Entwicklung zweier SW-Module an den Schnittstellen Mensch-Computer-Telefonie

Masterarbeit

vorgelegt von

Christopher Goeritz

Institut für Softwaretechnologie
Technische Universität Graz
Inffeldgasse 16b/II, 8010 Graz

Betreuer: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Graz, Juni 2011

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Zusammenfassung

Die vorliegende Arbeit befasst sich mit der Entwicklung eines Tools zur Unterstützung von Call-Agenten in einer Shared-Service-Factory bei ihrer täglichen Arbeit. Gezeigt wird die momentane Situation, die bereits genutzten Anwendungen und Technologien, und an welchen Stellen sich der allgemeine Arbeitsablauf noch verbessern lässt. Hierfür wird die Entwicklung eines Programms an der Schnittstelle Mensch-Computer-Telefonie dokumentiert, welche bestehenden Konzepte wieder verwendet werden können und jene, die speziell für dieses Tool neu entwickelt wurden. Die Herangehensweise an das Projekt sowie der Ablauf, die kundennahe Entwicklung, deren Feedback und die abschließende Testphase werden ebenfalls aufgezeigt.

Abstract

The following thesis focuses on the development of a tool to support Call-Agents in a Shared-Service-Factory in their daily work. The current working situation is shown; already utilized software and technologies are discussed as well as where there is still room for improvement in the overall workflow. To accomplish improvement the development of a software tool for the intersection of Person-Computer-Telephony is documented, it is analyzed which components of existing concepts can be reutilized and which have to be specially coded for this tool. The approach to this project as well as its process flow, the customer-oriented progression, customer feedback and the final testing phase are also illustrated.

Kapitel 1: Einleitung

Die Firma PIDAS (PIDAS AG, 2011) schrieb diese Masterarbeit aus, um ihre Call-Agenten bei der täglichen Arbeit zu unterstützen. „PIDAS ist ein Dienstleistungsunternehmen, das sich im Bereich Kundenservice auf den Aufbau, die Optimierung sowie den Betrieb von Service-Organisationen wie IT-Helpdesks und Customer Care Center spezialisiert hat.“ (PIDAS Österreich Ges.m.b.H., 2009) Für die in Graz betriebene Shared-Service-Factory (SSF) (PIDAS, 2011) wurde eine Softwarelösung gesucht, die die Zusammenarbeit der Agenten mit den bestehenden integrierten Helpdesk-Softwarelösungen verbessert.
Der Geschäftsbereich „Managed Services“ der PIDAS Österreich Gesellschaft m.b.H. übernimmt IT-Organisationen, betreibt und optimiert sie. PIDAS betreibt seit mehr als sechs Jahren auf Basis langjähriger Erfahrung im Bereich Aufbau, Optimierung und Betrieb von Service-Organisationen eine dezentrale Shared Service Factory in Graz und Basel (CH). Das Leistungsportfolio des stark skalierenden Servicecenters ist umfassend und stützt sich auf die Kompetenz der Servicemitarbeiter und das perfekte Zusammenspiel mit integrierten Helpdesk-Softwarelösungen.

1.1 Aufgabenstellung

Die ursprüngliche Aufgabenstellung sah die Entwicklung eines Agent-Call-Interface Tools (ACI) in Anlehnung an bekannte Instant-Messenger-Systeme vor. Die Leiter der SSF hatten bereits ein Pflichtenheft ausgearbeitet und die einzelnen zu realisierenden Funktionen definiert.
In Einzelprojekten hat PIDAS Solutions in der Vergangenheit bereits auf Basis von Projekten proprietäre Computer-Telefonie-Integrationen (CTI) zwischen dem Java EE basierendem PIDAS-TICKET-System (PIDAS, 2011) und diversen Telefonanlagen realisiert. Die Wiederverwertbarkeit dieser CTI-Clients war jedoch aufgrund der engen Bindung an bestimmte Telefonanlagen nur gering ausgeprägt.

Der sogenannte ACI-Client soll die nächste Generation dieser CTI-Integration darstellen. Als selbständiges Tool schafft er die Verbindung zwischen dem im Webbrowser des Anwenders laufenden PIDAS Ticket und der Telefonanlage des jeweiligen Callcenters bzw. IT-Helpdesk. Der ACI-Client ist so ausgelegt, dass er einerseits an die aktuell von der Shared Service Factory Graz verwendete Telefonieanlage und die PIDAS-Wissensdatenbank gekoppelt werden kann, andererseits aber auch so offen gestaltet, um auch mit möglichst geringem Konfigurationsaufwand an andere Telefonanlagen gebunden werden kann.

Abbildung 1-1: Früher ACI-Entwurf

1.2 Zielsetzung

Da sich seit der Erstellung des Pflichtenheftes und dem Beginn dieser Arbeit einige der in der SSF verwendeten Softwarelösungen geändert hatten, darunter die Einführung eines von allen Agenten genutzten Instant-Messenger-Systems (der Spark-Client (Jive Software, 2011) und der Openfire-Server (Jive Software, 2011)), wurde die Aufgabenstellung leicht angepasst. Da diese Produkte so konzipiert sind, dass sich leicht firmeneigene Funktionen nachträglich über Plugins hinzufügen lassen, wurde die Spezifikation des ACI Tools von einem eigenständigen System auf ein Plugin abgewandelt. Somit wurde das Ziel der Arbeit, sowohl ein clientseitiges Plugin für Spark als auch ein serverseitiges Plugin für Openfire zu entwickeln, das alle ursprünglichen Funktionen in die bestehende Programmstruktur integriert.

Kapitel 2: Grundlagen

Der folgende Abschnitt soll eine kurze Übersicht über die Thematik geben, den täglichen Ablauf in der SSF aufzeigen, sowie die Bereiche des Arbeitsalltages hervorheben, bei denen Verbesserungspotenzial besteht.

2.1 Java[1]

„Java“ bezeichnet einerseits die Programmiersprache Java, andererseits aber auch die Java Plattform, eine komplette Umgebung zum Erstellen und Ausführen von Programmen (Oracle, 2011). Diese Plattform, die „Virtual Machine“ (VM), agiert als eine Schicht zwischen dem Programm und dem Betriebssystem, auf dem es ausgeführt werden soll, und erlaubt so, Code in nur einer Sprache auf verschiedenen Systemen laufen zu lassen, denn die VM „führt nach der Übersetzungsphase den Bytecode aus. Somit ist Java eine kompilierte, aber auch interpretierte Programmiersprache – von der Hardwaremethode einmal abgesehen. Die virtuelle Maschine selbst ist in C++ programmiert, genauso wie einige Bibliotheken.“ (Ullenboom, 2009) Java basiert auf der Objektorientierten Programmierung, d.h. es benutzt Objekte um Funktionen zu implementieren. Es ist zu beachten, dass Java „nicht bis zur letzten Konsequenz objektorientiert [ist] (…). Primitive Datentypen wie Ganzzahlen oder Fließkommazahlen werden nicht als Objekte verwaltet“. (Ullenboom, 2009)

Objekte haben den Vorteil des Polymorphismus und der Vererbung. Polymorphismus bedeutet, dass einzelne Teile des Programms (bestimmte Objekte) ausgetauscht werden können, und die restlichen Teile trotzdem noch genauso funktionieren. Das System dahinter beruht auf dem Sender-Empfänger Prinzip. Ein Objekt, das eine Funktion erwirken möchte, sendet eine Anfrage an ein anders Objekt, das die Funktion dann ausführen kann. Das sendende Objekt muss (und sollte) dabei nicht wissen müssen, wie die Funktion genau implementiert ist, und das empfangende Objekt sollte nur die zur Kommunikation wichtigsten Funktionen zur Verfügung stellen und für andere unnötige Informationen verbergen. Daher kann jede andere Implementation des Senders bzw. des Empfängers, die die gleiche

[1] Vgl. (Langr, 2005)

Schnittstelle bedient, die alte ersetzen, ohne dass die anderen Bereiche geändert werden müssen.

Vererbung dagegen bedeutet, dass spezielle Objekte allgemeinere Objekte verfeinern. So kann ein spezielleres Objekt sämtliche Funktionen des allgemeineren Objektes „erben“, sodass es für andere Objekte noch die gleiche Schnittstelle definiert, und zusätzlich noch eigene Funktionen bereitstellen, die die Hauptfunktion detaillierter beschreiben. Außerdem können die geerbten Funktionen erweitert werden.

2.2 Agiles Programmieren[2]

Agiles Programmieren bezeichnet im Unterschied zu „normalem“ Programmieren, sich während des Entwicklungsprozesses leichter an Änderungen in den Anforderungen, Spezifikationen oder anderem anzupassen.

Das standardmäßig verwendete Wasserfallmodell (Royce, 1970) ist in seinem Ablauf sehr unflexibel und Änderungen in weiter oben liegenden Schichten ziehen sich in alle darunter liegenden hindurch. „Kritikpunkte an traditionellen Vorgehensmodellen, wie beispielsweise dem V-Modell, dem Spiral-Modell oder dem Rational Unified Process, sind häufig unter anderem mangelhafte Flexibilität (durch vorgegebene starre Strukturen), mangelnde Einbeziehung des Kunden in den Entwicklungsprozess sowie ein (scheinbar unnötig) hoher Dokumentationsaufwand. Agile Software-Entwicklungsprozesse sollen diese Nachteile beheben und ein höheres Maß an Flexibilität und Kundennähe bei einem Mindestmaß an Dokumentationen ermöglichen.“ (Alexander Schatten, 2010) Ich werde im Folgenden genauer auf eine spezielle agile Methode eingehen, nämlich „Extreme Programming“ (XP).

2.2.1 Extreme Programming[3]

XP ist ein Softwareentwicklungsstil, der sich auf klare Kommunikation, Programmiertechniken und Teamwork konzentriert. Er stellt eine Philosophie der Softwareentwicklung gründend auf Kommunikation, Feedback, Einfachheit, Mut und Respekt dar. Er beinhaltet eine Sammlung von Vorgehensweisen und Prinzipien, die sich als hilfreich bei der Verbesserung des Entwicklungsprozesses erwiesen haben.

[2] Vgl. (Langr, 2005)
[3] Vgl. (Beck, Extreme Programming Explained: Embrace Change, 2004)

Das „extreme“ daran besteht aus den kurzen Entwicklungszyklen, dem inkrementellen Planungsverlauf und dem flexiblen Implementationszeitplan, der sich schnell an wechselnde Bedürfnisse des Business anpassen kann. Weitere Elemente von XP beschäftigen sich mit der direkten Kommunikation zwischen Entwickler und Kunde, sowie einem testgetriebenen Entwicklungsprozesses (Test-Driven-Development, TDD).

Von Vertretern der XP-Gemeinde (z.B. Kent Beck) wird ausdrücklich empfohlen, XP schrittweise umzusetzen und an die eigenen Bedürfnisse anzupassen. Man sollte die verschiedenen Methoden ausprobieren und sehen, ob sie für sich von Nutzen sein können. So habe auch ich erst nur einen Teil der empfohlenen Prinzipien umgesetzt, wie zum Beispiel TDD, Kommunikation mit dem Kunden und frühes Feedback. Andere dagegen waren in dieser Diplomarbeit weniger gut umsetzbar, da es sich größtenteils um ein Einzelprojekt handelte, und so Methoden wie paarweises Programmieren sich nicht anboten.

2.2.2 Testgetriebene Entwicklung[4]

Testgetriebene Entwicklung (TDD) (Beck, 2002) ist ein einfacher, kurzlebiger Mechanismus, den man während des Programmiertages sehr häufig wiederholt. Die Idee dahinter ist, dass man zu einer gegebenen Spezifikation zuerst einen Testfall schreibt und erst hinterher den zu testenden Code. Der Vorteil davon ist, dass man genau testet was spezifiziert wurde, und nicht das, was bereits implementiert wurde. Dadurch steigt die Qualität des Codes, weil jeder Abschnitt des Programms auch einen entsprechenden Test besitzt, und des Designs, weil die Entwickler indirekt dazu gebracht werden entkoppeltere Klassen zu entwerfen die weniger stark voneinander abhängig sind. Dies führt zu einer viel besseren Wartbarkeit des Programms.

Ein weiterer Vorteil von TDD ist, dass jeder Test für sich als Dokumentation seiner zugehörigen Klasse dient. Aus dem Test lässt sich direkt herauslesen, was die Klasse tun soll und was nicht. Außerdem ist das Erweitern des Programms ungefährlicher, da man es sofort merkt wenn neue Teile bereits bestehenden schaden. So kann man seine Produktionszyklen erheblich verkürzen, da einem früher auffällt, wenn der momentan verfolgte Weg in die falsche Richtung führt.

[4] Vgl. (Langr, 2005)

Der Ablauf von TDD für eine neue Klasse sieht in etwa wie folgt aus:

- Schreibe eine Spezifikation für die Klasse
- Schreibe einen Test für die Klasse, der die Spezifikation (bzw. einen Teil davon) erfüllt aber fehlschlägt – noch gibt es die Funktion in der Klasse noch nicht.
- Implementiere die Funktion so, dass der Test erfolgreich verläuft.
- Erweitere den Test, um einen weiteren Teil der Spezifikation zu erfüllen, sodass er wieder fehlschlägt.
- Erweitere die Funktion bzw. implementiere eine neue um den Test wieder zu erfüllen.
- Wiederhole diese Schritte so lange, bis die Klasse komplett ist.
- Für eine neue Klasse beginne von vorne.

Wichtig ist, dass man immer, wenn man sein System testet, alle Tests für jede Klasse wieder mit laufen lässt. Nur so ist sichergestellt, dass spätere Änderungen keine Fehlfunktionen in bestehenden Bereichen erzeugen. Je umfangreicher das Projekt ist, desto positiver fallen die Vorteile von TDD auf, denn es ist oft umständlich, nachträglich Fehler in früher implementierten Methoden zu finden.

Ich hatte bisher noch kein so großes Projekt zu bearbeiten, und habe deswegen zuvor noch nie nach dem TDD-Konzept gearbeitet. Es bedarf durchaus einer gewissen Eingewöhnungszeit sich an das testgetriebene Programmieren zu gewöhnen, denn es passiert einem Neuling schnell, dass er doch wieder kleine Klassen schreibt, ohne vorher Tests zu definieren. Doch nachdem ich mich darauf eingestellt hatte, fiel mir schnell auf, dass man auf diese Art die zukünftige Klasse viel genauer durchdenkt.

Das schwierige dabei ist allerdings, dass man die Tests/Klassen unabhängig genug von anderen Klassen hält. Dies ist mir besonders beim Entwickeln der Plugins für Spark und Openfire aufgefallen, da es hier oft schwer war, Tests zu schreiben ohne gewisse Teile dieser Programme zu nutzen. Problematisch wurde es dadurch, dass das ACI-Tool und der ACI-Server keine eigenständigen Programme sind, sondern nur Plugins, die in bestehende Programme eingebunden werden, und somit nur richtig funktionieren, wenn auch alle Teile ihrer Hauptprogramme laufen. Einzelne Methoden oder Klassen ausführen oder testen ist oft gar nicht möglich, da die benötigten Services nicht gestartet sind.

2.3 Einführung in die Thematik

Die Firma PIDAS betreut eine große Anzahl an Unternehmen bei IT-bezogenen Fragen und Problemen (PIDAS AG, 2011). Jedes dieser Unternehmen (die Mandate) erreicht die SSF über eine eigene Hotline Nummer. Die Agenten sind mehreren Mandaten als Bearbeiter zugeordnet und haben somit jeder ihren speziellen Kundenstamm.

2.3.1 Ist-Situation

Die Agenten haben an ihren Arbeitsplätzen mindestens einen Rechner mit mehreren Bildschirmen (teilweise sogar mehrere Rechner/Laptops) und ihr Telefon. Sie benötigen zum Bearbeiten einer Kundenanfrage verschiedene, voneinander unabhängige Systeme, wie zum Beispiel ein Ticketing-System zum Erfassen des Problems des Kunden, eine Wissensdatenbank in Form eines internen Forums, Zugriff auf die Rechner der Kunden per Fernwartung, etc. Zur Kommunikation untereinander stehen ihnen der Instant Messenger Spark zur Verfügung, sowie ein Newsticker-Laufband für kürzlich aufgetretene Probleme.

Die Anrufe werden über einen Telefonie-Server an die entsprechenden Agenten weitergeleitet, bzw. in ihre Warteschleife eingereiht. Die (SIP-)Telefone der Agenten sind auf dem Server an einem bestimmten Kanal angemeldet, und der Benutzer kann diesen Kanal über das Telefon steuern. So kann er sich mittels bestimmter Tastenkombinationen am Server ein- und ausloggen und seinen momentanen Bereitschaftszustand verändern. Über einen Webbrowser können vom Server eine Liste der vergangenen Anrufe abgefragt, sowie Audioaufzeichnungen der geführten Gespräche angehört werden.

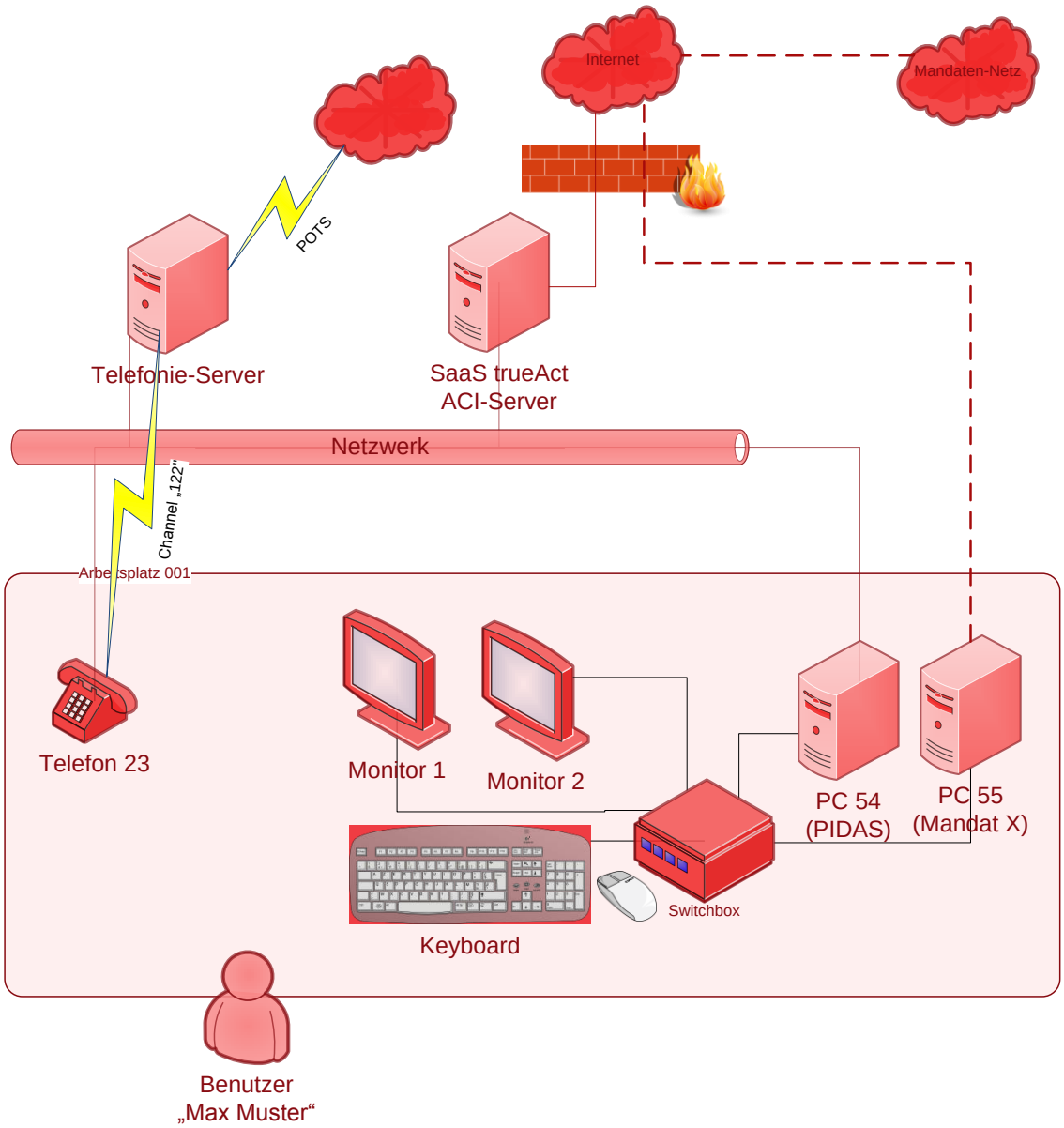

Abbildung 2-1: Standard-Arbeitsplatz eines SSF-Callagents

2.3.2 Ziele

Auf den ersten Blick fällt das Hauptziel direkt ins Auge: alle diese auf verschiedene Systeme verteilten Funktionen in einem einzigen übersichtlichen und leicht zu bedienendem Tool zu vereinen. Die Agenten sollen die Möglichkeit haben, ihren Kanal am Telefonie-Server zu steuern, aktuelle Neuigkeiten über Störfälle bei den Kunden zu sehen, Tickets und die Wissensdatenbank aufrufen zu können, und miteinander zu kommunizieren ohne dabei permanent zwischen unterschiedlichen Applikationen wechseln zu müssen.

Außerdem sollen im Zuge der Vereinfachung des Zugriffs auf diese Funktionen ebenfalls einige neue mit eingeführt werden. So soll für den Agenten zum Beispiel auf einen Blick ersichtlich sein, in welchem Bereitschaftszustand er sich gerade befindet (das Telefon zeigt dieses nämlich nicht an!), er soll seine persönliche Tagesstatistik betreffend Arbeitszeit, geführter Anrufe usw. einsehen können und eine kurze Übersicht über die Bereitschaftszustände seiner Kollegen der gleichen Mandate abrufen können. Außerdem soll er über den Zustand der Warteschleifen seiner einzelnen Mandate auf dem Laufenden gehalten werden.

Kapitel 3: Konzeption

Zur Realisierung dieser Ziele hatte ich zunächst verschiedene Konzeptideen, so zum Beispiel die Idee, einen eigenen Client zu bauen, der sich mit den anderen Systemen verbinden und die entsprechenden Daten abfragen kann, oder aber eine Client-Server-Architektur zu verwenden, die ähnlich einem Instant-Messenger funktionieren sollte. Da inzwischen jedoch firmenweit der Spark-Client als Instant-Messenger eingeführt wurde, und die Agenten nicht zwei dieser Clients gleichzeitig benutzen sollten (dies würde dem eigentlichen Ziel widersprechen), habe ich mich entschieden, das ACI-Tool als Erweiterung für den Spark zu entwickeln, zumal dieser das nachträgliche Einfügen von externen Plugins erlaubt und sogar ausdrücklich unterstützt.

3.1 Anforderungen

Die Anforderungen an solch ein Plugin bleiben die gleichen, als würde das Tool als eigenständiger Client entworfen werden. Es muss sich in die Benutzeroberfläche des Spark integrieren lassen, sämtliche geforderten Funktionen anbieten und sich auch wieder separat vom Spark beenden lassen. Es muss den Agenten bei der Arbeit unterstützen, allerdings soll er auch weiterhin seine Aufgaben wie gewohnt ohne Einschränkungen erledigen können, als wäre das Tool nicht vorhanden.
Das ACI soll bei Dienstantritt des Agenten gestartet werden und ihn während seiner Dienstzeit permanent unterstützen. Führt der Agent kein aktives Gespräch, dann sollen allgemeine Informationen und statistische Daten angezeigt werden. Im Newsfeed werden News aller Mandate des Agentendargestellt. Sobald ein Anruf eingeht, sollen die Anzeigen wie Mandat, Rufnummer, Anwender etc. entsprechend angepasst werden.

Da der Spark als Instant-Messenger bereits eine Client-Server-Architektur implementiert, nämlich mit dem Openfire als Instant-Messenger-Server, sollte das ACI Tool dieses ebenfalls nutzen. So muss auch auf dem Openfire ein Plugin integriert werden können, dass den Server für die Funktionen des ACI darstellt.

3.2 Planungsphase

Zunächst habe ich untersucht, ob diese beiden Applikationen für Plugins genug Möglichkeiten bieten, dass alle geforderten Funktionen umgesetzt werden könnten. Da Ignite Realtime (Jive Softwares Open Source Real Time Communications Projekt) (Jive Software, 2011) sie aber anscheinend bewusst so entwickelt hat, dass externe Plugins auf so gut wie jede interne Komponente der Programme zugreifen können, und sie zudem als Open-Source-Projekte unter der GNU-GPL (Openfire) (Free Software Foundation, Inc., 2010) bzw. GNU-LGPL (Spark) (Free Software Foundation, Inc, 2010) zur Verfügung stehen, bieten sie sich regelrecht für solch eine Erweiterung an.

Die Wahl der Programmiersprache stand damit auch direkt fest, da sowohl der Spark als auch der Openfire komplett in Java geschrieben sind. Somit konnte ich beginnen, die einzelnen Funktionen zu planen.

Eine der ersten Architekturentscheidungen musste natürlich die Verbindung zu den bestehenden anderen Systemen sein, die die Agenten benutzen. Es erschien mir als sinnvoll, nicht jeden Client einzeln mit ihnen kommunizieren zu lassen, zumal zu Beginn noch bedacht werden musste, dass einzelne Agenten eventuell aus fremden Netzwerken heraus arbeiten und somit keinen direkten Zugriff auf das firmeninterne Netz hätten. So habe ich mich entschieden, dass die ACI-Clients zu einem großen Teil nur mit dem ACI-Server in Kontakt stehen würden, und dieser die Weiterleitung an die Applikationen übernimmt. Diese Brücke hat zudem den Vorteil, dass wenn eine der Schnittstellen zu den anderen Systemen geändert werden müsste, so wäre dies nur am Server erforderlich.

Ich habe die einzelnen Funktionen in drei Schritten geplant: Zunächst habe ich nur die Funktionen behandelt, die komplett ohne den Zugriff auf die Telefonie auskommen, zum Beispiel die persönliche Tagesstatistik oder der Newsfeed, da so die Arbeit bereits ohne ein Telefon-Server-Testsystem im Hintergrund begonnen werden konnte. Im zweiten Schritt habe ich dann jene Funktionen betrachtet, die nur auf Ereignisse der Telefonanlage reagieren müssen, zum Beispiel das Anzeigen eines eingehenden Anrufs oder die Übersicht über die Warteschleifen, da dies mit einem eigenen kleinen Programm, das ausgewählte Bereiche der Telefonanlage imitiert, simuliert werden konnte. Der letzte Schritt beinhaltete alle Funktionen, die eine direkte Anbindung des Servers an ein Telefonie-System erfordern, wofür ein dem Live-Betrieb gleichendes Testsystem benötigt wurde.

3.3 Funktionen

Die Funktionen lassen sich grob in zwei Kategorien unterteilen: Jene, die Zugriff auf den Telefonie-Server erfordern (externe Funktionen), und jene, die nur den ACI-Server benötigen (interne Funktionen).

3.3.1 Interne Funktionen

- Anzeigen SSF Newsfeed

  Der ACI-Client soll es dem Agent ermöglichen, die aktuellen News zu seinen Mandaten zu sehen.

  Hierfür steht ihm ein automatischer Newsfeed zur Verfügung, der nach einem definierbaren Zeitintervall (z.B. alle 10 Sekunden) die Kopfzeile des nächsten Newseintrages anzeigt. Dieses selbständige Weiterschalten kann über einen Pause/Play-Button angehalten bzw. wieder aufgenommen und über jeweils einen Vor- und Zurück-Button gesteuert werden.

  Klickt der Agent einen Newseintrag an, so soll der vollständige Beitrag in einem Popup Fenster erscheinen.

  Der Agent soll nur solche News angezeigt bekommen, die einem seiner Mandate entspricht.

  Außerdem soll ein Button zur Verfügung stehen um die Historie der News aufrufen zu können. Diese soll ebenfalls in einem neuen Fenster angezeigt werden, alle Mandate des Agenten umfassen (bzw. komplett alle Mandate für bestimmte Rollen) und bei Auswahl eines Artikels diesen in den Newsfeed schreiben.

- Newsfeed-Editor

  Der ACI-Client soll es dem Agent ermöglichen, eigene Newsbeiträge zu erstellen bzw. vorhandene zu verwalten. Damit nicht jeder Agent Zugriff auf den Newsfeed-Editor bekommen kann, werden nur bestimmte Rollen dazu berechtigt.

  Nur berechtigte Agenten werden einen Button zum Hinzufügen und Bearbeiten von News unter dem Newsfeed angezeigt bekommen, welcher ein neues Fenster öffnet, das die momentan im Newsfeed befindlichen News enthält.

Hier kann er bestehende News ändern oder neue hinzufügen. Dafür stehen eine Kopfzeile und ein Textfeld zur Verfügung sowie ein Feld zur Bestimmung der Anzeigezeit (von wann bis wann die News im Newsfeed bleibt).
Ist die Anzeigezeit einer News abgelaufen, wird sie automatisch in die Historie geschrieben. Diese Funktion kann beim Editieren einer News über einen Button ausgelöst werden.

- Anzeigen des Agentenstatus

  Der ACI-Client soll es dem Agent ermöglichen, seinen momentanen Status zu sehen. Dieser soll dauerhaft in der Statuszeile des Clients angezeigt werden, inklusive der bisherigen Zeit, die der Agent in diesem Status verbracht hat.
  Zur schnellen Identifizierung des Status können verschiedenfarbige Icons benutzt werden, die den Statusmodi entsprechen (z.B. „Available“, „Away“, etc.). Die verfügbaren Modi werden aus der internen Datenbank des Servers ausgelesen.

- Anzeigen der Statuszeit

  Der ACI-Client soll es dem Agent ermöglichen, die Dauer seines momentanen bzw. seines letzten Anrufs zu sehen. Die Dauer des momentanen Anrufs wird in der Statuszeile mit einem Timer festgehalten, für die Dauer des letzten Anrufs ist ein Feld in der Statistikzeile des Clients vorgesehen.
  Wenn der Client bei Dienstantritt gestartet wird, ist das Feld für die Dauer des letzten Anrufes noch ausgeblendet.

- Anzeige & Verlinkung zum Ticketing-System

  Der ACI-Client soll es dem Agent ermöglichen, mit dem aktuellen Mandat direkt in das Ticketing-System verlinkt zu werden. Wird solch ein Link angeboten, dann wird das Mandat besonders markiert (zum Beispiel unterstrichen) und anklickbar.

- Anzeigen der Tagesstatistik des Call-Agenten

  Der ACI-Client soll es dem Agent ermöglichen, seine eigene Tagesstatistik abzurufen, indem er mit der Maus über das Feld „Tagesstatistik“ in der Statistikzeile des Clients fährt (eine kurze Verzögerung von 2-3 Sekunden wäre günstig, um versehentlichen Zugriff zu vermeiden). Seine aktuelle Statistik soll daraufhin in einem Tooltip-Fenster angezeigt werden. Sobald der Zeiger das Feld wieder verlässt, soll auch das Fenster verschwinden.

Die Statistik beinhaltet die Punkte „Eigener Status“, „Loginzeit“, „Anwesenheit“, „#eingehende Anrufe“, „#ausgehende Anrufe“ und „durchschnittliche Gesprächszeit“.

- WDB Suche

  Der ACI-Client soll es dem Agent ermöglichen, direkt aus dem Tool heraus eine mandatsbezogene Suchanfrage an die WDB zu senden. Hierfür steht ein Texteingabefeld in der Suchzeile des Clients zur Verfügung, in dem der zu suchende Text eingetragen werden kann. Als Mandat wird automatisch das Mandat des momentan geführten Anrufes bzw. das des zuletzt geführten Anrufes gewählt. Das Ergebnis der Suche wird in einem neuen Browserfenster angezeigt.

- Berechtigungssystem

  Der ACI-Client soll es dem Agent ermöglichen auf bestimmte Funktionen zugreifen zu können, sofern er die benötigte Berechtigungsstufe besitzt. Alle Agenten ab einer bestimmten Stufe sollen dazu ermächtigt sein, den Newsfeed Editor zu bedienen, sowie in der News-Historie Einträge zu allen Mandaten zu sehen anstatt nur zu den eigenen.

  Die Rollen sollen den Agenten über die ACI Seite der Adminkonsole des Openfire Servers zuordenbar sein.

- Administration

  Der ACI-Client soll es dem Agent ermöglichen, gewisse Einstellungen direkt zur Laufzeit zu verändern. Hierfür wird ein Untermenü in der Menüzeile des Spark bereitgestellt, über das ein Konfigurationsfenster erreicht werden kann. In diesem Fenster können zum Beispiel die Ports des Clients, die Anzeigezeit pro Newsbeitrag und andere Einstellungen geändert werden. Der Agent hat dabei die Möglichkeit, die Änderungen entweder zu übernehmen oder wieder zu verwerfen, es soll allerdings überprüft werden, ob die Eingaben auch zulässig sind.

  Die Einstellungen des ACI-Servers können über das Web-Interface der Adminkonsole vorgenommen werden. Hier können sämtliche Ports konfiguriert werden, sowie Zugriffe auf die interne Datenbank erfolgen. Jeder Eintrag der Tabellen kann dabei eingesehen, editiert oder gelöscht werden, und neue Einträge können angelegt werden.

- Persönliches Telefonbuch

  Der ACI-Client soll es dem Agent ermöglichen, sich ein eigenes Telefonbuch anlegen zu können. Darin soll zu jeder Nummer auch ein Name abgespeichert werden können.

  Wird in dem Telefonbuch eine Nummer angeklickt, so soll diese auch sofort angerufen werden können.

- Features ausblenden/minimieren

  Der ACI-Client soll es dem Agent ermöglichen, die einzelnen Bereiche der Oberfläche der Anwendung bei Bedarf ein- bzw. auszublenden. Hierbei wird die Ansicht in vier Bereiche aufgeteilt: den Call-Bereich (Begrüßung, Telefonnummer, Anrufername), den Statistik-Bereich (Tagesstatistik, Rufnummer-Historie, Statusübersicht, Dauer des letzten Calls), den Bereich für Warteschleife und WDB Suche und den Newsfeedbereich. Jeder davon soll unabhängig voneinander zu verbergen sein, und hinterher ist nur noch eine kurze Beschreibung des Bereichs zu sehen. Die Einstellungen, wie der Agent die einzelnen Bereiche angezeigt haben will, werden gespeichert und beim nächsten Programmstart automatisch entweder angezeigt oder ausgeblendet. Bestimmte Bereiche sollen bei gewissen Ereignissen allerdings zwingend angezeigt werden, so soll der Call-Bereich eingeblendet sein, während ein Telefonat geführt wird, die Warteschleife wird eingeblendet, wenn ein neuer Anruf eingereiht wird, und der Newsfeed wird sichtbar, wenn sich etwas an den Einträgen geändert hat.

- Arbeitsplatz konfigurieren

  Der ACI-Client soll es dem Agent ermöglichen, beim Login die Line bzw. den Kanal des Telefons frei zu wählen. Hierzu wird, bevor der Client gestartet wird, ein Login-Fenster angezeigt, welches neben den Feldern für die Daten zur Authentifizierung des Agenten ein Feld für die Wahl des Kanals bereitstellt. Der Agent kann sich entscheiden, die getroffene Auswahl für spätere Einlogvorgänge zu speichern, sodass er diese nicht immer wieder neu eingeben muss.

  Da die Verbindung zum Telefonie-Server über den ACI-Server abgewickelt wird, kann die IP-Adresse des Telefonie-Servers nicht direkt am Client verändert werden, sondern muss über die Admin-Konsole konfiguriert werden können.

3.3.2 Externe Funktionen

- Anzeigen der Rufnummer

  Der ACI-Client soll es dem Agent ermöglichen, die vollständige Rufnummer inkl. Landeskennung des Anrufers zu sehen. Dafür ist ein Feld am unteren Rand der Mandatszeile vorgesehen. Wird die eingehende Nummer unterdrückt, so soll anstelle der Nummer „anonym“ zu sehen sein.
  Die benötigten Daten werden aus der Telefonanlage ausgelesen.
  Wird kein aktiver Call geführt, so wird anstelle des Feldes zum Anzeigen der Rufnummer ein Texteingabefeld angezeigt, das zum Eingeben einer Rufnummer für den aktiven Rufaufbau genutzt wird.
- Anzeigen des Mandates

  Der ACI-Client soll es dem Agent ermöglichen, am oberen Rand der Mandatszeile das Mandat des aktuellen Anrufs zu sehen.
  Wird kein aktiver Call oder ein interner Call geführt, so wird stattdessen das Firmenlogo angezeigt.
- Anzeigen der Begrüßung

  Der ACI-Client soll es dem Agent ermöglichen, den zum aktuellen Mandat gehörenden Standard-Begrüßungstext angezeigt zu bekommen. Dieser soll in einem Feld direkt unter dem Mandatsnamen in der Mandatszeile eingeblendet werden.
  Wird kein aktiver Call geführt, so soll dieses Feld leer bleiben.
  Der Text wird anhand des Mandats aus einer eigenen Datenbank am Server ausgewählt, und mit den Daten des Agenten ergänzt.
- Rufnummer Historie

  Der ACI-Client soll es dem Agent ermöglichen, den Verlauf seiner Calls eines zu definierenden Zeitraums (z. B. der letzten 24 Stunden) aufgelistet zu bekommen, wobei die neuesten Anrufe oben stehen, die älteren unten. Hierfür steht ein Button in der Statistikzeile des Clients zur Verfügung. Dieser soll bei Betätigung ein neues Fenster öffnen, das die folgenden Informationen enthält: Zu jedem Call das Datum, die Uhrzeit, die Dauer des Anrufs, um welches Mandat es sich handelte, die genaue Rufnummer, wenn möglich den Namen des Users (sonst wird ein Platzhalter angezeigt) und ob es ein ein- oder ausgehender Anruf war.

Zusätzlich wird in dem Fenster rechts am Ende jeder Zeile ein Button angezeigt um sich die Aufzeichnung des Anrufs anhören zu können, sowie ein zweiter Button um die jeweilige Nummer direkt zurückrufen zu können.

- Anzeige des anrufenden Anwenders

  Der ACI-Client soll es dem Agent ermöglichen, den Namen des Anrufers zu sehen. Dafür ist ein Feld am unteren Rand der Mandatszeile vorgesehen.
  Wird kein aktiver Call geführt, so wird das Feld zum Anzeigen des Namens ausgeblendet.

- Anzeigen der Callagent Statusübersicht pro Mandat

  Der ACI-Client soll es dem Agent ermöglichen, eine Übersicht über die Status der anderen Agenten seiner Mandate einsehen zu können. Dies wird über ein ein-/ausblendbares Feld in der Statistikzeile des Clients ermöglicht.
  Ist das Feld eingeblendet, so soll pro Mandat des Agenten zu jedem Statusmodus die Anzahl der anderen Agenten, die sich in diesem Modus befinden, angezeigt werden.

- Anzeigen # Anrufer in Warteschleife

  Der ACI-Client soll es dem Agent ermöglichen, die Anzahl der wartenden Anwender in der Warteschleife zu sehen. Es sollen ihm hierbei nur die Anrufer seiner Mandate gezeigt werden. Ein Mandat ohne Anrufer in der Warteschleife wird ausgeblendet.
  Hierfür ist ein Feld in der Warteschleifenzeile des Clients vorgesehen, in dem zusätzlich zu der Anzahl der Anwender in der Warteschleife auch die Dauer des am längsten wartenden Calls angezeigt wird.
  Befindet sich kein Anrufer in einer der Warteschleifen der Mandate des Agenten, so wird in der Warteschleifenzeile stattdessen das Wort „Warteschleife“ in grauer Schrift angezeigt.
  Erhöht sich die Anzahl der Calls in der Warteschleife, so wird der Agent darüber über ein kleines Popup in der rechten unteren Ecke des Bildschirms informiert. Dieses Fenster soll den Fokus des aktuellen Fensters nicht beeinflussen, und nach einer kurzen Zeitspanne automatisch wieder verschwinden.

- Anzeige der Landessprache

  Der ACI-Client soll es dem Agent ermöglichen, bei einem eingehenden Anruf die Landeskennung des Ursprungslandes des Calls zu sehen. Hierfür sollen die jeweils gängigen Länderkürzel vor der Telefonnummer des Anrufers eingeblendet werden.

- Aktiver Wechsel des Telefonstatus

  Der ACI-Client soll es dem Agent ermöglichen, entsprechend den verfügbaren Status, seinen momentanen Status zu ändern. Dies wird über ein Drop-Down-Menü in der Statuszeile des Clients ermöglicht.

  Der ACI-Client soll es dem Agent nicht ermöglichen, eigene Status zu definieren, daher müssen die verfügbaren Status aus einer externen Quelle (TA, DB, o.a.) eingelesen werden, um auch später von einem Admin modifizierbar zu sein. Der ACI-Client muss beim Starten auf diese Quelle zugreifen können und Informationen wie Statusname („Bereit“, „Telefoniert“, etc.), zugehöriger Statusmodus („Available“, „Away“, „Extended Away“, „Do not Disturb“, etc.) und evtl. Pausengrund abrufen können.

  Wechselt der Agent seinen Status, muss der geänderte Agentenstatus auch an die TA weitergegeben werden können, bzw. wenn der Agent seinen Status direkt am Telefon ändert muss ebenfalls der Server und auch der ACI-Client benachrichtigt werden.

- Rufaufbau

  Der ACI-Client soll es dem Agent ermöglichen, seinen Telefonkanal direkt über das Tool zu steuern. Hierfür wird zunächst der Off-Hook-Modus benötigt, wodurch zu wählende Rufnummern am Client eingegeben bzw. ausgewählt werden können. Wird eine Nummer angewählt, so soll sich das System verhalten, als wäre das Telefon benutzt worden.

  Ebenso soll es möglich sein, eingehende Anrufe im Off-Hook-Modus über den Client anzunehmen, und laufende Anrufe zu beenden.

  Hierfür ist ein Feld in der Mandatszeile des Clients unter dem Begrüßungsfenster vorgesehen, in das der Agent eine zu wählende Nummer eintragen kann. Funktionen zum annehmen und beenden eines Anrufs werden dem Feld zum Anzeigen der Rufnummer hinzugefügt, sofern sich der Agent im Off-Hook-Modus befindet. Dieser soll ebenfalls über einen Menüpunkt ein- bzw. auszuschalten sein.

Kapitel 4: Realisierung

Damit die geschriebenen Plugins von Spark bzw. Openfire erkannt und geladen werden, muss man sich bei der Programmierung an spezielle Konventionen halten.

4.1 Spark-Plugins

Ein Projekt, das als Spark-Plugin geplant wird, muss eine bestimmte Ordnerstruktur aufweisen. Im Root-Verzeichnis des Projektes muss es die Ordner „src“, „lib“ und „build“ geben, sowie eine XML-Datei namens „plugin.xml“, die Spark die nötigen Informationen über das Plugin liefert. Diese Datei muss folgenden Aufbau haben:

```
<plugin>
    <name>Plugin Name</name>
    <version>x.y.z</version>
    <author>Author Name</author>
    <homePage>URL</homePage>
    <email>eMail</email>
    <description>Text</description>
    <class>Plugin Main Class</class>
    <minSparkVersion>x.y.z</minSparkVersion>
</plugin>
```

Der Ordner „build“ beinhaltet das ANT-Script zum automatischen Erstellen des Plugins, „lib“ beherbergt etwaige Ressourcen, die das Plugin zum Kompilieren benötigt und in „src“ befindet sich der Quellcode.

4.2 Openfire-Plugins

Ein Projekt, das als Openfire-Plugin geplant wird, muss eine bestimmte Ordnerstruktur aufweisen. Im Root-Verzeichnis des Projektes, das sich im Source-Pfad des Openfire im Unterordner „Plugins“ befinden muss, sind die Ordner „src“ und „lib“ erforderlich, sowie eine XML-Datei namens „plugin.xml“, die Openfire die nötigen Informationen über das Plugin liefert. Diese Datei muss folgenden Aufbau haben:

```
<plugin>
        <class>Plugin Main Class</class>
        <name>Plugin Name</name>
        <description>Text</description>
        <author>Author Name</author>
        <version>x.y.z</version>
        <date>dd.mm.yy</date>
        <minServerVersion>x.y.z</minServerVersion>
        <databaseKey>Plugin Database Key</databaseKey>
        <databaseVersion>Plugin Database Version</databaseVersion>
        <adminconsole>HTML Elements for Plugin Content in the Openfire
                Adminconsole</adminconsole>
</plugin>
```

Optional kann man im Root-Verzeichnis noch Inhalte für die Plugin-Übersicht in der Openfire-Adminkonsole bereitstellen. Dazu gehören ein Plugin-Logo im GIF-Format namens „logo_small.gif“ und „logo_large.gif“, eine kurze Beschreibung zum Plugin in „readme.html“ und eine Übersicht über den Änderungsverlauf des Plugins in „changelog.html“. Der Ordner „lib“ beherbergt etwaige Ressourcen, die das Plugin zum Kompilieren benötigt und in „src“ befindet sich der Quellcode. Letzterer Ordner wird aufgeteilt in die Unterordner „java“, wo sich die Java-Dateien des Plugins befinden, „database“, der eine Reihe von SQL-Dateien beinhalten kann, wenn das Plugin die Openfire-interne Datenbank nutzt, „web“, in dem sich sämtliche HTML und JSP Dateien befinden, die das Plugin braucht falls ein eigener Bereich in der Adminkonsole eingerichtet werden soll, und „i18n“, wo die PROPERTIES-Dateien untergebracht werden können, sollte das Plugin mehrsprachig angelegt sein.

4.3 Aufbau ACI-Client

Der ACI-Client ist in sechs Bereiche unterteilt:

- Statuszeile: Hier wird der Name des Agenten und sein Telefonstatus verwaltet.
- Mandatszeile: Hier werden alle Daten zum momentanen Mandat angezeigt.
- Statistikzeile: Hier können verschiedene statistische Daten abgerufen werden.
- Warteschleifenzeile: Hier werden Daten über die Anrufe in Warteschleife angezeigt.
- Suchzeile: Von hier aus kann eine Suchanfrage an die WDB geschickt werden.
- Newsfeed-Bereich: Hier wird der automatische Newsfeed der SSF verwaltet.

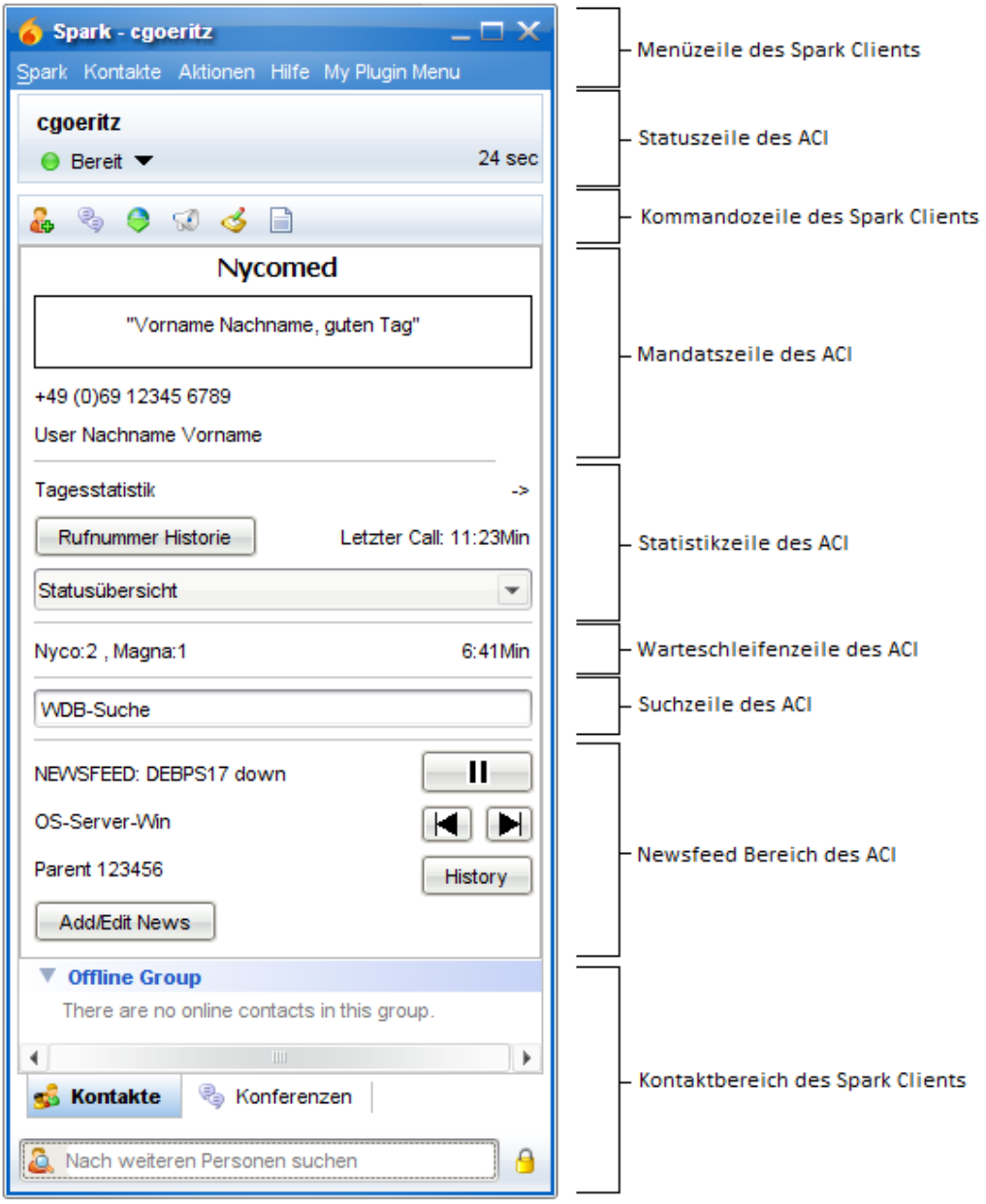

Abbildung 3: Gliederung ACI-Client

4.4 Aufbau ACI-Server

Das ACI-Server Plugin wird beim Start des Openfire-Servers geladen und richtet einen Listener ein, an dem sich die ACI-Clients anmelden können. Für jeden Client wird ein eigener Thread geführt, sodass sich die Kommunikation der Clients mit dem Server nicht gegenseitig überlappen kann.
Daneben stehen 4 weitere Services zur Verfügung, die von jedem Client-Thread genutzt werden können:

- Ein Datenbank-Manager zum Zugriff auf die Openfire-interne Datenbank
- Ein Datentransfer-Manager zum Senden/Empfangen von Dateien
- Der Newsfeed-Manager, der den aktuellen Newsfeed sowie die Newsfeed-Historie verwaltet
- Die Schnittstelle zur TA, über die sämtliche telefoniebezogenen Anfragen abgehandelt werden

Der Server führt eine Liste über alle verbundenen Agenten. Sie beinhaltet die Socket-Informationen für jede Verbindung. Loggt sich ein neuer Agent ein, wird seine Verbindung in diese Liste eingetragen und er wird als aktiver User mit seinen Daten aus der TA angelegt. Desweiteren wird eine Liste über alle angelegten User geführt, wobei von der Socket-Information auf den User gemappt werden kann. So kann jederzeit für jede Verbindung eindeutig der zugehörige User samt seinen Daten abgefragt werden.
Loggt sich ein Agent wieder aus, so wird er aus der Liste der User entfernt, und sein Socket wird aus der Liste der aktiven Verbindungen entfernt.
Als Schnittstelle zum Benutzer wird ein für den ACI-Server eigener Bereich im Webinterface der Openfire Adminkonsole bereitgestellt, über den Einstellungen des Servers vorgenommen und die Datenbank verwaltet werden können.

4.5 Client-Server-Architektur

Das ACI Plugin wird im Spark Client geladen und baut eine von Spark unabhängige Verbindung zum ACI-Server auf, der als Plugin auf dem Openfire-Server läuft. Der ACI-Server liest daraufhin die Daten des Agenten aus der TA aus und sendet sie an den ACI-Client, damit dieser eine neue User-Session anlegen kann. Diese Brücke zwischen Client und

TA ist nötig, damit sich auch Clients, die in Fremdnetzen laufen, aus denen normalerweise keine Kommunikation mit der internen Telefonie möglich ist, mit dieser verbinden können. Damit der ACI-Server nicht an eine bestimmte TA gebunden ist, wird die Verbindung zwischen diesen beiden Komponenten über eine spezielle API geregelt, die dann für verschiedene Telefonanlagen separat implementiert werden kann.
Am Openfire-Server können parallel andere Plugins betrieben werden, der ACI-Server legt sich einen eigenen Bereich in der Openfire-internen Datenbank an, sowie einen persönlichen Unterordner im Installationspfad zum Speichern der Newsfeed Daten. Er besteht aus drei großen Komponenten: dem Newsfeed-Teil, dem Datenbank-Teil und dem Telefonie-Teil. Der Newsfeed-Teil verwaltet den Newsfeed in einer XML-Datei im persönlichen Unterordner, sowie die News-History im eigenen Datenbank Bereich. Der Datenbank-Teil verwaltet den Zugriff der anderen beiden Komponenten auf die Openfire-interne Datenbank, erstellt die benötigten Tabellen und bietet Zugriff auf die Daten über die Openfire Admin-Konsole. Der Telefonie-Teil verwaltet die Kommunikation zwischen ACI-Client und TA, sowie die statistischen Daten der Agenten und die Begrüßungskonfiguration der Mandate.

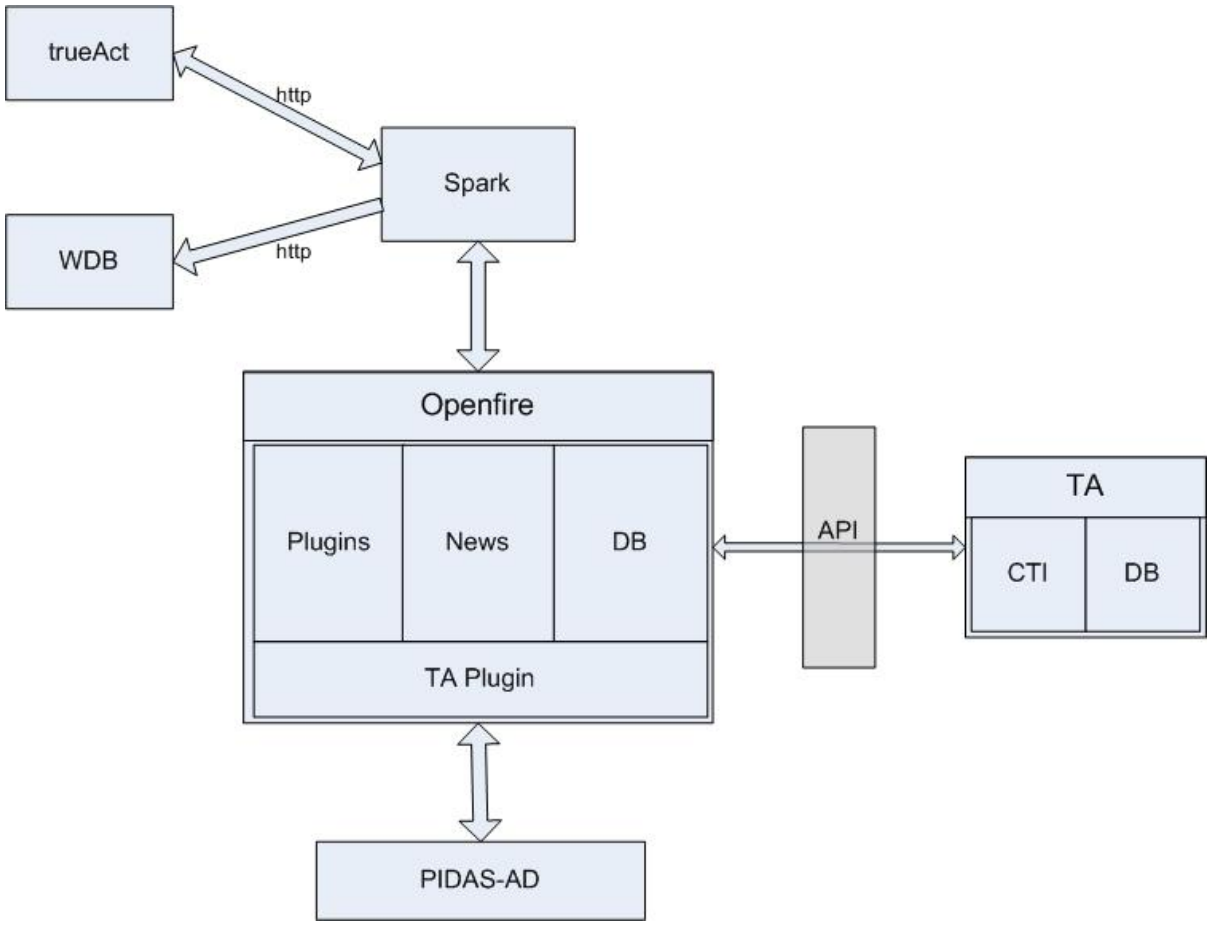

Abbildung 4: Aufbau Client-Server Architektur

Die Kommunikation zwischen ACI-Client und Server findet hierbei über eine eigene SSL-Verbindung statt, und jeder Client bekommt seinen eigenen Socket zugewiesen sowie einen eigenen Thread zum Behandeln der eingehenden Anfragen. Dieser Thread verwaltet zum einen den Telefon-Listener, der auf Ereignisse der Telefonanlage reagiert und diese an den Client weiterleitet, falls er davon betroffen ist. Zum anderen wartet er auf Eingaben des Clients um die gewünschten Funktionen auszuführen. Das Aufrufen einer Funktion dieser Art umfasst mehrere Datentransfers zwischen Client und Server, und damit keine anderen Threads gleichzeitig den Socket benutzen wird er für diese Zeitspanne beidseitig gesperrt.

Wenn der Client eine Anfrage sendet, antwortet der Server zunächst mit einer Empfangsbestätigung (bzw. gleich mit einem Fehler, falls etwas Unerwartetes passiert und er nicht zum Empfangen von Anfragen bereit ist, sodass der Client reagieren kann). Werden für die gewünschte Funktion noch Daten vom Client benötigt, so werden diese nun übertragen, danach wird die Funktion ausgeführt und der Server sendet das Ergebnis an den Client zurück. Ist dieser Austausch abgeschlossen, so wird der Socket auf beiden Seiten wieder freigegeben, und es kann auf neue Anfragen gewartet werden.

4.6 Nicht-Telefonie Teil

Zunächst habe ich einige Versuche unternommen, mich in das bestehende Spark-System einzuarbeiten. Zu diesem Zweck habe ich testweise neue Panels in das Spark-GUI eingefügt, auf bereits vorhandene Elemente zugegriffen und diese verändert und erweitert, sowie die voreingestellten Statusmodi modifiziert. Diese Bereiche habe ich als erste Testversuche ausgewählt, da sie die Grundlage für nahezu alle Funktionen des ACI bilden. Die Vorgehensweise ähnelte sich dabei bei allen Versuchen, da die Implementation des Spark für Plugin-Entwickler einen zentralen Zugangspunkt bereitstellt: den Spark Manager.

Der Spark Manager gibt einem Plugin Zugriff auf alle wichtigen Unterfunktionen des Spark; so bekommt man zum Beispiel mit SparkManager.getConnection() Zugriff auf die zugrunde liegende XMPP Verbindung des Spark zum Openfire, und kann dieser beispielsweise einen eigenen ConnectionListener hinzufügen, der auf Ereignisse die Verbindung betreffend reagieren kann. SparkManager.getWorkspace() erlaubt dem Plugin, auf die Elemente des Hauptfensters des Spark zuzugreifen, zum Beispiel auf die Status Zeile oder die Kontaktliste, SparkManager.getMainWindow() stellt das den Workspace beinhaltende Fenster zur Verfügung und somit auch zum Beispiel die Menüzeile des Spark. Da nun sichergestellt war, dass ausreichende Elemente des Spark vom Plugin bearbeitet

werden können, konnte ich mit der Arbeit an den Funktionen, die ohne die Telefonie auskommen, beginnen.

Bevor jedoch erste Implementierungen gemacht werden konnten, habe ich ein Dokument erstellt, welches eine Übersicht über den grundlegenden Aufbau der beiden Plugins beschreibt, sowie zu jeder der geforderten Funktionen die Anforderung aus dem Pflichtenheft, die User-Akzeptanzkriterien und einen ersten Lösungsansatz liefert. Dieses Dokument bin ich dann mit den Kunden – also den Verantwortlichen der SSF – durchgegangen und wir haben jede Funktion noch einmal besprochen, sodass von vornherein ein Konsens zwischen dem Entwickler und den Kunden bestand was den genauen Inhalt jeder Funktion angeht. Außerdem wurden auch noch von Kundenseite einige Änderungsvorschläge und –wünsche vorgebracht, die ich in das Dokument einarbeiten konnte.

Als nächstes entwarf ich ein provisorisches GUI für das ACI Plugin, damit für spätere Vorführungen beim Kunden etwas Greifbares vorhanden war. Die erste zu planende Funktion sollte der Newsfeed werden, nebst jenen Funktionen, die damit zusammenhängen (z.B. der Editor). Ich habe bei der Entwicklung eine Herangehensweise entsprechend der Test-Driven-Development(TDD)-Vereinbarung gewählt, und zunächst Tests für die zu implementierenden Methoden, die der Newsfeed benötigt, geschrieben.

Der Newsfeed stellt den größten von der Telefonie unabhängigen Teil dar, und würde nach der Fertigstellung bereits eine erste Präsentation vor den Kunden rechtfertigen, weswegen diese Funktion als Ausgangspunkt gewählt wurde. Lösungsansatz:

> Der Newsfeed wird als RSS-XML-Datei vom Server verwaltet und nur beim ersten einloggen eines Clients an diesen übertragen oder nach einer Änderung des Inhalts an alle verbundenen Agenten übermittelt. Das ACI-Tool verwaltet den Newsfeed in einem eigenen Thread. Dieser liest die Datei aus und erstellt eine Liste der Einträge, die den Mandaten des Agenten zugeordnet sind. Ungültige Einträge aufgrund von fehlenden oder falsch formatierten Feldern werden dabei übersprungen. Aus dieser Liste wird nach einem vordefinierten Zeitintervall der jeweils nächste Eintrag, dessen Gültigkeitszeitraum bereits begonnen hat, im Newsfeed Bereich dargestellt. Hierfür wird nur die erste Zeile ausgelesen und der Rest zunächst verworfen. Ist das Ende der Liste erreicht, springt er zurück zum Anfang und überprüft zuvor, ob eine neue Version der Datei vorliegt.

Jede Zeile des Newsfeed verfügt über einen Mouse-Listener, und wenn ein Klick erfolgt wird ein neues Fenster geöffnet und der gesamte Eintrag, der dem aktuellen Index der Liste entspricht, ausgegeben.

Der Pause/Play-Button sendet bei Aktivierung einfach ein „suspend“ bzw. „resume“ an den Newsfeed-Thread, die den Timer entweder pausieren bzw. wiederaufnehmen.

Die Vor- und Zurück-Buttons variieren den momentanen Index der Liste um eins nach oben bzw. unten. Somit wird die nächste bzw. die vorige News ausgewählt.

Die News-History wird ebenfalls vom Server in einer eigenen Datenbank verwaltet. Wird am Client der History-Button betätigt, wird eine Anfrage an die serverseitige Datenbank gesendet und die entsprechenden Einträge ausgelesen (nur die Mandate des Agenten betreffend, bzw. zu allen Mandaten, je nach Rolle). Diese werden in einem neuen Fenster nach Datum sortiert bis ein Jahr zurückliegend dargestellt, und wenn einer davon angeklickt wird, wird der Newsfeed automatisch pausiert und der Eintrag temporär in diesem angezeigt. Sobald der Newsfeed wieder fortgesetzt wird, kehrt er zu seiner aktuellen Liste zurück.

Der Newsfeed-Editor stellt praktisch einen Teil der Newsfeed-Funktion dar, und wurde somit in einem Zug mit dem Newsfeed entworfen. Lösungsansatz:

Der Newsfeed-Editor soll die serverseitige Newsfeed-XML-Datei modifizieren können. Da dies mehreren Agenten möglich sein wird, muss das gleichzeitige Bearbeiten der Datei verhindert werden. Will ein Agent den Newsfeed ändern während dies gerade ein anderer unternimmt, so soll er darüber informiert werden und keinen Zugriff auf die Datei bekommen.

Wenn ein Agent den Editor öffnet, wird eine Benachrichtigung an den Server gesendet, dass dieser die Datei sperren kann. Da der Agent bereits die aktuelle Datei vorliegen hat, muss sie nicht neu übermittelt werden. Er bekommt dann in einem neuen Fenster den gesamten Inhalt der Datei und im oberen Teil des Fensters einen Editierbereich angezeigt.

Will der Agent einen neuen Eintrag erstellen, muss er mindestens die erste Zeile des Newseintrages sowie die Felder „Mandat“ und „Anzeigezeitraum von/bis“ ausfüllen. Die restlichen Felder sind optional. Das Mandat wird aus einer vordefinierten Liste ausgewählt, sodass nur gültige Mandate eingetragen werden können. Der Anzeigezeitraum soll bis auf die Minute genau angegeben werden können, wobei

Datum und Uhrzeit aus einem einblendbaren Kalender ausgewählt oder manuell eingegeben werden können. Die Felder „von“ und „bis“ sind standardmäßig mit dem aktuellen Datum bzw. mit dem Datum des nächsten Tages vorinitialisiert, sodass eine neue News ohne Änderung des Zeitraums genau 24 Stunden gültig ist.
Will der Agent einen bestehenden Eintrag ändern, klickt er diesen unten im Fenster an, und die Daten des Eintrags werden in die Felder des Editierbereichs geschrieben. Der Button „Erstellen“ ändert sich in „Ändern“ und der Agent kann genauso vorgehen wie beim Erstellen eines neuen Eintrages (inklusive der gleichen Beschränkungen).
Wird der Button betätigt, wird in der Datei entweder ein neues Item angelegt oder ein bestehendes verändert, und das Fenster wird wieder geschlossen. Die veränderte Datei wird an den Server gesendet, und dieser muss sie an alle verbundenen Clients weiterleiten.
Das Fenster kann aber auch geschlossen werden, ohne dass der Button betätigt wurde. In beiden Fällen muss der Server darüber informiert werden, damit er die Datei wieder für andere Agenten freigeben kann.
Der dritte Button („Abgelaufen“) ist nur beim Editieren eines bestehenden Newsbeitrages bedienbar und setzt dessen Ablaufzeit auf die momentane Uhrzeit, sodass er bei nächster Gelegenheit vom Server in die News-History verschoben wird.
Die News-History-Datenbank kann von den Agenten nicht bearbeitet werden – sie wird vom Server selbst verwaltet. Dieser überprüft in regelmäßigen Abständen (z.B. jede Minute) den Gültigkeitszeitraum jedes Eintrages in der Newsfeed-XML-Datei. Ist einer davon abgelaufen, so wird der Eintrag aus der Datei entfernt und in die History-Datenbank kopiert. Die veränderte Datei muss an alle verbundenen Clients weitergeleitet werden.

Des Weiteren wurden einige der weniger umfangreichen Funktionen implementiert, deren Entwurf größtenteils bereits in der Versuchsphase geschehen ist, nämlich das Anzeigen und ändern des Agentenstatus, sowie das Mitprotokollieren der Statuszeiten. Lösungsansätze:

Beim Anmelden eines Clients an der TA erhält dieser eine Liste mit allen verfügbaren Statusmodi, sowie die Information, in welchem Status sich der Agent befindet. Der Client baut daraufhin die Statuszeile auf und startet den Timer.

Findet ein Statuswechsel statt, so wird der Server darüber benachrichtigt, die aktuelle Zeit des Timers an diesen übermittelt, und der neue Status mit einem frischen Timer im Client angezeigt. Dadurch kann der Server die Statuszeit des Agenten in dem entsprechenden Datenbankeintrag aktualisieren. Gleichzeitig wird der neue Status an die TA weitergegeben, damit der Status des Clients mit dem am Telefon übereinstimmt.

Ging der Statuswechsel vom Telefon aus, so wird der Listener der API eine Benachrichtigung an den Client senden, die den neuen Status enthält und einen Wechsel erzwingt.

Der Client stellt einen Presence-Listener zur Verfügung, der jedes Mal aufgerufen wird, wenn sich der Status des Agenten ändert. Dieser prüft zunächst, ob der bisherige Status der Telefon-Status war, damit das Feld für die Dauer des letzten Calls aktualisiert werden kann, und sendet daraufhin eine Status-Ändern-Nachricht sowie die Zeit des Status-Timers an den Server.

Der Server liest die Zeit des bisherigen Status des Agenten aus der agentenspezifischen Datenbank aus, addiert die übermittelte Zeit und schreibt das Ergebnis wieder zurück. Erst danach wird der Status des Agenten auch am Server geändert, damit die Zeit nicht in die falsche Tabelle geschrieben wird, und die TA wird mittels der API benachrichtigt, auf welcher Line sich der Status ändert.

Nachdem der Statuswechsel auf der Clientseite bereits funktionierte, lag es nahe, am Server die Tagesstatistiken der Agenten mitzuschreiben, und den Clients zur Verfügung zu stellen.
Lösungsansatz:

Das Tooltip-Fenster des Tagesstatistik-Feldes beinhaltet Platzhalter, die durch die entsprechenden Daten, die der Client vom Server erhält, ersetzt werden. Schickt der Client die Anfrage nach der Statistik an den Server, so fragt dieser seine Datenbank ab und sucht die erforderlichen Daten des betreffenden Agenten heraus. Daraus wird eine Antwort für den Client gebaut, sodass dieser die übrigen Werte berechnen und die Felder des Tooltips befüllen kann.

Eine weitere weniger aufwändige Funktion stellt die WDB-Suche dar, da hier im Endeffekt lediglich ein String an den Browser übergeben werden muss, weswegen sie in einer provisorischen Form ebenfalls mit eingebaut wurde. Lösungsansatz:

Dem Server steht eine eigene Tabelle in der internen Datenbank zur Verfügung, in der die einzelnen Foren-IDs den Mandaten zugeordnet werden können. So können die IDs auch im Nachhinein noch angepasst werden, sollte sich am Forum etwas ändern.
Bei einem eingehenden Anruf wird diese Datenbank abgefragt und die entsprechende ID ausgelesen. Sollte für das Mandat keine eigene ID gefunden werden, so wird die ID zur Suche in allen Mandatsforen ausgewählt.
Die ermittelte ID wird in die Benachrichtigung über einen neuen Anruf für den Client eingebaut und übertragen. Der Client extrahiert sie aus der Nachricht und speichert sie ab, damit sie auch noch verfügbar ist, nachdem der Anruf beendet wurde.
Wird nun eine Suchanfrage an die WDB vom Client aus gestartet, so schickt dieser einen Aufruf an den Standardbrowser des Systems, in den die gespeicherte ID und der eingegebene Suchtext eingebettet wurde.

4.7 Telefonie Teil (sim)

Um die Telefonfunktionen planen und vor allem auch vorführen zu können, ohne ein dem originalen Telefonieserver ähnelndes System zur Verfügung zu haben, habe ich ein separates Programm geschrieben, dass die Grundfunktionen der Telefonie simuliert. Es konnte einen eingehenden Anruf auf einem bestimmten Kanal mit einer Anrufernummer und einem Mandat melden, es konnte Anrufe in die Warteschleife einreihen bzw. entfernen, und den Status eines Agenten ändern. Damit die daraus resultierenden Ergebnisse auch auf das eigentliche System übertragen werden konnten, schrieb ich wiederum vorher Tests, um die Methoden genau zu beschreiben. Mit der Möglichkeit im Sinn, das ACI System später auch an andere Telefonsysteme anbinden zu können, sollte die Kommunikation zwischen ACI-Server und Telefonserver über eine definierte Schnittstelle (API) erfolgen, die gegebenenfalls für andere Systeme neu implementiert werden kann. Aufbau API:

Die API muss folgende Funktionen beschreiben:

- Listener für eingehende Anrufe anlegen
- Rufnummer und Mandat bei eingehenden Anrufen melden
- Die Agenten einloggen und den Mandaten zuordnen
- Den Status eines Agenten auslesen und schreiben
- Die Rufnummer-History eines Agenten abfragen
- Die Call-Queue einer Line abfragen
- Statistische Daten eines Agenten abfragen
- Den Off-Hook-Modus für die Agenten steuern (ein/aus, Anruf annehmen, Anruf beenden, Anruf aufbauen)

Die API selbst muss folgende Methoden bereitstellen:

```
public interface TelephoneAPI {

    public void addTelephoneListener(TelephoneListener cl);

    public void removeTelephoneListener(TelephoneListener cl);

    public String[] agentLogin(String[] loginData);

    public void agentLogout(String line);

    public void changePresence(String line,String newPresence);

    public String getCallHistory(String line);

    public String getReadyStatus();

    public String getPhoneStatus();

    public void initiateCall(String line,String number);

    public void acceptCall(String line);

    public void endCall(String line);

    public boolean getOffHookMode(String line);

    public void setOffHookMode(String line,boolean mode);

    public String listOperators(String mandate);

}
```

Eine Implementation dieses Interfaces sollte stets dem Singleton Design Pattern folgen, da jeder Agent seinen eigenen Thread hat, aber alle auf die gleiche API zugreifen müssen. Eine API kann verschiedene Pausen-Status für die Agenten definieren, aber sie muss immer genau einen Bereit-Status und einen Telefon-Status bereitstellen.
Die Methode addTelefoneListener fügt der Liste einen neuen Listener hinzu und wird immer dann gerufen, wenn sich ein neuer Agent anmeldet.
Die Methode removeTelefoneListener löscht einen bestimmten Listener wieder aus der Liste und wird immer dann gerufen, wenn sich ein Agent vom Server abmeldet.
Die Methode agentLogin muss alle wichtigen Login Daten des Agenten erhalten, meldet ihn an der TA an, liest seine persönlichen Daten aus und gibt sie an den Client zurück. Sie wird zusammen mit addTelefoneListener aufgerufen. Der Aufbau der Daten erscheint wie folgt:
{Skill-Liste, Rolle, Start-Status, verfügbare Status}
Skill-Liste: alle dem Agent zugeordneten Mandate
Mandat1/Mandat2/…./MandatN
Rolle: Die Berechtigungsebene des Agenten
Start-Status: der Status in dem sich der Agent gerade an der TA befindet, der Client startet in diesem Status
verfügbare Status: alle dem Agent zur Verfügung stehenden Status
Statusart1:Statustext1/…./StatusartN:StatustextN
Statusarten: available, phone, away, dnd

Die Methode agentLogout muss die Line des Agenten kennen, damit dieser an der TA abgemeldet werden kann. Sie wird zusammen mit removeTelefoneListener aufgerufen.
Die Methode changePresence teilt der TA mit, dass der Agent seinen Status über den Client ändern möchte und auf welcher Line dieser Wechsel stattfindet.
Die Methode getCallHistory ruft für eine bestimmte Line die Liste der geführten Telefonate aus der TA ab und liefert diese an den Client weiter.
Die Methoden getAvailableStatus und getPhoneStatus lesen den jeweiligen Statustext aus der TA aus.
Die Methode initiateCall startet (im Off-Hook-Modus) einen Anruf auf einer bestimmten Line zu einer bestimmten Telefonnummer.
Die Methode acceptCall nimmt einen Anruf (im Off-Hook-Modus) auf einer bestimmten Line an.
Die Methode endCall beendet einen Anruf (im Off-Hook-Modus) auf einer bestimmten Line.

Die Methode getOffHookMode prüft, ob eine bestimmte Line im Off-Hook-Modus ist.
Die Methode setOffHookMode setzt den Off-Hook-Modus für eine bestimmte Line auf aktiv bzw. inaktiv.
Die Methode listOperators sucht alle eingeloggten Agenten zu einem bestimmten Mandat sowie deren momentanen Status heraus.

Mit Hilfe der API standen dann alle Methodenzum Entwickeln der Telefonfunktionen des ACI-Clients zur Verfügung. Da die meisten davon auf einen eingehenden Anruf reagieren müssen, wurde als erstes ein Telefon-Listener entworfen, der alle wichtigen Informationen an den Client weiterleiten kann. Aufbau Listener:
Die API führt eine Liste über alle angemeldeten Listener, sodass jeder bei einem entsprechenden Ereignis informiert werden kann. Die Struktur des Listeners sieht wie folgt aus:

```
public abstract class TelephoneListener {

public abstract void incomingCall(String line,String telNr,
                                  String mandate,String country);
public abstract void endCall(String line);

public abstract void changePresence(String line,String presence,
                                    boolean outboundCall);
public abstract void updateCallQueue(String line,String queue,
                                  String time,boolean newCall);
}
```

Jede dieser Methoden hat als Parameter eine bestimmte Line für die sie aufgerufen wird. Somit wird sichergestellt, dass nur der richtige Client angesprochen wird.
Die Methode incomingCall wird gerufen, wenn ein neuer Anruf auf einer bestimmten Line eingeht, und teilt dem Client auch gleich die Telefonnummer des Anrufers sowie das zugehörige Mandat und die Länderkennung mit.
Die Methode endCall informiert den Client lediglich darüber, dass ein zuvor geführter Anruf nun beendet ist.
Die Methode changePresence teilt dem Client einen Statuswechsel, der von der TA ausgeht, mit, und übergibt auch gleich den neuen Status, sodass dieser gesetzt werden kann und ob der Wechsel aufgrund eines ausgehenden Anrufs passierte, damit die Statistik des Agenten angepasst werden kann.

Die Methode updateCallQueue wird von der TA jedes Mal benutzt, wenn sich die Warteschleife für eine bestimmte Line irgendwie geändert hat. Der Client bekommt dann die neue Warteschleife sowie die Zeit des momentan am längsten wartenden Anrufs und ob die Änderung auftrat weil ein neuer Call hinzukam.

Aus diesen Daten kann der Client dann sämtliche anzeigende Funktionen versorgen.
Lösungsansatz Anzeigen der Rufnummer/Anzeigen des Mandates/Anzeigen der Begrüßung/Anzeige des anrufenden Anwenders/Anzeige der Landessprache:

Der Server meldet für jeden verbundenen Agent einen eigenen Listener an der API an, der eine Benachrichtigung an den Agent zurücksendet, sobald auf seiner Line ein Anruf eingeht. Die Benachrichtigung besteht aus den Informationen für Anrufernummer, Mandat, Begrüßung, WDB-ID, Landeskennung und Ticketsystem-Link.
Für das Anzeigen der Rufnummer wird das Eingabefeld für eine zu wählende Telefonnummer ausgeblendet und das Feld zur Darstellung der Nummer des aktiven Calls eingeblendet.
Sollte der Client an dieser Stelle keine Zahl empfangen, so wird er davon ausgehen, dass die Nummer unterdrückt wurde, und stattdessen das Wort „anonym“ anzeigen.
Sobald der Anruf beendet ist, schickt der Listener wiederum eine Benachrichtigung über dieses Ereignis an den Client, der dann das Telefonnummer Feld ausblendet und das Eingabefeld für zu wählende Telefonnummern wieder einblendet.

Hat der Server das Mandat empfangen, fragt er die Ticket-Link-Datenbank ab, ob es dort für dieses Mandat einen Eintrag gibt. Ist dies der Fall, wird der Link in die Antwortnachricht eigebettet. Der Client kann dann den Text zusätzlich unterstreichen und mit einem Mouse-Listener versehen, der bei einem Klick einen entsprechenden Link zum Ticketsystem generiert und an den Standardbrowser übergibt.
Sobald der Anruf beendet ist, schickt der Listener wiederum eine Benachrichtigung über dieses Ereignis an den Client, der dann das Mandat durch das Firmenlogo ersetzt. Dies passiert ebenso, wenn ein interner Anruf geführt wird, indem der Listener diese Information in die Benachrichtigung einbaut.

Der Server verfügt in der internen Datenbank über einen separaten Bereich zur Speicherung von Begrüßungstexten, wobei jedem Mandat ein eigener Text zugeordnet

werden kann. Die Daten können über einen eigenen Bereich in der Admin-Konsole des Servers eingegeben, geändert oder gelöscht werden.
Wenn die API einen eingehenden Anruf meldet, erhält der Server das zugehörige Mandat. Somit kann er aus der Datenbank den entsprechenden Begrüßungstext auslesen und mit in die Benachrichtigung für den Client einbinden. Schlüsselwörter im Text werden zuvor durch die persönlichen Daten des Agenten ersetzt.
Wird kein passender Eintrag in der Datenbank gefunden, bleibt dieser Teil der Benachrichtigung leer, der Client zeigt somit ein leeres Feld an.
Sobald der Anruf endet wird das Feld wieder geleert.

Für das Anzeigen der Landeskennung ermittelt die API das Kürzel anhand der Vorwahl der Telefonnummer und fügt es direkt in die Antwortnachricht ein.
Sobald der Anruf beendet ist, schickt der Listener wiederum eine Benachrichtigung über dieses Ereignis an den Client, der dann das Feld für die Länderkennung wieder ausblendet.

Außer auf eingehende Anrufe kann der Listener ebenfalls auf Wechsel des Agentenstatus und Änderungen einer Warteschleife reagieren. Lösungsansatz Anzeigen # Anrufer in Warteschleife/Aktiver Wechsel des Telefonstatus:

Die Call-Queue wird ausschließlich von der TA verwaltet, und jedes Mal, wenn sich die Warteschleife für einen bestimmten Agent (bzw. seine Line) ändert, wird der entsprechende Client über den Listener der API darüber informiert. Der Server sendet dann eine Nachricht an den Client, der die Warteschleifenzeile daraufhin neu aufbauen kann. Da dies jedes Mal geschieht, wenn sich die Warteschleife ändert (egal ob ein Call hinzukommt oder wegfällt), muss der Client zusätzlich noch über die Art der Änderung informiert werden, damit er bei einem hinzukommenden Call eine entsprechende Nachricht anzeigen kann.
Erhält der Client eine Nachricht mit einem Call-Queue-Präfix, springt er zum Ende des Präfixes und zählt die Mandate (wie viele Calls des gleichen Mandats in der Warteschleife sind ist nicht explizit Teil der Nachricht, es ergibt sich aus der Anzahl der Wiederholungen des Mandats). Diese können dann in der Warteschleifenzeile angezeigt werden.

Die Zeit des am längsten wartenden Anrufers wird zum Zeitpunkt der Änderung von der API übermittelt (in Millisekunden), sodass der Client einen Timer beginnend mit der entsprechenden verstrichenen Zeit starten kann.
Das Feld newCall ist entweder true oder false, damit bei true eine kleine Nachricht am unteren Bildschirmrand eingeblendet werden kann, die über einen neuen Anrufer in der Warteschleife informiert.
Leert sich die Warteschleife später wieder, wird von der API zwar die gleiche Nachricht übermittelt, nur dass dieses Mal das Mandatsfeld leer ist. Bekommt der Client eine Nachricht ohne Mandate, wird der Timer ausgeblendet und statt den Mandaten das Wort „Warteschleife“ in grau angezeigt.

Eine von der TA ausgehende Statusänderung aktiviert den Listener des Servers, der eine Nachricht an den Client sendet. Diese enthält den Befehl, den Status zu ändern, sowie den neuen Status. Wenn der Client daraufhin den neuen Status setzt, werden wiederum die gleichen Schritte ausgeführt wie beim Wechsel am Client beschrieben.

Die Anzeige der Callagent Statusübersicht pro Mandat benötigte allerdings ein laufendes Originalsystem, da auch die Daten der Agenten, die das ACI nicht benutzen, angezeigt werden sollen. Daher habe ich hierfür vorerst nur die Daten der mit dem Openfire verbundenen ACI-Clients ausgelesen, und somit diese Funktion noch nicht über die API geregelt. Lösungsansatz:

Wenn ein Agent die Statusübersicht anfordert, werden seine sämtlichen Mandate geprüft. Für jedes davon wird die Liste aller verbundenen Agenten durchsucht und die Userdaten abgefragt, ob sie das Mandat ebenfalls haben und ist dies der Fall, so wird ihr momentaner Status notiert.
Der Client, der die Anfrage gestellt hat, bekommt eine Nachricht zurück, die zu jedem seiner Mandate eine Liste von Status enthält. Diese kann er dann zählen und daraus die Statusübersicht zusammensetzen.

4.8 Telefonie Teil (live)

Da die Telefoniefunktionen größtenteils bereits im ACI-Tool implementiert waren, musste als nächster Schritt die Verbindung zwischen ACI-Server und TA hergestellt werden. Hierfür musste ich die Implementation der API von der Simulation auf das Livesystem ändern. Diesbezüglich fand ein Treffen mit den Herstellern der Telefonserver-Software statt (itCampus technology), da diese in der SSF gerade einen Backup-Server installierten, und ich konnten einige Details in der Umsetzung klären. Für eine genauere Einführung in den Umgang mit dem Telefonserver haben sie mir eine Tagesschulung in Wien angeboten, wobei sie auch gleich einen Testserver einrichten wollten. Das Ansprechen der Serverschnittstelle funktioniert relativ simpel über eine JQuery in einem http Aufruf, der ein JSON Objekt als Antwort liefert. Beispiel:
Aufruf:
http://atgrzstest01:9999/itctec/cti/2010/08/jquery?func=System.GetMecctidInfo
Antwort:
{"result":{"Title":"mecctid","Vendor":"itCampus technology GmbH","Version":"3.3.5"}}

Somit konnte ich die bestehenden Funktionen an diese Schnittstelle anpassen ohne eine größere Änderung an der Grundarbeitsweise vornehmen zu müssen. Außerdem konnten jetzt die noch fehlenden Funktionen nachgeholt werden, da der Testserver hierfür die nötigen Mittel bereitstellte.

Der Zugriff auf den Datenbankserver der Telefonanlage musste separat vom Zugriff auf die Openfire-interne Datenbank erfolgen, damit später im Livebetrieb die Datenbanken voneinander getrennt betrieben werden können. Der Telefonserver benutzt eine SQLExpress Datenbank, aus der die Rufnummer-Historie eines Agenten ausgelesen werden kann.
Lösungsansatz:

Will ein Agent seine persönliche Call-History abrufen, wird eine entsprechende Anfrage über den Server an die API gesendet. Diese liest die benötigten Daten anhand der Line des betreffenden Agenten aus der TA aus, setzt sie zu einer einzigen Nachricht zusammen und schickt sie an den Client zurück. Dieser zerlegt die Nachricht wieder in ihre Bestandteile und listet das Ergebnis in einem neuen Fenster auf.

Das Fenster hat dabei eine feste Größe, in dem maximal acht Zeilen angezeigt werden können. Befinden sich mehr Anrufe in der History, wird ein Scrollbalken eingeführt.

Felder, deren Text über ihre Grenzen hinausgeht, können zwar nicht vergrößert werden, allerdings wird der vollständige Text als Tooltip angezeigt.

Außerdem hat die SSF sich in ihrer Livedatenbank bereits eigene Tabellen und Views eingerichtet, die das Abfragen von Agenten- und Mandatsdaten vereinfachen.

Die letzte Aufgabe bestand darin, über das ACI-Tool nicht nur Gespräche zu erhalten, sondern auch aktiv Telefonate einleiten zu können. Hierfür wird der Off-Hook-Modus des Telefonservers benötigt, der es Clientprogrammen erlaubt, über die JQuery Schnittstelle den Kanal des Agenten zu steuern. Dadurch kann der Benutzer direkt über das Tool Anrufe annehmen, beginnen oder beenden. Lösungsansatz:

Im ACI-Untermenü der Menüzeile des Spark wird eine Option hinzugefügt, den Off-Hook-Mode ein- bzw. auszuschalten. Hierfür wird bei jedem Aufruf des Menüs zunächst der momentane Zustand abgefragt, und wenn die Option gewählt wird, wird der Zustand gewechselt.

Befindet sich der Agent im OHM, so kann er in das Feld für zu wählende Telefonnummern eine Nummer eingeben und anrufen. Dabei wird eine Anfrage an die API gesendet, die auf dem Kanal des entsprechenden Agenten einen Anruf initialisiert zu der Nummer, die zuvor übertragen wurde.

Befindet sich der Agent nicht im OHM und versucht trotzdem eine Nummer zu wählen, so wird er darüber informiert, dass zuerst der OHM aktiviert werden muss.

Bei eingehenden Anrufen wird ebenfalls der Zustand des OHM auf diesem Kanal abgefragt, und wenn er aktiviert ist, dann werden der Zeile zur Anzeige der Rufnummer Buttons zum Annehmen bzw. Beenden des Anrufs hinzugefügt. Werden diese betätigt, so wird ebenfalls eine Anfrage an die API gesendet, auf dem Kanal des Agenten entweder einen wartenden Call anzunehmen oder einen laufenden Call zu beenden.

Die gleichen Funktionen werden benutzt, wenn eine Nummer aus der Call-Historie zurückgerufen oder eine aus dem Telefonbuch angerufen werden soll.

Kapitel 5: Administration / Webinterface

Um vor allem die Grundeinstellungen des ACI-Clients und –Servers später an Veränderungen anpassen zu können, musste ich eine Möglichkeit schaffen, diese variabel zu halten. Der Client bietet hierfür das Einstellungsfenster (s.o.), der Server dagegen hat einen eigenen Bereich im Webinterface der Adminkonsole des Openfire zur Verfügung. Hier kann der Administrator über Webseiten die Konfiguration des ACI-Servers anpassen, sowie auf die ACI-Datenbank, die in der Openfire-internen Datenbank eingebettet ist, zugreifen.
Lösungsansatz:

Die Integration der Serverkonfiguration in die Adminkonsole erfolgt über JSPs, die die momentanen Werte der Ports aus den JiveGlobals des Openfire auslesen. Dabei handelt es sich um Properties, wodurch Änderungen auch ohne Neustart des Servers übernommen werden können. Dies wird dadurch ermöglicht, dass die Server-Sockets nur eine begrenzte Zeit auf einen Client warten und danach beendet und wieder neu instanziiert werden. Die Ports werden bei jeder Instanziierung neu aus den Properties ausgelesen.
Die Kommunikation mit der Datenbank läuft über den DbConnectionManager des Openfire und sendet lediglich vorgefertigte, auf jede Zeile der aktuellen Tabelle zugeschnittene SQL-Anfragen.

„Eine JSP ist eine Datei, die im Allgemeinen aus einem HTML-Grundgerüst mit eingestreuten Java-Code-Fragmenten besteht. (…) Eine JSP-Datei wird von einem JSP-Server ausgeführt [hier der Openfire Server]. Dort wird ein HTML-Dokument erzeugt, das zum Webbrowser geschickt und dort angezeigt wird.“ (Balzert, 2003)

Kapitel 6: Vorführungen / Feedback

6.1 1. Präsentation

In der ersten Präsentation habe ich den Kunden hauptsächlich den Newsfeed vorgeführt, mit Fokus auf die Darstellung im Spark sowie das Erstellen eines neuen Beitrages.

6.1.1 Feedback

Die vorgeführten Funktionen fanden größtenteils Zustimmung, es wurden jedoch Änderungsvorschläge für die Darstellung eines Newsbeitrages vorgebracht. So sollten nicht mehr die ersten drei Zeilen des Beitrages (die bei einer zu großen Länge abgekürzt worden) zu sehen sein, sondern nur noch ausschließlich die erste Zeile, die bei einer zu großen Länge dynamisch auf mehrere Zeilen aufgeteilt werden sollte. Die Änderung, die sich hieraus ergab, war nicht sehr umfangreich, es mussten lediglich die beiden unteren Zeilen aus dem GUI entfernt werden, und die erste Zeile bei Bedarf vergrößert werden. Allerdings hatte dies ebenfalls eine Änderung am Editor bewirkt, da vorher die drei sichtbaren Zeilen einzeln befüllt werden konnten, nun aber nicht mehr benötigt wurden. Daher wurde das Fenster zum Erstellen eines neuen Beitrages auf eine sichtbare Zeile und einen Textblock für eine ausführlichere Beschreibung der News reduziert.

6.2 2. Präsentation

In der zweiten Präsentation habe ich den Kunden hauptsächlich die Darstellung der Daten eines eingehenden Anrufs vorgeführt, sowie die rudimentär implementierten Funktionen, die auf die TA zugreifen.

6.2.1 Feedback

Die Grundzüge der einzelnen Funktionen trafen genau die Erwartungen der Kunden, allerdings gab es einige Designvorschläge, deren Integration in das System nicht weiter schwierig war. So wurde unter anderem gewünscht, dass bei bestimmten Ereignissen wie zum Beispiel dem Eintreffen eines neuen Anrufs oder einer Änderung in der Warteschleife eine kleine Toaster-Nachricht in der Bildschirmecke darüber informiert. Diese soll auf einen Klick das Spark-Fenster in den Vordergrund holen. Diese Modifizierung erwies sich als sehr einfach, da der Spark bereits über solche Nachrichtenfenster verfügt, und sie auch den Plugins zur Verfügung stellt. Mit einem Mouse Listener versehen können sie dann den gewünschten Effekt erzielen.

In Verbindung mit der Funktion, einzelne Bereiche des ACI zu verbergen, kam noch der Vorschlag auf, dass zusätzlich zu der Toaster-Nachricht der betreffende Bereich auf jeden Fall automatisch sichtbar gemacht wird.

6.3 3. Präsentation

In der dritten Präsentation habe ich den Kunden die Interaktion des ACI-Servers mit dem Telefonserver vorgeführt, sowie die Steuerung des Kanals des Agenten über das ACI-Tool.

6.3.1 Feedback

Die Funktionen an sich stießen dabei auf große Akzeptanz, es wurde jedoch geäußert, dass nachdem der zweite Server in der SSF installiert wurde, das ACI auch zwischen den beiden wechseln können müsse. Da diese Funktion allerdings nicht Gegenstand des ursprünglichen Pflichtenheftes war, wird ihre Implementierung auf ein folgendes Release verschoben.

Kapitel 7: Tests

„Bei aller Begeisterung für das Programmieren darf (…) nicht übersehen werden, dass Programmieren eine äußerst anstrengende Tätigkeit ist, die viel Disziplin, Sorgfalt, Ausdauer und Systematik erfordert.“ (Balzert, 2003)

7.1 Tests vorher

Im Sinne des TDD habe ich vor der Implementierung der Methoden zuerst Testfälle dazu geschrieben, die die gewünschte Funktionsweise beschreiben und festlegen. Der Ablauf ist dabei immer wie folgt:

Zuerst überlege ich mir, was die Methode genau können soll und was nicht.

Bsp.: Arbeitszeit berechnen

> Ich möchte eine Methode getWorkingTime erstellen, die eine Uhrzeit in String-Format bekommt, die Differenz dieser Zeit zur momentanen Uhrzeit berechnet und das Ergebnis wieder als String zurückliefert.

Entsprechend dieser Spezifikation schreibe ich einen Testfall, der zunächst den einfachsten Fall der Funktion beschreibt.

```
Bsp.: testGetWorkingTime() {
        String time; // time sei mit der aktuellen Uhrzeit im Format „hh:mm:ss“ initialisiert
        assertEquals(“kein Unterschied“, “00:00:00“, getWorkingTime(time));
}
```

Damit dieser Testfall auch ausgeführt werden kann, muss ich die Methode erstellen. Vorerst sollte nur ein Gerüst geschrieben werden, dass die benötigten Parameter und Rückgabewerte definiert.

```
Bsp.: public String getWorkingTime(String time) {
        String result = new String();
        return result;
}
```

Jetzt kann ich den Testfall ausführen, er wird allerdings noch fehlschlagen. Deshalb werde ich als nächstes die Methode so implementieren, dass sie den Testfall erfüllt – mehr aber auch nicht.

```
Bsp.: public String getWorkingTime(String time) {
        String result = new String();
        result = "00:00:00";
        return result;
}
```

Nun werde ich den Testfall erweitern, sodass mehrere Fälle behandelt werden, die im Rahmen der Spezifikation liegen und die die Methode bearbeiten können muss.

```
Bsp.: testGetWorkingTime() {
        String time; // time sei mit der aktuellen Uhrzeit im Format „hh:mm:ss“ initialisiert
        assertEquals("kein Unterschied", "00:00:00", getWorkingTime(time));
        String time_2; // time sei 2 Stunden vor der aktuellen Uhrzeit im Format „hh:mm:ss“
                        initialisiert
        assertEquals("2 Stunden", "02:00:00", getWorkingTime(time_2));
}
```

Daraufhin wird der Test wieder erst einmal fehlschlagen, weswegen ich als nächstes wieder die Methode anpassen werde.

Bsp.: public String getWorkingTime(String time) {
```
        String result = new String();
        String[] split_time = time.split(":");
        int hour = Integer.parseInt(split_time[0]);
        int min = Integer.parseInt(split_time[1]);
        int sec = Integer.parseInt(split_time[2]);
        // build calendar from time
        Calendar cal = Calendar.getInstance();
        cal.set(Calendar.HOUR_OF_DAY, hour);
        cal.set(Calendar.MINUTE, min);
        cal.set(Calendar.SECOND, sec);
        longtime_in_millis = cal.getTimeInMillis();
        cal.setTime(new Date());
        // compute difference
        cal.setTimeInMillis(cal.getTimeInMillis()-time_in_millis);
        // mind calendar offset of 1 hour
        cal.add(Calendar.HOUR_OF_DAY, -1);
        DateFormat TIME_READER = new SimpleDateFormat("HH:mm:ss",
                Locale.ENGLISH);
        result = TIME_READER.format(cal.getTime());
        return result;
    }
```

Diese Schritte muss ich dann so lange wiederholen, bis die Spezifikation vollständig erfüllt ist. Dabei ist darauf zu achten, dass bei allen Tests – während der Implementation dieser Methode sowie bei allen nachfolgenden auch – immer alle bisher geschriebenen Testfälle zu allen Methoden mit ausgeführt werden. Dadurch kann sichergestellt werden, dass spätere Änderungen keine unbemerkten Seiteneffekte auf frühere Funktionen haben.
Da ich zuvor noch keine großen Erfahrungen im Bereich des TDD hatte, hatte ich am Anfang noch ein paar Schwierigkeiten, die man wohl erst durch eine gewisse Übung auf diesem Gebiet überwinden kann. So musste ich zum Beispiel im Nachhinein häufiger frühere Methoden noch ändern, da entweder die Spezifikation zu ungenau oder die Schnittstellen ungenügend waren. TDD erfordert die Fähigkeit, Methoden so unabhängig und selbstständig wie möglich vom Rest des Programms zu halten, dass sie auch nach der Einführung von etlichen weiteren Funktionen noch Bestand haben.
Bei der Entwicklung der Plugins war das allerdings stellenweise gar nicht möglich, da manche Funktionen die Anwesenheit gewisser Teile des Spark bzw. Openfire Hauptprogramms benötigen und somit gar nicht ohne diese getestet werden können. Ich habe in solchen Fällen, soweit es mir möglich war, die Methode zuerst außerhalb des Plugins in

einer kleinen Testumgebung geschrieben und dabei die TDD Vorgehensweise angewendet. Die so gewonnene Methode habe ich dann in das Plugin übernommen und an die Schnittstellen des Hauptprogramms angepasst. Dabei geht nur leider der Vorteil verloren, dass man bei allen späteren Tests diese Methode mit testen kann.

7.2 Tests nachher

Nach der Fertigstellung beider Plugins wurden sie zunächst ausgiebig in meiner Testumgebung getestet. Ich hatte dafür den Telefonie-Testserver zur Verfügung, sowie ein physisches Telefon wie es auch in der SSF benutzt wird und zwei virtuelle SIP-Telefone für Windows. So konnte ich sämtliche geforderten Funktionen direkt überprüfen und auch einen längeren Betrieb der Plugins überwachen.
Außerdem habe ich einen Lasttest für das Server-Plugin geschrieben, bei dem immer wieder eingehende Verbindungen von vermeintlichen Client-Plugins aufgebaut werden (wodurch die Behandlung vieler gleichzeitiger Verbindungen simuliert wird), diese sich an der Telefonanlage anmelden (wodurch die Schnittstelle mit dem Telefonie-Server und der Datenbank betroffen ist) und danach ihre Verbindung zum Server trennen. Hierbei wurde ein Problem mit dem Datenbank-Manager offensichtlich, bei dem die geöffneten Verbindungen nicht ordnungsgemäß wieder geschlossen wurden, und somit eine Überlastung geschaffen wurde.

7.3 Testbetrieb

Die Nicht-Telefoniefunktionen wurden nach der Fertigstellung bereits in der SSF getestet und haben ein in der Entwicklungsumgebung übersehenes Problem aufgezeigt: Es bestand eine Inkompatibilität des Newsfeedteils mit WindowsXP (die eigenen Tests fanden unter Windows7 statt). Der XML Parser kann hierbei nicht direkt auf den Pfad der Datei zugreifen wenn dieser Leerstellen enthält, sondern muss ein File-Objekt bekommen, das auf den Pfad zeigt. (Bekanntes Problem, Bug ID 6506304 in der Sun Bug-Datenbank für Java).
Der Testbetrieb des fertigen Tools hat gezeigt, dass es nicht vollständig mit anderen Konfigurationen des Openfire kompatibel war. So wurde in der SSF ausschließlich SSL-

verschlüsselter Datenverkehr genutzt, was vom ACI-Tool nicht unterstützt wurde. (Änderung auf SSL-Sockets nötig, vorhanden im javax.net.ssl Paket).
Diese Änderung zog allerdings weitere Folgen nach sich, da nun der Server zwar im Live-Betrieb getestet werden konnte, dort aber Speicherprobleme auftraten. Lief das Plug-In eine gewisse Zeit ununterbrochen durch, so stieg der Verbrauch im Hauptspeicher an bis aufs Maximum, wodurch der Rechner blockierte. Diese Art Fehler wird Memory Leak genannt und kann mitunter sehr schwer zu lokalisieren sein. So fiel es auch mir sehr schwer, die Ursache dafür zu finden. Natürlich hatte ich zunächst die SSL-Sockets im Verdacht, da sie die letzte große Änderung darstellten, aber nach mehreren erfolglosen Code-Modifikationen schienen sie nicht die Quelle des Problems zu sein. Daraufhin habe ich andere Teile des Servers untersucht und potenzielle Speicherlücken mit diversen Tools getestet, leider ohne Erfolg. Da zudem die Tests meist über Nacht laufen mussten, weil der Speicher nur sehr langsam voll lief, stellte dieses Problem den größten Zeitverlust im ganzen Projekt dar.
Am Ende stellten sich dann doch die SSL-Sockets als Fehlerquellen heraus, und zwar weil die für die Verschlüsselung generierten Keys in einem Cache zwischengespeichert werden, der sich nicht manuell leeren lässt. Die Threads des Servers hielten ihre Sockets jeweils nur für 1 Sekunde offen und erstellten danach einen neuen (damit die Ports während der Laufzeit leicht zu ändern sind), wodurch immer wieder neue Keys generiert wurden. Ich hatte dieses Verhalten zwar schon vorher im Verdacht die Ursache zu sein, da aber Einschränkungen der Cache-Größe keine Auswirkungen hatten, verwarf ich diese Idee zu früh. Die Lösung war, die Sockets länger offen zu halten (z.B. 1 Minute lang), sodass der Cache geleert werden kann.

Kapitel 8: Fazit

Das Arbeiten mit Spark und Openfire stellte sich als sehr angenehm heraus, da man als Entwickler eines Plugins völlig freie Hand hat und im Grunde ein eigenes kleines Programm in die bestehenden Strukturen integriert. So ließen sich die Kundenwünsche sehr detailgetreu umsetzen und mit den Vorgaben von Ignite Realtime vereinbaren. Auch die Anbindung an die Telefonanlage von itcTec gestaltete sich nach kurzer Einarbeitungszeit problemlos, und ich konnte einige neue Erkenntnisse über die beteiligten Technologien gewinnen und einiges über die internen Vorgänge einer Shared Service Factory lernen.

8.1 Ausblick

Am Ende dieser Arbeit gingen der ACI-Client sowie der dazugehörige Server in Version 1.0 über, und wird bereits seit der Testphase vom größten Teil der Callagents genutzt. Abgesehen von einigen Bugreports sind bereits weitere kleinere Features für den Client geplant, die in späteren Versionen verwirklicht werden sollen. Da allerdings auch der Spark und der Openfire weiter entwickelt werden, muss in Zukunft sichergestellt werden, dass die Plugins auch mit neueren Versionen ihrer Hauptprogramme kompatibel bleiben. So wird sich die weitere Arbeit an diesem Projekt also hauptsächlich auf den Ausbau der Funktionsliste und den reibungslosen Einsatz im Livebetrieb konzentrieren, was durch den modularen Aufbau der Plugins und den niedrigen Deployment-Aufwand erheblich vereinfacht wird.

Referenzen

Alexander Schatten, S. B.-F. (2010). *Best Practice Software-Engineering: Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen.* Spektrum Akademischer Verlag.

Apache Software Foundation. (2011). *Apache HttpComponents*. Abgerufen am Juni 2011 von http://hc.apache.org/

Balzert, H. (2003). *JSP für Einsteiger. Dynamische Websites mit JavaServer Pages erstellen.*W3l.

Beck, K. (2002). *Test Driven Development by Example.* Amsterdam: Addison-Wesley Verlag.

Crockford, D. (2006). *The application/json Media Type for JavaScript Object Notation (JSON).* The Internet Society.

Free Software Foundation, Inc. (September 2010). *GNU Lesser General Public License v2.1 - GNU Project*. Abgerufen am Juni 2011 von http://www.gnu.org/licenses/lgpl-2.1.html

Free Software Foundation, Inc. (September 2010). *GNU General Public License v2.0 - GNU Project*. Abgerufen am Juni 2011 von http://www.gnu.org/licenses/old-licenses/gpl-2.0.html

Freixas, T. (kein Datum). *JCalendar - Introduction*. Abgerufen am Juni 2011 von http://flib.sourceforge.net/JCalendar/doc/index.html

itCampus technology. (kein Datum). *itCampus technology*. Abgerufen am Juni 2011 von http://www.itctec.com/

JDOM Project. (kein Datum). *JDOM*. Abgerufen am Juni 2011 von http://www.jdom.org/

Jive Software. (2011). *Ignite Realtime: a real time collaboration community site*. Abgerufen am Juni 2011 von http://www.igniterealtime.org/

Jive Software. (2011). *Ignite Realtime: Openfire Server*. Abgerufen am Juni 2011 von http://www.igniterealtime.org/projects/openfire/

Jive Software. (2011). *Ignite Realtime: Spark IM Client*. Abgerufen am Juni 2011 von http://www.igniterealtime.org/projects/spark/index.jsp

Oracle. (2011). *Lernen über Java-Technologie*. Abgerufen am Juni 2011 von http://www.java.com/de/about/

PIDAS. (2011). *SSF Fact Sheet.* Abgerufen am Juni 2011 von http://www.pidas.com/fileadmin/download/factsheets/PIDAS_FactSheet-MS_SharedServiceFactory_V11.04.pdf

PIDAS. (2011). *trueAct Fact Sheet.* Abgerufen am Juni 2011 von http://www.pidas.com/fileadmin/download/factsheets/PIDAS_Solutions-Folder_V11.04.pdf

PIDAS AG. (2011). *PIDAS - Wissensmanagement*. Abgerufen am Juni 2011 von www.pidas.com

PIDAS AG. (2011). *Referenzen - Kunden*. Abgerufen am Juni 2011 von http://www.pidas.com/de/referenzen/kunden/industrie

PIDAS Österreich Ges.m.b.H. (2009). Master- bzw. Diplomarbeit - Thema: Entwicklung zweier SW-Module an den Schnittstellen Mensch-Computer-Telefonie. Graz, Steiermark, Österreich.

Royce, D. W. (1970). Managing the developement of large Software Systems.

The jTDS Project. (kein Datum). *jTDS JDBC Driver*. Abgerufen am Juni 2011 von http://jtds.sourceforge.net/

Ullenboom, C. (2009). *Java ist auch eine Insel: Programmieren mit der Java Standard Edition Version 6.* Galileo Computing.

Quellen

Beck, K. (2004). *Extreme Programming Explained: Embrace Change.* Amsterdam: Addison-Wesley Longman.

Cooper, A. (2004). *The Inmates Are Running the Asylum: Why High-tech Products Drive Us Crazy and How to Restore the Sanity.*Sams.

Duane K. Fields, M. A. (2001). *Web Development with Java Server Pages.* Manning.

Langr, J. (2005). *Agile Java: Crafting Code with Test-Driven Development.*Prentice Hall International.

Anhang A: Verwendete Technologien

A.1 Client

Der ACI-Client verwendet folgende externe Bibliotheken:

- jCalendar (Freixas), ein auf Swing basierender Datums-Wahl-Kalender von Tony Freixas aus seiner FLib Bibliothek (benutzt für den News-Editor zur einfacheren Wahl des Gültigkeitszeitraumes)
- jCert, eine javax.security Bibliothek von Sun zum Validieren von Sicherheitszertifikaten (benutzt für die SSL-Kommunikation)
- jDom (JDOM Project), ein Java-XML-Parser von Jason Hunter (benutzt für das Einlesen der XML-Dokumente)
- jNet, eine javax.net Bibliothek von Sun zum Verwalten von (Server-)Sockets (benutzt als (Server-)Socket-Factory für die SSL-Kommunikation)
- jSON (Crockford, 2006), ein textbasierter offener Standard für menschenleserlichen Datenaustausch (benutzt für die Kommunikation mit der TA)

A.2 Server

- jCert, eine javax.security Bibliothek von Sun zum Validieren von Sicherheitszertifikaten (benutzt für die SSL-Kommunikation)
- jDom (JDOM Project), ein Java-XML-Parser von Jason Hunter (benutzt für das Einlesen der XML-Dokumente)
- jNet, eine javax.net Bibliothek von Sun zum Verwalten von (Server-)Sockets (benutzt als (Server-)Socket-Factory für die SSL-Kommunikation)
- jSON (Crockford, 2006), ein textbasierter offener Standard für menschenleserlichen Datenaustausch (benutzt für die Kommunikation mit der TA)
- jTDS(The jTDS Project), ein Open-Source Java JDBC Treiber für MS SQL Server (benutzt für die Kommunikation der der TA Datenbank)
- httpComponents (Apache Software Foundation, 2011), ein Java HTTP Toolset (benutzt für die Kommunikation mit der TA)

Anhang B: Datenstrukturen und Datenbankdesign

B.1 Aufbau der Client-Server-Kommunikationsstrings

- Aufbau der Antwortnachricht eines eingehenden Anrufs:

TelNr;Mandate;Greeting;ForumID;Country;Ticket-Link

- Aufbau der Antwortnachricht der Call-History-Abfrage:

Date1;Time1;Duration1;Mandate1;TelNr1;Inbound1/Date2;Time2…….Inbound2/

- Aufbau der Antwortnachricht der Agentenübersicht:

Mandat1:Agent1Status,Agent2Status,/Mandat2:Agent1Status,Agent3Status,/

- Aufbau der Antwortnachricht der Warteschleife:

Notification+Mandate1/Mandate2/…/MandateN/;LongestTime;newCall

- Aufbau der Antwortnachricht der Tagesstatistik:

LoginZeit;ZeitInBereit;ZeitInPause;ZeitInTelefoniert;EingehendeCalls;AusgehendeCalls

- Aufbau der Browseranfrage für die WDB-Suche:
https://wissen.pidas.com/search.php?keywords=Text&terms=all&author=&fid[]=ID&sc=1&sf=all&sr=posts&sk=t&sd=d&st=0&ch=300&t=0&submit=Search
Die Keywords „Text“ und „ID“ werden durch die entsprechenden Werte ersetzt.

B.2 Aufbau der verwendeten XML-Dateien

Beispielaufbau der Telefonbuch-XML-Datei:

```
<?xmlversion="1.0" encoding="UTF-8"?>
<phoneBook>                                                                    // hier beginnt das Telefonbuch
        <entry>                                  // Jeder Eintrag besteht aus einem Namen und einer Nummer
                <Name>Name 1</Name>                                            // Der Name in diesem Eintrag
                <Number>Number 1</Number>                        // Die Telefonnummer in diesem Eintrag
        </entry>
        ….
        <entry>
                <Name>Name N</Name>
                <Number>Number N</Number>
        </entry>
</phoneBook>
```

Beispielaufbau der Newsfeed-XML-Datei (RSS2.0.11 konform):

```
<?xml version="1.0" encoding="UTF-8"?>
<rssxmlns:dc="http://purl.org/dc/elements/1.1/" version="2.0">
<channel>
<title>SSF-Newsfeed</title>
<link>http://www.pidas.com/</link>
<description>This is the Newsfeed of PIDAS SSF Graz</description>
<item>                                                    // jedes Item entspricht einem Newseintrag
<title>. . .</title>                                          // Titel des Newseintrages (optional)
<link />                          // normale RSS-Feeds verlinken auf ihre Newseinträge, hier unnötig
<category>Nycomed</category>          // das Mandat, dem der Newseintrag zugeordnet ist (Pflichtfeld)
<description>Zeile 1                                    // der eigentliche Text des Newseintrags (Pflichtfeld)
                Rest          // die erste Zeile wird im Client angezeigt, alles unter „Rest“ erst im eigenen Fenster
</description>
<pubDate>Fri, 5 Nov 2010 10:33:12 GMT</pubDate>           // Beginn des Gültigkeitszeitraums (Pflichtfeld)
<dc:date>2010-11-05T10:33:12Z</dc:date>                        // alternative Datumsschreibweise
                                                        (wird von manchen RSS-Readern benötigt, optional)
<guid />                  // Feld zur eindeutigen Identifizierung eines Eintrages (optional), hier unnötig
<validUntil>Wed, 31 Dec 2004 23:00:00 GMT</validUntil>     // Ende des Gültigkeitszeitraums (Pflichtfeld)
      (Dieses Feld ist ein von uns selbst erstelltes Feld –
     solche Felder sind legitim und werden von Standard-RSS-Readern ignoriert.)
</item>
<item>                                                                          // weitere Newseinträge
        . . .
</item>
</channel>
</rss>
```

B.3 Aufbau der internen Datenbank

Das ACI Server Plugin erweitert die Openfire interne Datenbank um sieben zusätzliche Bereiche: Newsfeed-History, mandatsspezifische Begrüßungen sowie spezielle WDB-Foren-IDs und Ticket-System-Links, Agent-Statistiken, Agent-Daten und eine Liste aller verfügbarer Status.

Newsfeed-History

Table name: newsfeed_history

Feld	Beschreibung
RID	INT NOT NULL AUTO_INCREMENT PRIMARY KEY
Category	VARCHAR(255) NOT NULL Repräsentiert die Mandatszugehörigkeit
Description	VARCHAR(255) NOT NULL Beinhaltet den Text des Newseintrages
pubDate	VARCHAR(255) NOT NULL Beschreibt das Erstellungsdatum des Eintrages

Tabelle 1: Newsfeed-History Datenbank

Mandatsspezifische Begrüßungen

Table name: greetings

Feld	Beschreibung
RID	INT NOT NULL AUTO_INCREMENT PRIMARY KEY
mandate	VARCHAR(255) NOT NULL UNIQUE Das Mandat zu dem die Begrüßung gehört
text	VARCHAR(255) NOT NULL Der Begrüßungstext

Tabelle 2: Begrüßungsdatenbank

Mandat-IDs für die WDB-Suche

Table name: mandate_forum_ID

Feld	Beschreibung
RID	INT NOT NULL AUTO_INCREMENT PRIMARY KEY
mandate	VARCHAR(255) NOT NULL UNIQUE Das Mandat zu dem die ID gehört
forumID	VARCHAR(255) NOT NULL Die ID des Forums für die WDB-Suche

Tabelle 3: ID-Datenbank

Mandatsspezifische Ticket-System-Links

Table name: mandate_ticket

Feld	Beschreibung
RID	INT NOT NULL AUTO_INCREMENT PRIMARY KEY
mandate	VARCHAR(255) NOT NULL UNIQUE Das Mandat zu dem der Link gehört
ticket	VARCHAR(255) NOT NULL Der Link zum Ticketsystem dieses Mandats

Tabelle 4: Ticket-Link Datenbank

Verfügbare Status

Table name: AvailableStatus

Feld	Beschreibung
RID	INT NOT NULL AUTO_INCREMENT PRIMARY KEY
name	VARCHAR(255) NOT NULL Der Name des Status
type	VARCHAR(255) NOT NULL Die Art des Status (available,phone,away,dnd)

Tabelle 5: Status Datenbank

Agentenspezifische Daten

Table name: AgentData

Feld	Beschreibung
RID	INT NOT NULL AUTO_INCREMENT PRIMARY KEY
agent	VARCHAR(255) NOT NULL UNIQUE Die Nummer des Agenten
role	INT NOT NULL Die Berechtigungsebene des Agenten

Tabelle 6: Berechtigungs-Datenbank

Agentenspezifische Statistiken

Table name: firstName_lastName

Feld	Beschreibung
date	DATE NOT NULL PRIMARY KEY Der Tag, an dem die Daten erhoben wurden
first_login	TIME NOT NULL Der Zeitpunkt des ersten Logins an diesem Tag
ready_time	TIME NULL DEFAULT '00:00:00' Die Zeit, die der Agent an diesem Tag im Zustand „Bereit“ war
away_time	TIME NULL DEFAULT '00:00:00' Die Zeit, die der Agent an diesem Tag im Zustand „Pause“ war
phone_time	TIME NULL DEFAULT '00:00:00' Die Zeit, die der Agent an diesem Tag im Zustand „Telefoniert“ war
in_calls	SMALLINT NULL DEFAULT '0' Die Anzahl der an diesem Tag angenommenen Anrufe
out_calls	SMALLINT NULL DEFAULT '0' Die Anzahl der an diesem Tag getätigten Anrufe

Tabelle 7: Agenten-Datenbank

Anhang C: Zeitlicher Ablauf

C.1 Geplant

30 ECTS	750 Std.
Schreiben der Arbeit	150 Std.
Arbeit am Projekt	600 Std.
- Planung / Dokumentation	- 90 Std.
- Implementation	- 210 Std.
- Testen	- 300 Std.
• Tests „vorher“	• 90 Std.
• Tests „nachher“	• 210 Std.

C.2 Tatsächlich

30 ECTS	745,75 Std.
Schreiben der Arbeit	132 Std.
Arbeit am Projekt	613,75 Std.
- Planung / Dokumentation	- 93 Std.
- Implementation	- 312,75 Std.
- Testen	- 208 Std.
• Tests „vorher“	• 52 Std.
• Tests „nachher“	• 156 Std.

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

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

Graz, am 7.10.2011 ..... (Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

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

7.10.2011 .....
date

..... (signature)