Evaluierung mit dem fertigen Modul durchzuführen (vgl. Abschnitt 3.10). Dafür müssen allerdings eigene Testdaten zusammengestellt werden.

### 3.9.6 Entwurf des dynamischen Bayesschen Netzes

Die SBC DEVELOPMENT PLATFORM bietet derzeit keinen integrierten Editor für Bayessche Netze. Daher muss auf eine externe Lösung wie *Hugin Researcher* zurückgegriffen werden. Die genannte Software erlaubt das Skizzieren von Knoten und Kanten, Eintragung der CPTs und Testen der eingegeben Werte anhand von Beispielszenarien. Abb. 3.28 veranschaulicht einen solchen Testfall. Links in der Abbildung sind die Testdaten zu sehen, während rechts das Netz dargestellt ist. In diesem Fall wurden alle drei Klassifizierer-Knoten instanziiert, so dass ihr Einfluss auf die Eigenschaftsknoten beobachtet werden kann.

Das fertige Netz wird gespeichert und durch den *JavaDBN* in eine *C++*-Klasse um-

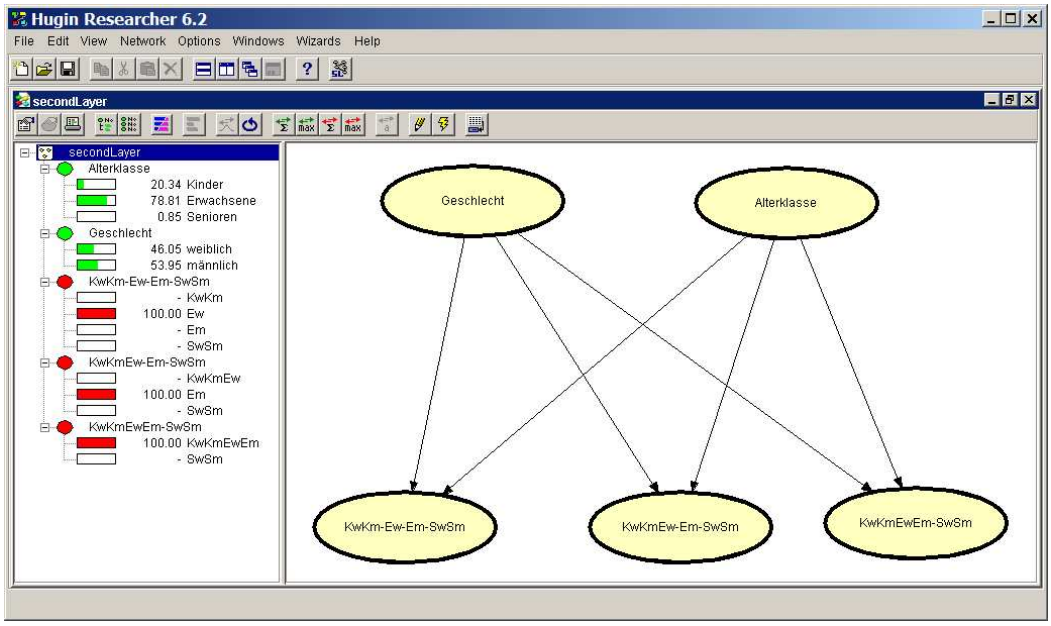



Abbildung 3.28: Ein Bayessches Netz wird in der Anwendung *Hugin Researcher* entworfen und getestet

gewandelt, welche ein dynamisches Bayessches Netz modelliert (s. Abb. 3.29). Die so erzeugte Quellcodedatei wird in ein Verzeichnis der DEVELOPMENT PLATFORM gespeichert, damit sie, basierend auf der gewählten Konfiguration, später automatisch in das Modul integriert werden kann. Der Name der Klasse wird, wie in Abschnitt 3.9.3 angegeben, in die Modul-Konfigurationsdatei eingetragen. Auf die Art und Weise lässt sich später einfach zwischen mehreren Netzen umschalten.

### 3.9.7 Exportvorgang

Durch den Export wird der Anforderung an die Modularität der Architektur Rechnung getragen. Ohne die Möglichkeit zum Export der Daten aus der Entwurfsphase müssten de facto alle Änderungen, wie z.B. die Integration der Klassifizierer, manuell am Code des Klassifikationsmoduls vorgenommen werden, was also bedeutet, dass ein einziges Modul immer wieder verändert wird. Beim Exportvorgang dagegen wird mit mehreren parallelen Konfigurationen gearbeitet, die auf der Entwicklungsplattform verwaltet werden. Zwar wird aus technischer Sicht auch hier immer die gleiche Codebasis mit der jeweils gewählten Konfiguration aktualisiert, jedoch geschieht dies vollständig transparent.

Der Exportvorgang besteht aus drei Teilen: dem Export der Klassifizierer, dem Ko-

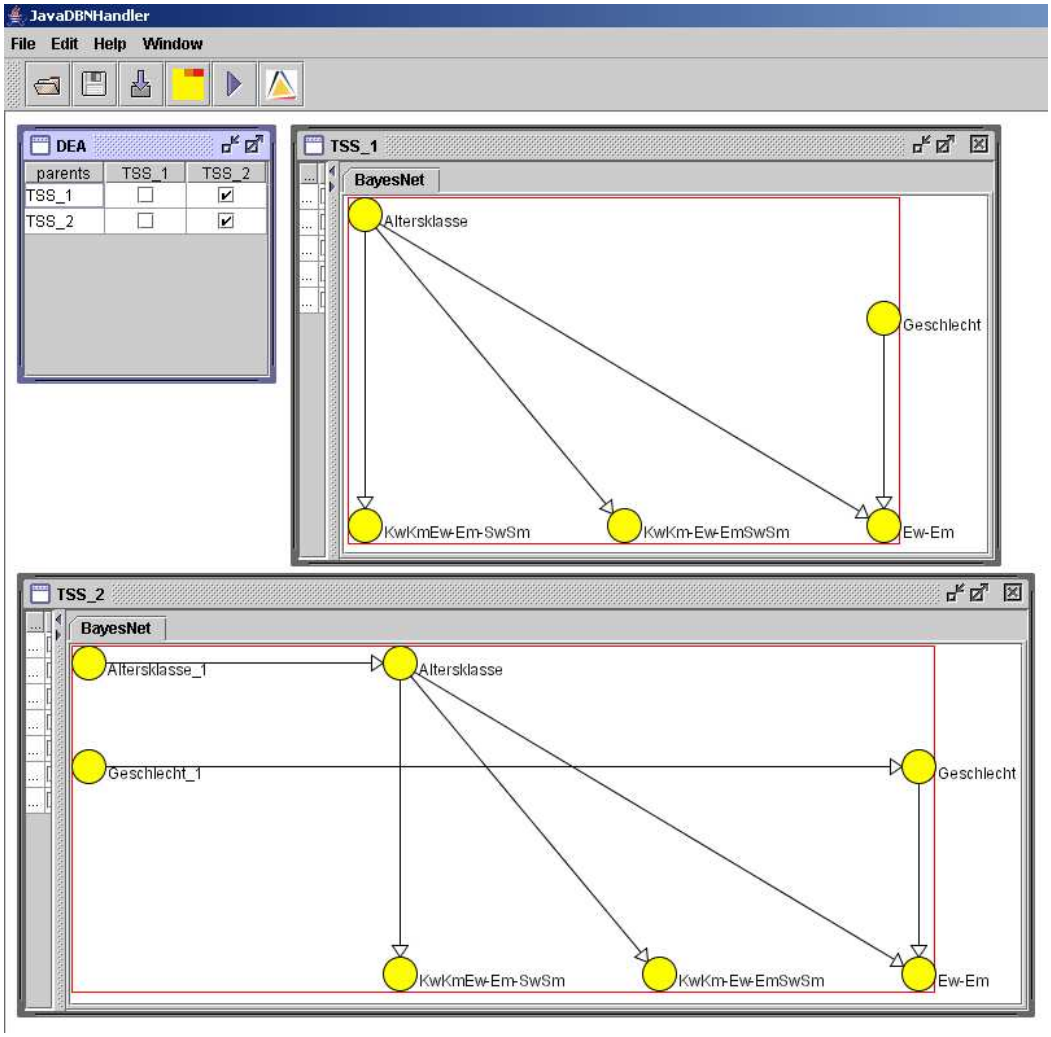



Abbildung 3.29: Darstellung des dynamischen Bayesschen Netzes in *JavaDBN*

pieren der zweiten Ebene und dem Schreiben der Modul-Konfiguration. Bezüglich der Klassifizierer werden zuerst unter den geladenen Objekten diejenigen ermittelt, welche exportierbar sind. Diese Klassifizierer müssen zwei Bedingungen erfüllen: Sie müssen die Schnittstelle `SourceExportableClassifier` unterstützen und es muss eine *Export-Schablone* für sie vorliegen. Die Schnittstelle `SourceExportableClassifier` beinhaltet die Methoden `getExportTemplate` und `getTemplateSectionVars`. `getExportTemplate` gibt den Dateinamen der Schablone an, welche für den Export verwendet werden soll. Die zweite Methode wird zusammen mit der Schablone zum Erzeugen der Klasse verwendet. Eine Schablone ist im Wesentlichen eine *C++*-Klasse mit Inline-Implementierung, d.h. Header und Implementierung in einer Datei, und mit benannten *Platzhaltern* der Form `%name%` für die internen Datenstrukturen oder sonstigen instanzspezifischen Code des Klassifizierers. Platzhalter werden am häufigsten für Felder und Initialisierungscode im Konstruktor verwendet. Mit Hilfe der Funktion `getTemplateSectionVars` werden für einen trainierten Klassifizierer die Werte für alle in der Schablone auftretenden Platzhalter ermittelt. Einige Variablen werden unabhängig vom Klassifiziertyp bestimmt, da sie bei allen Verfahren benötigt werden, z.B. die Klassifizierer-ID, der Profilpfad, sowie Mittelwerte und Standardabweichung der Merkmale (zur Normierung). Anschließend können die Platzhalter durch diese Werte ersetzt werden. Bei diesem Vorgang entsteht eine Datei pro Klassifizierer, welche in das Verzeichnis des Klassifikationsmoduls geschrieben wird.

Der zweite Teil des Exports besteht lediglich im Kopieren der Codedatei für das Bayessche Netz in das entsprechende Verzeichnis des Moduls. Den dritten Teil stellt die Konfigurationsbeschreibung dar (vgl. Abschnitt 3.7.7). Auch hierfür gibt es eine Schablone, bestehend aus den Dateien `ModuleConfig.h` und `ModuleConfig.cpp`. Die Platzhalter in diesen Dateien sind fest vordefinierte Variablen, die je nach Modulkonfiguration ersetzt werden. Einige davon enthalten einfache Werte, wie z.B. den Namen der Klasse für die zweite Ebene, während andere durch vollständige Funktionen ersetzt werden, wie z.B. `CreateModuleClassifier` zum Erstellen eines Klassifizierers basierend auf dem Index. Diese Datei dient auch dazu, die Verbindung zu den einzelnen Klassifizierer-Dateien herzustellen: Sie besitzt im Gegensatz zu diesen nämlich einen festen Namen und kann so vom Projekt des Klassifikationsmoduls dauerhaft referenziert werden. Die Klassifizierer-Dateien schließlich werden über Inklusionsanweisungen in der Konfigurationsbeschreibung referenziert.

Um den Export zu starten, muss zu dem Knoten *home* → *sbc* → *actions* → *Export* gewechselt werden. Dort wird der Vorgang über die Schaltfläche *Export* gestartet. Dieser Vorgang läuft aufgrund der geringen Komplexität sehr schnell ab. Vor dem ersten Export müssen die Pfadangaben für die Schablonen und das Klassifikationsmodul in der Programm-Konfigurationsdatei überprüft werden.

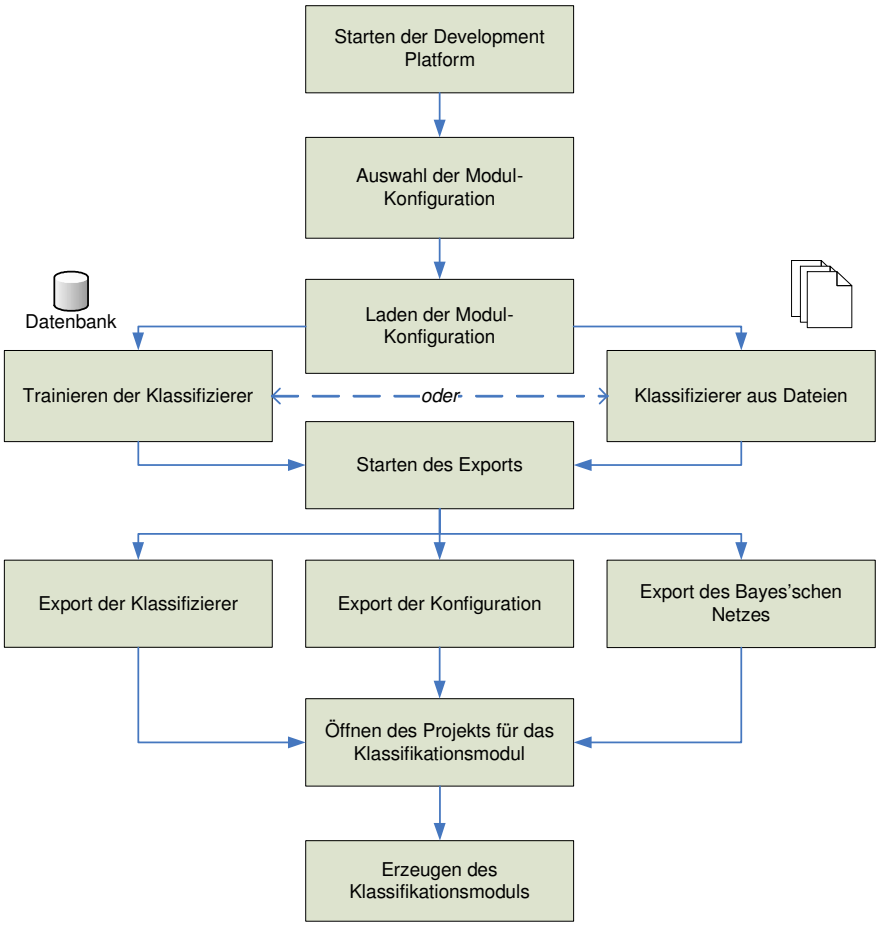



Abbildung 3.30: Schritte zur Erzeugung eines Klassifikationsmoduls (Build-Vorgang)

### 3.9.8 One-Click Build-Vorgang

Abbildung 3.30 fasst den kompletten Build-Vorgang in einem Diagramm zusammen. Wird dieser Vorgang über die grafische Benutzeroberfläche ausgeführt, so handelt es sich immer noch um mindestens sechs Schritte, von denen einige mehrerer Mausklicks bedürfen. Hier besteht also ein Bedarf zur weiteren Automatisierung des Prozesses.

Eine Lösung stellt der *One-Click Build-Vorgang* dar. Dieser ermöglichst es, in nur einem Schritt (aus Sicht des Entwicklers) den kompletten Ablauf anzustoßen und mit der Erzeugung des Moduls abzuschließen. Ausgangsbasis ist ein Skript für jede Modul-Konfiguration, welches die zur Erzeugung notwendigen Befehle enthält, die direkt durch das Betriebssystem ausgeführt werden.

Der erste Teil des Build-Vorgangs besteht aus dem Export durch die Entwicklungsplattform. Dieser kann alternativ zur GUI-Variante auch über den Konsolenmodus aus-

geführt werden. Dies kann noch weiter vereinfacht werden, indem beim Aufruf des Programms der Name der zu exportierenden Konfiguration als Argument angegeben wird. Es ergibt sich also die folgende Befehlszeile:

```
DevelopmentPlatform console export=<Konfiguration>
```

Dadurch wird die Anwendung gestartet, die Konfiguration geladen und exportiert. Danach wird das Programm wieder automatisch beendet.

Der zweite Teil beinhaltet die Kompilierung des Moduls zu einer Bibliothek. *Visual Studio* unterstützt das Kompilieren eines Projekts auch über die Befehlszeile durch die folgende Syntax:

```
DevEnv.exe <Projektdatei> /build Release
```

Als *Projektdatei* muss der vollständige Pfad zur Datei mit der Erweiterung .sln angegeben werden, welche das eingebettete Modul enthält. Für die Host-Applikation besteht im Allgemeinen auch eine Möglichkeit zur automatisierten Erstellung. Die entsprechenden Befehle können dann ebenfalls in die Stapelverarbeitungsdatei aufgenommen werden.

Wird das fertige Skript nun gestartet, so wird der vollständige Build-Vorgang ohne weitere Eingaben des Benutzers ausgeführt.

## 3.10 Webserver-basiertes Evaluierungsmodul

In manchen Situationen kann es wünschenswert sein, einen Testlauf oder eine gesteuerte Evaluierung direkt mit dem Klassifikationsmodul selbst durchzuführen. Ein wichtiger Grund dafür wurde in Abschnitt 3.3.4 genannt. Das SBC-Klassifikationsmodul ist eine Bibliothek und kann daher nicht direkt aufgerufen werden. Um aber dennoch obiges Szenario ohne viel Aufwand realisieren zu können, wurde eine Konsumenten-Applikation für das Klassifikationsmodul geschrieben, welche über die Befehlszeile gestartet werden kann. Sie nimmt als Argument den Pfad und Dateinamen einer Audiodatei und führt die Klassifikation durch. Da sie im Quelltext vorliegt, kann sie mit jeder gewünschten Modulkonfiguration kompiliert werden.

Das Ergebnis ist eine codierte Zeichenfolge, welche die extrahierten Merkmale, das Ergebnis jedes einzelnen Klassifizierers und die auf zweiter Ebene vorhergesagten Klassen enthält.

Da sich die Datenbank häufig auf einem anderen Rechner befindet als der, auf dem die Entwicklung und Analyse erfolgt, wurde zudem ein Web Service geschrieben, welcher als Schnittstelle zwischen dem Entwicklungsrechner und der Klassifikationsanwendung fungiert. Er liest den Namen der zu klassifizierenden Audiodatei über die URL aus, führt intern die oben beschriebene Anwendung zur Klassifizierung aus, und liefert die Ergebnisse über das HTTP-Protokoll zurück. Der Web Service verwendet *ASP.NET* und benötigt einen entsprechenden Webserver, wie z.B. *Internet Information Server 5.0* und höher oder *ASP.NET Web Matrix*.

Über die Webschnittstelle lassen sich vom Entwicklungsrechner aus komfortabel Klassifizierungen durchführen, und zwar auch parallel oder stapelweise. M3I CAT enthält beispielsweise eine Funktion zur Visualisierung der Ergebnisse einzelner Klassifizierer, welche diese Schnittstelle nutzt. Hierbei werden für jeden Klassifizierer die Gauß-Verteilungen aller Merkmale in einem Diagramm dargestellt und die extrahierten Werte eingetragen. Zusammen mit dem Klassifizierungsergebnis erhält der Entwickler Aufschluss darüber, ob ein bestimmtes Merkmal gut oder weniger gut geeignet ist, und ob die Werte in einem konkreten Fall stark vom Erwartungswert abweichen bzw. in eine andere Klasse fallen und daher eine Fehlklassifikation begründen.

## 3.11 SBC-Gesamtkonzept

Bislang wurde die SBC-Architektur hauptsächlich auf Komponentenebene und aus Sicht der Anwendung, welche die Dienste der Sprecherklassifikation nutzt, betrachtet. Tatsächlich aber werden im Rahmen von SBC auch in der Zukunft Verbesserungen vorgenommen und neue Möglichkeiten zur Klassifikation in die bestehende Architektur integriert. Abbildung 3.31 stellt die Zusammenhänge zwischen den Prozessen in der Weiterentwicklung von Verfahren und Komponenten und denen der Integration in Anwendungen dar.

Bei der Weiterentwicklung des Verfahrens, also AGENDER, ist zu unterscheiden zwischen Verbesserungen der bestehenden Methoden einerseits, wie z.B. genauerer Klassifikation durch neue Mustererkennungsverfahren und Algorithmen oder feinere Aufteilung der bestehenden Klassen, und der Entwicklung komplett neuer Methoden anderseits, z.B. Sprachenerkennung oder Kontextklassifikation. Als Grundlage für die genannten beiden Bereiche dienen die konsequente Fortführung der Forschung auf den entsprechenden Gebieten einschließlich Literaturstudien, die Analyse neuer Daten, sofern diese verfügbar sind, durch die existierenden Werkzeuge, sowie die Anforderungen aus der Industrie nach neuen Technologien und die Rückmeldung aus konkreten Projekten. Letzteres ist für SBC besonders durch die Mitwirkung an solchen Projekten interessant: Wird beispielsweise in dem Telefonie-Szenario für bestimmte Typen von Verbindungen (z.B. Mobiltelefone) eine deutliche Schwankung der Klassifikationsgenauigkeit beobachtet, so können die klassifizierten Werte direkt zur Analyse und zum Training wieder verwendet werden.

Auch die Software-Komponenten von SBC, insbesondere Klassifikationsmodul und DEVELOPMENT PLATFORM, werden verbessert. Sehr deutlich wird dies, wenn AGENDER um neue Funktionen erweitert wird, für die eine Schnittstelle zur Applikation geschaffen werden muss. Aber auch für ein neu entwickeltes Klassifikationsverfahren muss eine performante *C++*-Implementierung für das eingebettete Modul bereitgestellt werden. Allerdings ist nicht jede Tätigkeit in diesem Bereich an entsprechende Änderungen des Verfahrens geknüpft. So sind auch unabhängig von diesem fortwährende Optimierungen und weitere Verbesserungen vorgesehen. Auch kann von Seiten der Industrie Bedarf nach der Unterstützung neuer Plattformen bestehen, was ebenfalls lediglich Anpassungen auf Software-Ebene erfordert.

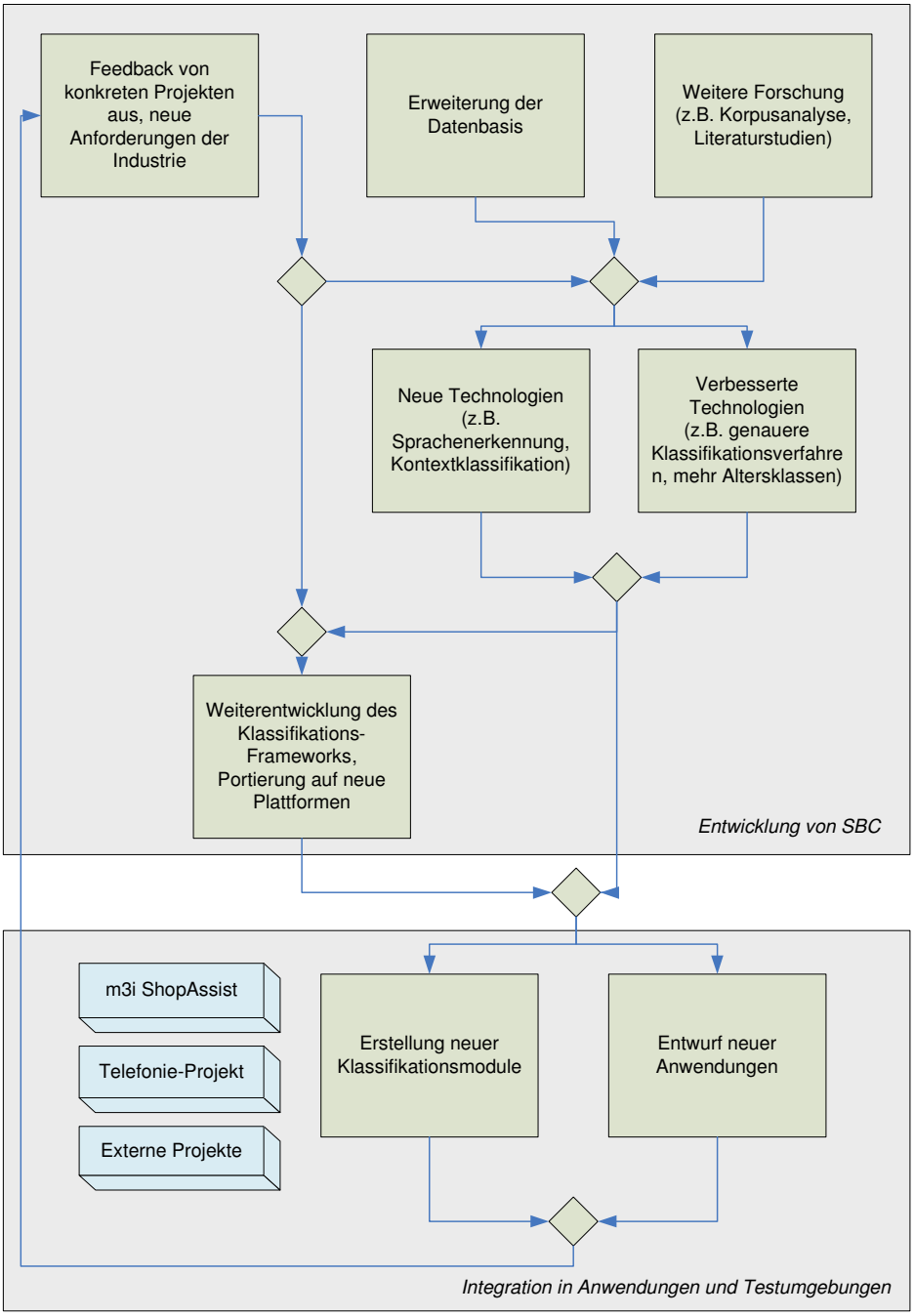



Abbildung 3.31: Beziehungen zwischen Entwicklung und Anwendung des Systems (SBC-Gesamtkonzept)

# 4 Anwendungen

In diesem Kapitel werden zwei konkrete Anwendungen für SBC beschrieben, welche die vielfältigen Einsatzmöglichkeiten hervorheben. Bei der ersten Applikation handelt es sich um einen digitalen Einkaufsassistenten, welcher von dem Endbenutzer auf einem mobilen Gerät bedient wird. Im zweiten Szenario wird die Sprecherklassifikation auf einem Server ausgeführt, der in einem Callcenter für die Verarbeitung eingehender Anrufe zuständig ist.

## 4.1 m3i Mobile ShopAssist

Der MOBILE SHOPASSIST (oder kurz SHOPASSIST) wurde von Wasinger, Krüger und Jacobs (2005) ebenfalls am DFKI im Rahmen des Projekts M3I entwickelt. Die Anwendung dient als Demonstrationsobjekt für eine Reihe von Technologien wie z.B. multimodaler Interaktion, besitzt aber auch einen engen Bezug zur Praxis. Sie läuft vollständig auf einem *PocketPC* und modelliert ein Hilfsmittel für den Endbenutzer, welches als Informationsquelle und Berater beim Einkauf genutzt werden kann. Im Kern enthält das Programm eine Produktdatenbank, welche die Artikel eines Geschäfts enthält, in dem sich der Benutzer gerade befindet. Auf Wunsch können zu einem Artikel Preis oder Zusatzinformationen erfragt oder die technischen Daten zweier Artikel verglichen werden. Diese Funktionalität lässt sich dabei auf verschiedene Arten steuern, unter anderem per Sprache oder Schrift, durch Gestik unter Zuhilfenahme des Zeigegeräts (Stylus), durch Auswahl realer entsprechend präparierter Artikel im Ladenregal sowie durch eine Kombination aus den genannten Modi. Eine typische Sprachinteraktion im Kontext von Abb. 4.1 wäre die folgende:

> **Benutzer:** „Vergleiche diese Kamera *[Benutzer tippt auf eine Kamera auf dem Bildschirm]* mit dieser *[Benutzer nimmt ein Produkt aus dem Regal]*."
>
> **PocketPC:** „Produkt 1: Canon PowerShot S1 IS, 3.2 Megapixel, 599 Euro. Produkt 2: Canon PowerShot S50, 5 Mega-pixel, 599 Euro."

Um die gesprochene Eingabe zu verstehen, wird von der Anwendung *Spracherkennung* in Kombination mit *Natural Language Technologie* eingesetzt. Das mobile Gerät gibt die Informationen über die beiden Produkte sowohl per *Sprachsynthese* (*Text-to-Speech*, TTS) als auch per visueller Darstellung aus.


Abbildung 4.1: Benutzerinteraktion mit dem SHOPASSIST (vgl. Wasinger et al., 2005, S. 299ff)

### 4.1.1 Verwendung von SBC im Mobile ShopAssist

In dem beschriebenen Rahmen bietet es sich an, AGENDER zur *Benutzeradaption* für den SHOPASSIST zu verwenden. Dabei könnte die Information über Alter und Geschlecht des Benutzers beispielsweise dazu genutzt werden, zielgruppenspezifische Produktempfehlungen zu geben oder einen Benutzer gezielt zu beraten, wenn er sich zwischen mehreren Produkten nicht entscheiden kann. Darüber hinaus sind auch klassische Optionen wie Anpassen der Lesbarkeit für ältere Benutzer denkbar. Ausgangsbasis für die Sprecherklassifikation sind die Sprachbefehle, welche der Anwender zum Steuern der Anwendung im normalen Betrieb gibt. Die Klassifikation kann daher beiläufig erfolgen, ohne dass hierzu künstlich Spracheingaben angefordert werden müssen.

Von zentraler Bedeutung beim Einsatz von AGENDER auf einem mobilen Gerät sind dessen beschränkte Ressourcen. So muss davon ausgegangen werden, dass die Klassifikation eine gewisse Zeit (mehrere Sekunden, auf langsamen Geräten evtl. sogar über

eine Minute) in Anspruch nimmt und auch dass sie möglicherweise nicht vollständig im Hintergrund erfolgen kann, d.h. der Benutzer könnte die Klassifikation in Form einer Verlangsamung der Reaktionsgeschwindigkeit des SHOPASSIST wahrnehmen. Aus diesem Grund ist es sinnvoll, wie unter Abschnitt 3.6 beschrieben die Zusammenstellung der Klassifizierer im Klassifikationsmodul qualitativ und quantitativ insbesondere nach Ressourcenaspekten zu beurteilen, was wiederum bedeutet, dass Einbußen bei der Klassifikationsgenauigkeit in Kauf genommen werden.

Eine Grundannahme bei M3I ist das „*always on*"-Prinzip, d.h. die (bis auf unvorhergesehene Ausfälle) ständige Verfügbarkeit einer Netzwerkverbindung über WLAN, UMTS oder ein anderes Funknetzwerk. Der SHOPASSIST nutzt dies beispielsweise aus, um auf die Datenbank des Geschäfts zuzugreifen, in dem sich der Benutzer befindet. Hierfür ist auf der Seite des Anbieters ein Server erforderlich, welcher die Daten zur Verfügung stellt und die Kommunikation verwaltet. Ein solcher bereits existierender Server und die damit verbundene Netzwerk-Infrastruktur kann auch von SBC dazu genutzt werden, eine serverbasierte Klassifizierung durchzuführen. Der dafür auf dem mobilen Gerät anfallende Arbeitsaufwand ist verglichen mit der eigentlichen Klassifizierung äußerst gering, da lediglich die Audiodatei an den Server übermittelt und das Ergebnis abgefragt werden muss. Dies belastet primär die Netzwerkverbindung, nicht aber die CPU. Ein ähnlicher Ansatz wird auch in anderen Arbeiten verfolgt. So nutzen beispielsweise Park und Jung (2004) komputationell anspruchsvolle Künstliche Neuronale Netze auf einem Server, um eine präzise Textlokalisierung für ein Client-Gerät vorzunehmen.

Abbildung 4.2 stellt das beschriebene Szenario dar. Wie der Architekturskizze zu entnehmen ist, wird das SBC-Klassifikationsmodul nicht direkt in den SHOPASSIST integriert, sondern über eine weitere Schnittstelle, die SBC-Client-Bibliothek, angesprochen. Der Grund für diese Entscheidung liegt darin, dass die Kombination von server- und clientgestützter Klassifikation in einer Anwendung je nach Verfügbarkeit für viele Anwendungen auf der mobilen Plattform eine sinnvolle Lösung ist, da ihnen allen das Problem der Ressourcenbeschränktheit gemeinsam ist. Die hier vorgestellte Client-Bibliothek kann universell von beliebigen *PocketPC*-Anwendungen genutzt werden. Die SHOPASSIST-Architektur kann also in drei Komponenten unterteilt werden: Die eigentliche Applikation, die SBC-Client-Bibliothek und den SBC-Server. Applikation und Bibliothek laufen auf demselben Gerät, sogar im gleichen Prozess. Der Server dagegen ist – sofern verfügbar – über ein beliebiges Netzwerk mit einem oder mehreren Clients verbunden. In der Regel wird es sich aus praktischen Gründen um ein Funknetzwerk handeln, für das entsprechend besondere Faktoren wie die inhärente Instabilität in Bezug auf Verbindungen zu berücksichtigen sind.

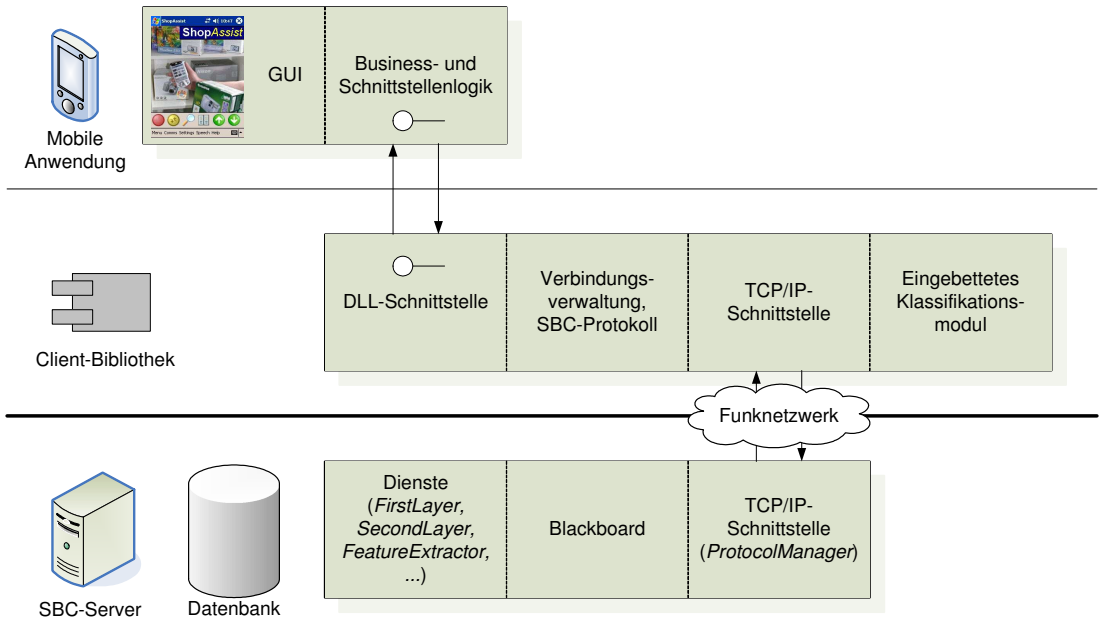



Abbildung 4.2: Client-Server-Architektur für Klassifikation auf mobilen Endgeräten

### 4.1.2 SBC-Client-Bibliothek

Die Client-Bibliothek stellt die Schnittstelle zwischen der Anwendung und einem Dienst dar. Bei dem Dienst handelt es sich meist um die Sprecherklassifikation, aber durch ein universelles Protokolldesign lassen sich auch weitere Dienste einsetzen. Die Ausführung des Dienstes erfolgt entweder auf einem entfernten Rechner oder – bei Sprecherklassifikation mittels AGENDER – lokal auf dem Client.

Die Bibliothek wurde als DLL unter *Windows CE* realisiert. Dadurch lassen sich die Vorzüge der gemeinsamen Verwendung der Komponente durch mehrere Anwendungen nutzen. Neben der möglichen Speicherplatzersparnis bietet dies auch den Vorteil, dass die Komponente ausgetauscht werden kann, wenn Aktualisierungen vorliegen. Zur Entscheidung gegen eine statische Bibliothek trug auch die Annahme bei, dass bei den meisten mobilen Anwendungen die Sprecherklassifikation nicht Hauptzweck der Anwendung ist, sondern unterstützende Funktion hat, z.B. als Bestandteil der Benutzeradaption. Umgekehrt ist der mit der dynamischen Bibliothek einhergehende Performanzverlust, verglichen mit der generell niedrigen Rechenleistung der mobilen Plattform, sehr gering. Die Performanz ist jedoch eher dann von Bedeutung, wenn die Klassifikation eine Kernfunktion der Anwendung darstellt. Außerdem entsteht durch die Klassifikation auf dem Server, eine primäre Aufgabe der Bibliothek, auch immer eine Latenz durch die Datenübertragung über das Netzwerk, welche ebenfalls um ein Vielfaches höher ist als der Aufwand von dynamischem Linken.

Die DLL stellt eine feste Menge öffentlicher Funktionen bereit (im Gegensatz zu Objekten wie beim Klassifikationsmodul). Dies bietet den weiteren Vorteil, dass sich diese Funktionalität aus fast allen Programmiersprachen erschließen lässt. So kann beispielsweise auch aus *C#*-Anwendungen, welche mit Hilfe des *.NET Compact Framework* auf dem *PocketPC* ausgeführt werden, ohne eine zusätzliche Wrapper-Bibliothek direkt auf die Klassifikationsroutinen von SBC zugegriffen werden.

Die Client-Komponente arbeitet *verbindungsorientiert*. Üblicherweise erstellt jede Applikation eine Verbindung und erhält diese während der Laufzeit des Programms über aufrecht. Werden mehrere SBC-Anwendungen zur gleichen Zeit ausgeführt, so kann die Bibliothek mehrere Verbindungen parallel verwalten und Ressourcen zweckmäßig gemeinsam nutzen. Prinzipiell kann auch eine einzelne Anwendung mehrere Verbindungen erstellen, z.B. falls verschiedene Server oder Dienste angesprochen werden sollen. Eine Verbindung stellt einen virtuellen Datenkanal dar. Dieser kann mit einem externen Server verbunden sein, auf das eingebettete SBC-Klassifikationsmodul zurückgreifen, eine Kombination aus beidem ermöglichen oder auch in einem physikalisch nicht verbundenen Zustand sein. Welche Variante davon tatsächlich zum Einsatz kommt, hängt von der Konfiguration der Verbindung und von der Netzwerkverfügbarkeit ab. Für jede Verbindung verwaltet die Bibliothek ein aktuelles Benutzerprofil, welches die Ergebnisse des Klassifikationsmoduls oder des Servers speichert.

Tabelle 4.1 listet alle öffentlichen Funktionen und deren Parameter auf. Diese können entweder dynamisch oder über die Header-Datei `ClientFunctions.h` in die Anwendung eingebunden werden. Die meisten Funktionen erhalten als weiteres Argument einen optionalen Zeichenfolgen-Ausgabeparameter, welcher im Falle eines Fehlschlags der Funktion eine Fehlerbeschreibung enthalten kann.

Das Anlegen einer neuen Verbindung erfolgt durch die Funktion `CreateConnection`. Diese Funktion kann als Argument den Namen und Anschluss eines Servers für die servergestützte Klassifikation erhalten. Rückgabewert ist eine *Verbindungs-ID*, welche in allen weiteren Funktionen spezifiziert werden muss. Beim Aufruf dieser Funktion wird weder eine tatsächliche Verbindung zum Server hergestellt, noch das interne Klassifikationsmodul initialisiert. Dies geschieht erst, wenn Daten zum Senden vorhanden sind, oder wenn die Applikation den Verbindungsaufbau durch Aufruf von `Connect` explizit anfordert. Vorher besteht die Möglichkeit, mittels `SetConnectionOption` einige Optionen zu konfigurieren. Jede Option ist durch eine Konstante definiert und akzeptiert einen spezifischen Datentyp. Eine Auswahl der derzeit möglichen Optionen kann Tabelle 4.2 entnommen werden.

Eine Besonderheit stellen die Verarbeitungsmodi dar. Hier wird festgelegt, ob die Klassifikation intern oder extern erfolgt. Die Einstellung lässt sich auch im laufenden Betrieb ändern. Der Wert *Off* gibt an, dass keine Form von Klassifizierung oder Aktualisierung des Benutzerprofils erfolgen soll. Dies ist eine Möglichkeit für die Anwendung zum De-

| **Funktionsname** | **Parameter** | **Rückgabewert** |
|---|---|---|
| `CreateConnection` | Hostadresse, Port | *Integer* |
| `CloseConnection` | connID (Verbindungs-ID) | |
| `Connect` | connID | *Boolean* |
| `Disconnect` | connID | |
| `NewTransmission` | connID | *Boolean* |
| `SetConnectionOption` | connID, Option, Wert | *Boolean* |
| `IsConnected` | connID | *Boolean* |
| `IsInternalServer-Started` | connID | *Boolean* |
| `IsSending` | connID | *Boolean* |
| `IsClassifying` | connID | *Boolean* |
| `AddFile` | connID, Dateiname | *Boolean* |
| `AddFileWH` | connID, Dateiname, Abtastfrequenz, Abtasttiefe, Kanäle | *Boolean* |
| `AddRaw` | connID, Daten (Bytes) | *Boolean* |
| `AddRawWH` | connID, Daten, Abtastfrequenz, Abtasttiefe, Kanäle | *Boolean* |
| `AddMeta` | connID, Einstellung, Wert | *Boolean* |
| `AddClassifierRequest` | connID, Klassifizierer-ID | *Boolean* |
| `AddProfileRequest` | connID, Profilpfad | *Boolean* |
| `AddSpeakerSwitch-Notification` | connID | *Boolean* |
| `AddActionRequest` | connID, Aktionsdaten | *Boolean* |
| `Send` | connID | *Boolean* |
| `SendAsync` | connID | |
| `WaitForResult` | connID, Daten (out), Timeout | *Boolean* |
| `GetCurrentProfileInfo` | connID, Profilpfad | *Boolean* |

Tabelle 4.1: Funktionen der Client-Bibliothek

| Option | Datentyp | Beschreibung |
|---|---|---|
| `AUTO_RECONNECT` | *Boolean* | Gibt an, ob im Falle eines Verbindungsabbruchs automatisch im Hintergrund die Verbindung wiederhergestellt werden soll |
| `AUTO_TIMESTAMP` | *Boolean* | Falls `True`, werden an jedes Segment Metadaten mit einem aktuellen Zeitstempel angehängt |
| `SPEECH_PROCESSING` | *Konstante* | Gibt den Verarbeitungsmodus an (siehe Text) |
| `CONNECT_TIMEOUT` | *Integer* | Gibt an, wie lange beim Herstellen einer Verbindung auf eine Antwort vom Server gewartet wird |
| `SEND_TIMEOUT` | *Integer* | Timeout beim Senden von Daten |
| `PING_TIMEOUT` | *Integer* | Maximale Zeitspanne zwischen einer *Ping*-Nachricht zum Server und dessen Antwort, oder 0 um den Heartbeat-Mechanismus zu deaktivieren (siehe Text) |
| `PROCESSING_TIMEOUT` | *Integer* | Maximale Zeit für die interne Klassifizierung |
| `PROFILE_UPDATE_TIMEOUT` | *Integer* | Zeit, nach der automatisch das Benutzerprofil vom Server aktualisiert wird, oder 0 zum Deaktivieren |

Tabelle 4.2: Optionen der Client-Bibliothek

aktivieren der SBC-Funktionalität, ohne weitere Funktionsaufrufe im Code ändern zu müssen; Befehle zum Klassifizieren von Daten werden in diesem Modus ignoriert. In der Einstellung *Server* wird nur die serverbasierte Klassifikation verwendet. Dazu wird eine Verbindung zum Server hergestellt, sobald diese benötigt wird. Es werden die Verbindungsdaten verwendet, die beim Aufruf von `CreateConnection` angegeben wurden. Bei dem Server kann es sich um den SBC-Server (vgl. Abschnitt 4.1.3) oder eine andere Software handeln, welche über TCP/IP angebunden ist und zum Datenaustausch das SBC-Netzwerkprotokoll (vgl. Abschnitt 4.1.5) verwendet. Der Modus *Local* gibt an, dass nur das Klassifikationsmodul verwendet und keine Netzwerkverbindung aufgebaut werden soll. Dieser Modus ist immer verfügbar. Im Modus *Auto* wird zuerst die Verbindung zu dem angegeben Remote-Server versucht. Kommt diese nicht zustande oder bricht ab, so wird stattdessen die Klassifizierung auf dem mobilen Gerät verwendet. Dabei handelt es sich um einen typischen Fallback-Mechanismus. Der Server wird vorgezogen, da er über mehr Rechenleistung verfügt und möglicherweise für den Anwendungskontext oder die Umgebung optimierte Klassifizierer enthält. Schließlich existiert noch ein weiterer Modus *Parallel*, in dem die lokale und servergestützte Verarbeitung derselben Daten gleichzeitig erfolgen. Dadurch wird einerseits sichergestellt, dass im Falle eines Verbindungsproblems die Wartezeit bis zum Feststellen dieses Problems nicht ungenutzt verstreicht, aber andererseits benötigt das interne Klassifikationsmodul in jedem Fall gewisse CPU-Ressourcen, auch wenn letztlich das Ergebnis des Servers verwendet wird. Aus den schon beim Modus *Auto* genannten Gründen wird auch hier der Antwort des Servers eine höhere Priorität beigemessen. Derzeit sind lokale und servergestützte Klassifikation entkoppelt, d.h. das Benutzerprofil enthält zu jedem Zeitpunkt nur die Daten aus maximal *einer* der beiden möglichen Quellen. In einer zukünftigen Version sollen jedoch die Ergebnisse beider Varianten in ein Bayessches Netz eingegeben werden, so dass das endgültige Benutzerprofil dynamisch von allen verfügbaren Eingaben abhängt.

Normalerweise erfolgt die Initialisierung der jeweils verwendeten Verbindungsart (intern, extern oder beides) automatisch, sobald Daten zur Übermittlung vorliegen. Die Anwendung kann dies durch den Aufruf der Funktionen `Connect` und `Disconnect` aber auch selbst steuern. So könnte eine Anwendung, bei der die Klassifikation nur eine untergeordnete Bedeutung für den Betrieb spielt, beim Abbruch der Verbindung zum Server den Wiederaufbau dieser Verbindung komplett aussetzen oder einige Zeit verzögern. Eine andere Applikation könnte den Benutzer fragen, ob die Verbindung wiederhergestellt oder stattdessen die lokale Klassifikation verwendet werden soll.

Das Erkennen eines Verbindungsabbruchs in einem Funknetzwerk allein durch die Methoden der Netzwerkschnittstelle ist häufig nicht verlässlich. Um bei der Verwendung eines Servers derartige Abbrüche zu erkennen, wird daher ein so genannter *Heartbeat-Mechanismus* eingesetzt. Dabei wird in regelmäßigen Intervallen ein einfaches Datenpaket (*Ping*) zum Server gesendet und auf dessen Antwort gewartet. Wird die Antwort

nicht innerhalb einer konfigurierbaren Zeitspanne erhalten, wird eine Zeitüberschreitung vermerkt. Treten mehrere Zeitüberschreitungen in Folge auf, so wird dies als Verbindungsabbruch interpretiert.

Die Daten von der Anwendung werden in Form von *Segmenten* an die Bibliothek übergeben. Jedes Segment kann mehrere zusammengehörige Informationen bestimmter vorgegebener Typen enthalten. Audiodaten, die z.B. vom Klassifikationsmodul verarbeitet werden, sind dabei eine mögliche Art von Informationen in einem Segment. Weitere Datentypen können übertragen werden, um zusätzliche serverbasierte Dienste zu realisieren. Ein wichtiger Datentyp sind *Metadaten*. Dabei handelt es sich um Paare aus Name und Wert, die zur Steuerung des Servers verwendet werden können. Sie dienen beispielsweise zur Übermittlung genauer Zeitstempel an den Server oder zur Ankündigung eines Sprecherwechsels. Ein neues Segment wird durch den Aufruf von `NewTransmission` angelegt und durch `Send` oder `SendAsync` gesendet. Die Funktion `SendAsync` unterscheidet sich von `Send` dadurch, dass sie *asynchron* arbeitet, d.h. das Segment nur in eine Sendewarteschlange einreiht und die Kontrolle an den Aufrufer zurückgibt, bevor der Sendevorgang abgeschlossen ist. Dadurch tritt für den Anwender keine Verzögerung bei der Übertragung auf. Beim Anlegen eines neuen Segments wird ein eventuell vorhandenes, aber noch nicht gesendetes Segment verworfen. Alle in Tabelle 4.1 auf Seite 120 aufgeführten Funktionen mit dem Präfix `Add...` dienen dazu, Daten zum aktuellen Segment hinzuzufügen, bevor dieses gesendet wird. SBC schränkt die Art und Anzahl dieser Daten von Seiten des Protokolls nicht ein, allerdings können die Client-Bibliothek und der SBC-Server nur bestimmte Daten interpretieren.

`AddFile` und `AddRaw` fügen jeweils binäre Daten zum Segment hinzu, welche entweder aus einer Datei oder einem Byte-Datenfeld gelesen werden. Der SBC-Server und das Klassifikationsmodul versuchen diese Daten immer als Audiodaten zu interpretieren und leiten eine Klassifikation ein, wenn solche Daten eintreffen. `AddFileWH` und `AddRawWH` arbeiten analog, jedoch fügen sie den Daten noch Kopfdaten des WAVE-Formats hinzu, die aus den angegeben Parametern übernommen werden. Dies ist nur geeignet für unkomprimierte PCM-Audiodaten. Durch `AddMeta` können benutzerdefinierte Metadaten hinzugefügt werden. Der SBC-Server ignoriert alle bis auf die in Abschnitt 4.1.5 angegebenen Metadaten. Eine andere Server-Implementierung oder eine zukünftige Version des SBC-Servers könnten jedoch weitere Werte verwenden. `AddSpeakerSwitchNotification` fügt eine Meta-Information zum Segment hinzu, dass es sich bei den nachfolgenden Audiodaten um Äußerungen eines neuen Sprechers handelt. Dies bewirkt eine Rücksetzung des dynamischen Bayesschen Netzes. Es handelt sich dabei um die einzige Meta-Information, die bei der lokalen Klassifikation interpretiert wird.

Das SBC-Klassifikationsmodul wird durch statisches Linken in die Client-Bibliothek integriert. Eine zwischengeschaltete Klasse erhält die von der Anwendung erstellten Segmente und wandelt sie in Aufrufe an die `ProcessingPipeline` um, sofern die lo-

kale Klassifikation aktiviert ist. Momentan werden jeweils die im Klassifikationsmodul enthaltenen Klassifizierer verwendet, so dass entweder die Bibliothek bei der Auswahl eines neuen Moduls mit kompiliert oder ein ausreichend generisches Modul verwendet werden muss. Sowohl das Protokoll als auch das Klassendesign von Modul und Client-Bibliothek, sehen die Möglichkeit vor, Klassifizierer bei erstmaliger Verbindung vom Server auf das Gerät zu übertragen (vgl. Methode `FromString` in Abschnitt 3.7.3). Die Funktion `AddClassifierRequest` dient dazu, eine solche Anforderung an den Server zu stellen. Dabei kann der Bezeichner des Klassifizierers angegeben werden, welcher auf dem Gerät benötigt wird. Ein solcher Klassifizierer würde zumindest für die Dauer der Anwendung und möglicherweise noch darüber hinaus in einem Cache erhalten bleiben. In einem typischen Fall wird für alle Knoten der zweiten Ebene auf dem Gerät eine Anfrage an den Server gestellt.

Nach dem Aufruf von `Send` bzw. `SendAsync` erfolgt bei Verwendung der Klassifikation auf einem Server keine automatische Benachrichtigung über die Aktualisierung des Benutzerprofils. Die jeweils aktuellen Daten müssen vom Client manuell erfragt werden (*Poll-Update*). Mittels der Option `PROFILE_UPDATE_TIMEOUT` (s. Tabelle 4.2 auf Seite 121) kann diese Abfrage zumindest teilweise automatisiert werden, da sie dann jeweils eine bestimmte (einstellbare) Zeit nach dem Sendevorgang im Hintergrund durchgeführt wird. Intern wird dabei ein Segment erzeugt und `AddProfileRequest` aufgerufen, welches eine Profilanfrage erzeugt. Als Parameter kann dieser Funktion optional der Profilpfad übergeben werden, welcher vom Server ausgelesen werden soll. Wird der Parameter nicht angegeben, so werden alle verfügbaren Informationen des Benutzermodells aktualisiert. Bei lokaler Klassifikation ist eine solche Anfrage nicht erforderlich, da alle Ergebnisse direkt in das lokal gespeicherte Benutzerprofil übertragen werden.

Die Ansteuerung weiterer serverbasierter Dienste, welche nicht in direktem Zusammenhang mit SBC stehen, ist entweder über `AddMeta` oder die Funktion `AddActionRequest` möglich. In beiden Fällen werden benutzerdefinierte Metadaten gesendet, welche über die Funktionsparameter angepasst werden können. Die Funktion `WaitForResult` kann aufgerufen werden, um das Ergebnis einer solchen Anfrage vom Server abzurufen. Dieses wird als Zeichenfolge über einen Ausgabe-Parameter zurückgeliefert. Ein Beispiel für einen solchen Dienst wäre Spracherkennung, wie sie im SHOPASSIST z.B. zur Steuerung der Anwendung verwendet wird. Dadurch, dass diese auf den Server verlagert wird, kann die Performanz erhöht werden. Es ist kein zusätzlicher Serverrechner und keine zusätzliche Verbindung erforderlich, da alle benötigten Anfragen über das SBC-Protokoll abgewickelt werden. In diesem Fall würde die Audiodatei, welche schon zur Sprecherklassifikation verwendet wird, zusätzlich mit einer Aktionsanfrage in Form von Metadaten versehen, welche die Spracherkennung einleitet. Das Ergebnis des Spracherkenners kann von der Anwendung dann (ggf. synchron) mit `WaitForResult` abgerufen und verarbeitet werden.

### 4.1.3 SBC-Server

Bei dem Server, welcher zusammen mit dem ShopAssist oder einer anderen Anwendung eingesetzt wird, um die Klassifizierung zu übernehmen, handelt es sich um den unveränderten m3i-Server, welcher in Müller (2005, S. 207ff) ausführlich beschrieben ist. Er wird in dieser Arbeit als sbc-Server bezeichnet, um seine Integration in die neue Architektur zu verdeutlichen. In seinem aktuellen Stadium ist er jedoch als Prototyp zu betrachten, welcher im Rahmen von sbc die Einsatzmöglichkeiten und Vorteile servergestützter Klassifizierung demonstriert.

Nach dem Start des Programms wird der Dienst Live-Klassifikation gestartet. Damit wird auch der *Protokoll-Manager* geladen, welcher auf eingehende Netzwerkverbindungen an einem zuvor festgelegten Anschluss wartet. Ist der Server z.B. über einen WLAN Access Point mit einem Funknetzwerk verbunden, so kann er von mobilen Geräten mit sbc-Client-Software erreicht werden, z.B. von der ShopAssist-Anwendung.

Nach einem initialen *Handshake-Vorgang* (vgl. Abschnitt 4.1.5) weist der Server dem Client eine interne ID zu, um für weitere Geräte verfügbar zu bleiben, und legt ein Benutzerprofil an. Danach werden alle Segmente des Clients gelesen und verarbeitet. Bei eingehenden Audiodaten werden diese klassifiziert, wobei dies je nach Konfiguration des Servers entweder lokal oder verteilt auf mehreren Cluster-Rechnern erfolgen kann. Auf Anfrage wird das aktuelle Benutzerprofil an den Client gesendet.

### 4.1.4 Gegenüberstellung SBC-Server und Klassifikationsmodul

Wie aus Abschnitt 3.8 bekannt ist, lässt sich das sbc-Klassifikationsmodul beliebig in andere Applikationen einbetten und ist daher grundsätzlich auch für den Einsatz in einem Klassifikations-Server geeignet. An dieser Stelle soll kurz auf einige Unterschiede zwischen den beiden Varianten eingegangen werden sowie deren jeweilige Vor- und Nachteile.

**Netzwerk-Anbindung:** Der sbc-Server enthält bereits eine vollständige Netzwerkschnittstelle, welche *Java Sockets* verwendet und sehr robust ist. Der Server nutzt diese Schnittstelle exklusiv. Sollen über die gleiche Schnittstelle noch andere Daten übertragen werden oder soll statt TCP/IP ein anderes Protokoll eingesetzt werden, so muss das Klassifikationsmodul in eine entsprechende Netzwerkschnittstelle integriert werden. Dabei ist zu bedenken, dass auch das darüber liegende sbc-Protokoll erst implementiert werden muss.

**Klassifizierungsverfahren:** Für den sbc-Server stehen durch das *WEKA*-Paket von vorne herein sehr viel mehr Algorithmen zur Verfügung. Für die eingebettete Plattform müssen diese Methoden erst neu implementiert werden.

**Performanz/Skalierbarkeit:** Die einfache Ausführungsgeschwindigkeit des Klassifikationsmoduls ist aufgrund der zugrunde liegenden Sprache und verschiedener Optimierungen zwar höher als die des SBC-Servers, doch enthält der SBC-Server die Möglichkeit des *Clusterings*, was zu einer besseren Skalierbarkeit führt.

**Anpassbarkeit:** Da der eigentliche Server nicht Bestandteil des Moduls ist, bietet dieses eine sehr große Flexibilität in Bezug auf die Gestaltung der eigentlichen Netzwerkschnittstelle. Um jedoch nur einen zusätzlichen Dienst zu integrieren, wie z.B. Spracherkennung, genügt es in der Regel, diesen Dienst bei dem Server auf dem Blackboard zu registrieren.

**Setup-Aufwand:** Da der SBC-Server eine eigenständige Anwendung ist, lässt er sich viel schneller einsetzen als das Modul. Dieses muss als Komponente erst in eine Host-Anwendung integriert werden.

**Training:** Der SBC-Server ist direkt mit einer Datenbank verbunden und kann Klassifizierer beim Programmstart trainieren. Die jeweils aktuellen Klassifizierer können über das Netzwerk-Protokoll an verbundene Clients weitergegeben werden. Letzteres ist zwar auch beim eingebetteten Modul für eine zukünftige Version geplant, allerdings muss im Fall von Änderungen an Datenbank oder Modul-Konfiguration jeweils erst ein neues Modul erzeugt werden.

Aus den genannten Punkten resultiert, dass beide Server-Varianten eigene Vorteile besitzen. Es kommt auf die Anforderungen der Anwendung an, welche Gründe letztlich überwiegen. Für den SHOPASSIST waren wichtige Gründe, die für den SBC-Server sprechen, die bereits integrierte Netzwerkschnittstelle, die Verfügbarkeit einer größeren Anzahl von Algorithmen und das einfachere Training. Dies ist natürlich auch unter dem Gesichtspunkt zu betrachten, dass es sich bei der Applikation um einen Demonstrator handelt.

### 4.1.5 Netzwerkprotokoll

Das eigens für die Kommunikation zwischen Client und Server entwickelte Netzwerkprotokoll ist ein nachrichtenorientiertes Protokoll. Für die Richtung vom Client zum Server wird jede Nachricht durch einen binären Datenblock repräsentiert, welcher mit einer Kennung für den Typ der nachfolgenden Daten beginnt. Je nach Typ werden die Daten unterschiedlich interpretiert. Zeichenfolgen werden übertragen, indem zuerst die Länge der Folge und dann die einzelnen *Unicode*[1]-Zeichen gesendet werden. Für Nachrichten

[1] Standard zur Repräsentation von Zeichen verschiedenster Sprachen. Jedes Zeichen wird durch 2 Byte dargestellt.

vom Server an den Client werden ausschließlich Text-Nachrichten verwendet, da sich alle benötigten Informationen auf diese Weise darstellen lassen.

Folgende Typen von Nachrichten kann der Client an den Server senden:

### Authentifizierung (Handshake):

Dies ist die erste Nachricht, welche an den Server gesendet werden muss, bevor andere Nachrichten gesendet werden können. Erst beim Erhalt dieser Nachricht legt der Server ein Sprecherprofil für den Benutzer des Client-Geräts an. Der Client teilt dem Server hier auch mit, unter welchem Namen das Gerät angesprochen wird, um z.B. bei einem Verbindungsabbruch später mit dem bereits erstellten Profil weiterarbeiten zu können. Zur besseren Einstellung auf den Client und zu statistischen Zwecken wird auch der Typ des Geräts übermittelt, welches die Verbindung aufnimmt, z.B. *PocketPC* oder *Smartphone*. Eine Optimierung der Dienste, abhängig von dem Gerätetyp, findet in der aktuellen Version des SBC-Servers noch nicht statt. Die Antwort des Servers auf diese Nachricht ist eine `hello`-Nachricht.

| Datentyp | Inhalt |
|---|---|
| Int32 | 6 (`SBC_MSGTYPE_HELLO`) |
| Int32 + WChar[] | Name des Clients (falls bekannt, sonst leer) |
| Int32 | Gerätetyp (0=unbekannt, 1=Desktop, 2=PocketPC, 3=Smartphone) |

### Abmeldung:

Dies ist die bevorzugte Art und Weise, eine Verbindung zum Server zu beenden. Hierbei wird das Sprecherprofil direkt gelöscht.

| Datentyp | Inhalt |
|---|---|
| Int32 | 7 (`SBC_MSGTYPE_BYE`) |

### Binäre Daten (Paket):

Dient zur Übertragung von Bytedaten an den Server, z.B. einer Audiodatei. Die Verarbeitung, welche auf dem Server erfolgt, hängt von den auf dem Server installierten Diensten und den zuvor übertragenen Metadaten ab. Zur Klassifizierung erwartet der SBC-Server PCM-Daten einschließlich Formatbeschreibung.

| Datentyp | Inhalt |
|---|---|
| Int32 | 1 (`M3I_MSGTYPE_BINARY_PACKET`) |
| Int32 | Länge der Daten in Bytes |
| Byte[] | Binärdaten |

### Binäre Daten (Stream):

Diese Nachricht ist vorgesehen, um Audiodaten als Stream zu übertragen. Dies ist nützlich, wenn längere Äußerungen übertragen werden sollen und die Klassifikation bereits vor Fertigstellung der Aufnahme oder Eintreffen der kompletten Nachricht eingeleitet werden soll. Es werden daher bereits Daten gesendet, während der Client noch die Aufnahme fortsetzt. Da die Länge der Nachricht nicht im Voraus bekannt ist, wird statt der Längenangabe im Paket eine Endmarkierung im Datenstrom verwendet. Streaming wird von der Client-Bibliothek und dem SBC-Server gegenwärtig noch nicht unterstützt, ist jedoch eine für nachfolgende Versionen geplante Funktion. Um derweil eine ähnliche Funktionalität mit Paketdaten zu erreichen, können die Daten vom Client aus in kleineren Teilpaketen gesendet werden. In beiden Fällen ist zu bedenken, dass die berechneten Sprachmerkmale je nach Länge des Ausschnitts aus einer Äußerung leicht variieren können. Es ist aber davon auszugehen, dass solche Abweichungen auf der zweiten Ebene wieder ausgeglichen werden.

| **Datentyp** | **Inhalt** |
|---|---|
| Int32 | 3 (`SBC_MSGTYPE_BINARY_STREAM`) |
| Byte[] | Binärdaten mit Endmarkierung |

### Metadaten:

Die Nachricht einhält ein einzelnes Name-Wert-Paar in Textform. Manche Metainformationen (z.B. Zeitstempel) beziehen sich auf Audiodaten. In diesem Fall wird die Information dem nächsten binären Datenpaket zugeordnet.

| **Datentyp** | **Inhalt** |
|---|---|
| Int32 | 2 (`SBC_MSGTYPE_META`) |
| Int32 + WChar[] | Name des Attributs |
| Int32 + WChar[] | Wert des Attributs |

Folgende Attribute werden derzeit vom SBC-Server interpretiert:

| **Attribut** | **Wert** |
|---|---|
| `new_speaker` | 1, um einen Sprecherwechsel anzukündigen. Dadurch wird das Benutzerprofil zurückgesetzt. |
| `requested_action` | Name einer Aktion, welche durch einen zusätzlichen registrierten Dienst ausgeführt werden soll, z.B. Spracherkennung |
| `timestamp` | Zeitstempel für die nachfolgende Äußerung |

#### Ping:

Diese Nachricht enthält keine weiteren Daten. Sie ist lediglich als Aufforderung für eine Antwort vom Server bestimmt und Bestandteil des Heartbeat-Mechanismus (vgl. Abschnitt 4.1.2).

| **Datentyp** | **Inhalt** |
|---|---|
| Int32 | 4 (`SBC_MSGTYPE_PING`) |

#### Klassifizierer-Anfrage:

Wird verwendet, um einen neuen Klassifizierer oder die Aktualisierung eines vorhandenen Klassifizierers vom Server zu beziehen. Der Server antwortet in diesem Fall mit einer `classifier`-Nachricht.

| **Datentyp** | **Inhalt** |
|---|---|
| Int32 | 5 (`SBC_MSGTYPE_REQ_UPDATE_CLASSIFIER`) |
| Int32 + WChar[] | Klassifizierer-ID |

#### Abfrage des Benutzerprofils:

Über diesen Nachrichtentyp wird das aktuell auf dem Server gespeicherte Benutzerprofil abgefragt. Durch Angabe eines Parameters kann die Ausgabe auf einen bestimmten Profilpfad eingeschränkt werden. Andernfalls werden alle verfügbaren Informationen an den Client gesendet. Die angefragten Informationen werden in Form von einer oder mehreren `profile`-Nachrichten an den Client übermittelt.

| **Datentyp** | **Inhalt** |
|---|---|
| Int32 | 8 (`SBC_MSGTYPE_REQ_UPDATE_PROFILE`) |
| Int32 + WChar[] | Profilpfad oder leere Zeichenfolge |

Folgende Nachrichten können vom Server an den Client gesendet werden:

#### hello:

Diese Nachricht fungiert als Anmeldebestätigung und schließt den Handshake-Vorgang ab. Die Syntax lautet:

`hello <Client-Name> iam <Server-Name>`

*Client-Name* ist der Name, den der Client vom Server zugewiesen bekommt. Dies kann der gleiche Name sein, mit welchem sich der Client zuvor beim Server gemeldet hat. *Server-Name* ist eine Identifikation des Servers, die z.B. dem Benutzer zu Informationszwecken mitgeteilt werden kann.

**Beispiel:** `hello Pocket_PC iam SBC_Server`

**profile:**

Dient zur Übermittlung des aktuellen Sprecherprofils an den Client. Für jede Sprechereigenschaft wird eine eigene Nachricht verwendet.

`profile <Profilpfad>=<Wert>`

*Profilpfad* enthält den Profilpfad der Sprechereigenschaft. *Wert* kann jede Zeichenfolge bzw. Zahl sein, welche den Wert der Eigenschaft darstellt.

**Beispiel:** `profile user::gender::female=0.8`

**classifier:**

Durch diese Nachricht kann ein Klassifizierer an den Client übertragen werden, welcher von diesem angefordert wurde. Dabei wird eine codierte Zeichenfolge zur Beschreibung verwendet, die auf der Seite des Clients geparst werden muss. Nicht alle Klassifizierer lassen sich auf diese Weise übertragen. Ist der angeforderte Klassifizierer auf dem Server verfügbar und serialisierbar, so lautet die Syntax der Nachricht:

`classifier <Klassifizierer-ID> <Typ> <Codezeichenfolge> <Sprachmerkmale> <Klassennamen>`

*Klassifizierer-ID* ist der Bezeichner des angeforderten Klassifizierers. Sie wird verwendet, um dem Ergebnis einen Knoten im Bayesschen Netz zuzuordnen. Der *Typ* beschreibt die Implementierung des Klassifizierers, damit der Codestring korrekt interpretiert werden kann. Tabelle 3.6 auf Seite 102 enthält die möglichen Werte. *Codezeichenfolge* ist die serialisierte Text-Repräsentation für den Klassifizierer. *Sprachmerkmale* enthält eine Liste der aus den Audiodaten benötigten Merkmale in der vom Klassifizierer erwarteten Reihenfolge. *Klassennamen* definiert Bezeichnungen für die Klassen, die den vom Klassifizierer gelieferten Ergebnissen zugeordnet werden.

Ist der angeforderte Klassifizierer nicht verfügbar, so wird ein Fehler in der folgenden Form an den Client gemeldet:

`classifier <Klassifizierer-ID> unavailable`

Das folgende Beispiel sendet einen Geschlechtsklassifizierer an den Client:

```
classifier user::gender main:N13860000;N13860000:4,164.143420053499:
L-N26d4f11:L-Nc5c3ac20;N26d4f11:4,135.904189192268:L-N1662dc82:
L-N161d36b8;N1662dc82:0,0.0252816261612662:C-1:L-N147c5fc3;N147c5fc3:
0,0.0294405188383751:C-1:L-N1174b074;N1174b074:3,0.181494516057268:
L-N3eca905:C-1;N3eca905:4,112.294029347258:L-N64dc116:L-N1ac1fe47;
N64dc116:2,0.0676532066056226:C-1:C-0;N1ac1fe47:2,0.0842698333868203:
C-0:C-1;N161d36b8:2,0.0858304026720833:L-N17f1ba39:C-1;N17f1ba39:0,
0.0202226683664841:L-N1ef8cf310:L-N1d520c412;N1ef8cf310:4,
151.703008221803:C-1:L-Necd7e11;Necd7e11:0,0.00858860463548902:C-1:C-0;
N1d520c412:2,0.0653707517488461:L-N15a3d6b13:L-N16fd0b715;N15a3d6b13:4,
153.041843016558:L-N1764be114:C-0;N1764be114:3,0.0772023896307453:C-1:
C-0;N16fd0b715:0,0.0377747032777348:L-N1ef9f1d16:L-Nb753f817;N1ef9f1d16:
2,0.0740389913484848:C-0:C-1;Nb753f817:1,0.0361345695424885:C-0:
L-N1e9cb7518;N1e9cb7518:4,146.0992889243:C-0:L-N2c84d919;N2c84d919:4,
157.325212119803:C-1:C-0;Nc5c3ac20:2,0.100948051699542:L-N1b16e5221:
L-N4413ee24;N1b16e5221:4,180.808790447296:L-N1c1ea2922:C-0;N1c1ea2922:3,
0.0614579559446241:L-N1f436f523:C-0;N1f436f523:0,0.0115656239000967:C-1:
C-0;N4413ee24:3,0.243164826646892:L-N1786e6425:C-1;N1786e6425:4,
204.158826802576:C-1:C-0; 5,mean_pitch,jitterRAP,jitterPPQ,shimmerAPQ5,
shimmerAPQ11 2,male,female
```

**pong:**

Dies ist die Antwort des Servers auf eine *Ping*-Anfrage des Clients. Durch ihren Erhalt wird bestätigt, dass die Verbindung funktioniert und der Server bereit ist.

In Abb. 4.3 ein Beispiel für eine mögliche Kommunikation zwischen der Client-Bibliothek und dem SBC-Server als Sequenzdiagramm dargestellt. Auf den Heartbeat-Mechanismus wurde in diesem Diagramm aus Gründen der Übersichtlichkeit verzichtet.

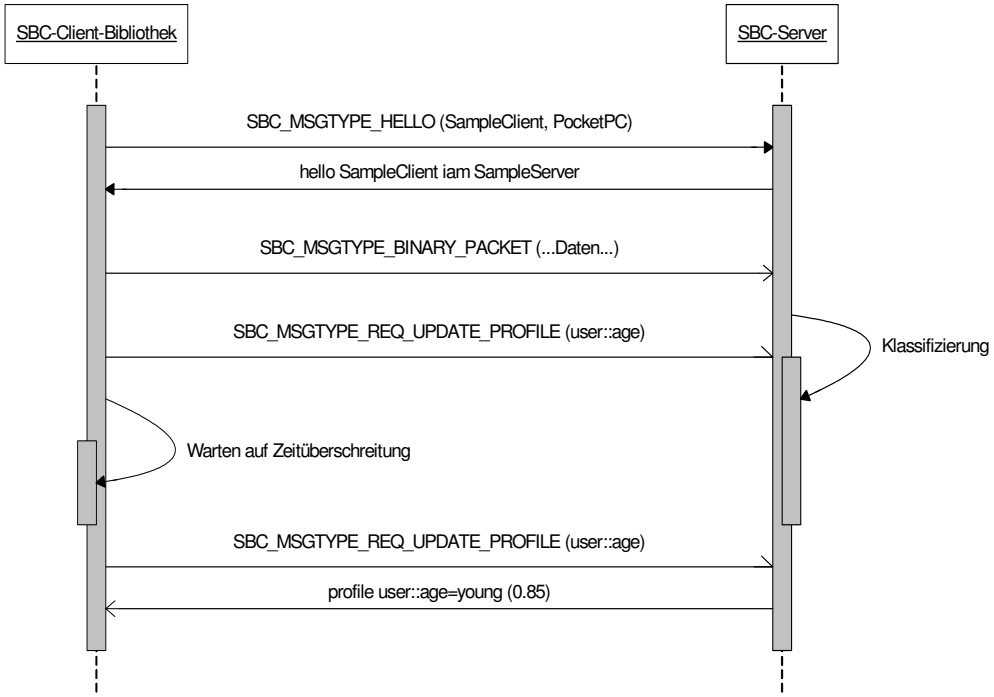



Abbildung 4.3: Beispiel für eine mögliche Kommunikation zwischen Client und Server über das SBC-Protokoll

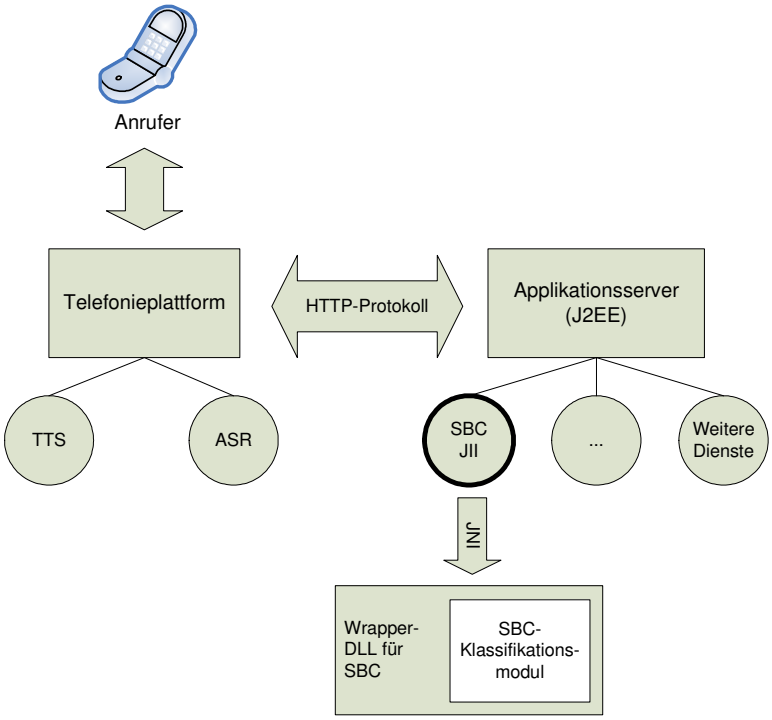



Abbildung 4.4: Architektur des Callcenter-Szenarios

## 4.2 SBC für Sprachapplikations-Server

Als zweites Beispiel für eine mögliche Anwendung von Sprecherklassifikationsmodulen soll ein Projekt dienen, welches für ein Telekommunikationsunternehmen entwickelt wurde. Abb. 4.4 stellt die Architektur dieses Systems dar.

Ein Anrufer ist über eine bestehende Infrastruktur (Telefonzentrale) mit einer *Telefonieplattform* verbunden. Diese erlaubt das Aufzeichnen von Äußerungen des Anrufers als Audiodateien, automatische Spracherkennung, Umwandlung von Text in Audiodaten (*Text-to-Speech*) und Abspielen von Audiodaten für den Anrufer. Alle aufgezeichneten Daten werden an einen *Sprachapplikationsserver* weitergeleitet, welcher den Sprachdialog steuert. Dieser Server läuft unter der *Java 2 Enterprise Platform* (J2EE) und kann mehrere Dienste zur Unterstützung des Dialogsystems integrieren.

Ein Dialog könnte folgendermaßen ablaufen: Ein Anrufer bei einer technischen Supporthotline hört sich eine kurze Einleitung an, an deren Ende er aufgefordert wird, sein Problem zu schildern. Daraufhin spricht der Anrufer einige Sätze, welche von der Telefonieplattform aufgezeichnet werden. Durch die Spracherkennung kann ein Abgleich mit einer Datenbank vordefinierter Begriffe für bestimmte häufig auftretende Probleme erfolgen. Bei einem Treffer wird der entsprechende Lösungsvorschlag abgespielt und

beim Anrufer nachgefragt, ob das Problem dadurch behoben wurde. Falls kein Treffer gefunden wurde oder das Problem noch besteht, erfolgt die Vermittlung an einen menschlichen Agenten.

Durch SBC/AGENDER kann dieses Szenario auf zwei Arten verbessert werden. Die erste Möglichkeit besteht darin, die Alters- und Geschlechtsinformation in die Suche nach möglichen Lösungen einzubeziehen. So wäre es denkbar, dass bestimmte Lösungsvorschläge nur bei älteren Menschen abgespielt werden, welchen die im Alter häufig zunehmende Vergesslichkeit als Problemursache zugrunde liegt. Analog dazu können solche Datenbankeinträge, welche sich mit Funktionen eines Produkts befassen, die im Allgemeinen nur bei Kindern und Jugendlichen Verwendung finden, bei Erwachsenen ausgeschlossen werden. Statt dem vollständigen Ausschluss eines Eintrags kann auch lediglich die Priorität verringert werden.

Die zweite Verbesserungsmöglichkeit besteht darin, die durch eine Klassifizierung der Problembeschreibung gewonnenen Informationen dazu zu verwenden, eine spätere Weiterleitung an einen Berater gezielt zu gestalten (vgl. Beschreibung von ACD in Abschnitt 1.1).

In beiden beschriebenen Fällen wird eine von der Telefonieplattform aufgezeichnete Äußerung zur Klassifikation des Anrufers verwendet. Dazu wird die Audiodatei zuerst an den Applikationsserver weitergeleitet. Dessen Geschäftslogik entscheidet darüber, in welchen Fällen eine Klassifizierung stattfindet. Da der Server in *Java* implementiert ist, kann das Klassifikationsmodul nicht direkt eingebunden werden. Es wird eine Schnittstelle benötigt, welche das Marshalling der Daten zwischen *Java* und *C++* übernimmt.

Die Wahl von *Java* von Seiten des Auftraggebers wurde primär aus Gründen der Plattformunabhängigkeit der generellen Architektur (d.h. auch solche Versionen ohne SBC) getroffen. Zur Optimierung der Geschwindigkeit scheint jedoch die Auslagerung des Klassifizierungscodes in *C++* gerechtfertigt. Solche oder ähnliche Vorgehensweisen werden in der Praxis häufig angewandt, vgl. beispielsweise Bramberger et al. (2005, S. 107).

Für die eigentliche *Java*/*C++*-Kommunikation bietet *Java* eine Technologie mit der Bezeichnung *Java Native Interfaces* (JNI) an, durch welche sich Plattformaufrufe durchführen lassen, also Aufrufe von Funktionen in dynamischen Bibliotheken. Da es sich bei dem SBC-Klassifikationsmodul an sich nicht um eine dynamische Bibliothek handelt, ist eine Wrapper-DLL (vgl. Abschnitt 3.8.3) erforderlich, die das Klassifikationsmodul kapselt. Damit von Seiten des Auftraggebers keine JNI-Aufrufe durchgeführt werden müssen, wird zusätzlich eine *Java*-Schnittstelle für den Applikationsserver zur Verfügung gestellt. Diese trägt den Namen SBC JAVA INTEGRATION INTERFACE (JII). Neben einer vereinfachten Handhabung bietet sich durch sie auch die Gelegenheit, einige projektspezifische Funktionen ohne Rückgriff auf JNI direkt in *Java* zu implementieren.

Beim Aufruf von JNI-Methoden ist zu bedenken, dass ein wesentlich größerer Aufwand

entsteht als beim Aufruf von Funktionen innerhalb von *Java* oder *C++*. Operationen, die zusätzliche Zeit beanspruchen, sind u.a. die Konvertierung von Daten, Fixierung von Speicherbereichen, Thread-Synchronisierung und Auflösung von Funktionsverweisen. Die Anzahl der JNI-Aufrufe sollten also möglichst gering gehalten werden.

### 4.2.1 Wrapper-DLL

Das SBC-Klassifikationsmodul arbeitet mit *C++*-Objekten wie z.B. der Klasse `ProcessingPipeline`. Diese Objekte können jedoch nicht direkt mit *Java* ausgetauscht werden. Daher werden stattdessen Bezeichner (oder Handles) verwendet, um der DLL bei nachfolgenden Aufrufen mitzuteilen, mit welcher Pipeline gearbeitet werden soll.

Über die Funktion `newUser` wird eine neue `ProcessingPipeline` angelegt und deren Handle zurückgeliefert. Analog dazu kann diese mit `destroyUser` wieder zerstört werden. Zum Ausführen der Klassifizierung wird `processUtterance` mit einem Byte-Datenfeld aufgerufen, welches die Audiodaten in einem bestimmten zuvor festgelegten Format enthält. Um die Ergebnisse der Klassifizierung abzurufen, wird die Funktion `getResults` verwendet, welche intern für jeden Eintrag des Zeichenfolgenwörterbuchs eine *Java*-Funktion aufruft (*Callback*).

Teile der Wrapper-DLL, im Speziellen die Header-Datei für die öffentliche Schnittstelle, welche auch den Teil von JNI für *C* einschließt, wurden mit dem Hilfsprogramm *javah* aus dem *Java Development Kit* (JDK) automatisch generiert.

Eine DLL kann von mehreren Anwendungen gleichzeitig genutzt werden. Unter dem Windows-Betriebssystem wird jeder Prozess, der auf eine DLL zugreift, dieser beim Laden und Entladen durch Aufruf von `DllMain` mitgeteilt. Die SBC-Wrapper-DLL verwendet diese Funktion zur Verwaltung eines Referenzzählers, welcher angibt, wie viele Prozesse auf die DLL zugreifen. Erreicht der Zähler 1, so werden die internen Datenstrukturen (z.B. für Tracing) initialisiert. Sinkt der Zähler wieder auf 0, wird eine Speicherbereinigung durchgeführt und das Tracing beendet.

### 4.2.2 Java Integration Interface

Auf der *Java*-Seite bildet das Paket `integrationinterface` die Schnittstelle zur Steuerung von SBC. Das Paket enthält drei Klassen: `Classifier`, `ClassificationResult` und die Ausnahme `ClassificationFailedException`.

Für die UML-Darstellung von `Classifier` kann Abb. 4.5 betrachtet werden. `Classifier` kapselt im Wesentlichen 1:1 die `ProcessingPipeline`-Objekte des Klassifikationsmoduls. Die Bezeichnung `Classifier` wurde gewählt, da sie für Personen ohne tiefgehende Kenntnis von SBC und AGENDER intuitiver ist als `ProcessingPipeline` und auch auf den gesamten Prozess bezogen werden kann. Beim Erstellen eines `Classifier`-Objekts wird automatisch eine `ProcessingPipeline` erzeugt und die *SpeakerID*-Handle

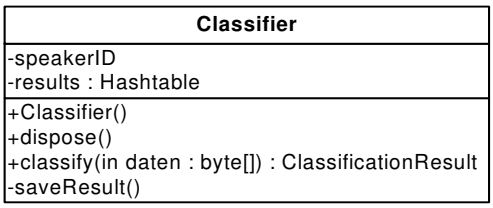



Abbildung 4.5: UML-Diagramm der Java-Klasse `Classifier`

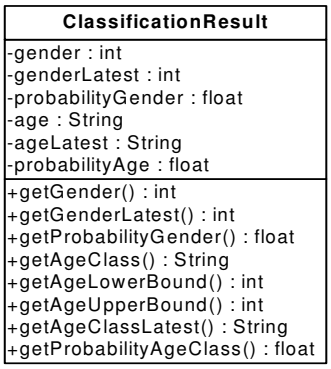



Abbildung 4.6: UML-Diagramm der Java-Klasse `ClassificationResult`

der Wrapper-DLL in einem privaten lokalen Feld gespeichert, so dass diese vollständig transparent für die Applikation ist. Da *Java* allerdings keine Destruktoren kennt, welche wie in *C++* automatisch aufgerufen werden, ist es wichtig, die `ProcessingPipeline` manuell durch Aufruf der Methode `dispose` zu zerstören und den Speicher freizugeben, da andernfalls ein Speicherleck auftritt.

Die Methode `classify` leitet ein Byte-Datenfeld zur Klassifizierung weiter an die Funktion `processUtterance` der Wrapper-DLL. Schlägt die Klassifikation fehl, so wird die Ausnahme `ClassificationFailedException` ausgelöst, welche von der Applikation abgefangen und behandelt werden kann. Bei erfolgreicher Klassifikation liest `classify` danach das Ergebnis über `getResults` aus. Als Callback-Funktion dient `saveResult`, welche die Ergebnisse temporär in der lokalen Hashtabelle `results` speichert. Nach Abschluss der Funktion `getResults` wird ein neues Objekt vom Typ `ClassificationResult` erstellt, welches die für das Anwendungsszenario interessanten Ergebnisse enthält. Dabei werden die zurückgelieferten Zeichenfolgenpaare geparst und in privaten Feldern des `ClassificationResult`-Objekts gespeichert. Eine `ClassificationFailedException` wird auch dann ausgelöst, wenn nicht alle relevanten Sprechereigenschaften ermittelt werden konnten.

`ClassificationResult` (s. Abb. 4.6) ist eine einfache Hilfsstruktur zum Zugriff auf die Ergebnisse eines Klassifizierungsvorgangs. Die Methode `getGender` liefert 1 für

*männlich*, 2 für *weiblich* und 0 für *unsicher* zurück, wobei *unsicher* in Abhängigkeit von der Wahrscheinlichkeit mittels eines festen Schwellenwerts definiert wird. Ein Aufruf von `getAgeClass` gibt die Altersklasse in einer Randwert-Notation $x$`S`$y$ an, welche für „Sprecher ist zwischen $x$ und $y$ Jahren alt“ steht, z.B. `20S65`. Am oberen bzw. unteren Intervallende können $x$ bzw. $y$ auch weggelassen werden. Um nur die beiden Randwerte ohne Parsen in numerischer Form abzufragen, können auch `getAgeLowerBound` und `getAgeUpperBound` verwendet werden. `getGenderLatest` und `getAgeClassLatest` wurden auf Wunsch des Auftraggebers hinzugefügt und sollen das Ergebnis der jeweils letzten Klassifizierung ohne Berücksichtigung früherer Ergebnisse ausdrücken. Die Funktionalität ist bisher noch nicht in *C++* implementiert. Später würde hierzu ein zweites Bayessches Netz erstellt werden, welches parallel zur eigentlichen zweiten Ebene klassifiziert, jedoch nach jeder Äußerung wieder zurückgesetzt wird. Die Funktionen `getProbabilityGender` und `getProbabilityAgeClass` schließlich geben die Wahrscheinlichkeiten für Geschlecht und Alter auf der zweiten Ebene an.

# 5 Leistungsdaten

In diesem Kapitel sollen einige Zahlen zur derzeitigen Leistung der SBC-Architektur gegeben werden. Der Fokus liegt dabei auf der Geschwindigkeit, da hier im Vergleich zur M3I-Architektur deutliche Verbesserungen festzustellen sind. Die Klassifikationsgenauigkeit des eingebetteten Moduls unterscheidet sich durch die Re-Implementierung vorhandener Methoden im Optimalfall nicht von den in der Client/Server-Umgebung ermittelten Werten. Da überdies die qualitative Optimierung der Methoden nicht Gegenstand der vorliegenden Arbeit ist, werden an dieser Stelle keine Statistiken zur Genauigkeit aufgeführt und stattdessen auf Müller (2005, S. 135ff) verwiesen.

Die Laufzeitperformanz der Plattform wurde durch *Benchmark-Tests* mit dem Klassifikationsmodul ermittelt. Dazu wurde eine *Timer*-Hilfsklasse in den Code des Moduls integriert, mit deren Hilfe die benötigte Zeit für die gesamte Klassifikation einer Äußerung sowie der einzelnen Teilschritte innerhalb der Verarbeitungs-Pipeline gemessen werden konnte.

Die erste Testreihe wurde mit jeweils mehreren Audiodateien ähnlicher Länge in einem gemeinsamen Format auf einem Desktop-PC durchgeführt. Als Format wurden 8 kHz bei 16 Bit und Mono-Aufnahme gewählt. Die Messungen wurden einmal mit und einmal ohne die in Feld (2005, S. 27) beschriebene Optimierung vorgenommen. Bei dem verwendeten Testsystem handelt es sich um einen *Intel Pentium 4* mit 2,53 GHz. Für die Klassifizierung wurde der *MultiGauss*-Algorithmus (vgl. Abschnitt 3.7.4) und die zweite Ebene aus Abb. 3.29 auf Seite 108 eingesetzt. Die Ergebnisse sind in Tabelle 5.1 dargestellt.

| **Äußerungslänge** | **Dauer ohne Optimierung** | **Dauer mit Optimierung** |
|---|---|---|
| 2-3 Sekunden | 0,96s | 0,09s |
| 9-11 Sekunden | 3,24s | 0,32s |
| ca. 2 Minuten | 30,59s | 3,96s |

Tabelle 5.1: Dauer zur Klassifikation von Äußerungen unterschiedlicher Länge auf einem Desktop-System, mit und ohne Optimierung

Wie zu erkennen ist, liegen die Werte bei kurzen Äußerungen unter der Zeitdauer, die für einen Menschen als spürbare Latenz wahrgenommen werden kann. Dies ist beispielsweise im Callcenter-Szenario wichtig, damit für den Anrufer keine Wartezeiten bei

der Behandlung der Anfrage entstehen. Ferner bestätigen die Messungen mit und ohne Optimierung, dass jene einen deutlichen (und in der Praxis möglicherweise unverzichtbaren) Performanzgewinn von ca. Faktor 9 bewirkt. Eine weitere Vermutung anhand der vorliegenden Daten besteht darin, dass die Klassifikationsdauer in etwa linear zur Äußerungslänge wächst. Dies würde unter anderem bedeuten, dass sich durch die Zerlegung einer großen Äußerung in mehrere kleinere Teilstücke keine Vor- oder Nachteile im Hinblick auf die Geschwindigkeit ergeben würden.

Das Resultat der Messung der einzelnen rechenintensiven Teilschritte innerhalb der Verarbeitungs-Pipeline (Datei lesen, Merkmalsextraktion, erste Ebene, zweite Ebene) bedarf keiner grafischen Veranschaulichung. Dies liegt daran, dass die Klassifikationszeit fast ausschließlich zur Merkmalsextraktion benötigt wird, und die für die verbleibenden Prozesse gemessenen Zeiten unter der durch den System-Zeitgeber messbaren Auflösung lagen (gerundet also 0 ms). Es sei allerdings darauf hingewiesen, dass komplexere Klassifizierungsmethoden oder andere Hardware-Plattformen in dieser Hinsicht ein etwas anderes Bild bewirken könnten, wenngleich nicht davon auszugehen ist, dass sich bei den gängigen Verfahren die Tendenz grundlegend ändert.

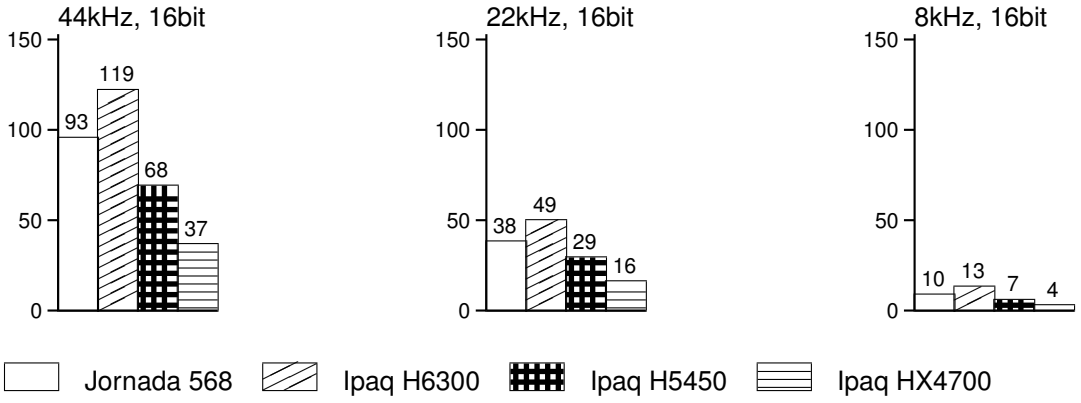



Abbildung 5.1: Dauer zur Klassifikation einer Äußerung von 2,3s auf verschiedenen *PocketPCs*, optimierter Algorithmus (vgl. Feld, 2005, S. 28)

Weitere Messungen wurden im Rahmen von Feld (2005, S. 26ff) auf verschiedenen mobilen Geräten durchgeführt. Abb. 5.1 zeigt das Ergebnis für eine einzelne Audiodatei von 2,3 Sekunden Länge in unterschiedlichen Formaten und auf vier verschiedenen Geräten (jeweils mit Performanz-Optimierung). Neben der Tatsache, dass die Klassifizierungsdauer erwartungsgemäß auch von der Abtastfrequenz abhängt, wird deutlich, dass es gerade bei der mobilen Plattform große Unterschiede zwischen den Geräten gibt, und dass der technologische Fortschritt die momentan noch relativ hohen Zeiten nach und nach denen der heutigen Desktop-Systeme annähern wird. Bei einem Anwendungs-

szenario wie dem SHOPASSIST, bei dem die Adaption im Hintergrund durchgeführt wird und der Benutzer nicht auf den Abschluss des Vorgangs angewiesen ist, sind Zeiten im Bereich von mehreren Sekunden jedoch vollkommen akzeptabel.

# 6 Zusammenfassung

In dieser Arbeit wurde eine Architektur (SBC) vorgestellt, welche Sprecherklassifikation für Real-World-Applikationen auf verschiedenen Plattformen verfügbar macht. Die Architektur unterstützt die Klassifikation von Alter und Geschlecht eines Sprechers basierend auf dem AGENDER-Verfahren und lässt sich auch flexibel auf andere Sprechermerkmale erweitern. Durch Verwendung von AGENDER ist gewährleistet, dass sich eine hohe Klassifikationsgenauigkeit durch die Wahl eines optimal auf das konkrete Klassifikationsproblem abgestimmten Modells erreichen lässt, z.B. durch die Kombination mehrerer Klassifizierer.

Ein von Grund auf in *C++*, neu implementiertes eingebettetes Klassifikationsmodul, dessen Aufbau anhand ausführlicher Beschreibungen von Klassen und Methoden sowie der zugehörigen UML-Diagramme im Detail beschrieben wurde, verfügt durch vielfältige Optimierungen auf Architektur-, Algorithmus- und Codeebene über eine hohe Laufzeitperformanz und erlaubt die schnelle und parallele Klassifikation von Äußerungen. So zeigt sich die Performanzorientierung beispielsweise bei der Pipeline-Architektur des Klassifikationsmoduls, dem Klassifizierer-Cache, der Integration von *Praat* und der Optimierung der Autokorrelation aus Boersma (1993).

In der Arbeit wurde beschrieben, wie für ein Klassifikationsproblem mit Hilfe der SBC DEVELOPMENT PLATFORM Daten und Modelle zusammengestellt werden können, welche dann trainiert und als Laufzeit-Klassifizierer zu einem Klassifikationsmodul kompiliert werden. Dieser als Build-Vorgang bezeichnete Prozess beschreibt eine Möglichkeit, Klassifizierer äußerst effizient zu implementieren und dennoch eine hohe Modularität des Systems zu gewährleisten, indem z.B. mehrere Modul-Konfigurationen parallel verwendet werden. Als technisches Werkzeug wurde der One-Click-Buildvorgang herausgestellt, welcher die Erzeugung eines Klassifikationsmoduls bei vorhandener Konfiguration in nur einem Arbeitsschritt ermöglicht. Um die Verfügbarkeit auf verschiedenen Plattformen sicherzustellen, wurden einige Verfahren aufgegriffen, die eine plattformspezifische Entwicklung auf transparente Art und Weise erlauben, z.B. Erstellungs-Konfigurationen und bedingte Kompilierung. Weitere in dieser Arbeit entwickelte Werkzeuge, die zur Konfiguration und zum Testen von Modulen genutzt werden können, sind die automatische Evaluierung in der DEVELOPMENT PLATFORM, die exakte manuelle Evaluierung über das Webserver-basierte Evaluierungsmodul und die Plausibilitätsüberprüfung von Merkmalswerten im eingebetteten Modul, welche dank der integrierten Tracing-

Funktionen möglich ist. Als externe Hilfsmittel wurden M3I CAT und *JavaDBN* angeführt.

Durch den sparsamen Umgang mit Systemressourcen ist die Architektur auch für den Einsatz auf mobilen Geräten geeignet. Dies belegt die Portierung auf *Windows CE* (*PocketPC*) und die Integration in den SHOPASSIST und ein eigenständiges Klassifikations-Testprogramm. Durch einen anwendungsgesteuerten Objekt-Lifecycle besitzt auch die Host-Applikation eine eigenständige Kontrolle über die verwendeten Ressourcen. Im Zusammenhang mit dem ShopAssist wurde die SBC Client-Bibliothek vorgestellt, die über den SBC-Server auch die servergestützte Sprecherklassifikation erlaubt oder in einem erweiterten Fallback-Modus arbeiten kann. Das zum Einsatz kommende Protokoll wurde beschrieben, wie auch einige der verwendeten Techniken, z.B. der Heartbeat-Mechanismus.

Die vorgestellte Schnittstelle für eine Callcenter-Applikation (*JII*) zeigt, dass SBC auch die Anforderungen an ein Server-System mit hohem Datenaufkommen erfüllt, beispielsweise eine gute Verfügbarkeit, Robustheit und Skalierbarkeit. Letzteres ist insofern gegeben, als dass sich Äußerungen parallel klassifizieren lassen und ein anwendungsgesteuerter Lastausgleich möglich ist, einschließlich der Verarbeitung auf einem Cluster. Darüber hinaus wird die Interoperabilität mit anderen Programmiersprachen wie *Java* über Wrapper-DLLs demonstriert.

Ein für viele Anwendungsarchitekten wichtiger Punkt ist die Integrierbarkeit einer Komponente wie dem SBC-Klassifikationsmodul. Hier wurden neben der Beschreibung der verschiedenen Integrationsschichten eine genaue Schnittstellenbeschreibung (`ProcessingPipeline`-Objekt) und Informationen zur Einbindung gegeben. Daneben wurde erörtert, aus welchen Gründen der gewählte Ansatz einer statisch verknüpften Bibliothek einer DLL vorgezogen wurde. Zum Austausch der Klassifikationsergebnisse wurde ferner die Möglichkeit einer universellen Benennungskonvention vorgeschlagen.

Schließlich wurde das Gesamtkonzept aufgezeigt, in dessen Zentrum die Gegenüberstellung der Entwicklung von Anwendungen auf der einen Seite und der Rahmenarchitektur (SBC/AGENDER) auf der anderen Seite steht.

# 7 Ausblick

Mit der Implementierung von SBC, die begleitend zu dieser Arbeit vorangetrieben wurde, lassen sich bereits jetzt alle wesentlichen Funktionen der Sprecherklassifikation nach Alter und Geschlecht nutzen. Damit ist die Arbeit in diesem Bereich aber keineswegs abgeschlossen. Die weitere Vorgehensweise erlaubt eine Einteilung in zwei Bereiche: die Verbesserung der Methoden, also AGENDER, und die Weiterentwicklung der Architektur, d.h. SBC.

Der AGENDER-Ansatz soll auch in Zukunft auf konzeptueller Ebene verbessert werden. Fortschritte in diesem Bereich kommen allen SBC-Anwendungen zugute, da sie eine noch besser auf die Anforderungen eines Szenarios zugeschnittene Auswahl der Modelle erlauben und die Genauigkeit verbessern. Als konkrete Möglichkeit seien neue Sprachmerkmale als Basis für die Klassifizierung genannt, beispielsweise die von Müller (2005) vorgeschlagenen *cepstralen* Merkmale (vgl. ebd., S. 48f.), *F1* und *F2* (vgl. ebd., S. 38ff). Auch die Suche nach neuen Gruppierungen für die Klassen nach den genannten oder vorhandenen Merkmalen könnte sich positiv auf die Leistung auswirken. Daneben wäre es denkbar, dass vollständig neue Klassifizierungsverfahren entwickelt werden. Alle neuen Merkmale und Modelle erfordern neben einer Implementierung in der Entwicklungsplattform bzw. M3I CAT jeweils auch eine plattformspezifische SBC-Implementierung.

Ein weiterer Gesichtspunkt ist die Modellierung noch zu bestimmender Eigenschaften eines Sprechers oder des Kontexts. Durch den flexiblen Aufbau von AGENDER ist es möglich, solche neuen Sprechereigenschaften mit den bestehenden Mitteln klassifizieren zu können, vorausgesetzt, es stehen geeignete Merkmale und Sprachkorpora zur Verfügung. Bereits angedacht sind beispielsweise eine *Sprachenerkennung*, d.h. Bestimmung der Sprache, in der etwas gesprochen wird, mit Hilfe von paralinguistischen Merkmalen. Verfolgt man diesen Ansatz weiter, so könnten auch der *Dialekt* oder weitere kulturelle Charakteristika extrahiert werden. Neben den Sprechereigenschaften liefert eine Aufnahme aber häufig auch Informationen zur Umgebung. Hier wurden in Müller (2005, S. 104ff) bereits erste Untersuchungen zu einer Unterscheidung der drei Klassen *Quiet*, *Voicy* und *Noisy* durchgeführt. In einem späteren Stadium ist die Aufnahme dieser Merkmale in AGENDER sehr viel versprechend. Da SBC grundsätzlich nicht an bestimmte Sprechereigenschaften gebunden ist, lassen sich diese Erweiterungen auch problemlos in das Framework übernehmen. Durch die Angabe entsprechender Profilpfade liegt die Handhabung dieser zusätzlichen Daten allein bei der Applikation. Wie

dem SBC-Gesamtkonzept (s. Abb. 3.31 auf Seite 114) zu entnehmen ist, besteht hier die Hoffnung, dass Industrieprojekte Anstöße dazu liefern können, welche Informationen in der Praxis am ehesten relevant sind.

Unabhängig von der Verbesserung der Methoden gibt es auch einige zukünftige Aufgaben, die lediglich die Architektur betreffen. Manche davon wurden bereits in den vorangegangenen Kapiteln angesprochen. Hohe Priorität besitzt fortwährend die Optimierung der Performanz. Eine Überarbeitung des gesamten Codes und vor allem der Algorithmen kann zusätzliche Geschwindigkeitsvorteile mit sich bringen. So wurden bisher plattformspezifische Optimierungen von Funktionen nur am Rande betrachtet. Auch lässt sich die Arbeitsweise des `FeatureExtractor` noch verbessern und somit der Prozess der Merkmalsextraktion kompakter und schneller gestalten, indem nur die tatsächlich benötigten Merkmale berechnet werden (vgl. Abschnitt 3.7.2). Neben besagten Feinabstimmungen ist auch die Implementierung weiterer Merkmale, insbesondere der Sprechverhaltensmerkmale *Artikulationsgeschwindigkeit* und *Sprechpausen*, ein wichtiger Anknüpfungspunkt. Dies wird nach dem jetzigen Stand auf die Portierung der Programme *mrate* und *srsad* hinauslaufen, da diese bereits erfolgreich im M3I-Server getestet wurden.

In der aktuellen Version von SBC sind lediglich zwei Klassifizierungsverfahren im eingebetteten Modul verfügbar. Daher werden sich zukünftige Bemühungen darauf konzentrieren, weitere Verfahren zu implementieren, insbesondere *Künstliche Neuronale Netze*, die im Durchschnitt die beste Genauigkeit bei der Evaluierung erzielten, *Entscheidungsbäume* aufgrund deren guter Genauigkeit bei sehr hoher Performanz sowie ein *k-Nearest-Neighbor-Algorithmus*, welcher je nach Struktur der Daten die anderen Verfahren übertreffen kann. Derartige Neuerungen erfordern möglicherweise auch die Erweiterung der Hilfsklassen. Die vorhandenen Klassifizierer bieten ebenfalls noch Möglichkeiten zur Weiterentwicklung. Beispielsweise werden bei dem *WeightedUniGaussClassifier* die Gewichte gegenwärtig über die Kreuzvalidierung berechnet. Der *Expectation-Maximization-Algorithmus* verspricht hier eine Steigerung der Qualität.

Ein weiterer Aspekt, der bei SBC bisher noch nicht betrachtet wurde, ist die *Vorverarbeitung* von Audiodaten. So könnten diese durch integrierte *Resampling*-Routinen in ein gemeinsames Format überführt werden, auch wenn diese aus Aufnahmequellen mit unterschiedlichen Formaten stammen. Weiter besteht die Möglichkeit, die Qualität der Merkmale durch das Herausfiltern von Hintergrundgeräuschen oder Artefakten zu verbessern, was einer Segmentierung der Daten nach Duda et al. (2000) entspricht. Acero (1990) stellt mehrere für diesen Zweck geeignete Algorithmen vor. Das gleiche gilt für das Zuschneiden (*Trimming*) der Aufzeichnung, um die Pause bis zum Sprechbeginn und nach Äußerungsende zu identifizieren und zu entfernen.

Es gibt eine Reihe weiterer Verbesserungsmöglichkeiten des Klassifikationsmoduls, die den Weg in zukünftige Versionen finden könnten. Dazu zählt die echte Ressourcenadaptivität, d.h. die dynamische Anpassung an die tatsächlich vorhandenen Ressourcen wie

Arbeitsspeicher, um die Leistung für die Hard- und Softwareumgebung zu optimieren. Eine interessante Fragestellung ist auch die nach persistenter Speicherung eines Benutzermodells, also dem Laden und Speichern von Benutzermodellen. Auch wenn dies eventuell den Rahmen von SBC sprengt und in der Regel eher Aufgabe der Anwendung ist, so hätte es den Vorteil, dass ein Lastausgleich sowie Clustering-Funktionalität mit geringem Aufwand direkt im Modul integriert werden könnten. Die Voraussetzungen für Clustering wären dadurch gegeben, dass verschiedene Rechner je nach Anforderung auf ein zentral gespeichertes, persistentes Benutzermodell zugreifen und dieses aktualisieren könnten.

Wichtig für den Erfolg von SBC und dessen Weiterentwicklung ist die Integration in verschiedene Applikationen. Hier wird davon ausgegangen, dass es zukünftig noch mehrere praktische Anwendungsszenarien, aber auch Demonstratoren geben wird. Solche Projekte liefern wertvolles Feedback und möglicherweise auch neues Trainingsmaterial; sie machen aber vermutlich auch die Portierung von SBC auf weitere Plattformen erforderlich, wie z.B. *Linux* oder *Macintosh*. Auch existiert bis jetzt noch keine Anwendung, welche die nativen Funktionen der *Smartphone*-Plattform (z.B. Telefonie) ausnutzt, um mobile Adaption zu betreiben. Bei einer großen Anzahl zu unterstützender Plattformen könnte die Einführung einer PAL (vgl. Abschnitt 3.1.4) die Wartung des Codes deutlich vereinfachen und stellt somit eine weitere Verbesserungsmöglichkeit dar. Zur leichteren Integrierbarkeit in Anwendungen, die in verschiedenen Sprachen geschrieben sind, würde es sich ferner anbieten, seitens SBC die dazu notwendigen Wrapper-Bibliotheken für die am weitesten verbreiteten Sprachen wie *C#* und *Visual Basic* (für *Java* existiert bereits *JII*) zu erstellen.

In Hinblick auf den SBC-Server ist ebenfalls eine Überarbeitung denkbar. Bisher existiert nur die in Abschnitt 4.1.3 vorgestellte *Java*-Implementierung des M3I-Servers. Ein Server, welcher intern das SBC-Klassifikationsmodul verwendet, aber auch das Netzwerkprotokoll (vgl. Abschnitt 4.1.5) in vollem Umfang unterstützt, könnte eine bedeutende Alternative darstellen. So würde der Server nicht nur unmittelbar von der verbesserten Performanz und Ressourcensparsamkeit profitieren, sondern ließe sich auch mit verhältnismäßig geringem Mehraufwand in einen vollwertigen *Web Service* umwandeln, der sich vom Server lediglich durch die Schnittstelle (TCP/IP vs. SOAP[1]/HTTP) und die Art der Host-Applikation (Executable vs. Webserver) unterscheidet. Als weiterer Effekt hiervon würde ein eigenes Evaluierungsmodul wie das in Abschnitt 3.10 beschriebene überflüssig. Im Zuge der Überarbeitung des Servers könnten weitere wünschenswerte Merkmale in die Client/Server-Architektur aufgenommen werden, beispielsweise die Serialisierung von Klassifizierern, für die eine Infrastruktur zum Datenaustausch bereits vorhanden ist (vgl. Abschnitt 3.7.3 Seite 80), die Möglichkeit zum Klassifizieren eines konstanten Datenstroms (vgl. Abschnitt 4.1.5 Seite 128) sowie die Fusion der Ergebnisse

[1] XML-basiertes Protokoll zum Austausch von Objekten und Nachrichten

von lokaler und serverbasierter Klassifizierung in der Client-Bibliothek bei Verwendung der Modi *Auto* und *Parallel* (vgl. Abschnitt 4.1.2 Seite 119).

Die bisher angesprochenen Möglichkeiten betreffen hauptsächlich die Ausführungsphase. Da die Entwicklung seitens der Anwendungsarchitekten aber einen festen Bestandteil des SBC-Konzepts darstellt, gibt es auch in diesem Bereich Bedarf für Weiterentwicklungen. Das zentrale Entwicklungswerkzeug von SBC ist die DEVELOPMENT PLATFORM. Seit den frühen M3I-Server-Prototypen wurde die GUI nicht wesentlich verändert. Hier wäre zu überlegen, ob eine dokumentenorientierte Ansicht, welche ein Modul als Dokument ansieht, möglicherweise den aktuellen Anforderungen eher gerecht wird als die derzeit verwendete Kombination aus Browser und „Explorer"-Ansicht, die nicht zuletzt von der internen Repräsentation des Blackboard und der Dienste abgeleitet wurde. Auch müssen die Modul-Konfigurationen im Augenblick noch „von Hand", d.h. mittels eines externen Texteditors, bearbeitet werden. Eine grundlegende Überholung der GUI könnte die Bedienbarkeit des Programms verbessern. Darüber hinaus wäre es auch sinnvoll, den Entwurf von Bayesschen Netzen und deren automatische Kompilierung als Teil des Build-Vorgangs vollständig über die DEVELOPMENT PLATFORM zu steuern (vgl. Abschnitt 3.9.6).

Die in Kapitel 5 ermittelten Werte für die zur Klassifizierung einer einzelnen Äußerung benötigten Zeitdauer wurden über eine benutzerdefinierte Timing-Routine ermittelt. Für die einen SBC-Entwickler könnte es aber auch von Interesse sein, solche Werte jeweils beim Erzeugen eines neuen Klassifikationsmoduls - analog zur automatische Evaluierung - ausgeben zu lassen. Daher wird für spätere Versionen auch die Integration eines Benchmarks in die Entwicklungsplattform und – optional – in das Klassifikationsmodul angestrebt.

# Literaturverzeichnis


Acero, A. (1990). *Acoustical and Environmental Robustness in Automatic Speech Recognition.* Dissertation, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Boersma, P. (1993). Accurate short-term analysis of the fundamental frequency and the harmonics-to-noise ratio of a sampled sound. In *Proceedings of the Dutch Institute of Phonetic Sciences (IFA)* (Bd. 17, S. 97–110). Amsterdam, Netherlands.

Boersma, P. (2001). PRAAT, a system for doing phonetics by computer. *Glot International*, *9*(5), 341–345.

Bramberger, M., Brunner, J., Rinner, B. und Schwabach, H. (2004). Real-Time Video Analysis on an Embedded Smart Camera for Traffic Surveillance. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004)* (S. 174–181). Toronto, Canada.

Bramberger, M., Rinner, B. und Schwabach, H. (2004). An Embedded Smart Camera on a Scalable Heterogeneous Multi-DSP System. In *Proceedings of the European DSP Education and Research Symposium (EDERS 2004).* Birmingham, United Kingdom.

Bramberger, M., Rinner, B. und Schwabach, H. (2005). Resource-Aware Dynamic Task-Allocation in Clusters of Embedded Smart Cameras by Mobile Agents. In *Proceedings of the IEE International Workshop on Intelligent Environments.* Colchester, United Kingdom.

Brandherm, B. und Jameson, A. (2004). An extension of the differential approach for Bayesian network inference to dynamic Bayesian networks. *International Journal of Intelligent Systems*, *19*(8), 727–748.

DeVaul, R., Sung, M., Gips, J. und Pentland, A. S. (2003). MIThril 2003: Applications and Architecture. In *Proceedings of the Seventh IEEE International Symposium on Wearable Computers.*

Dietterich, T. (1995). Overfitting and Undercomputing in Machine Learning. *Computing Surveys*, *27*(3), 326–327.

Duda, R. O., Hart, P. E. und Stork, D. G. (2000). *Pattern Classification* (2. Aufl.). New York, USA: Wiley-Interscience.

Durand, J. und Laks, B. (Hrsg.). (2002). *Phonetics, Phonology, and Cognition.* USA: Oxford University Press.

Feld, M. (2005). *Portierung von Merkmalsextraktion auf die PocketPC-Plattform* (Technical Memo Nr. TM-05-01). Saarbrücken, Germany: DFKI, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH.

Galassi, M., Davies, J., Theiler, J., Gough, B., Jungman, G., Booth, M. und Rossi, F. (2003). *GNU Scientific Library Reference Manual* (2. Aufl.). Network Theory Ltd.

Hastings, R. und Joyce, B. (1991). Purify: Fast Detection of Memory Leaks and Access Errors. In *Proc. of the Winter 1992 USENIX Conference* (S. 125–138). San Francisco, California.

Heckmann, D. (2005). *Ubiquitous User Modeling.* Dissertation, Fachbereich 6.2 Informatik, Universität des Saarlandes, Deutschland.

Heckmann, D., Schwartz, T., Brandherm, B. und von Wilamowitz-Moellendorff, M. (2005). GUMO, the General User Model Ontology. In *Proceedings of the 10th International Conference on User Modeling* (S. 428–432). Edinburgh, Scotland: Springer, Berlin Heidelberg.

Hergula, K. (2003). *Daten- und Funktionsintegration durch Föderierte Datenbanksysteme.* Dissertation, Technische Universität Kaiserslautern.

Heutte, L., Paquet, T., Nosary, A. und Hernoux, C. (2000). Handwritten text recognition using a multiple-agent architecture to adapt the recognition task. In L. Schomaker und L. Vuurpijl (Hrsg.), *Proceedings of the Seventh International Workshop on Frontiers in Handwriting Recognition* (S. 413–422). Nijmegen: International Unipen Foundation.

Jones, M. B., Roşu, D. und Roşu, M.-C. (1997). CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *16th ACM Symposium on Operating Systems Principles.* Saint-Malo, France.

Kent, R. D. und Read, C. (2002). *Acoustic Analysis of Speech.* Canada: Thomson Learning (Singular).

Kneissler, J., Kienappel, A. K. und Klakow, D. (2003). Information retrieval based call classification. In *Proceedings of Eurospeech 2003* (S. 1213–1216). Geneva, Switzerland.

Ladefoged, P. (2001). *A Course in Phonetics* (4. Aufl.). Orlando: Harcourt College Publishers.

Moon, T. K. (1996). The expectation-maximization algorithm. *Signal Processing Magazine, IEEE, 13*(6), 47–60.

Morgan, N. und Fosler, E. (1998). Combining Multiple Estimators of Speaking Rate. In *Proceedings of the 23rd International Conference on Acoustics, Speech, and Signal Processing (ICASSPP'98)* (S. 729–732).

Morgan, N., Fosler, E. und Mirghafori, N. (1997). Speech Recognition using On-line Estimation of Speaking Rate. In *Proceedings of the 5th European Conference on Speech Communication and Technology, EUROSPEECH* (S. 2079–2082). Rhodes, Greece.

Müller, C. (2005). *Zweistufige kontextsensitive Sprecherklassifikation am Beispiel von Alter und Geschlecht.* Dissertation, Fachbereich 6.2 Informatik, Universität des Saarlandes, Deutschland.

Müller, C., Großmann-Hutter, B., Jameson, A., Rummer, R. und Wittig, F. (2001). Recognizing Time Pressure and Cognitive Load on the Basis of Speech: An Experimental Study. In M. Bauer, P. Gmytrasiewicz und J. Vassileva (Hrsg.), *UM2001, User Modeling: Proceedings of the Eighth International Conference* (S. 24–33). New York - Berlin: Springer.

Noll, P. (1997). MPEG Digital Audio Coding – Setting the Standard for High-Quality Audio Compression. *IEEE Signal Processing Magazine, Special Issue on MPEG Audio and Video Coding, 14*(5), 59–81.

Orfanidis, S. J. (1996). *Introduction to Signal Processing.* New Jersey: Prentice Hall.

Park, A. und Jung, K. (2004). PDA-Based Text Localization System Using Client/Server Architecture. In C. Zhang, H. W. Guesgen und W.-K. Yeap (Hrsg.), *PRICAI 2004: Trends in Artificial Intelligence: 8th Pacific Rim International Conference on Artificial Intelligence, Auckland, New Zealand* (Bd. 3157, S. 833–842). Springer.

Parnas, D. L. (1972). On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the Association of Computing Machinery, 15*(12), 1053–1058.

Plichta, B. (2002). Best Practices in the Acquisition, Processing, and Analysis of Acoustic Speech Signals. (Verfügbar unter `http://www.historicalvoices.org/flint/extras/Audio-technology.pdf`)

Plichta, B. und Kornbluh, M. (2001). *Digitizing Speech Recordings for Archival Purposes.* Matrix, Michigan State University. (Working Paper)

Quinlan, R. (1993). *C4.5: Programs for Machine Learning.* San Mateo, CA, USA: Morgan Kaufmann Publishers.

Racine, J. (2000). The Cygwin tools: a GNU toolkit for Windows. *Journal of Applied Econometrics*, *15*(3), 331–341.

Rumelhart, D. E., Hinton, G. E. und Williams, R. J. (1986). Learning Internal Representations by Error Propagation. In D. E. Rumelhart, J. L. McClelland und PDP research group (Hrsg.), *Parallel distributed processing: Explorations in the microstructure of cognition, Volume 1: Foundations.* Cambridge: MIT Press.

Smith, D., Townsend, J., Nelson, D. und Richman, D. (1999). A Multivariate Speech Activity Detector Based on the Syllable Rate. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'99)* (S. 73–76). Phoenix, USA.

Sung, M. und Pentland, A. S. (2004). *LiveNet: Health and Lifestyle Networking Through Distributed Mobile Devices* (Tech. Rep. Nr. TR-575). Cambridge, Massachusetts: Human Dynamics Group, MIT Media Laboratory.

Tack, G., Kornstaedt, L. und Smolka, G. (2005). Generic pickling and minimization. In *Electronic Notes in Theoretical Computer Science (ENTCS).* (Verfügbar unter `http://www.ps.uni-sb.de/Papers/abstracts/pickling_minimization.pdf`)

Turpeinen, M. (2000). Customizing News Content for Individuals and Communities (Dissertation, Helsinki University of Technology, Espoo, Finnland). *Acta Polytechnica Scandinavica, Mathematics and Computing Series No. 103.*

Wahlster, W. (2000). Künstliche Intelligenz: Werden Computer zu intelligenten Assistenten für jedermann? In Brockhaus-Redaktion (Hrsg.), *Visionen 2000* (S. 172–175). Mannheim, Germany: Brockhaus.

Wasinger, R., Krüger, A. und Jacobs, O. (2005). Integrating Intra and Extra Gestures into a Mobile and Multimodal Shopping Assistant. In *Proceedings of the 3rd International Conference on Pervasive Computing (Pervasive)* (S. 297–314). München, Germany.

Wasinger, R., Stahl, C. und Krüger, A. (2003). M3I in a Pedestrian Navigation & Exploration System. In *Proceedings of the Fourth International Symposium on Human Computer Interaction with Mobile Devices* (S. 481–485). Pisa, Italy.

Weiser, M. (1991). The computer for the 21st century. *Scientific American*, 94–104.

Witten, I. H. und Frank, E. (2005). *Data Mining: Practical machine learning tools and techniques* (2 Aufl.). San Francisco: Morgan Kaufmann.

Wu, C., Lubensky, D., Huerta, J., Li, X. und Kuo, H.-K. J. (2003). A Framework for Large Scalable Natural Language Call Routing Systems. In *Proceedings of the International Conference on Natural Language Processing and Knowledge Engineering.* Beijing, China.