Entwicklung einer Anwendung zur computergestützten Telefonie (CTI) auf Basis des Java Telephony API (JTAPI)

Jan Hasert

Konstanz, 31. Mai 2005

Diplomarbeit

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Informatiker (FH)

an der

Fachhochschule Konstanz
Hochschule für Technik, Wirtschaft und Gestaltung

Fachbereich Informatik
Studiengang Wirtschaftsinformatik

Thema:	**Entwicklung einer Anwendung zur computergestützten Telefonie (CTI) auf Basis des Java Telephony API (JTAPI)**
Diplomand:	Jan Hasert Carl-Maria-von-Weber-Straße 11 71083 Herrenberg
1. Prüfer:	Prof. Dr. Ralf Leibscher Fachhochschule Konstanz
2. Prüfer:	Dipl.-Phys. Thomas Jaekel Canoris GmbH & Co. KG IT-Dienstleistungen Otto-Lilienthal-Straße 36 71034 Böblingen
Abgabedatum:	31. Mai 2005

Zusammenfassung

Thema:	Entwicklung einer Anwendung zur computergestützten Telefonie (CTI) auf Basis des Java Telephony API (JTAPI)
Diplomand:	Jan Hasert Carl-Maria-von-Weber-Straße 11 71083 Herrenberg
Ort:	Canoris GmbH & Co. KG IT-Dienstleistungen Otto-Lilienthal-Straße 36 71034 Böblingen
Betreuer:	Prof. Dr. Ralf Leibscher Fachhochschule Konstanz
Abgabedatum:	31. Mai 2005
Schlagworte:	Computer Telephony, CTI, JTAPI, komponentenorientierte Software, PBX, Nortel Meridian I, Java, RMI, JNDI, LDAP, CSTA, TSAPI, TAPI

Die Zielsetzung dieser Arbeit ist die Entwicklung der Serverdienste einer Client/Server-Software für die computergestützte Telefonie am Arbeitsplatz. Die Software ermöglicht die Steuerung und Statusüberwachung von Telefonapparaten mit dem Ziel, die Benutzung ergonomischer zu gestalten und das zugrunde liegende Telekommunikationssystem stärker in die Informationstechnik zu integrieren. Die Software soll mit der Telekommunikationsanlage *Meridian I* von Nortel zusammenarbeiten, um die Manipulation der angeschlossenen Telefonapparate zu ermöglichen. Beim Softwareentwurf ist für die Zukunft die Unterstützung weiterer Telekommunikationsanlagen zu berücksichtigen. Zudem soll ein API definiert werden, welches die Implementierung eines Clients in Form eines HTTP-Dienstes für die Nutzung über einen Web-Browser ermöglicht. Ebenso soll der Weg für eine in Zukunft zu entwickelnde dedizierte Client-Software bereitet werden.

Für die Umsetzung der Projektziele wurde ein komponentenorientiertes Middleware-Konzept auf Basis des Java Telephony Application Programming Interface (JTAPI) verwirklicht. Dafür wurde eine JTAPI-konforme Provider-Implementierung realisiert, die das zugehörige Zustands- und Objektmodell umsetzt. Für die Kommunikation mit der Telekommunikationsanlage wurde eine Treiberkomponente entwickelt, die das proprietäre Protokoll des CTI-Links implementiert. Schließlich wurde ein einheitliches, RMI-basiertes API spezifiziert, das für die Entwicklung der Client-Software in Form eines HTTP-Dienstes oder einer selbständigen, fensterbasierten Anwendung eingesetzt werden kann. Die komponentenorientierte Gesamtarchitektur ermöglicht darüber hinaus die Entwicklung weiterführender CTI-Dienste.

Inhaltsverzeichnis

Typographische Vorbemerkungen:
Mit Rücksicht auf die Lesbarkeit dieser Arbeit wird ausschließlich die maskuline Form verwendet, was jedoch keine Diskriminierung des weiblichen Geschlechts darstellen soll.
Besonders zu betonende Wörter, Wort- und Satzteile sowie Zitate werden *kursiv* gesetzt. Schlüsselwörter, Elemente des Programmcodes (Klassen-, Schnittstellen- und Methodennamen) sowie Komponentenbezeichner erstellter sowie verwendeter Modelle werden im Text durch `Maschinenschrift` hervorgehoben.
Enthalten Assoziationsbeziehungen in den abgebildeten Klassendiagrammen keine Angaben zur Kardinalität, gilt der Standardwert von „1“.

1 Einleitung

1.1 Aufgabenstellung

Im Rahmen dieser Diplomarbeit soll für die *Canoris GmbH & Co. KG* das Produkt *webDial* weiterentwickelt werden. Bei *webDial* handelt es sich um eine *CTI*[1]-Software in Client/Server-Architektur. Projektziel ist die Neuentwicklung der *Server-Komponenten*, um das bisherige System abzulösen, das aufgrund seiner uneinheitlichen Konzeption und prozeduraler Implementierung schwer wartbar und erweiterbar geworden ist. Es soll eine Lösung entstehen, die über den Funktionsumfang des alten Systems verfügt, darüber hinaus aber auf modernen Softwaretechnologien beruht, die Aspekten wie Wartbarkeit und Erweiterbarkeit Rechnung tragen. Die Entwicklung einer entsprechenden Client-Anwendung ist hingegen nicht Teil der Projektaufgabe.

1.2 Die Canoris GmbH

Die Canoris GmbH & Co. KG ist ein IT-Dienstleistungsunternehmen mit Sitz in Stockach und Standorten in Stuttgart-Leinfelden und Böblingen. Das Unternehmen verfügt derzeit über neun Mitarbeiter. Zu den angebotenen Dienstleistungen gehören die Installation und Konfiguration von Callcenter-Systemen, die Durchführung und Auswertung der Telefonabrechnung komplexer Telefonanlagen sowie die Entwicklung individueller Softwarelösungen für die Telekommunikation. Ein Beispiel ist die Software *TelAccount*, die die automatisierte Erfassung von Gebührendaten und deren Auswertung sowie Exportierung in ein ERP-System von SAP ermöglicht. Es existieren weitere Produkte für das Monitoring von Callcentern sowie für die webbasierte, zeitgesteuerte Konfiguration von Anrufweiterleitungen. Zu den Kunden gehören Unternehmen wie DaimlerCrysler, T-Systems, EADS, Natenco sowie die Motorflug GmbH Baden Baden.

Das Unternehmen wurde 1997 in Stockach als Einzelunternehmen für Consulting im Bereich Telekommunikation gegründet. 2002 erfolgte der Aufbau der Außenstelle in Böblingen, worauf 2004 die Eintragung der Canoris GmbH & Co KG in das Handelsregister folgte.

1.3 Das Produktkonzept

Bei *webDial* handelt es sich um eine Softwarelösung für die computergestützte Steuerung von Telefonapparaten am Arbeitsplatz. Über eine einheitliche Benutzeroberfläche können Telefongespräche und Konferenzen initiiert sowie Anrufer weitergeleitet werden. Der Dienst wird

[1] Computer Telephony Integration, vgl. Kapitel 2.1

über das Intranet den Mitarbeitern eines Unternehmens zur Verfügung gestellt. Die HTML-basierte Oberfläche sorgt für eine betriebssystem- und plattformunabhängige Verfügbarkeit, wobei der Anwender nur einen aktuellen Web-Browser benötigt.

Durch die Einbeziehung von LDAP[2]-fähigen Verzeichnisdiensten wird eine höhere Integration der Telefonieinfrastruktur in die IT-Umgebung eines Unternehmens erzielt. So können anrufende Teilnehmer über das Corporate Directory identifiziert und nähere Informationen angezeigt werden. Zudem kann über den LDAP-Dienst die Benutzerauthentifizierung mittels bestehender User-Accounts erfolgen, sodass zunächst keine zusätzliche Benutzerverwaltung notwendig ist.

Folgender Ablauf soll das Konzept von *webDial* verdeutlichen. Ein typischer Anwendungsfall stellt hierbei ein Telefonanruf dar. Am Arbeitsplatz wählt der Anwender an seinem Computer über die Benutzeroberfläche die Telefonnummer eines Kontaktes[3] aus. Der *webDial*-Dienst stellt dann, mittels Interaktion mit einer zentralen Telekommunikationsanlage (im Folgenden *TK-Anlage* oder *PBX*[4] genannt), die Verbindung mit dem Gesprächspartner her. Konnte die Verbindung erfolgreich aufgebaut werden, so klingelt der Telefonapparat des Angerufenen, während der des Anrufers durch Aktivierung der Freisprechfunktion selbsttätig abnimmt. Wenn der Anruf entgegengenommen wird, kann das Gespräch beginnen. Dem *webDial*-Nutzer bleibt es dann überlassen, die Freisprechfunktion weiter zu nutzen, den Telefonhörer abzunehmen oder ein geeignetes Head-Set einzusetzen.

Dieser Ablauf zeigt die Verbesserung des Kommunikationsprozesses durch Reduzierung der notwendigen, direkten Interaktion mit dem Telefoniesystem. Weiter wird die Benutzung des Telefons in ein dem Benutzer vertrautes Umfeld verlagert. Dieses Umfeld kann zum Beispiel aus einem Who-is-Who[5], das im Intranet der Organisation abrufbar ist, oder aus einem Groupwaresystem wie Lotus Notes oder Outlook Exchange bestehen. Somit entfallen immer wiederkehrende Tätigkeiten wie das Wählen von Rufnummern oder die Belegung von Schnellruftasten am Telefonapparat. Gemessen an der intuitiven Nutzbarkeit einer HTML-Oberfläche werden diese Vorteile noch durch die meist komplizierte Bedienung der Telefonapparate verstärkt. So erfordert beispielsweise das Belegen von Schnellruftasten die Konfiguration der digitalen Systemtelefone[6]. Dies ist, zum Teil aufgrund der kleinen Darstellungsfläche der Telefondisplays, oft schwierig und wenig intuitiv.

In Kombination mit einem ergonomischen Benutzerinterface ist *webDial* in der Lage, den täglichen Umgang mit dem Telefon zu erleichtern und effizienter zu gestalten.

[2] Lightweight Directory Access Protocol
[3] Datenelement zur Repräsentation der Kontaktinformationen einer Person oder Organisation
[4] Public Branch Exchange
[5] Mitarbeiterverzeichnis einer Organisation
[6] Telefone zur Verwendung an bestimmten PBX zur Nutzung erweiterter Funktionen

2 Grundlagen

Das folgende Kapitel ist eine thematische Einführung in die Integration von Telefonie und Informationstechnologie. Im ersten Unterkapitel wird anhand des Begriffs der *Computer Telephony Integration* das Potential und die Möglichkeiten dieser Integrationsbestrebungen aufgezeigt. Darauf folgt eine Abgrenzung zum umfassenderen Begriff der *Computer Telephony (CT)*. Das dritte Unterkapitel gewährt einen Überblick über aktuelle CT-Standards, während das vierte aus Sicht der TK-Anlagen einen Einblick in die leitungsvermittelnde Telekommunikationstechnologie bietet.

2.1 Computer Telephony Integration

Die Telefonie ist seit der zweiten Hälfte des 19. Jahrhunderts, als 1861 Philipp Reis die ersten menschlichen Worte für eine elektrische Leitung modulierte und über eine Entfernung von nur wenigen Metern übertrug, nicht mehr wegzudenken. Sie ist trotz ihrer langen Entwicklungsgeschichte, auch wenn sich durch den technischen Fortschritt (drahtlose Datenübertragung, Digitaltechnik, etc.) einige Rahmenbedingungen geändert haben, in ihrem Grundprinzip unverändert geblieben: Es geht nach wie vor um die Übermittlung von Informationen in Form von gesprochener Sprache.

Im Gegensatz zum Telefon war der Computer nicht von Beginn an als ein Medium der Kommunikation gedacht. Ausgehend von der anfänglichen Aufgabe komplexe Berechnungen durchzuführen, wurde er zu einem modernen Werkzeug der Datenerfassung, -haltung und -verarbeitung weiterentwickelt. Diese zentrale Rolle im Arbeitsalltag vieler Menschen in der heutigen Industrie- und Dienstleistungsgesellschaft führte zu einer immer mehr computerzentrierten Arbeitswelt.

Ein weiterer Aspekt ergab sich in zunehmendem Maße durch den technischen Fortschritt und die Verbreitung des Internets innerhalb der letzten 10 Jahre. Mit zunehmendem Grad der Vernetzung entwickelte sich der Computer immer mehr zu einem Medium der Kommunikation. Neben den bisher etablierten, computerbasierten Kommunikationswegen wie E-Mail, Foren, Newsgroups und Instant Messaging ist es eine konsequente Fortsetzung dieser Tendenz, dass die Telefonie ebenfalls Berücksichtigung in der Informationstechnologie findet. Mit dem Begriff der *Computer Telephony Integration (CTI)* beschreibt man den Vorgang der Integration von Telefonie und Computerinfrastruktur.

The integration of the telephony function with computer applications, commonly used to automate call centers. [NEWT04]

Mit der daraus resultierenden Verknüpfung von Computer- und Telekommunikationssystemen können jedoch sehr unterschiedliche Ziele verfolgt werden. Zum einen kann die Motivation darin bestehen, die Benutzung der Telefonie durch einheitliche Bedienungsabläufe und intuitive Benutzerführung einfacher und effizienter zu gestalten. Mit diesem Vorsatz ergibt sich für eine CTI-Lösung ein zu unterstützender Funktionsumfang, der dem eines Telefonapparates nachempfunden wird. Bei der so genannten *Screen-based Telephony* werden die von einem Telefon bekannten Funktionen abgebildet, wie dem Auf- und Abbau einer Telefonverbindung (Einzelgespräch oder Konferenzschaltung) oder der Annahme und Weiterleitung von Anrufen. Zudem sind Funktionen wie die Identifizierung des Anrufers, die Auflistung entgangener Anrufe beziehungsweise getätigter Gespräche oder die Verwaltung der Sprachmailbox denkbar.

Aber gerade die *softwarebasierte* Anbindung der Telefonieinfrastruktur bietet im Gegensatz zum Einsatz spezialisierter, hardwarenaher Lösungen, meist aus der Hand der Anbieter der Telekommunikationsanlagen, die Möglichkeit der kostengünstigen Implementierung anderer Funktionsaspekte. So kann bei der *Call-based Data Selection (CBDS)* durch Auswertung und Darstellung der Anruferinformationen der weitere Anrufverlauf beeinflusst werden. Bei Anrufeingang wird entschieden, ob eine Weiterleitung in eine Warteschleife, an ein *IVR-System*[1] oder an einen zuständigen Operator erfolgt. Diese Art der Anwendung kommt bei Callcentern unter der Bezeichnung *Automatic Call Distribution (ACD)*[2] zum Einsatz.

Ein anderer Aspekt von CTI-Lösungen stellt das *Coordinated Call Monitoring (CCM)* dar. Hierbei geht es um das Überwachen und Protokollieren der Aktivitäten eines Telefoniesystems zu unterschiedlichsten Auswertungszwecken wie beispielsweise

- dem Accounting von Telefongebühren,
- der Auslastungsanalyse von Operatoren und Callcentern
- und der Lastbetrachtung der verfügbaren Amtsleitungen.

Diese und weitere Anwendungsgebiete wie dem *Application Controlled Routing (ACR)*, zur kostenoptimierten Nutzung der vorhandenen Telefonieinfrastruktur, dem Transport von Daten oder der simultanen Zuordnung von Gesprächsdaten und zugehörigen Informationen im Rahmen eines CRM[3]-Systems verdeutlichen die unterschiedlichen Ansätze möglicher CTI-Lösungen.

Nach ihrer Architektur kann man CTI-Systeme in zwei Kategorien unterteilen. Hierbei wird unterschieden wo die Schnittstelle zum Telefoniesystem implementiert wurde. Beim *First Party Call Control Modell* findet die Kommunikation mit dem Telefoniesubsystem direkt an der Workstation des Benutzers statt (vgl. Abbildung 2-1). Der Computer mit der CTI-Anwendung verfügt damit über eine direkte Verbindung zum *Terminal* (Telefon, Fax, Modem, etc.). Dabei ist es nicht von Bedeutung, ob das Terminal auf herkömmliche Weise an einer TK-Anlage

[1] System zur sprachbasierten Menüsteuerung; Steuerung kann über Tastendruck (DTMF - Dual-Tone Multi-Frequency) oder Spracheingabe erfolgen

[2] automatisierte Verteilung eingehender Anrufe auf bestimmte Gegenstellen gemäß gegebener Kriterien

[3] Customer Relationship Management

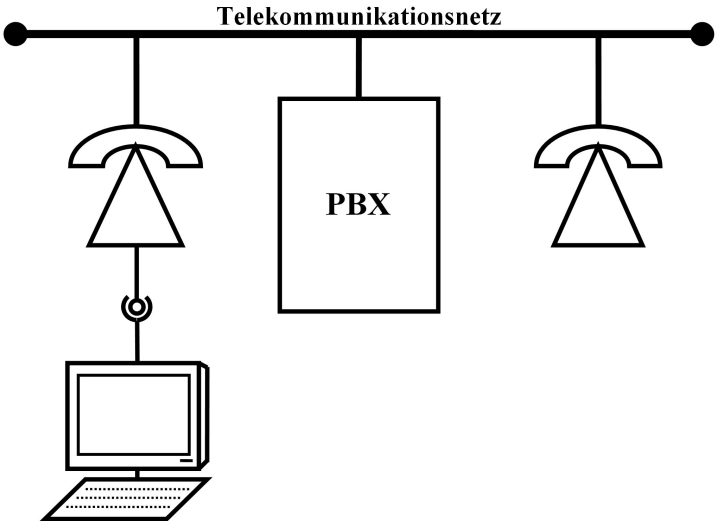

Abb. 2-1: Das First Party Call Control Modell

angeschlossen ist oder es sich um ein IP-basiertes Endgerät handelt. Der Vorzug dieser Variante liegt in der einfachen Implementierbarkeit aufgrund der geringen Komplexität einer Einzelplatzanwendung. Allerdings kann dabei nur ein Aspekt, nämlich das der Screen-based Telephony, abgedeckt werden. Jede weiterführende Funktionalität erfordert die Interaktion an zentralen Stellen der Infrastruktur.

Das *Third Party Call Control Modell* hebt diese Einschränkung auf, indem die Kommunikation über einen *CTI-Server* erfolgt. Dieser ist mit der PBX des Telefonnetzes verbunden und tritt als Vermittler zwischen den Client-Computern der Anwender und der PBX auf (vgl. Abbildung 2-2). Deshalb wird dieser Server auch als CTI-Proxy bezeichnet. Durch die direkte Anbindung an die Schaltstelle des Telefoniesystems ist es möglich an Informationen unterschiedlichster Ausprägungen wie den Status der angeschlossenen Terminals, Gebühreninformationen und Gesprächsdaten zu gelangen. Auch das Ausführen von Kommandos für die Umsetzung der zuvor aufgeführten Funktionen ist an dieser Stelle möglich. Das größere Potential dieser Lösung wird durch eine höhere Komplexität einer Client/Server-Architektur erkauft, die zusätzlich dem zeitkritischen Charakter der Telefonie gerecht werden muss.

Eine Grundvoraussetzung für beide Modelle ist natürlich, dass Terminals und TK-Anlagen über geeignete Schnittstellen verfügen, die eine Steuerung der Geräte durch eine CTI-Software ermöglichen. Damit steht der implementierbare Funktionsumfang in direkter Abhängigkeit zu den Fähigkeiten der CTI-Schnittstellen der verwendeten Telefoniesysteme.

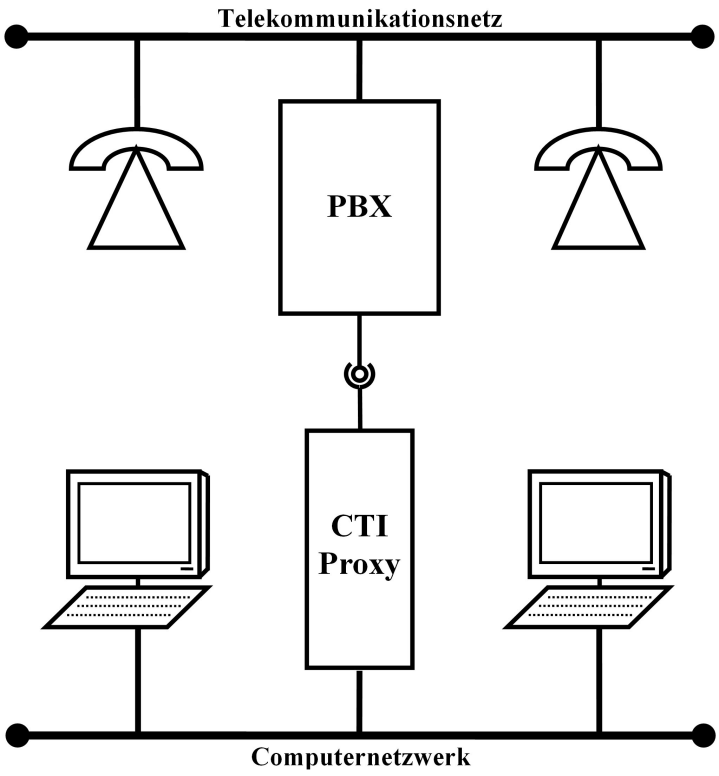

Abb. 2-2: Das Third Party Call Control Modell

2.2 Computer Telephony Integration versus Computer Telephony

Die oft synonym verwendeten Begriffe *Computer Telephony Integration* und *Computer Telephony* unterscheiden sich im Verständnis der computerunterstützten Telefonie. *Computer Telephony Integration* ist der ältere Begriff und durch ein traditionelles Verständnis der Telekommunikation geprägt. Dabei sind Hardwarekomponenten wie Telefonapparate und Telekommunikationsanlagen, aber auch analoge und digitale Erweiterungskarten für PCs, die Funktionsträger der Telefonie. Kernaspekt ist damit die Integration traditioneller, PBX-basierter Telefoniesysteme in die Infrastrukturen moderner Informationstechnologien. Der Begriff der *Computer Telephony* wird hingegen einer zunehmenden Verwischung der Grenzen zwischen beiden Elementen gerecht und vertritt damit ein allumfassenderes Verständnis. Neben der traditionellen Telefonie sind auch Aspekte wie packetgebundene Kommunikation mittels Voice-over-IP, E-Mail und Voice-Mail enthalten, die nicht zwangsläufig an dedizierte Telekommunikationshardware gebunden sind. Eine exemplarische Teildisziplin stellt das *Unified Messaging* dar, das sich mit der einheitlichen Nutzung unterschiedlicher nachrichtenbasierter Kommunikationstechnologien beschäftigt. Über eine zentrale Anwendung werden, ungeachtet der zugrunde liegenden Technologie, Nachrichten unterschiedlicher Ausprägungen wie E-Mails, Fax- und Voice-Mails empfangen, erstellt, versandt und bearbeitet. Dabei ist die Telefonie nur eine Komponente unter anderen.

2.3 Computer Telephony API-Standards

Als die CTI-Technologie mitte der 80er Jahre aufkam, lag die Entwicklung von Applikationen allein in der Hand der Hersteller der Telekommunikationssysteme. Dies wurde durch den hohen Entwicklungsaufwand und die starke Abhängigkeit von der zugrunde liegenden Hardware begründet. Die entstandenen Lösungen waren deshalb an die Hardware eines Anbieters gebunden. Es wurden proprietäre Protokolle eingesetzt, da herstellerübergreifende API-Standards nicht existierten. Diese Standards sind notwendig, um herstellerunabhängigen Anbietern zu ermöglichen, rentable Lösungen zu entwickeln, die einen möglichst großen Kundenkreis erschließen. Hierfür müssen verschiedene PBX-Typen verschiedener Hersteller unterstützt werden. Die Umsetzung wäre ohne umfassende Standardisierungen, aufgrund der Vielzahl an unterschiedlichen Telekommunikationssystemen am Markt, finanziell nicht rentabel.

Im Jahr 1992 wurde deshalb von der ECMA[4] die Phase I des *CSTA*[5]-Standards verabschiedet. Hierbei handelt es sich um die Spezifikation eines CTI-Protokolls für die Kommunikation zwischen Computer- und Telekommunikationsschicht. Der aus zwei Komponenten bestehende Standard beinhaltet die Definition der angebotenen Funktionen (Services) und den Aufbau des zur Umsetzung benötigten Protokolls. Ende 1994 erfolgte die Veröffentlichung der Phase II des Standards in einer überarbeiteten und um Funktionen erweiterten Fassung. Der aktuelle Stand wurde im Jahr 2003 mit der fünften Version der Phase III erreicht. Diese enthält unter anderem XML-Schemaspezifikationen für die Umsetzung eines CSTA-Protokolls in Form von Web-Services. CSTA ist heute ein weit verbreiteter und von vielen Herstellern akzeptierter Standard für die Computer-zu-PBX-Kommunikation. Allerdings lieferte die ECMA nie eine zugehörige Referenzimplementierung eines API und überließ dies den Herstellern der Telekommunikationssysteme.

Kurz nach dem Erscheinen der Phase I hatte 1993 *TSAPI*[6] sein Debüt als Teilimplementierung des CSTA-Standards. Der von Novell und AT&T, später Lucent Technologies, entwickelte Standard ermöglichte Lösungen nach dem *First* und *Third Party Call Control Modell*. Das API unterstützt Funktionen für

- Call Control,
- Call & Device Monitoring,
- Call Routing,
- Maintenance und
- ermöglicht die Integration von Verzeichnisdiensten.

Aufgrund der fortgeschrittenen Möglichkeiten eignete sich dieses API bereits zum Zeitpunkt der Einführung zur Implementierung von Call-Center-Applikationen. Um TSAPI nutzen zu können muss das Netzwerkbetriebssystem *NetWare* von Novell eingesetzt werden. Die Servererweiterung *Telephony Server Loadable Module* stellte dabei die Serverkomponente von

[4] European Computer Manufacturer Association
[5] Computer Supported Telecommunications Applications
[6] Telephony Services Application Programming Interface

TSAPI zur Verfügung. Hierin bestand jedoch auch der Hauptnachteil von TSAPI. Denn die Bindung an Novell war mit hohen Lizenzkosten verbunden, sodass sich der 1994 aufkommende TAPI[7]-Standard von Intel und Microsoft aufgrund höherer Akzeptanz durchsetzte. Erstaunlicherweise geschah dies, obwohl in der ersten veröffentlichten Version 1.3 nur das *First Party Call Control Modell* unterstützt wurde. Erst mit der 1997 erschienenen Version 2.1 wurde dieser Mangel behoben. Einer der Gründe für den Erfolg von TAPI war die Entscheidung, das API kostenlos in den jeweiligen Betriebssystemversionen von Microsoft Windows zu unterstützen. Von Windows 3.1 bis XP ist TAPI integriert. Ein weiterer Grund war ein besser abstrahierender Ansatz zur Trennung von Anwendung und Treiber, der in diesem Zusammenhang *Provider* genannt wird. TAPI besteht aus dem *Application Programming Interface* und dem *Service Provider Interface*. Das *Service Provider Interface* nutzen Hersteller der Hardware zur Implementierung der Provider, die den Anwendungsentwicklern bereitgestellt werden. Diese können dann anhand des *Application Programming Interface* die Anwendung entwickeln. Diese Struktur ermöglicht das Zusammenspiel von Applikationen mit unterschiedlichen Providern und dadurch mit unterschiedlicher Hardware. TAPI sieht zudem den direkten Einsatz des Computers als Telefon ohne zusätzliche Hardware vor, während TSAPI, ursprünglich beeinflusst von AT&T, diese voraussetzt. Auch CSTA wurde als Protokoll zur Integration traditioneller PBX-Systeme entwickelt. Da CSTA keine API-Spezifikation beinhaltet, bietet sich die Kombination mit beispielsweise TAPI an. So wird das *Service Provider Interface* auf Basis von CSTA implementiert und kann mit unterschiedlicher Hardware verwendet werden – sofern die CSTA-Spezifikation von allen Beteiligten eingehalten wurde.

Die Organisation ECTF[8] hat ihrerseits eine Reihe neuerer Standards veröffentlicht. Hauptziel dieser Standardisierungsbestrebung ist das Schaffen einer Basis für die verteilte Nutzung von Telefonieressourcen ungeachtet der Unterschiede der jeweiligen Hardware. Mit dem Dokument *S.100* ist zu Beginn der Grundstein in Form eines API zur Nutzung der Telefonieressourcen in CTI-Applikationen gelegt worden. Die Spezifikation *S.200* beschreibt ergänzend dazu einen Protokollstandard für die Umsetzung von Client/Server-Applikationen. Mit Verzögerung folgte die *S.300*-Spezifikation, die *S.100* und *S.200* zu einem Middlewarekonzept zusammenführt, indem nach dem Provider-Prinzip die Applikationsentwicklung von Hardwarepräferenzen abgeschirmt wird.

Einen verwandten, jedoch noch weiter abstrahierenden Ansatz stellt die *JTAPI*[9]-Spezifikation dar. Dieser Standard wurde von einem Konsortium namhafter Unternehmen wie Dialogic, IBM, Intel, Lucent, Nortel, Novell, Siemens und Sun Microsystems entwickelt. Ende 1996 erschien Version 1.0 des Standards. Es folgten die Versionen 1.2, 1.3 und im Jahr 2001 schließlich die aktuelle Version 1.4. An der Version 2.0 wird derzeit gearbeitet. JTAPI ist ebenfalls nach dem Provider-Konzept aufgebaut und damit ebenso auf vorhandene Provider angewiesen, die die Funktionen des API implementieren. Das API basiert auf einem relativ einfachen und flexiblen Call-Modell.

JTAPI ist eine reine API-Spezifikation zur Entwicklung Java-basierter CT-Applikationen. Die Besonderheit dabei ist die Freiheit bei der Wahl der darunter liegenden Protokolle und

[7] Telephony Applications Programming Interface
[8] Enterprise Computer Telephony Forum
[9] Java Telephony Application Programming Interface

Standardlösungen. So können Provider, die sich proprietärer Protokolle bedienen, benutzt werden, als auch solche, die auf bestehenden Standards wie CSTA, TSAPI und TAPI aufbauen. Sogar Provider sind möglich, die gänzlich auf zugrunde liegende Telefoniehardware verzichten. Unter `http://sourceforge.net/projects/gjtapi` ist im Rahmen des GJTAPI-Projekts eine auf dem SIP-Protokoll basierende Provider-Implementierung für IP-Telefonie zu finden. Somit sind beispielsweise Anwendungen möglich, die zwei unterschiedliche Provider für IP-basierte und traditionelle Telefonie einsetzen.

Abschließend muss bemerkt werden, dass sich bisher kein einzelner Standard als Patentrezept erwiesen hat. Begründet durch die Komplexität der Aufgabenstellung und der berechtigten Interessenskonflikte der Hersteller, hat sich bisher kein dominierender Standard industrieweit etablieren können.

2.4 Rolle und Aufgabe der Nebenstellenanlagen

Im folgenden Kapitel werden die wichtigsten Komponenten und Abläufe in traditionellen, leitungsvermittelnden Telekommunikationsnetzen erläutert und dabei die zentrale Bedeutung der Telekommunikationsanlagen vermittelt. Es wird jedoch nicht auf technologische Details wie angewandte Protokolle oder Prozesse eingegangen, da diese für die Problemlösung transparent sind und den Rahmen dieser Arbeit sprengen würden.

Eine zentrale Komponente privater Telefoniesysteme stellt die Anbindung an das öffentliche Telekommunikationsnetz dar. Telefonanbieter erbringen diese Dienstleistung, indem sie sowohl infrastrukturelle als auch administrative Voraussetzungen schaffen. Die Leistungen reichen vom Setzen der Hausanschlüsse bis hin zur Zuteilung der Rufnummern. Wenngleich seit der Liberalisierung des Telekommunikationsmarktes verschiedene Unternehmen diese Aufgaben erfüllen, soll im Folgenden zur Vereinfachung der geläufige Begriff *Amt*[10] für die zuständigen Vermittlungsstellen weiter verwendet werden.

Auf Verbraucherseite befinden sich meist nur Einzelanschlüsse für einfache Telefone, Faxgeräte oder Anrufbeantworter, die direkt mit dem Amt verbunden sind. Auf diese Weise lassen sich zum Beispiel bei einem digitalen ISDN[11]-Anschluss drei Endgeräte anschließen, wovon aufgrund der verfügbaren Übertragungskanäle zwei gleichzeitig von außen erreichbar sind. Benötigt man hingegen mehr Telefone und Amtsleitungen, muss die Zahl der Anschlüsse erhöht werden. Eine Kommunikation zwischen den Apparaten unter Ausschluß der Vermittlungsstelle ist ohne gesonderte Hardware nicht möglich.

Wie man erkennen kann, ist diese Vorgehensweise beschränkt und für private Telekommunikationsnetze nicht praktikabel. Sobald mehrere Telefonanschlüsse notwendig werden, sollte die Vergabe der Rufnummern innerhalb des Telefonsubnetzes und unabhängig vom Telefonanbieter erfolgen können. Dies ist besonders notwendig, um der Dynamik ständig wechselnder Konfigurationen an Standorten mittlerer bis größerer Organisationen gerecht zu werden. Je größer das private Telekommunikationsnetz wird, desto größer wird auch der interne Kommunikationsbedarf, sodass Sonderfunktionen wie Telefonkonferenzen, Anrufweiterleitungen,

[10] Im Englischen ist hierfür der Begriff Central Office geläufig.

[11] Integrated Services Digital Network - Standard in Europa für digitale, leitungsvermittelnde Telekommunikationsnetze

und elektronische Anrufbeantworter an Bedeutung gewinnen.

Um diese Anforderungen erfüllen zu können, werden *Telekommunikationsanlagen* (auch: *Nebenstellenanlagen*) eingesetzt. Diese Systeme werden anwenderseitig zwischen das Amt und das Kommunikationsnetz der Organisation geschaltet. Dabei verfügt die PBX zum Amt hin über einen Verbindungspool, dessen Größe die Anzahl gleichzeitig vermittelbarer Leitungen in das öffentliche Telekommunikationsnetz beschränkt. Die PBX erhält vom Telefonanbieter eine oder mehrere Kopfnummern zugewiesen, unter der sie von außerhalb erreichbar ist. Die *Terminals*[12] sind an der PBX angeschlossen, die nun deren Verwaltung und Steuerung übernimmt. Jedem angeschlossenen Terminal wird durch die PBX eine intern eindeutige *Terminal Number (TN)* zugewiesen. Diese Nummer entspricht der physischen Anschlussadresse. Über die Konfiguration der PBX, wird jeder TN eine *Directory Number (DN)* zugeordnet. Diese entspricht der internen Rufnummer eines Terminals, über die es innerhalb des lokalen Telekommunikationsnetzes angesprochen werden kann. Die Kombination aus Kopfnummer der Telefonanlage und Durchwahlnummer des Endgeräts ergibt die komplette Rufnummer, unter der man einen Apparat vom öffentlichen Netz aus erreichen kann. Für interne Verbindungen ergibt sich dabei ein positiver Nebeneffekt: Die Terminals lassen sich nun allein durch Wahl der internen DN erreichen ohne direkte Verbindungsgebühren zu verursachen. Auch die Administration des Telefoniesystems erfolgt jetzt an zentraler Stelle vor Ort, da an der PBX nach Bedarf Terminals angeschlossen, entfernt und konfiguriert werden können.

An dieser Stelle soll erwähnt werden, dass sich hinter einer DN mehrere Terminals mit unterschiedlichen TNs verbergen können. Gleichwohl kann ein Terminal über mehrere DNs verfügen. Auch muss die interne Rufnummer eines Terminals nicht notwendigerweise mit der von außen erreichbaren übereinstimmen. Des Weiteren sind auch Terminals denkbar, die nur von innerhalb des privaten Telekommunikationsnetzes erreichbar sind. All diese Sonderfälle können durch entsprechende Konfigurationsmöglichkeiten der PBX umgesetzt werden.

Die vollständig qualifizierte Telefonnummer, unter der ein Terminal von außerhalb des privaten Telekommunikationsnetzes erreichbar ist, setzt sich nun aus

- der Landesvorwahl,
- der Ortsvorwahl,
- der Kopfummer
- und der internen Durchwahlnummer

zusammen, wobei die ersten beiden Elemente *Landesvorwahl* und *Ortsvorwahl* weggelassen werden können, sofern man sich als Anrufer im gleichen Land beziehungsweise Ortsvorwahlbereich befindet. Die Kopfnummer entfällt bei internen Verbindungen. Tabelle 2-1 zeigt an einem Beispiel die einzelnen Elemente einer solchen Telefonnummer.

Erfolgt nun ein Anruf von außerhalb, so wird anhand der Rufnummer bis einschließlich der Kopfnummer der Amtskopf der PBX identifiziert und die Verbindung vermittelt. Die interne

[12] Endgeräte wie Telefone, Smartphones, Fax, Modems sowie CTI-Hardware in Form von internen und externen Erweiterungen für PCs

Landesvorwahl	Ortsvorwahl	Kopfnummer	Durchwahl
0049	711	972	93381

Tab. 2-1: Vollqualifizierte Telefonnummer

DN ist dabei für den Provider uninteressant, da erst die PBX diese auflöst, um die TN des Terminals zu erhalten. Ist dies geschehen ist der Endpunkt der Telefonverbindung ermittelt und die Leitung kann durchgeschaltet werden. Hierbei ist dem Amt nur die Kopfnummer der PBX bekannt. Die Ziffern der Durchwahl werden in Form einer Erweiterung mit übermittelt, damit die PBX die zuständigen Terminals ermitteln kann. Dies wird als *Direct Inward Dialing* bezeichnet. Ohne dieses Verfahren müssten alle eingehenden Verbindungen über eine Telefonzentrale weitervermittelt werden.

Wird hingegen ein Anruf von innerhalb des lokalen Telefonnetzes mit einer externen Nummer als Ziel initiiert, übernimmt die PBX den ersten Schritt des Vermittlungsprozesses. Hierfür wurde vorher eine Amtsvorwahlnummer definiert, die bei Anrufen mit externem Ziel der Zielrufnummer vorangestellt wird. Mit Hilfe dieser Amtsvorwahl (beispielsweise „0“) ermittelt die PBX, ob die Verbindung im lokalen Telefonnetz bleibt oder über eine Amtsleitung in das öffentliche Telefonnetz weitervermittelt werden soll.

Bei großen Organisationen kann allerdings der Einsatz mehrere TK-Anlagen an einem Standort notwendig sein. Zum einen kann die geforderte Zahl an anzuschließenden Geräten die Möglichkeiten einer Anlage überschreiten, zum anderen können auch geographische Begebenheiten mehrere PBX erfordern. Dabei ergibt sich eine neue Problematik, da ein Terminal über seine TN nur *einer* PBX bekannt ist. Denn nur die PBX, an der das Terminal angeschlossen ist, verfügt über die Information, welche Hardwareadresse sich hinter einer Durchwahlnummer verbirgt. Die anderen Anlagen müssen nun entsprechend konfiguriert werden, damit sie Verbindungsanfragen an Terminals mit unbekannter TN an die entsprechende PBX weiterleiten können. Dieser Sachverhalt lässt sich sehr gut mit der Situation mehrerer Router in einem IP-Netz vergleichen. Dabei verfügt jede PBX über Routingtabellen, in der die DNs den Hardwareadressen der Terminals zugeordnet sind. Erfolgt eine Verbindungsanfrage an eine bestimmte Rufnummer, so wird mittels dieser Tabellen die TN ermittelt. Bei Geräten die physisch an anderen PBX angeschlossen sind, verweist die Tabelle anstatt auf die Hardwareadresse auf die nächste, zuständige TK-Anlage, die ihrerseits die DN aufzulösen versucht.

Dies bedeutet für die Implementierung einer CTI-Lösung, dass durchaus mehrere Telefonanlagen vorhanden sein können, wobei jede über ihre eigene CTI-Schnittstelle verfügt. Im kompliziertesten Fall sind die Anlagen nicht einmal desselben Typs, sodass eine CTI-Lösung vor der Aufgabe steht, mit mehreren PBX über unterschiedliche Schnittstellen *nebenläufig* zu kommunizieren.

3 Ist-Situation

In diesem Kapitel wird die Ausgangssituation des Projektes dargestellt. Dabei werden neben der Beschreibung von Funktionsweise und Aufbau der ersten *webDial*-Implementierung auch die Rahmenbedingungen am T-Systems Standort in Stuttgart-Leinfelden beschrieben. Abschließend werden die Schwachstellen der bisherigen Lösung aufgezeigt, die in der Summe zur kompletten Neuentwicklung führten.

3.1 Die Vorgängerversion

Die erste Version von *webDial* ist als internes Projekt der T-Systems International GmbH am Standort Stuttgart-Leinfelden entstanden. *webDial* ist als einfaches und kostengünstiges Werkzeug für den unternehmensinternen Einsatz konzipiert worden. Dabei wurden nur Grundfunktionen eines Telefons implementiert. So können zum einen Telefongespräche mit internen sowie externen Teilnehmern eingeleitet werden, indem man entweder die HTML-Oberfläche von *webDial* zur Direkteingabe nutzt oder die Telefonnummer eines Kontakteintrages auf der Intranetseite des Corporate Directory anklickt. Weiter ist es auf Basis einer bestehenden Verbindung möglich, Konferenzen zu schalten oder einen Gesprächspartner weiterzuvermitteln.

webDial ist gänzlich mit den Skriptsprachen Python und PHP realisiert worden. Zur persistenten Speicherung anfallender Statusinformationen wurde die GNU-Bibliothek *GDBM*[1] verwendet. Die Software besteht aus drei Komponenten, die allesamt auf *einem* Linux-Server ausgeführt werden. Die webbasierte Benutzeroberfläche wurde aus einer Kombination von PHP- und Python-Skripten implementiert, die auf einem Apache Webserver aufsetzen. Die Erfassung von Benutzerinteraktionen erfolgt über HTML-Formulare, deren Eingabedaten mit Hilfe der HTTP/GET-Methode an den Webserverprozess übermittelt werden. Die Formulareingaben werden dabei in Form von Key/Value-Paaren der URI[2] angehängt, sodass sie vom Webserverprozess in Form eines Eingabestrings verarbeitet werden können. Die Kommunikation zwischen den GUI-Skripten und dem *webDial*-Dämon erfolgt mittels Socketverbindungen über die lediglich dieser Eingabestring durchgereicht wird. Der in Python geschriebene Dämon interpretiert nun diesen String und setzt die Anweisungen gegebenenfalls in Interaktionen mit den Telefonanlagen und dem Webserver um. Die dritte Komponente ist ein separater Prozess der in vordefinierten Zyklen dem Dämon Kommandos für diverse Wartungsaufgaben übermittelt. Dazu gehört beispielsweise das Sichern der zur Laufzeit angefallenen Daten in die GDBM-Datenbank oder die Bereinigung von veralteten Einträgen.

[1] GNU Standardbibliothek zur Realisierung einer dateibasierten Datenbank mit einem auf Hash-Routinen basierten Zugriffskonzept

[2] Uniform Resource Identifier

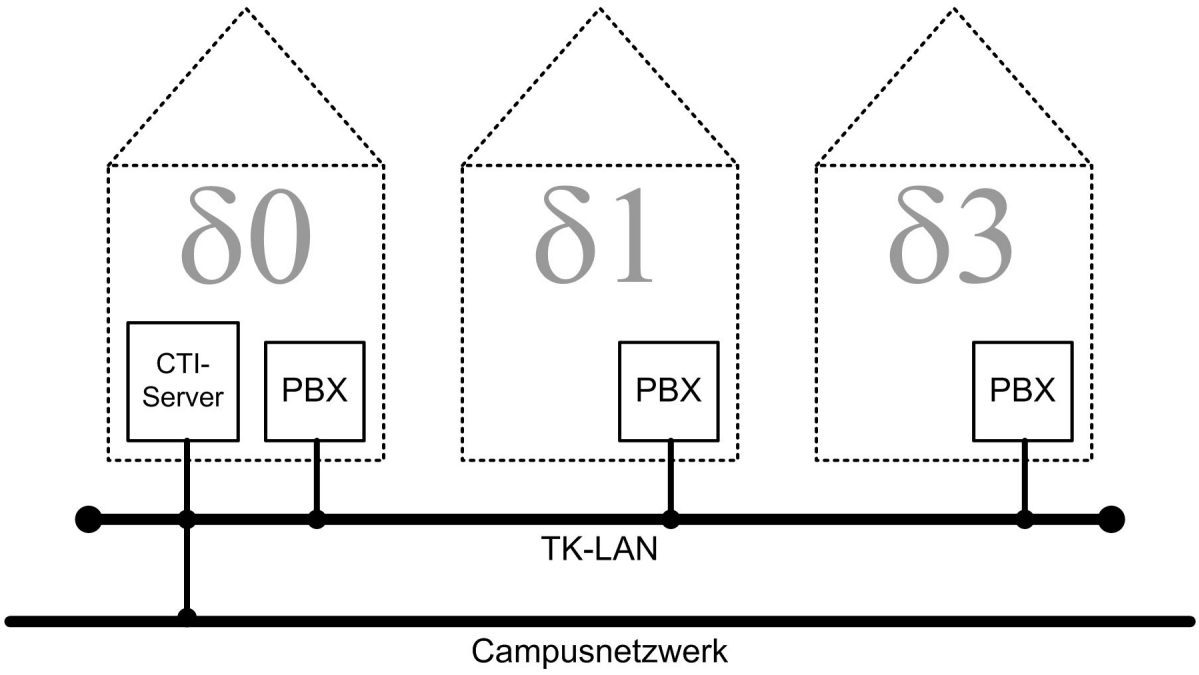

Abb. 3-1: Infrastruktur am Standort Leinfelden

3.2 Rahmenbedingungen am Standort

Da die neue Version von *webDial* zuerst am T-Systems-Standort in Stuttgart-Leinfelden eingesetzt werden soll, folgt nun eine kurze Beschreibung der dort anzutreffenden Rahmenbedingungen.

Die auf drei Gebäude verteilte Niederlassung verfügt über ein zusammenhängendes Telefonnetz mit ca. 2000 Teilnehmern. Erfahrungsgemäß nutzten etwa 10% die Vorgängerversion. Um die große Zahl an Terminals zu ermöglichen werden insgesamt drei Telefonanlagen vom Typ *Meridian I* von Nortel im Verbund betrieben. Alle PBX verfügen über ein so genanntes *Link-Modul*, das die Schnittstelle für CTI-Anwendungen bereitstellt. Neben seriellen Schnittstellen für Administrationsaufgaben, erweitert das Modul eine PBX um einen Ethernet-Anschluss, der das TCP/IP-Protokoll unterstützt. Auf diese Weise sind am Standort alle drei PBX über ein abgeschlossenes, privates[3] IP-Subnetz miteinander verbunden. Mit zwei Netzwerkinterfaces ist der CTI-Proxy nun sowohl mit dem Intranet von T-Systems als auch mit dem TK-Subnetz verbunden. Der auf diesem CTI-Server ausgeführte *webDial*-Dämon kann nun an zentraler Stelle mit allen drei PBX über deren Link-Modul kommunizieren (vgl. Abbildung 3-1).

Im Zuge der Erweiterung des Angebots auf andere Standorte wurde eine Prüfung auf die Verfügbarkeit des Dienstes erforderlich. Denn um die Nutzung möglichst unkompliziert zu gestalten, blieb die *webDial*-URI im unternehmensweiten Intranet einheitlich. Bei Aufruf der Startseite muss zunächst überprüft werden, ob der Dienst für den jeweiligen Benutzer verfügbar und gegebenenfalls welcher CTI-Server zuständig ist. Ein Pythonskript, das beim

[3] unter Verwendung privater, nicht gerouteter IP-Adressen

ersten Aufruf der Startseite ausgeführt wird, erfüllt diese Aufgabe. Je nach IP-Adresse des Clientcomputers oder der des zwischengeschalteten Proxys wird entschieden, ob der Arbeitsplatz des Benutzers sich an einem Standort befindet, an dem die Steuerung der Telefonapparate über *webDial* möglich ist. Dieses Prinzip setzt eine grobe Übereinstimmung von IP-Adressbereichen und TK-Infrastruktur voraus. Im schlechtesten Falle müsste sonst für jede einzelne IP-Adresse die Verfügbarkeit spezifiziert werden.

Das LDAP-fähige Corporate Directory enthält neben den Kontaktinformationen der Mitarbeiter auch deren Benutzerkonten zur Anmeldung an den jeweiligen Domänen. Somit ist durch Nutzung des Verzeichnisdienstes sowohl die Identifizierung von Gesprächspartnern als auch eine Zugriffsbeschränkung auf einen authentifizierten Personenkreis möglich.

3.3 Nachteile der bisherigen Lösung

Im Folgenden sollen die Defizite der bisherigen Lösung aufgezeigt werden, die ausschlaggebend für die Entscheidung waren *webDial* neu zu entwickeln.

Trotz der immer stärker werdenden Akzeptanz moderner, leicht zu erlernender Skriptsprachen wie Python und Perl, von denen sich viele kürzere Entwicklungszeiten versprechen, gibt es berechtigte Vorbehalte gegen deren Einsatz in einer Anwendung wie *webDial*. In der Vergangenheit hat sich der Pythoncode des Dämon zwar als performant genug erwiesen, jedoch ist zu bezweifeln, dass mit wachsender Funktionalität und Komplexität der Anwendung und steigenden Benutzerzahlen dies zu gewährleisten ist. Compilersprachen wie C++ und Java haben hier im Bereich der Performance eindeutige Vorteile gegenüber interpretierten Sprachen.[4] Python bietet durch seine flexible, dynamische Typisierung dem Entwickler eine erhöhte Flexibilität bei der Lösung von Aufgaben und reduziert damit den Entwicklungsaufwand. Allerdings birgt gerade dies erhebliche Gefahren, da der kürzere Entwicklungszyklus oft mit Defiziten bei der Gesamtkonzeption und Verständlichkeit des Quelltextes erkauft wird. So erfordert eine Skriptsprache vom Entwickler eine hohe Disziplin bei der Einhaltung bestimmter Entwurfskonzepte. Restriktivere Sprachen wie C++ und vor allem Java zwingen den Entwickler sich an bestimmte (objektorientierte) Grundsätze zu halten und erleichtern gerade durch die Möglichkeiten der Datenkapselung und des Information Hiding die Zusammenarbeit und Koordinierung mehrerer Projektmitglieder. Zudem vereinfacht ein klar strukturiertes, objektorientiertes Entwurfskonzept die Weiterentwicklung eines bestehenden Systems, ohne die Software als Ganzes anpassen zu müssen.

Somit ergibt sich die Forderung nach dem Umstieg auf eine andere Technologie. Eine mit weniger Aufwand verbundene Portierung auf eine andere Programmiersprache ist ebenfalls keine Option. Dies wird bedingt durch die fehlende, objektorientierte Konzeption und den stark prozeduralen Charakter. Zwar ist Python eine objektorientierte Skriptsprache, in der fast alle objektorientierten Paradigmen umgesetzt werden können. Trotzdem ist in der Implementierung von *webDial* ein eindeutig funktionsorientiertes Design zu erkennen. So besteht der Dämon aus insgesamt *vier* Klassen. Eine implementiert die Log-Funktionen, während zwei

[4] Sprachen, deren Sourcecode in Byte-Code übersetzt wird, welcher dann zur Ausführungszeit im Rahmen einer *Virtual Machine* interpretiert wird, sollen hier zu den compilierten Programmiersprachen gezählt werden.

weitere Klassen den Socket-gebundenen Server für die Verarbeitung der Client-Anfragen umsetzen. Alle weiteren Aspekte, wie die Kommunikation mit den Telefonanlagen und dem Webserver, die Implementierung des CTI-Protokolls sowie die Ablaufsteuerung diverser Anwendungsfälle, sind in einer einzigen Klasse mit über 60 Methoden realisiert. Zum Vergleich verteilen sich die 13 verbleibenden Methoden auf die drei anderen Klassen und den Main-Namensraum. Dieser Umstand erschwert die Verständlichkeit des Quelltextes und macht eine sinnvolle Erweiterung des Funktionsumfanges sehr problematisch. Die Unterstützung anderer TK-Anlagen ist sogar ohne Umstrukturierung der Implementierung nicht möglich. Ebenso wenig ist eine Konfiguration mit Telekommunikationsanlagen unterschiedlichen Typs nicht abbildbar, da die nebenläufige Kommunikation über verschiedene CTI-Protokolle nicht vorgesehen ist. Eine Fortführung der bisherigen Software würde zudem noch durch das Fehlen jeglicher Dokumentation des Quelltextes, abgesehen von einigen wenigen Kommentaren, und der Konfigurationsdateien behindert.

Einen weiteren Kritikpunkt bietet die fehlende Skalierbarkeit. Alle drei Prozesse (Webserver, Dämon und Wartungsprozess) müssen auf einer physischen Maschine ausgeführt werden. Das Verteilen auf mehrere, leistungsfähigere Maschinen ist nur unter Anpassungen zu erreichen. Ebenso fehlt eine modulare Datenbankanbindung, um bei Bedarf das System austauschen und einen dedizierten Server einsetzen zu können.

Zuletzt bleibt noch zu erwähnen, dass alle Statusinformationen über Telefone, Fehlermeldungen, geöffnete Socketverbindungen und Authentifizierungsinformationen in wenigen, verschachtelten und deshalb schwer zu überblickenden Dictionaries[5] gehalten werden. Zudem gibt es in Python keine Möglickeit diese Datenelemente zu schützen, da es prinzipiell keine privaten Membervariablen gibt. In Python sind alle Klassen, Attribute und Methoden *public*. In Hinblick auf zukünftige Erweiterungen ist dies kein wünschenswerter Zustand.

Zusammenfassend ist die ungenügende Modularisierung der Ablaufsteuerung, der Datenbank- und PBX-Anbindung als K.O.-Kriterium für die Portierung der bisherigen Lösung zu werten. Es fehlen klare Schnittstellen für Implementierungen weiterer GUIs und PBX-Treiber sowie ein objektorientiertes Gesamtmodell, das die relevanten Erkenntnisobjekte wie *Anruf*, *Gesprächsteilnehmer* oder *Terminal* erfasst. Damit ist sowohl ein Technologiewechsel als auch die Neuentwicklung des *webDial*-Dämons notwendig, um den zukünftigen Anforderungen gerecht werden zu können.

[5] Datentyp von Python zur einfachen Implementierung von Hash-Tabellen (Maps nach dem Key/Value-Zugriffsprinzip)

4 Anforderungen

Dieses Kapitel beschäftigt sich mit den Anforderungen an die neue Version der *webDial*-Serverkomponente. Nachdem sich das letzte Kapitel mit der Vorgängerversion beschäftigte, sollen nun die funktionalen und strukturellen Anforderungen an die neue Version herausgearbeitet werden.

An dieser Stelle soll nochmals betont werden, dass am Ende dieses Projektes ein funktionsfähiges Release der Serversoftware stehen soll. Design und Entwicklung der Benutzeroberfläche sowie die Implementierung neuer Funktionen sind nicht im Rahmen dieses Projektes vorgesehen. Das Ziel ist das Erreichen des Status quo bei stark verbesserter Anpassbarkeit und Erweiterungsfähigkeit.

4.1 Funktionale Anforderungen

Im Folgenden wird der geforderte Funktionsumfang aus Sicht des Anwenders beschrieben. Eine Verfügbarkeit dieser Funktionen ist allerdings nur in Verbindung mit digitalen Systemtelefonen gegeben, die die Sonderfunktionen der eingesetzten Telekommunikationsanlage unterstützen. Analoge Endgeräte verfügen in der Regel nicht über Möglichkeiten der Steuerung durch die PBX und können damit auch nicht nach dem *Third Party Call Control Modell* beeinflusst werden.

4.1.1 Make Call

Mit dieser Funktion ruft der Benutzer, unter Angabe einer Telefonnummer, einen anderen Teilnehmer an. Dabei ist es für ihn nicht von Bedeutung, ob sich der gewünschte Gesprächspartner ebenfalls im lokalen TK-Anlagennetz befindet. Diese Unterscheidung nehmen ihm *webDial* und die TK-Anlage ab, solange die Rufnummer den in Kapitel 2.4 erläuterten Konventionen entspricht. Über dies hinaus kann *Make Call* für die weiteren Funktionen *Conference*, *Transfer* und *Change Active Call* als Wegbereiter dienen. Um beispielsweise eine Konferenz einleiten zu können, muss der Initiator an seinem Apparat bereits an einem Gespräch beteiligt sein, welches er zuvor über *Make Call* aufgebaut haben kann.

Dieser Sachverhalt impliziert bereits, dass es sich beim anrufenden Terminal, um das des Benutzers handelt. An dieser Stelle lässt sich bereits der Vorteil des Third-Party-Modells verdeutlichen, da keine Unterscheidung zwischen anrufendem und angerufenem Terminal stattfindet. Deshalb ist es mit dieser Funktion grundsätzlich möglich zwischen zwei *beliebigen* Terminals Verbindungen aufzubauen. Einzige Bedingung hierbei ist, dass sich mindestens eines der beiden Terminals in der Domäne[1] der PBX befindet. Dies ermöglicht Anwendungen,

[1] Einflussbereich der PBX; Einheit der durch die PBX kontrollierten Endgeräte

die die Funktionalität eines digitalen Systemtelefons übersteigen.

4.1.2 Conference

Unter einer Konferenz versteht man eine Telefonverbindung zwischen mindestens drei Terminals. Das Besondere ist, dass alle Teilnehmer bidirektional miteinander kommunizieren können. Das gleichzeitige Sprechen und Hören aller Teilnehmer ist möglich. Für den Benutzer kann der Aufbau einer Konferenzschaltung folgendermaßen ablaufen: Er erhält einen Anruf von einem externen Mitarbeiter, der mit ihm und einem weiteren Kollegen eine Besprechung am Telefon durchführen möchte. Über *webDial* kann der Anwender nun den dritten Konferenzteilnehmer anrufen. Hierfür wird zuerst die Verbindung mit dem ersten Gesprächspartner gehalten, um eine freie Leitung für den Anruf des fehlenden Teilnehmers zu erhalten. Ist dieser erreichbar und bereit teilzunehmen, kann der Benutzer beide Gespräche zu einer Konferenzverbindung zusammenschalten.

4.1.3 Transfer

Diese Funktion ermöglicht es dem Benutzer, den Gesprächspartner eines aktiven Gespräches an einen anderen weiterzuverbinden. Hierfür ruft der Benutzer den neuen Teilnehmer an, während die Verbindung mit dem bisherigen gehalten wird. Nun hat der Benutzer die Wahl, den Teilnehmer sofort durchzustellen oder zu warten bis der Zielteilnehmer den Anruf entgegen nimmt, um erst nach einleitender Absprache zu verbinden.

4.1.4 Retrieve Main Call

Wie bereits erläutert ist für die Anrufweiterleitung oder Konferenzschaltung ein aktives Gespräch Voraussetzung. Ein dritter Teilnehmer wird in beiden Fällen über ein zweites Telefongespräch erreicht. Dieses zusätzliche Gespräch wird *Consultation Call* (Konsultationsgespräch) genannt. Wenn dieses Konsultationsgespräch aus verschiedenen Gründen fehlschlägt, ermöglicht die Funktion *Retrieve Main Call* es dem Benutzer das erste Gespräch, den *Main Call*, zurückzuholen, wobei das zweite beendet wird. Mit dieser Funktion kann auch ein gehaltenes Gespräch ohne beteiligtes Konsultationsgespräch wieder aktiviert werden.

4.1.5 Release Active Connection

Unter *Release Active Connection* versteht man das Beenden der Teilnahme an einem Telefongespräch. Mit der Konfiguration und Ausstattung der eingesetzten PBX kann ein Terminal maximal an zwei Gesprächen gleichzeitig beteiligt sein, wobei der Consultation Call aktiv ist und der Main Call gehalten wird. In diesem Fall wird durch Release Active Connection die Verbindung mit dem Consultation Call beendet. Die Verbindung zum anderen Gespräch bleibt bestehen, wird aber weiterhin gehalten. Erst durch *Retrieve Main Call* wird dieses aktiviert. Ist der Apparat nur an einem Gespräch beteiligt, führt die Aktion Release Active Connection zum Auflegen des Apparates, da danach keine weitere Verbindung verbleibt.

4.1.6 Anruferidentifizierung

Diese Funktion ermöglicht es dem Benutzer sich über einen ankommenden Anruf näher zu informieren. Ist die Rufnummer bekannt, wird mit ihrer Hilfe versucht nähere Informationen über den Anrufer zu ermitteln. Dies erfolgt durch Abfragen des lokalen LDAP-Dienstes mit der übermittelten Rufnummer als Suchkriterium. Diese Funktion kann nur angeboten werden, wenn die Rufnummer des Anrufers ermittelt werden kann. Aus verschiedenen Gründen kann dies nicht möglich sein. Während bei externen Anrufern der Serviceprovider die Rufnummerübermittlung als Funktion anbieten muss, ist es bei internen eine Sache der Konfiguration der TK-Anlage. Im Zuge der Digitalisierung der öffentlichen Netze steht mittlerweile die Funktion der Rufnummerübermittlung auch bei analogen Endgeräten zur Verfügung.

4.2 Strukturelle und konzeptionelle Anforderungen

Dieses Kapitel beschäftigt sich mit den strukturellen und konzeptionellen Anforderungen an die neue Version der Software.

Das *Third Party Call Control Modell* fand bereits bei der ersten Implementierung Anwendung und soll wegen der höheren Flexibilität und der Möglichkeiten, mehr Aufgabenstellungen als mit der First-Party-Variante realisieren zu können, beibehalten werden. Damit bleibt das Grundprinzip einer Client/Server-Architektur erhalten. Eine Umstrukturierung der Software auf Serverseite ist jedoch erforderlich.

Weiter soll *webDial* durch die Entwicklung eines entsprechenden Treibers wie der Vorgänger in der Lage sein, die unter Kapitel 4.1 aufgeführten Funktionen in Verbindung mit der *Meridian I* TK-Anlage anbieten zu können.

Eine zentrale Rolle bei der Betrachtung der Strukturanforderungen spielt die veränderte Zielsetzung des Softwareanbieters. Da *webDial* ursprünglich unter dem Gesichtspunkt eines internen Tools entwickelt wurde, dass bei möglichst geringem Entwicklungsaufwand die gegebenen Anforderungen erfüllen sollte, waren Kriterien wie Wiederverwendbarkeit, Erweiterbarkeit und Modularität von geringer Priorität. Da *webDial* nun zu einem kommerziellen Produkt weiterentwickelt werden soll, das an die heterogenen Infrastrukturen potentieller Kunden anpassbar ist, rücken diese Faktoren in den Mittelpunkt des Softwareentwurfes. Damit ist die Umsetzung der Anforderungen von T-Systems allein nicht mehr ausreichend. Für weitere Interessenten stellt es ein K.O.-Kriterium dar, ob ihre TK-Anlagen von *webDial* unterstützt werden und wie es um eine mögliche Ausweitung der Unterstützung weiterer TK-Anlagen bestellt ist. Beim Entwurf ist daher darauf zu achten, eine Abstraktionsebene zwischen *webDial*-Dienst und PBX vorzusehen, die gewährleistet, dass durch Einsatz anderer Telekommunikationssysteme und damit anderer CTI-Protokolle, so wenig Einfluss wie möglich auf den Rest der Anwendung entsteht.

Auch die Anbindung der graphischen Benutzeroberfläche soll möglichst universell gehalten werden, damit später der geplanten Implementierung eines Webinterfaces noch eine dedizierte Clientanwendung folgen kann. Um Doppelimplementierungen und erhöhten Wartungsaufwand zu vermeiden ist es hierfür notwendig, Ablaufsteuerungen möglichst nah am Server zu implementieren und – falls erforderlich – für bestimmte GUI-Implementierungen eigene

Schnittstellen zur Verfügung zu stellen.

Je nach Verlauf der Weiterentwicklungen von *webDial* kann der Datenhaltung eine höhere Bedeutung zukommen. Derzeit ist sie lediglich für das Speichern von Statusinformationen erforderlich, um dem Benutzer einen höheren Bedienkomfort bieten zu können. Es ist jedoch durchaus möglich, *webDial* beispielsweise um einen Bereich für das Accounting von Gebührendaten zu erweitern, wobei das Speichern und Weiterverarbeiten der Daten essentiell wird. Deshalb soll bei der Wahl der Technologie darauf geachtet werden, dass eine Anbindung gängiger Datenbanksysteme möglich ist.

Beim Entwurf zu berücksichtigen sind außerdem Sicherheitsaspekte. So soll eine Nutzung der Software erst nach erfolgter Authentifizierung über geeignete Verfahren möglich sein. Auch eine Verschlüsselung des Netzwerkverkehrs zwischen dem Clientcomputer des *webDial*-Nutzers und dem Server (Dämon- oder HTTP-Serverprozess, je nach Ausprägung der Benutzeroberfläche) soll möglich sein. Im Falle der zu veröffentlichenden Version von *webDial* soll die Benutzerauthentifizierung auf Basis des örtlichen, LDAP-fähigen Verzeichnisdienstes erfolgen.

Zuletzt bleibt zu erwähnen, dass möglichst freie Tools und Entwicklungsumgebungen eingesetzt werden sollen, um eine kostengünstige Entwicklung zu ermöglichen. Lizenzkonzepte die eine Offenlegung des Quelltextes erfordern, stellen hingegen keine Option dar. Da das Unternehmen Canoris im Sektor der CTI-Anwendungen ausschließlich Server mit PC-Hardware und SuSe-Linux Betriebssystem einsetzt, soll *webDial* auch primär für diese Plattform entwickelt werden.

5 Anforderungsanalyse

Nachfolgend werden die in Kapitel 4 dargelegten Anforderungen reflektiert und die daraus resultierende Technologiewahl begründet. Darüber hinaus werden die technologischen Rahmenbedingungen erläutert, die durch das proprietäre Protokoll des CTI-Links der *Meridian I* gesetzt werden. Am Schluss dieses Kapitels wird auf das CSTA-Protokoll eingegangen, um eine über die Grenzen dieses Projekts hinaus reichende Perspektive zu bieten.

5.1 Technologieentscheidung

Die Wahl der Programmiersprache für die Entwicklung von *webDial* fiel auf Java. Neben der Plattformunabhängigkeit, der Typsicherheit und dem Sandkastenprinzip der Laufzeitumgebung sprachen die Etablierung als Hochsprache und die zehnjährige Reifephase für Java. Dies zeigt sich vor allem am großen Umfang der Java-Klassenbibliothek des *Java 2 Software Development Kits*, was die Implementierung vieler Basisroutinen erübrigt. Schon in der frühen Phase des Projekts war zudem abzusehen, dass im hohen Maße auf *Multi-Threading* zurückgegriffen werden müsse. Java bietet zur Erzeugung und Steuerung von Threads relativ einfache und zudem ausgereifte Mechanismen.

Die Anforderungen an Modularität, Wiederverwendbarkeit und Anpassbarkeit der Software implizieren bereits den objektorientierten Ansatz, den eine rein objektorientierte Sprache wie Java bietet.[1] Die objektorientierte Skriptsprache Python konnte aufgrund der schlechteren Performance von vornherein ausgeschlossen werden. Somit konnte nur noch C++ in die nähere Auswahl gezogen werden. Gegen C++ sprach jedoch die Notwendigkeit, viele Routinen, die bei Java standardmäßig bereits existieren, implementieren zu müssen, was in Anbetracht der Komplexität der Aufgabe zeitlich nicht realisierbar war.

Die Ausführungsgeschwindigkeit ist, entgegen sich hartnäckig haltender Vorurteile, kein Argument mehr gegen den Einsatz von Java. Verschiedene Performancetests und Benchmarks belegen, dass Java zu C++ aufgeschlossen und in objektorientierten Disziplinen sogar überholt hat (vgl. [SCHU03], getestete JRE: Version 1.4.2). Zudem hat sich, nach Aussagen von Sun Microsystems, die Performance von Version 1.4 auf 5.0 (respektive 1.5) nochmals erhöht (vgl. [J2SE05@]).

Mit den APIs *JNDI*[2] und *JDBC*[3] verfügt Java über ausgezeichnete Voraussetzungen zur Integration von Namens- und Verzeichnisdiensten sowie (objekt)relationalen Datenbanken. Beide Technologien erfreuen sich hoher Akzeptanz, sodass die Mehrheit der Hersteller dieser

[1] Der Verfasser ist sich bewusst, dass die Primitiven nicht als Objekte abgebildet sind – wie beispielsweise bei Smalltalk – jedoch der objektorientierte Ansatz im Vergleich zu C/C++ zwingend ist.

[2] Java Naming and Directory Interface

[3] Java Database Connectivity

Systeme eine entsprechende Unterstützung über geeignete Treiber anbietet. So ist es beispielsweise möglich sowohl LDAP-fähige Dienste als auch Datenbanken von MySQL oder Oracle einzubinden. Den Internetseiten [JDBC05@] und [JNDI05@] sind die unterstützten Namens- und Verzeichnisdienste sowie Datenbanksysteme zu entnehmen.

Ein weiteres wichtiges Argument für Java bietet die *RMI*[4]-Technik. Mit ihr ist es sehr einfach möglich, verteilte Anwendungen auf Objektebene zu realisieren. Da es sich bei *webDial* um eine verteilte Anwendung handelt, entfällt durch Zurückgreifen auf diese Technologie die aufwendige Entwicklung eines eigenen Applikationsprotokolls für die Client/Server-Kommunikation.

Für die nach Abschluss dieses Projektes zu entwickelnden *webDial*-Clients bietet Java mit dem GUI-Framework *Swing* ausreichende Möglichkeiten zu deren plattformunabhängigen Implementierung. Darüber hinaus existieren mit *JSP*[5] und den *Servlets* etablierte Technologien zur Erstellung dynamischer, webbasierter Anwendungen für die Realisierung der HTML-Oberfläche.

Für die Verschlüsselung der Kommunikation verfügt Java seit der aktuellen Version 1.5 über eine einfache Möglichkeit jedwede TCP-basierte Datenkommunikation (auch RMI) über das *SSL*[6]-Protkoll abzusichern, indem eine andere bereitgestellte Socket-Implementierung benutzt wird.

Unter Berücksichtigung all dieser Argumente zeigte sich Java als geeignete Wahl die nichtfunktionalen Anforderungen zu erfüllen.

5.2 Einsatz von Standardlösungen

Die Wahl Java als Programmiersprache einzusetzen wurde noch durch die Existenz des JTAPI-Standards bestärkt. Dieser zeigte sich als geeignetes Mittel zur Abbildung eines standardisierten und leistungsfähigen Call-Modells. Der Einsatz bereits bestehender Middleware-Lösungen wie *CallPath* von IBM oder *TSAPI* von Novell erübrigte sich aufgrund der Lizenzkosten, die eine rentable Eigenentwicklung von *webDial* für Canoris nicht ermöglichen. Darüber hinaus waren während der Entwicklungsphase von Nortel keine APIs für das CTI-Link Modul der *Meridian I* verfügbar, was die Wahl hätte beeinflussen können.

Mit der Unterstützung des *First* und *Third Party Call Control Modells* bietet JTAPI die Möglichkeit ohne Lizenzkosten auf einen ausgereiften und erprobten Standard zurückzugreifen, ohne das Rad neu erfinden zu müssen. Kapitel 6.3 gibt eine Einführung in JTAPI.

5.3 CTI-Link Protokolle

Zentrale Rolle bei der Entwicklung einer CTI-Anwendung spielt die Interaktion zwischen Anwendungs- und PBX-Ebene. In diesem Zwischenraum werden Anweisungen der CTI-

[4] Remote Method Invocation
[5] Java Server Pages
[6] Secure Socket Layer

Applikation an die Telekommunikationsanlage in verständliche Befehle übersetzt. In die andere Richtung werden Statusinformationen über den Zustand des Telekommunikationssystems an die Applikation übermittelt. Dabei sind die Rollen eines Client/Server-Verhältnisses nicht zwingend fest vergeben. Es sind auch Abläufe denkbar, bei denen die Telekommunikationsanlage der Anwendung gegenüber als Client auftritt, um beispielsweise Handlungsanweisungen für ACD-Funktionen abzufragen, wie es bei Call-Center-Servern wie dem *Nortel Networks Symposium Call Center* für die *Meridian I* der Fall ist. Für diese Kommunikation verfügen aktuelle Telekommunikationsanlagen für den Einsatz in mittleren bis großen Organisationen über CTI-Schnittstellen – den CTI-Links. Meist handelt es sich hierbei um optionale, kostenpflichtige Ausstattungsmerkmale. Dies trifft auch auf das Link-Modul der *Meridian I* von Nortel zu.

Ein Hauptproblem bei der Kommunikation stellt heute noch immer die wenig ausgeprägte Standardisierung der verwendeten Protokolle dar. So setzt Nortel bei ihrem Link-Modul auf ein eigenes, proprietäres Protokoll, dessen Spezifikationen von Fremdentwicklern käuflich erworben werden müssen. Wie die Übersicht in Kapitel 2.3 bereits zeigte, haben Bestrebungen unterschiedlichster industrieller Gremien in diesem Rahmen zur Verabschiedung einer Reihe von Spezifikationen für Hardware- und Softwarestandards geführt. Leider gibt es keinen Standard der sich auf breiter Linie durchgesetzt hat. Der Idealfall, mit einer Protokollimplementierung alle TK-Anlagen verschiedenster Hersteller ansprechen zu können, bleibt weiterhin unerreicht.

Neben JTAPI ist für dieses Projekt noch die CSTA[7]-Spezifikation der ECMA[8] von Bedeutung. Bei CSTA handelt es sich um eine in drei Phasen erweiterte Spezifikation eines umfassenden Kommunikationsprotokolls für CTI-Anwendungen. Mittlerweile bieten führende Hersteller TK-Anlagen mit CSTA-Unterstützung an und stellen Treiber für CSTA-basierte Lösungen wie *CallPath* von IBM oder *TSAPI* von Novell bereit. Mit dem CSTA-Standard wird versucht, eine breite Basis für viele erdenkliche CTI-Applikationen zu schaffen. Deshalb wurde ein Netzwerkprotokoll mit einer Fülle von Funktionen spezifiziert, die notwendigerweise nicht von allen PBX-Systemen unterstützt werden müssen. Die CSTA-Spezifikation ist für *webDial* deshalb von Bedeutung, da die PBX *Alcatel OmniPCX 4400 R4.1/4.1.1* nach der *Meridian I* als nächstes unterstützt werden soll. Der dabei zu entwickelnde Treiber kann größtenteils auf der CSTA-Spezifikation basieren, da diese PBX eine weitgehend zu CSTA Phase II konforme CTI-Schnittstelle aufweist. Deshalb wird in Kapitel 5.3.2 kurz auf den Aufbau der Spezifikation eingegangen und, in Hinblick auf eine zukünftige Unterstützung von CSTA-basierten CTI-Links, werden diejenigen Protokollelemente aufgeführt, die zur Umsetzung der aktuellen funktionalen Anforderungen vorhanden sind.

5.3.1 Meridian Link

Da während des gesamten Projektes keine Protokollspezifikation zur Verfügung stand, musste der Inhalt dieses Kapitels durch Reverseengineering der ersten *webDial*-Version erarbeitet werden. Deshalb erhebt folgende Protokollbeschreibung keinerlei Anspruch auf

[7] Computer Supported Telecommunications Applications
[8] European Computer Manufacturer Association

Request		**Response**	
Bit-Vektor	Dezimal	Bit-Vektor	Dezimal
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1	3	1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1	32771
0 1 0 1 0 0 1 1 0 1 0 0 1 0 0 0	21320	1 1 0 1 0 0 1 1 0 1 0 0 1 0 0 0	54088

Tab. 5-1: Zusammenhang der Referenznummern zwischen Request- und Response-Nachrichten

Vollständigkeit. Es sind nur diejenigen Nachrichtentypen des Protokolls bekannt, die auch bereits in der Implementierung des Vorgängers Anwendung fanden.

Das Meridian-Link-Protokoll setzt auf *TCP/IP*[9] auf. Das Protokoll besteht aus einzelnen Nachrichten die im Payload der TCP-Pakete transportiert werden. So kann die Kommunikation zwischen PBX und CTI-Anwendung auf Basis von Sockets erfolgen. Die Nachrichten bestehen aus 8-Bit-codierten Zeichenfolgen, die zum besseren Verständnis in hexadezimaler Form dargestellt werden. Ein Zeichen wird dabei durch einen zweistelligen Hexadezimalwert repräsentiert. Für die Kommunikation ist eine stehende TCP/IP-Verbindung erforderlich. Bevor das Link-Modul Befehle akzeptiert, muss sich eine CTI-Anwendung registrieren. Im Erfolgsfall wird der Anwendung eine *Association-ID* zugewiesen (vgl. Kapitel 5.3.1.2 und 5.3.1.3). Diese ID ermöglicht bei mehreren gleichzeitig registrierten Anwendungen die korrekte Zuordnung der Nachrichten. Das Ergebnis einer erfolgreichen Registrierung ist eine so genannte *Association* zwischen CTI-Link und Anwendung.

Jede zum Link hin ausgehende Nachricht verfügt über eine fortlaufende, 16 Bit lange Referenznummer. Das höchstwertige Bit ist dabei immer null. Bei eingehenden Nachrichten, die sich auf vorherige beziehen, ist dieses Bit der Nachrichtenreferenznummer invertiert, sodass Request und Response einander zugeordnet werden können. Bei einer nummerischen Interpretation entspricht dies der Addition von 32 768 (2^{15}). Tabelle 5-1 zeigt diesen Sachverhalt anhand zweier Beispiele. Beziehen sich eingehende Nachrichten nicht auf andere, so haben sie eine Referenznummer von *00*.

Nach erfolgreicher Registrierung der Applikation muss der Telekommunikationsanlage mitgeteilt werden, welche Terminals über das CTI-Interface gesteuert und überwacht werden sollen. Dabei können die Terminals entweder einzeln oder alle auf einmal registriert werden (vgl. Kapitel 5.3.1.6). Nur von registrierten Terminals werden Statusinformationen über das Link-Modul übermittelt. Dementsprechend ist nur die Steuerung von registrierten Terminals möglich. Es können nur diejenigen Terminals registriert, gesteuert und überwacht werden, die auch physisch an der PBX angeschlossen sind. Für einen Verbund mehrerer Telekommunikationsanlagen bedeutet dies, dass Registrierung, Überwachung und Steuerung am richtigen Link-Modul erfolgen müssen.

Die Telefonanlage *Meridian I* organisiert registrierte Apparate in so genannten *Associate Set Assignments (AST)*. Nur Terminals, die als AST spezifiziert sind, stehen CTI-Anwendungen zur Verfügung. In den Fällen in denen sich mehrere Terminals eine DN teilen, kann nur dasjenige manipuliert werden, das als AST konfiguriert wurde. Bei gleicher DN kann nur eines der

[9] Transmission Control Protocol/Internet Protocol

Position	1	2	3	4	5	6	7	8	9	10
Zeichen	xFF	x0A	x00	x0D	x0C	x00	x07	x01	x05	x00

Tab. 5-2: Meridian-Link – Aufbau des Header-Elementes

Terminals das AST-Flag tragen, wodurch die Steuerung und Überwachung von Terminals mit gleicher Telefonnummer bei der *Meridian I* ohne entsprechende Erweiterung[10] *nicht* möglich ist.

5.3.1.1 Information Elements

Wie bereits erwähnt, besteht eine Nachricht aus einer Folge von Zeichen. Diese sind innerhalb einer Nachricht in so genannten *Information Elements (IE)* gruppiert. Eine Nachricht besteht immer aus mindestens einem solchen Informationselement, dem Header-Element (vgl. Tabelle 5-2).

Ein Header-Element besteht immer aus zehn Zeichen und steht, sollte die Nachricht aus mehren Informationselementen bestehen, immer am Anfang. Das erste Zeichen *xFF* signalisiert den Beginn eines neuen Header-Elementes und damit den Beginn einer neuen Nachricht. Das zweite Zeichen gibt die Länge des ganzen Header-Elementes an, während die Bytes drei und vier die Länge der gesamten Nachricht, inklusive aller noch folgenden IEs, spezifiziert. Das fünfte Byte trägt die Association-ID und die Bytes sechs und sieben die Nachrichtenreferenznummer. Bei den Bytes Acht und Neun handelt es sich um den Schlüssel, der den Nachrichtentyp kennzeichnet. Die bekannten Nachrichtentypen werden in den anschließenden Kapiteln näher erläutert. Tabelle 5-3 zeigt eine Liste der Nachrichtentypen mit hexadezimalem Schlüssel und jeweiliger Senderichtung. Tabelle 5-2 zeigt das Header-Element einer Nachricht des Typs *DNRegisterRequest*. Das letzte Byte des Header-Elements erfüllt keine bekannte Funktion und ist bei Nachrichten meist *x00*.

Je nach Nachrichtentyp folgen mehrere Informationselemente auf das Header-Element, wobei die ersten beiden Bytes nach demselben Schema wie das Header-IE aufgebaut sind. Das erste Byte spezifiziert dabei den IE-Typ, während das zweite die Länge des Elements angibt. Nachfolgende Bytes kodieren die eigentliche Information. Dabei gibt es drei unterschiedliche Möglichkeiten der Interpretation von Bytes als

- ASCII-Zeichen,
- numerischen Wert oder
- einfaches Token (ohne kodierte Information).

In Tabelle 5-5 sind die bekannten Informationselemente neben ihren hexadezimalen Schlüsseln aufgeführt.

[10] Diese Funktionalität wird für gewöhnlich im Rahmen von Call-Centern benötigt und durch entsprechende Erweiterungen bereitgestellt.

Nachricht	Schlüssel	Senderichtung
ApplicationRegisterRequest	x01 x01	Applikation→PBX
ApplicationRegisterResponse	x01 x02	PBX→Applikation
ApplicationReleaseRequest	x01 x03	Applikation→PBX
ApplicationReleaseResponse	x01 x04	PBX→Applikation
DNRegisterRequest	x01 x05	Applikation→PBX
DNRegisterResponse	x01 x06	PBX→Applikation
MakeCall	x08 x01	Applikation→PBX
ConferenceInitiation	x08 x02	Applikation→PBX
ConferenceCompletion	x08 x03	Applikation→PBX
TransferInitiation	x08 x04	Applikation→PBX
TransferCompletion	x08 x05	Applikation→PBX
RetrieveCall	x08 x06	Applikation→PBX
ReleaseConnection	x08 x0B	Applikation→PBX
StatusChange	x08 x0F	PBX→Applikation

Tab. 5-3: Meridian-Link – Protokollübersicht Nachrichtentypen

Header	xFF	x0A	x00	x1D	x01	x00	x2A	x08	x01	x00
Origination DN	x30	x08	x00	x08	x34	x37	x32	x30		
Destination DN	x31	x08	x00	x08	x34	x35	x30	x31		
Call Manner	x34	x03	x00							

Tab. 5-4: Meridian-Link – Aufbau einer Nachricht am Beispiel MakeCall

Tabelle 5-4 zeigt eine vollständige Nachricht vom Typ *MakeCall* mit den einzelnen Informationselementen. Neben dem Header-Element besteht diese aus Informationselementen für die DN des anrufenden (*Origination DN*) und des angerufenen Terminals (*Destination DN*). Das vierte Element (*Call Manner*) beschreibt die Art des Anrufs. Das erste Byte eines jeden Informationselementes spezifiziert den Elementtyp (*FF*, *30*, *31* und *34*), während das zweite die Elementlänge angibt. Erst ab dem dritten Byte werden die Nutzdaten kodiert. Im Falle des Informationselements *Origination DN* tragen die Bytes fünf bis acht die Telefonnummer des Anrufers. Diese besteht aus vier Ziffern, die als ASCII-Zeichen zu interpretieren sind. Die Bytes *34*, *37*, *32* und *30* entsprechen in dezimaler Schreibweise *52*, *55*, *50* und *48* und damit der Zeichenfolge *4720*. Das Informationselement *Destination DN* dieses Beispiels spezifiziert die Telefonnummer des angerufenen Teilnehmers mit *4501*. Diese Nachricht würde der PBX den Befehl geben einen Anruf zwischen den Teilnehmern mit den Nummern 4720 und 4501 zu initiieren.

Nachdem die Grundstruktur einer Nachricht aus dem Meridian-Link-Protokoll behandelt wurde, wird nun auf den spezifischen Aufbau der Nachrichtentypen eingegangen. Die sich nun anschließenden Kapitel erläutern daher deren Funktion und Zusammensetzung aus Informationselementen.

Information Element	Schlüssel
Header	xFF
Application Protocol	x01
Application ID	x04
Host Name	x05
Process ID	x06
Meridian Mail Name	x07
Machine ID	x08
Service ID List	x09
Service ID	x0A
Meridian 1 Machine Name	x0B
Password	x0C
Meridian 1 Customer Name	x0D
Link Type	x0E
Associated DN List	x0F
Association List	x10
Resource List	x11
Link Identifier	x1A
Link Status	x1C
Connection Status	x2E
Origination Address	x30
Destination Address	x31
Call Type	x32
Call Manner	x34
Event	x35
This Device DN	x36
This Device TN	x37
This Device Status	x38
Other Device DN	x39
Other Device TN	x3A
Origination TN	x3F
Destination TN	x40
Transferred Party DN	x4C
Transferred Party TN	x4D
Transferred Party Call ID	x53
Conferenced Party Call ID	x54
Enhanced Time Stamp	x5F
Result	x71
Cause	x78
Call ID	x96

Tab. 5-5: Meridian-Link – Protokollübersicht Information Elements

5.3.1.2 ApplicationRegisterRequest

Die Nachricht *ApplicationRegisterRequest* registriert eine Applikation beim Link-Modul. Das positive Ergebnis ist eine so genannte *Association*, wobei der Applikation eine entsprechende ID zugeteilt wird. Das Ergebnis der Anfrage wird von der PBX durch die Nachricht *ApplicationRegisterResponse* mitgeteilt (vgl. nächstes Kapitel). Das Header-Element dieser Nachricht trägt, aufgrund fehlender vorangegangener Registrierung, eine Association-ID von *00*. Die folgende Tabelle 5-6 zeigt die Zusammensetzung der Nachricht aus Informationselementen und deren Bedeutung.

Information Element	**Schlüssel**	**Bedeutung**
Header	xFF	Header, Nachrichtentyp x01 x01
Application Protocol	x01	verwendetes Protokollformat
Application ID	x04	ID der Anwendung auf PBX-Seite
Service ID	x0A	unbekannt, konstanter Wert x62
Host Name	x05	Hostname der PBX
Meridian 1 Machine Name	x0B	Machinen-ID der PBX
Meridian 1 Customer Name	x0D	unbekannt, konstanter Wert x00

Tab. 5-6: ApplicationRegisterRequest – IE-Struktur

Die Gültigkeitsdauer einer Association ist nicht an das Fortbestehen der TCP/IP-Verbindung angewiesen. Sollte diese einmal unterbrochen werden und später wieder zustande kommen, ohne dass die Applikation abgemeldet (vgl. Kapitel 5.3.1.4) oder das Link-Modul neu gestartet wurde, ist die Association-ID noch immer gültig.

5.3.1.3 ApplicationRegisterResponse

ApplicationRegisterResponse ist die Antwort der PBX auf die Nachricht ApplicationRegisterRequest. Es handelt sich dabei entweder um eine positive oder eine negative Antwort. Im Falle einer positiven Antwort, enthält das Header-Element die Association-ID, die zukünftig für Anfragen verwendet werden soll. Ist die Antwort negativ, jedoch die Association-ID verschieden von *x00*, handelt es sich um den Versuch einer wiederholten Registrierung bei noch gültiger Association. Die ID gibt dann diejenige der letzten Registrierung an. Tabelle 5-7 zeigt den Aufbau einer positiven und Tabelle 5-8 einer negativen Nachricht.

5.3.1.4 ApplicationReleaseRequest

Die Nachricht *ApplicationReleaseRequest* löst die Association zwischen Meridian-Link und Applikation. Im Erfolgsfall ist die Association beendet und die ID nicht mehr gültig. Um danach die Kommunikation wieder aufnehmen zu können, ist eine erneute Registrierung mit ApplicationRegisterRequest erforderlich. Tabelle 5-9 zeigt den Aufbau dieser Nachricht, die nur aus dem Header Element besteht.

Information Element	Schlüssel	Bedeutung
Header	xFF	Header, Nachrichtentyp x01 x02
Result	x71	Ergebnis des Requests, positiv

Tab. 5-7: ApplicationRegisterResponse – IE-Struktur einer positiven Antwort

Information Element	Schlüssel	Bedeutung
Header	xFF	Header, Nachrichtentyp x01 x02
Result	x71	Ergebnis des Requests, neagtiv
Cause	x78	Fehlercode, Codes unbekannt

Tab. 5-8: ApplicationRegisterResponse – IE-Struktur einer negativen Antwort

5.3.1.5 ApplicationReleaseResponse

ApplicationReleaseResponse ist die Antwort der PBX auf die Nachricht ApplicationRelease-Request. Es handelt sich dabei entweder um eine positive oder eine negative Antwort. Tabelle 5-10 zeigt den Aufbau einer positiven, Tabelle 5-11 den einer negativen Nachricht.

5.3.1.6 DNRegisterRequest

Eine CTI-Anwendung registriert mit der Nachricht *DNRegisterRequest* die zu steuernden Terminals. Erst wenn ein Terminal auf diese Weise registriert wurde, übermittelt das Link-Modul dessen Statusinformationen und akzeptiert Steuerungsbefehle. Dabei gibt es die Möglichkeit pauschal alle als AST konfigurierten Endgeräte für die Steuerung zu aktivieren (vgl. Tabelle 5-12) oder diese über ihre DN einzeln zu registrieren. Bei der Einzelregistrierung kann auch eine Liste mehrerer Geräte angegeben werden (vgl. Tabelle 5-13).

5.3.1.7 DNRegisterResponse

DNRegisterResponse ist die Antwort der PBX auf die Nachricht DNRegisterRequest. Es handelt sich dabei entweder um eine positive oder eine negative Antwort. Tabelle 5-14 zeigt den Aufbau einer positiven, Tabelle 5-15 den einer negativen Nachricht.

5.3.1.8 MakeCall

Mit *MakeCall* wird ein Anruf zwischen zwei Terminals initiiert. Dabei muss eines der beiden Terminals an der PBX des Meridian-Links angeschlossen sein. An das zweite Terminal sind

Information Element	Schlüssel	Bedeutung
Header	xFF	Header, Nachrichtentyp x01 x03

Tab. 5-9: ApplicationReleaseRequest – IE-Struktur

Information Element	Schlüssel	Bedeutung
Header	xFF	Header, Nachrichtentyp x01 x04
Result	x71	Ergebnis des Requests, positiv

Tab. 5-10: ApplicationReleaseResponse – IE-Struktur einer positiven Antwort

Information Element	Schlüssel	Bedeutung
Header	xFF	Header, Nachrichtentyp x01 x04
Result	x71	Ergebnis des Requests, negativ
Cause	x78	Fehlercode, Codes unbekannt

Tab. 5-11: ApplicationReleaseResponse – IE-Struktur einer negativen Antwort

Information Element	Schlüssel	Bedeutung
Header	xFF	Header, Nachrichtentyp x01 x05
Associated DN List	x0F	alle AST-Terminals, Wert x7F

Tab. 5-12: DNRegisterRequest – IE-Struktur einer Pauschal-Registrierung

Information Element	Schlüssel	Bedeutung
Header	xFF	Header, Nachrichtentyp x01 x05
Associated DN List	x0F	DN-Liste, AST-konfigurierte Terminals

Tab. 5-13: DNRegisterRequest – IE-Struktur einer Einzel-/Batchregistrierung

Information Element	Schlüssel	Bedeutung
Header	xFF	Header, Nachrichtentyp x01 x06
Result	x71	Ergebnis des Requests, positiv

Tab. 5-14: DNRegisterResponse – IE-Struktur einer positiven Antwort

Information Element	Schlüssel	Bedeutung
Header	xFF	Header, Nachrichtentyp x01 x06
Result	x71	Ergebnis des Requests, negativ
Cause	x78	Fehlercode, Codes unbekannt

Tab. 5-15: DNRegisterResponse – IE-Struktur einer negativen Antwort

Information Element	Schlüssel	Bedeutung
Header	xFF	Header, Nachrichtentyp x08 x01
Origination Address	x30	DN, anrufender Teilnehmer
Destination Address	x31	DN, angerufener Teilnehmer
Call Manner	x34	unbekannt, mögliche Werte x00, x01, x03

Tab. 5-16: MakeCall – IE-Struktur

Information Element	Schlüssel	Bedeutung
Header	xFF	Header, Nachrichtentyp x08 x02
Origination Address	x30	DN, anrufender Teilnehmer
Destination Address	x31	DN, angerufener Teilnehmer

Tab. 5-17: ConferenceInitiation – IE-Struktur

hingegen keine besonderen Anforderungen gestellt. Neben den beiden Teilnehmern ist noch der Parameter *Call Manner* mit den möglichen Werten

- x00 für *polite*,
- x01 für *belligerent* und
- x03 für *semipolite*

anzugeben. Deren einzelnen Auswirkungen beim Aufbau von Telefongesprächen ist aufgrund der nicht verfügbaren Spezifikation unbekannt. Tabelle 5-16 zeigt den Aufbau von MakeCall.

5.3.1.9 ConferenceInitiation

Mit *ConferenceInitiation* wird ein so genannter *Consultation Call* gestartet. In diesem Fall handelt es sich um einen Call, der in Beziehung zu einem bestehenden *Main Call* steht. Ist der Aufbau der Verbindung erfolgreich und nimmt der Teilnehmer das Gespräch an, können mit dem Befehl *ConferenceCompletion* (vgl. nächstes Kapitel) beide Gespräche zu einer Konferenz zusammengeschaltet werden. Dabei werden die Teilnehmer des Consultation Calls dem Main Call hinzugefügt. Der Consultation Call wird beendet. Tabelle 5-17 zeigt den Aufbau dieses Befehls.

5.3.1.10 ConferenceCompletion

Mit dieser Nachricht wird ein Consultation Call beendet und dessen Teilnehmer dem Main Call des Initiators hinzugefügt. Tabelle 5-18 zeigt den Aufbau dieser Nachricht.

Information Element	Schlüssel	Bedeutung
Header	xFF	Header, Nachrichtentyp x08 x03
Origination Address	x30	DN, Initiator eines Consultation Calls

Tab. 5-18: ConferenceCompletion – IE-Struktur

Information Element	Schlüssel	Bedeutung
Header	xFF	Header, Nachrichtentyp x08 x04
Origination Address	x30	DN, anrufender Teilnehmer
Destination Address	x31	DN, angerufener Teilnehmer

Tab. 5-19: TransferInitiation – IE-Struktur

5.3.1.11 TransferInitiation

Mit *TransferInitiation* wird ein Consultation Call mit dem Ziel der Anrufweiterleitung gestartet. Dieser Consultation Call steht analog zum Befehl ConferenceInitiation durch einen gemeinsamen Teilnehmer in Beziehung zu einem bestehenden Main Call. Ist der Aufbau der Verbindung erfolgreich, kann mit dem Befehl *TransferCompletion* (vgl. nächstes Kapitel) der angerufene Teilnehmer des Consultation Calls mit dem wartenden Teilnehmer des Main Calls verbunden werden. Der Consultation Call wird beendet und der Initiator des Transfers verlässt beide Gespräche, wobei dessen Apparat aufgelegt wird. Tabelle 5-19 zeigt den Befehlsaufbau.

5.3.1.12 TransferCompletion

Der Befehl *TransferCompletion* schließt die mit TransferInitiation vorbereitete Anrufweiterleitung ab. Der Initiator des Transfers verlässt beide Calls, während der über den Consultation Call hinzukommende Teilnehmer dessen Platz im Main Call einnimmt. Tabelle 5-20 zeigt die Informationselemente dieses Befehls.

5.3.1.13 RetrieveCall

RetrieveCall veranlasst das Zurückkehren zum Main Call. Es gibt hierfür zwei Ausgangssituationen. Das betreffende Terminal ist an einem Consultation Call beteiligt oder der Main Call wird lediglich gehalten, da der Consultation Call beendet wurde oder nicht zustande gekommen ist. Das Kommando RetrieveCall bewirkt in beiden Fällen das Zurückholen des

Information Element	Schlüssel	Bedeutung
Header	xFF	Header, Nachrichtentyp x08 x05
Origination Address	x30	DN, Initiator eines Consultation Calls

Tab. 5-20: TransferCompletion – IE-Struktur

Information Element	Schlüssel	Bedeutung
Header	xFF	Header, Nachrichtentyp x08 x06
Origination Address	x30	DN, Terminal mit Main Call *on hold*

Tab. 5-21: RetrieveCall – IE-Struktur

Information Element	Schlüssel	Bedeutung
Header	xFF	Header, Nachrichtentyp x08 x0B
Origination Address	x30	DN, an einem Call beteiligt

Tab. 5-22: ReleaseConnection – IE-Struktur

Main Calls und, falls vorhanden, das Beenden[11] eines aktiven Consultation Calls. Sind beide Situationen nicht zutreffend, ist das Kommando ergebnislos. Tabelle 5-21 zeigt den Nachrichtenaufbau.

5.3.1.14 ReleaseConnection

Mit dieser Nachricht wird die Teilnahme am aktiven Call beendet. Nimmt das Terminal als Initiator an einem Consultation Call teil, wird dieser dabei beendet, während der Main Call weiterhin gehalten wird. Mit RetrieveCall kann dieser wieder aktiviert werden. Bei nur einem Gespräch ist ReleaseConnection gleichbedeutend mit dem Auflegen des Apparats. Die Informationselemente dieses Nachrichtentyps sind Tabelle 5-22 zu entnehmen.

5.3.1.15 StatusChange

Neben den Nachrichten zum Auf- und Abbau der Association und der Registrierung von Terminals, haben die übrigen Kommandos zur Implementierung der CTI-Features wie Anruf- und Konferenzaufbau keine expliziten Antwortnachrichten als Gegenstück. Erfolg oder Fehlschlag zeigen sich bei diesen Kommandos durch Eintreten oder Ausbleiben bestimmter Statusänderungen im Telefoniesystem. Diese Statusänderungen registrierter AST-Terminals werden über die Nachricht *StatusChange* einer CTI-Anwendung mitgeteilt. Eine Nachricht bezieht sich immer auf den Zustand eines Terminals und spezifiziert deshalb immer dessen DN und TN sowie dessen neuen Status. Tabelle 5-24 zeigt eine Auflistung der möglichen Status. Die Nachricht StatusChange kann darüber hinaus

- die DN und TN eines anderen Terminals mit Beziehung zur Statusänderung,
- die Call-ID,
- die Art des Anrufs und
- einen Zeitindex

[11] Implizite Ausführung von *ReleaseConnection*

Information Element	Schlüssel	Bedeutung
Header	xFF	Header, Nachrichtentyp x08 x0F
Call Type	x32	DN, Call-Typ
This Device DN	x36	DN, betroffenes Terminal
This Device TN	x37	TN, betroffenes Terminal
This Device Status	x38	Status (vgl. Tabelle 5-24)
Other Device DN	x39	DN, in Beziehung stehendes Terminal
Other Device TN	x3A	TN, in Beziehung stehendes Terminal
Call ID	x96	Call-ID, fortlaufende Nummer
Enhanced Time Stamp	x5F	Zeitindex

Tab. 5-23: StatusChange – mögliche Struktur

enthalten. Tabelle 5-23 zeigt alle Informationselemente, die im Rahmen dieser Nachricht übermittelt werden können. Da sich eine Nachricht abhängig vom Status aus unterschiedlichen, sinnvollen Informationselementen zusammensetzt, gibt diese Tabelle nur eine Möglichkeit wieder, die bei Meldungen mit dem Status *Unringing* auftritt.

Es werden nur Statusänderungen übermittelt. Es ist kein Befehl für die explizite Statusabfrage bestimmter Geräte bekannt.

5.3.2 Alcatel OmniPCX 4400 - CSTA Phase II

Die Protokollimplementierung der PBX *Alcatel OmniPCX 4400* befindet sich auf dem Versionsstand der *CSTA Phase II* und entspricht damit nicht mehr dem aktuellen Stand. Die nun folgende Einführung bezieht sich deshalb ebenfalls auf die Phase II der CSTA-Spezifikation.

5.3.2.1 Services

CSTA beschreibt eine Architektur für die Interoperabilität von Komponenten der Informations- und Telekommunikationstechnik, unabhängig von deren technischen Umsetzung. Im Dokument [ECMAa94] ist dies folgendermaßen formuliert:

The objective of CSTA Architecture is to define the inter-working mechanisms among Computing, Switching and Special Resource Functions independently from their physical implementations.

CSTA beschreibt die Funktionalität in Form von Diensten, die die Interaktion von Telefonie- und IT-Hardware ermöglichen. Diese Dienste werden in die Kategorien

- Switching Function Services,
- Status Reporting Services,
- Computing Function Services,

Status	Schlüssel	Bedeutung
UNKNOWN	0x00	unbekannt
ONHOOK	0x01	aufgelegt
OFFHOOK	0x02	abgehoben
RINGING	0x03	klingelt
ACTIVE	0x04	Call aktiv
DISCONNECT	0x05	Call verlassen
UNRINGING	0x06	Klingeln beendet
RETRIEVE	0x07	Main Call zurückgeholt
TRANSFERCOMPLETEDTO	0x09	Transfer abgeschlossen
CONFERENCECOMPLETEDTO	0x0A	Konferenzaufbau abgeschlossen
CONFERENCETOSIMPLE	0x0B	Call mit zwei verbleibenden Teilnehmern
HOLD	0x0D	Call gehalten
TRANSFERINITIATION	0x0E	Consultation Call für Transfer begonnen
CONFERENCEINITIATION	0x0F	Consultation Call für Konferenz begonnen

Tab. 5-24: StatusChange - mögliche Device-Zustände

- Bi-directional Services,
- Input/Output (I/O) Services und
- Voice Unit Services

unterteilt. An dieser Kategorisierung lässt sich bereits erkennen, dass CSTA keine feste Client/Server-Rollenverteilung propagiert. Mit den *Switching Function Services* werden alle Funktionen zusammengefasst, die die Telefoniehardware einer CTI-Softwarelösung anbietet. Dazu gehören Features wie *Answer Call Service*, *Make Call Service* oder *Conference Call Service*. Andererseits enthält die Kategorie *Computing Function Services* Dienste, die die CTI-Software der PBX bereitstellt. Hierbei handelt es sich für gewöhnlich, mit Features wie *Route Select Service* oder *Re-Route Service*, um Funktionen zur Beeinflussung des Routingverhaltens. Soll ein Gespräch in das öffentliche Telefonnetz geroutet werden und sind dabei mehrere Pfade möglich, so kann die CTI-Anwendung beispielsweise unter Gesichtspunkten der Lastverteilung oder Kostenminimierung den optimalen Pfad ermitteln. Die PBX tritt dabei der Anwendung gegenüber als Client auf, indem sie den optimalen Pfad abfragt.

Unter *Bi-directional Services* sind Dienste zusammengefasst, die von beiden Seiten in Anspruch genommen werden können. Dazu gehört beispielsweise der Austausch von Systemstatusmeldungen. Die Kategorie *Status Reporting Services* fasst Dienste zum Melden von Events zusammen, die Statusänderungen in der Telefoniehardware kommunizieren. So können Call-relevante Events wie das Auflegen oder Klingeln eines Telefonapparates mitgeteilt werden. Unter *Input/Output Services* sind Mechanismen zur Nutzung zusätzlicher Datenübertragungswege zusammengefasst, die manche Telefoniesysteme bereitstellen. Dabei kann es sich beispielsweise um die Übertragung von Textnachrichten oder Video-Streams

(Bildtelefonie) handeln. Die letzte Kategorie *Voice Unit Services* beschäftigt sich mit der Nutzung von Voice-Mailboxen und IVR-Systemen.

5.3.2.2 Protokoll

Der zweite Teil der Spezifikation befasst sich mit dem Aufbau des Applikationsprotokolls. CSTA verwendet dabei einen Protokollstack von

- TCP,
- IP,
- *ACSE*[12] und
- *ROSE*[13].

Die Protokolle ROSE und ACSE sind durch die ITU-T[14] spezifiziert. ACSE wird als Protokoll zum Aufbau der Association zwischen PBX und CTI-Applikation und zur Authentifizierung verwendet. Dabei werden die unterstützten CSTA-Protokollversionen und -Services ausgetauscht, um sich auf einen gemeinsamen Nenner zu verständigen.

ROSE definiert ein RPC-Protokoll für die Implementierung von Request/Response-basierten Applikationsprotokollen, worauf CSTA direkt aufbaut. So sind die CSTA-Messages aus so genannten *PDUs*[15] aufgebaut. Diese Protokolldatenelemente sind vergleichbar mit den Information Elements aus Kapitel 5.3.1ff. In der CSTA-Protokollspezifikation [ECMAb94] sind die ROSE-konformen PDUs in der *ASN.1*[16]-Notation definiert. Diese Notation bietet eine Möglichkeit Protokolle in mathematisch eindeutiger Form zu definieren. ASN.1 ist damit einer Dokumentendefinitionssprache wie XML sehr ähnlich. Mit dem Unterschied, dass mit den

- Basic Encoding Rules (BER),
- Canonical Encoding Rules (CER),
- Distinguished Encoding Rules (DER) und
- Packed Encoding Rules (PER)

plattformunabhängige Regeln zur Codierung existieren. Darüber hinaus existieren Regeln zur Codierung in XML (ASN.1-zu-XML-Schema-Mapping). Dies bedeutet, dass die letztendliche Binärcodierung der ASN.1-PDUs durch die Implementierung auf Basis von bestimmten Regeln erfolgt.

[12] Association Control Service Element, ITU-T Recommendation X.217 (ISO 8649) und X.227 (ISO 8650)
[13] Remote Operation Service Element
[14] International Telecommunication Union - Telecommunication Standardization Sector
[15] Protocol Data Units
[16] Abstract Syntax Notation One – ITU-T Recommendations X.680 - X.683

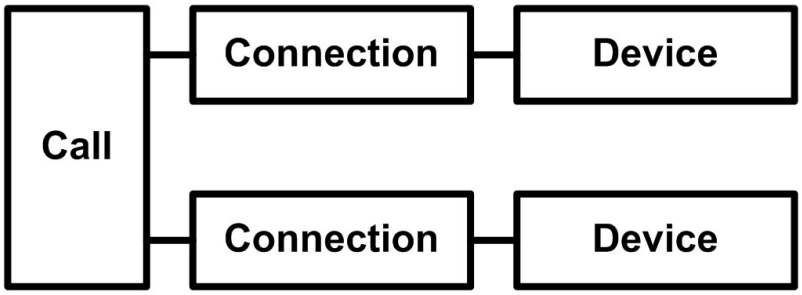

Abb. 5-1: CSTA – Call mit zwei Teilnehmern

5.3.2.3 Call-Modell

Dem CSTA-Service-Modell liegt ein Zustandsmodell mit den Elementen

- Device,
- Call und
- Connection

zugrunde. Ein *Device* repräsentiert ein Gerät oder System, das dem Benutzer den Zugriff auf Telekommunikationsdienste ermöglicht. Ein Device ist im Sinne von CSTA steuer- und überwachbar. Es kann sich dabei sowohl um physische Geräte wie Knöpfe, Leitungen, Trunks[17] oder Telefone handeln als auch um virtuelle wie Geräte- oder ACD-Gruppen. Jedes Device ist CSTA-weit über einen eindeutigen *CSTA Device Identifier* identifizierbar.

Ein *Call* stellt eine logische Verbindung von mehreren Devices dar. Auch Calls sind überwachbar und manipulierbar und verfügen über einen eindeutigen Identifier. Die Verknüpfung von Call und Device wird über das *Connection*-Objekt erreicht. Die Abbildung eines einfachen Telefongesprächs erfolgt demnach mittels eines Call-Objektes und zweier Device-Objekte, die jeweils über ein eigenes Connection-Object mit dem Call-Objekt verbunden sind (vgl. Abbildung 5-1). Der Verbindungsstatus wird über den Zustand des Connection-Objektes wiedergegeben. Die Abbildung 5-2 zeigt das zugehörige Zustandsmodell und die möglichen Statusübergänge.

5.3.2.4 CSTA-Services zur Umsetzung funktionaler Anforderungen

In Kapitel 5.3.2.1 wurden die Services als funktionale Bestandteile von CSTA beschrieben. Es folgt nun eine Auflistung konkreter Services, die zur Umsetzung der in Kapitel 4.1 aufgestellten Anforderungen in Frage kommen.

Zuvor soll noch angemerkt werden, dass auch bei CSTA die Statusüberwachung von Devices (Terminals) aktiviert werden muss. Ähnlich dem Kommando *DNRegisterRequest* (vgl. Kapitel 5.3.1.6) steht CSTA mit *Monitor Start Service* ein vergleichbares Kommando zur

[17] Physischer oder logischer Verbindungskanal zwischen vermittelnden Systemen (PBX)

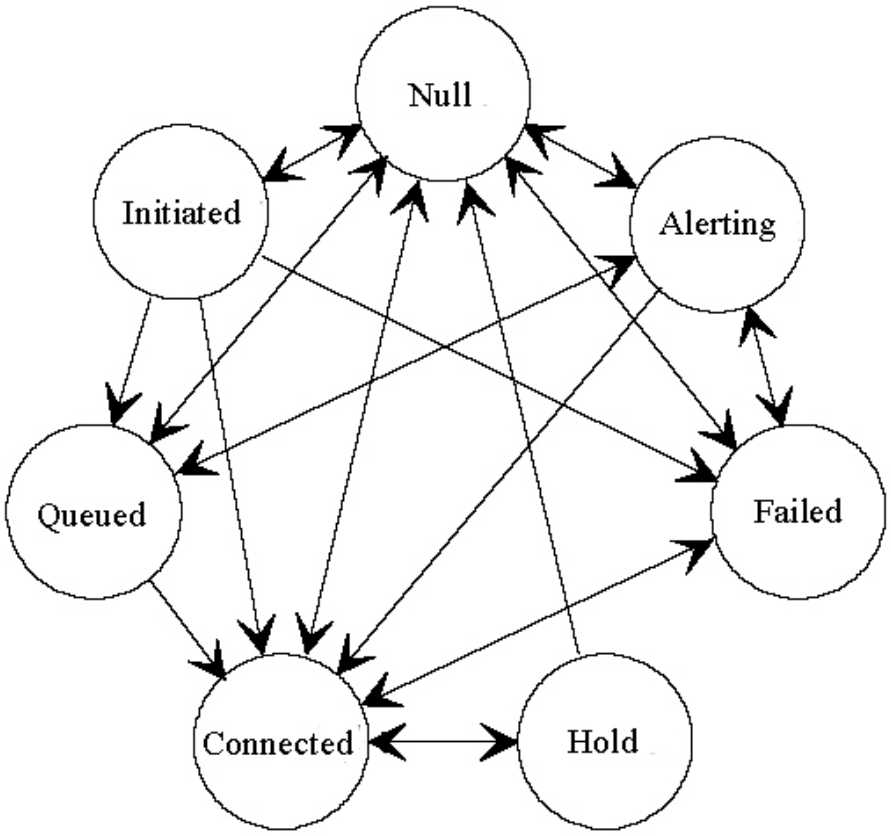

Abb. 5-2: CSTA – Zustandsdiagramm des Connection-Objekts (nach [ECMAa94])

Verfügung. CSTA verwendet für die Überwachung der Devices so genannte Monitore, die über Filter konfiguriert, auf bestimmte Ereignisse warten und bei deren Eintreten den Client, der den Monitor gesetzt hat, mittels *Event Reports* verständigen. Dieser Lösungsweg entspricht dem Listener-Konzept aus der objektorientierten Programmierung. Darüber hinaus verfügt CSTA über die Möglichkeit mit *Monitor Stop Service* gesetzte Monitore zurückzunehmen und mit *Change Monitor Filter Service* die Ereignisfilter bestehender Monitore zu verändern. Die übermittelten Event Reports unterteilen sich in die Kategorien

- Agent State,
- Call,
- Feature,
- Maintenance,
- Private und
- Voice Unit Event Reports.

In Anbetracht des derzeitigen Anforderungsprofils sind momentan nur die *Call Event Reports* relevant. Tabelle 5-25 zeigt eine Auflistung der Events dieser Kategorie und liefert eine kurze Erklärung. Für die Implementierung der Telekommunikationsfunktionen stehen folgende CSTA-Services zur Verfügung:

(1) *Make Call Service* startet einen Anruf zwischen zwei Devices und liefert die ID des Connection-Objekts des verursachenden Devices zurück.

(2) *Consultation Call Service* ist nur bei einem aktiven Call anwendbar, erzeugt einen Consultation Call und hält das bestehende Gespräch. Das betreffende Device verfügt damit über zwei Connection-Objekte im Status `Hold` und `Established`.

(3) *Conference Call Service* fügt bestehenden Consultation und Main Call unter Einbeziehung aller Teilnehmer zu einem gemeinsamen Call zusammen. Das resultierende Gespräch verfügt dann über mindestens drei Teilnehmer.

(4) *Transfer Call Service* vermittelt die Teilnehmer eines Main Calls und die des zugehörigen Consultation Calls unter Ausschluss des Terminals, das die Zusammenführung verursacht und beide Calls verlässt.

(5) *Retrieve Call Service* holt ein gehaltenes Gespräch zurück, wobei das betreffende Connection-Objekt vom Status `Hold` auf `Connected` wechselt.

(6) *Clear Connection Service* löst die Verbindung eines Devices mit einem Call. Der Teilnehmer verlässt das Gespräch.

(7) *Reconnect Call Service* ist eine funktionale Verschmelzung aus Clear Connection Service und Retrieve Call Service, indem die Verbindung zu einem bestehenden Consultation Call unterbrochen und ein gehaltenes Gespräch zurückgeholt wird.

Unter anderem existieren noch Services für

- das explizite Abfragen des Device-Status,
- Single-Step-Transfers[18] und Konferenzen sowie
- *Predictive Dialing*[19].

[18] *Single Step* nimmt bezug auf das Auslassen des Zwischenschrittes eines Consultation Calls beim Aufbau von Konferenzen oder bei der Anrufweiterleitung.

[19] Besondere Art der Anrufvermittlung bei der zuerst die Verbindung zum Empfänger aufgebaut wird. Erst bei Erfolg wird der Anrufer benachrichtigt.

Call Event Report	**Bedeutung**
Call Cleared	Call aufgelöst, alle Connections entfernt
Conferenced	zwei Calls zu Konferenz vereinigt
Connection Cleared	Connection des betroffenen Terminals entfernt
Delivered	Call hat Device erreicht, dieses wird benachrichtigt
Diverted	Call wurde vom betroffenen Device abgelenkt
Established	Call aufgebaut, Device hat Call entgegengenommen
Failed	Call-Aufbau fehlgeschlagen oder Connection im Status Failed
Held	Call gehalten, Connection im Status Hold
Network Reached	Call hat Netzwerkgrenze erreicht (bspw. Öffentliches Telefonnetz), eventuell eingeschränkte Statusüberwachung
Originated	Call vorbereitet, Connection des verusachenden Devices aufgebaut
Queued	Call wurde in eine Warteschlange am betroffenen Device eingereiht
Retrieved	gehaltener Call zurückgeholt
Service Initiated	Device abgehoben und erhält Freizeichen oder andersartiger Dienst wird in Anspruch genommen
Transferred	Call wurde weiterverbunden

Tab. 5-25: CSTA - Call Event Reports Übersicht

6 Eingesetzte Technologien

Dieses Kapitel liefert einen Überblick über die Technologien, die bei der Lösung der Projektaufgabe eingesetzt worden sind. Damit kann das Softwaredesign in Kapitel 7 darauf aufbauend erläutert werden. Zunächst wird die Java-Technologie im Allgemeinen und JNDI und RMI im Besonderen beschrieben. Es folgt eine knappe Beschreibung der eingesetzten Entwicklungsumgebung Eclipse, bis mit JTAPI die Kernkomponente von *webDial* ausführlich erläutert wird.

6.1 Java 5.0

Bei Java handelt es sich um eine Programmiersprache der dritten Generation, die im Gegensatz zu C++ von Beginn an als objektorientierte Sprache entwickelt wurde. Deshalb ist eine Vermischung prozeduraler und objektorientierter Programmierung nicht möglich, wenngleich die Syntax von Java sich an der von C++ anlehnt. Trotz der syntaktischen Ähnlichkeiten fand eine Bereinigung der Sprache statt. So wurden Sprachkonstrukte wie `goto`, `struct` oder `union` in Java nicht übernommen.

Ein wesentlicher Unterschied zwischen beiden Sprachen liegt in der Art und Weise der Erzeugung ausführbaren Maschinencodes. Während man im Falle von C++ den Sourcecode für jede zu unterstützende Hardwareplattform kompilieren muss[1], erstellt der Java Compiler einen Bytecode, der aus plattformunabhängigen Befehlen besteht. Dieser Bytecode wird zur Laufzeit von der *Java Virtual Machine (JVM)* interpretiert und in Maschinencode übersetzt. Dieser Vorgang wird als *Just-in-Time-Compilation (JIT)* bezeichnet. Der sich daraus ergebende Performanceverlust kann durch intelligente Optimierung für die eingesetzte Hardware wieder aufgeholt werden.

Bei der JVM handelt es sich um eine plattformabhängige Ausführungsumgebung, die in Form einer Vermittlungsinstanz zwischen Java-Programm und Betriebssystemebene auftritt. Jede Nutzung von Systemressourcen wird durch die JVM autorisiert und durchgesetzt. Die Schnittstelle, die durch die JVM Programmen zur Nutzung von Systemressourcen angeboten wird, stellt einen gemeinsamen, abstrahierten Nenner unterschiedlicher Betriebssysteme wie Microsoft Windows, Linux, Solaris oder Mac OS dar und erreicht damit Plattformunabhängigkeit. Sofern über das *Java Native Interface (JNI)* bei der Softwareentwicklung keine plattformabhängigen Codeelemente eingebunden wurden, ist der Bytecode eines Java-Programms auf jedem System ausführbar, für das eine Version der JVM existiert. Sun Microsystems verteilt mit der *Java Runtime Environment (JRE)* ein Softwarepaket, das neben

[1] Dem Verfasser ist durchaus bekannt, dass für C/C++ Interpreter existieren, jedoch deren Verwendung nicht als Normalfall anzusehen ist.

der *J2SE-API* (vgl. Kapitel 6.1.2) auch eine Implementierung der JVM enthält. Bei der JVM handelt es sich um eine standardisierte Softwarekomponente, sodass neben Sun Microsystems andere Hersteller wie IBM oder Microsoft[2] eigene Implementierungen der JVM entwickelt haben. Neben kommerziellen JVMs existieren mit Kaffe, SableVM und SuperWaba ebenfalls Open-Source-Projekte.

Neben dem nicht vorhandenen Präprozessor ist das Fehlen des Linkers, der kompilierte Codeteile zu einem ausführbaren Programm zusammenfügt, ein weiteres Unterscheidungsmerkmal gegenüber C++. Der durch den Java Compiler erzeugte Bytecode wird in `class`-Dateien[3] gespeichert. Diese sind in einer Verzeichnisstruktur abgelegt, die der Paket- und Klassenstruktur des Programms entspricht. So kann die JVM bei Bedarf zur Laufzeit die benötigten Codeelemente[4] laden und interpretieren. Alternativ können die `class`-Dateien in einer `jar`-Datei in komprimierter Form abgelegt werden. Dabei handelt es sich um ein Archiv im Zip-Format, das die compilierten Java-Dateien in ihrer Paket- und Verzeichnisstruktur enthält.

Im Gegensatz zu den Sprachen Smalltalk oder Ruby werden nicht alle Daten als Objekte abgebildet. Aus Gründen der Performance stellen einfache Datentypen wie `int`, `bool` oder `float` in Java keine Objekte dar. Die daraus entstandene Problematik der Gleichbehandlung von Objekten und Primitiven wurde mit Wrapper-Klassen begegnet, die einfache Basisdatentypen aufnehmen und kapseln. Die sich daraus ergebende Flut aus expliziten Typumwandlungen wurde im Java Release 5.0 durch die Einführung von Autoboxing und -unboxing[5] entschärft.

Eine direkte Manipulation des Programmspeichers ist in Java nicht möglich, da zu Gunsten geringerer Fehleranfälligkeit völlig auf das Pointerkonzept verzichtet wurde. Die Speicheradressen aller Objekte werden in Form von nicht manipulierbaren Referenzen gehalten. Ein weiterer Unterschied ergibt sich bei der Freigabe von nicht mehr gebrauchtem Speicher. Diese erfolgt nicht mehr manuell, sondern automatisch über den so genannten *Garbage Collector (GC)*. Dabei handelt es sich um einen Thread, der bei jeder Instanz der Laufzeitumgebung zu Beginn automatisch gestartet wird und zyklisch den Speicher nach Objekten durchsucht, die in den aktiven Programmteilen nicht mehr referenziert werden. Gefundene Speicherfragmente werden dann gelöscht und stehen wieder zur Verfügung. Das aus C/C++ bekannte Löschen dynamisch angeforderten Speichers mit dem Befehl `delete` gehört damit der Vergangenheit an. Allerdings kann die Problematik nicht völlig entschärft werden. Soll ein Objekt oder eine ganze Objekthierarchie vom GC aufgeräumt werden, muss dafür gesorgt sein, dass keine einzige Referenz mehr auf Elemente dieses Speicherbereichs verweist.

Unter Berücksichtigung all dieser Eigenschaften, lässt sich Java abschließend als wegweisende und zukunftsträchtige Technologie charakterisieren, deren Weiterentwicklung sich an den Bedürfnissen der Wirtschaft orientiert. Hierfür trägt der offene Entwicklungsprozess bei, der mit dem *Java Community Process (JCP)* von Sun ins Leben gerufen wurde. Dabei können

[2] Nach einem Rechtsstreit mit Sun wegen Bruch der Spezifikation und daraus resultierender Inkompatibilität zu anderen JVMs wurden die Weiterentwicklung und später der Support eingestellt.

[3] Dateien mit der Endung `.class`

[4] Da Java nur Klassen und Pakete kennt, enthält eine `class`-Datei immer mindestens eine Klasse, aber höchstens eine mit Sichtbarkeit `public`.

[5] Das Einpacken in oder Auspacken aus Wrapper-Klassen wird als Boxing beziehungsweise Unboxing bezeichnet.

Paket	Beschreibung
java.awt	Abstract Windowing Toolkit zur Erstellung graphischer Oberflächen
java.applet	Paket für das Erstellen von Applets
java.io	Paket für Input/Output-Aufgaben
java.lang	elementare Klassen wie `Object` oder `System`
java.net	Paket für die Netzwerkprogrammierung
java.util	häufig benötigte Sprachkonstrukten wie beispielsweise `Date`

Tab. 6-1: Klassenbibliothek von Java 1.0 (nach [CHAN05])

Mitglieder dieses Standardisierungsgremiums, das sich aus Vertretern verschiedener Unternehmen zusammensetzt, auf die weitere Entwicklung mittels so genannter *Java Specification Request (JSR)* Einfluss nehmen. Diese werden dann im Rahmen eines vordefinierten Verfahrens im Gremium diskutiert, angepasst und abschließend entweder angenommen oder verworfen.

6.1.1 Geschichte

Die anfängliche Absicht hinter der Entwicklung, die später zu Java führen sollte, war nicht die Erfindung einer neuen Programmiersprache. Denn das ursprünglich 13-köpfige Entwicklerteam von Sun hatte eigentlich eine neue Betriebssystemumgebung für die vernetzte Steuerung von Haushaltsgeräten im Bereich Home Entertainment zum Ziel. Es entstand die Umgebung Oak, die bereits das Konzept einer Virtual Machine beinhaltete. Entgegen der ursprünglichen Zielsetzung wurde die Entwicklung in Richtung Internet und Vernetzung forciert, um das Laden und Ausführen von Anwendungen über das Internet in Form von *Applets* zu ermöglichen. Der entscheidende Wendepunkt war die 1995 beschlossene Integration in den damals führenden Web-Browser von Netscape. Ein Jahr später erschien die Version 1.0 von Java inklusive einer kostenlosen Entwicklungsumgebung – dem *Java Development Kit (JDK)*. Es setzte ein großer Erfolg ein, der in der weiteren Entwicklung zu Erkennen ist. Es folgten die Versionen 1.1, 1.2, 1.3, 1.4 und schließlich die aktuelle Version 5.0.

Die Version 1.0 verfügte als erste Version der Java-Umgebung über eine, im Vergleich zum heutigen Umfang, überschaubare Klassenbibliothek. Tabelle 6-1 zeigt die damals verfügbaren Pakete.

1997 wurde das JDK 1.1 veröffentlicht. Dieses enthielt Verbesserungen im Bereich der Performance von AWT. Weitere Neuerungen waren RMI, JDBC, Objektserialisierung, JavaBeans und das Jar-Archiv, für effizienteres Laden der Java class-Dateien.

Im Jahr 1998 erschien Version 1.2. Von nun an wurde Java als Java 2 bezeichnet und die Entwicklungsumgebung als *Java 2 Standard Edition SDK (J2SE SDK)*. Die ursprüngliche Versionierung wurde parallel fortgeführt. Eine wesentliche Neuerung brachte die neue Swing-Bibliothek, die das Abstract Windowing Toolkit als API zur GUI-Entwicklung ablöste und für eine verbesserte Performance sorgte. Zudem wurde RMI durch die Integration von *CORBA*[6]

[6] Common Request Broker Architecture

ergänzt.

Das J2SE SDK 1.3 brachte 2000 im Wesentlichen Verbesserungen des Perfomanceverhaltens und die Integration der JNDI-API. Die *Java 2 Platform, Standard Edition (J2SE)* wurde durch die *Java 2 Platform, Enterprise Edition (J2EE)* um Elemente zur Entwicklung von Unternehmenslösungen mit hohen Ansprüchen an Vernetzung und Business-Integration erweitert. Das J2EE SDK ergänzt nun das J2SE SDK unter anderem um APIs für die Entwicklung von *Enterprise Java Beans*, *Servlets* und *Java Server Pages*. Tragende Komponenten der J2EE-Spezifikation sind Technologien wie Applikationsserver sowie JSP- und Servlet-Container.

Erweiterungen in den Bereichen XML, Kryptographie, Sicherheit und die Integration Regulärer Ausdrücke wurden im Jahr 2002 durch die Version 1.4 des J2SE SDK eingeführt. Auf Sprachebene wurden die Assertions und im Bereich des Deployments das Tool *Java Web Start*[7] hinzugefügt.

Die aktuelle Version Java 2 SE SDK 5.0 erschien 2004. Der Versionssprung von 1.4 auf 5.0 verfolgt dabei die Absicht, auf die zehnjährige Reifephase hinzuweisen und sich vom vermeintlichen Makel einer niedrigen Versionsnummer zu distanzieren. Mit dieser Version änderte sich relativ viel an der Sprache selbst. Zu den neuen Sprachelementen gehören

- Generische Datentypen (Templates oder parametrisierbare Klassen),
- `for`-Schleifen mit `foreach`-Charakter,
- Autoboxing und -unboxing,
- Enumerations,
- variable Parameterübergabe bei Methoden,
- statische Imports und
- Metadaten (sinngemäß aus [CHAN05]).

Wenngleich die Bedeutung der neuen Sprachelemente überwiegt, wurden dennoch zahlreiche Änderungen an den Paketen der Klassenbibliothek vorgenommen. So wurde im Paket `java.util` beispielsweise das Collections-Framework an die neuen generischen Typen angepasst, während mit dem Paket `java.util.concurrent` neue Möglichkeiten der Thread-Steuerung und -Synchronisation eingeführt wurden. Weitere Änderungen erfolgten in den Bereichen XML, RMI (SSL/TLS Socket Factory-Klassen) und JDBC (RowSets).

6.1.2 Java 2 Standard Edition Software Development Kit 5.0

Das *Java 2 Software Development Kit (JDK)* wird von Sun Microsystems zum Download[8] angeboten und ist frei von Lizenzgebühren. Es handelt sich dabei im Wesentlichen um die Java Runtime Environment mit einer Reihe zusätzlicher Tools zur Softwareentwicklung wie

[7] Deployment-Tool zur Verteilung von Stand-Alone-Applikationen über des Netzwerk
[8] http://java.sun.com/j2se/1.5.0/download.jsp

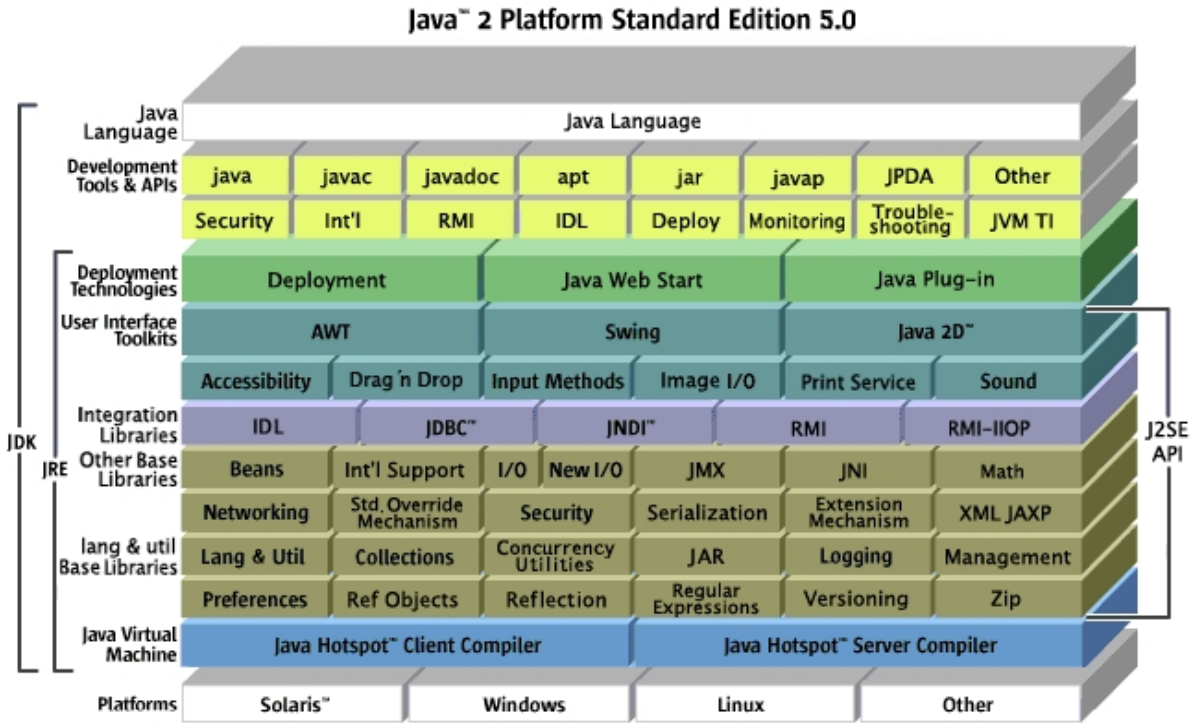

Abb. 6-1: Struktur der Java 2 Platform Standard Edition Version 5.0 [J2SE04b@]

beispielswiese dem Java-Compiler `javac`. Zudem enthält das JDK den Quellcode der J2SE-API. Die Abbildung 6-1 zeigt schematisch den Aufbau der Java-Platform.

Aus der umfangreichen Java-Plattform sollen im Folgenden für die Implementierng von *webDial* zwei relevante Komponenten näher betrachtet werden - die Remote Method Invocation (RMI) und das Java Naming Directory Interface (JNDI).

6.1.2.1 JNDI - Java Naming Directory Interface

Bei JNDI handelt es sich um eine API, die den einheitlichen Zugriff auf Namens- und Verzeichnisdienste ermöglicht und dabei Implementierungsdetails der jeweiligen Lösung verbirgt. Namens- und Verzeichnisdienste erleichtern das Auffinden von Daten und Information verschiedenster Art. Ein Namensdienst ermöglicht das Finden von Objekten über ihren Namen. Ein bekanntes und häufig anzutreffendes Beispiel dafür ist das Domain Name System (DNS), das IP-Adressen über Domain-Namen abbildet und damit dem Benutzer den Umgang mit abstrakten Nummern erspart. Aber auch die RMI-Registry (vgl. nächstes Kapitel), ein Dateisystem wie FAT32 oder die Windows-Registry stellen allgegenwärtige Vertreter dar, die den namensgebundenen Zugriff auf Informationen ermöglichen.

Verzeichnisdienste kann man als Namensdienst mit erweiterter Funktionalität auffassen. Während ein Namensdienst Objekte lediglich über ihre Namen beschreibt, kann ein Objekt in einem Verzeichnisdienst über zusätzliche Attribute verfügen. Das ermöglicht zum einen die Gruppierung nach funktionalen Aspekten und zum anderen die Suche nach bestimmten Attribut/Wert-Kombinationen. Somit kann ein Corporate Directory implementiert werden, das

als zentraler Dienst Auskunft über Informationen wie Mitarbeiter, Anlagegüter oder Softwarekomponenten bietet und an einheitlicher Stelle Konfigurations- und Accountdaten der Benutzer speichert. In Hinblick auf Java sind zudem Objekt-Repositories von Bedeutung, da sie Java-Objekte wie EJBs, JDBC-Datenquellen oder RMI-Server-Objekte verfügbar machen. Bekannte Verzeichnisdienste sind die

- Microsoft Active Directory Services (ADS),
- Novell Directory Services (NDS),
- Sun Network Information Services (NIS/NIS+) und
- Sun ONE Directory Server.

Da unterschiedliche Implementierungen über unterschiedliche Funktionen verfügen können, unterscheiden sich auch deren APIs. JNDI bietet hier eine standardisierte Schnittstelle mit einer Sammlung an Methoden zum Lesen, Löschen, Erzeugen und Durchsuchen von Verzeichnisdiensten, sodass – immer einen entsprechenden Treiber vorausgesetzt – unterschiedliche Implementierungen mit derselben API angesprochen werden können.

Allerdings muss kritisch bemerkt werden, dass aufgrund einer fehlenden einheitlichen Abfragesprache, ähnlich SQL für relationale Datenbanken, die unterschiedlichen Namenskonventionen der Lösungen nicht völlig abstrahiert werden können. Beispielsweise trennt das Domain Name System Komponenten durch Punkte (.), während LDAP-fähige Dienste hierfür das Komma (,) verwenden. Dies hat zur Folge, dass ein Entwickler Kenntnisse über das eingesetzte System haben muss. In [THOM02] wird dies wie folgt zusammengefasst:

[...] JNDI does not completely abstract the implementation of a particular directory service from you; you must still know the naming convention used in the target service.

Der wesentliche Vorteil bleibt jedoch: JNDI erspart die Einarbeitung in proprietäre APIs. Bei der Abstraktion der Implementierungsdetails greift JNDI auf ein bewährtes Konzept zurück. Es trennt das API zur Applikationsentwicklung vom Service-Provider, sodass sie unabhängig vom eingesetzten Namens- oder Verzeichnisdienst bleibt. Der Service-Provider ist dabei nichts anderes als ein Treiber, der den JNDI-Spezifikationen entspricht und die Schnittstelle für das *JNDI Service Provider Interface (JNDI SPI)* implementiert (vgl. Abbildung 6-2). Die Provider werden entweder vom Hersteller bereitgestellt oder sind im Falle von LDAP, COS[9], RMI, NIS, NDS, DSML[10] und DNS von SUN zu erhalten. Überdies werden von Sun Service-Provider für das Dateisystem[11] sowie für die Windows-Registry zum Download angeboten. Tabelle 6-2 zeigt die JNDI-Paketstruktur.

Nachfolgend soll die Nutzung von JNDI am Beispiel eines LDAP-fähigen Verzeichnisdienstes erfolgen. Bevor ein Verzeichnis- oder Namensdienst genutzt werden kann, muss

[9] CORBA – Common Object Services Naming Service
[10] Directory Service Markup Language
[11] Da Java Systemressourcen wie das Dateisystem abstrahiert, ist nur ein Provider für alle Dateisysteme, wie beispielsweise Fat32, HPFS, NTFS oder ext2, erforderlich!

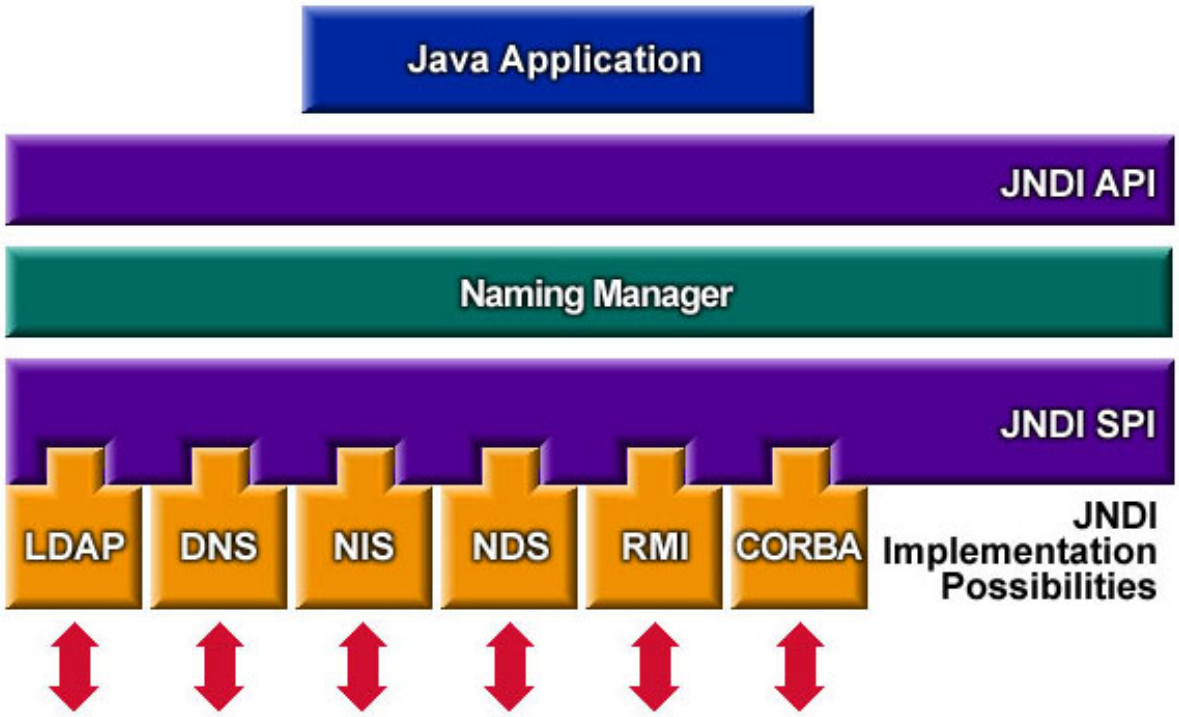

Abb. 6-2: Architektur des Java Naming Directory Interface [JNDI02@]

Paket	**Beschreibung**
javax.naming	Klassen und Schnittstellen zur Nutzung von Namensdiensten
javax.naming.directory	Klassen und Schnittstellen zur Nutzung von Verzeichnisdiensten (erweitert `javax.naming`)
javax.naming.event	ermöglicht nach dem Listener-Konzept das Verarbeiten von Ereignissen bei Objektveränderungen im Verzeichnis- oder Namensdienst
javax.naming.ldap	Erweiterungen für LDAP v3
javax.naming.spi	definiert das Service-Provider-Interface

Tab. 6-2: Die JNDI-Paketstruktur

ein so genannter `InitialDirContext` beziehungsweise `InitialContext` erzeugt werden. Dieser Startkontext ist der Einstiegspunkt in die Hierarchie eines Verzeichnis- oder Namensdienstes, von dem aus weitere Operationen möglich sind. Die Erzeugung dieses Objektes ist gleichbedeutend mit dem Aufbau der Verbindung (inklusive Login und Authentifizierung). Listing 6-1 zeigt die Erzeugung eines solchen Startkontextes am Beispiel eines LDAP-Verzeichnisdienstes[12]. Der Konstruktor der Klasse `InitialDirContext` akzeptiert einen Parameter vom Typ `Hashtable`, der die Konfigurationsparameter enthält. Es müssen mindestens zwei Parameter angegeben werden. Mit dem Schlüssel `Context.INITIAL_CONTEXT_FACTORY` wird der vollständige Klassenname des Service-Providers angegeben. Für einen LDAP-fähigen Dienst ist das `com.sun.jndi.ldap.LdapCtxFactory`. Der zweite Pflichtparameter mit dem Schlüssel `Context.PROVIDER_URL` spezifiziert die URL des Dienstes. Ist eine Authentifizierung erforderlich muss mit den Schlüsseln

- `Context.SECURITY_PRINCIPAL`,
- `Context.SECURITY_CREDENTIALS` und
- `Context.SECURITY_AUTHENTICATION`

```
1  Hashtable env = new Hashtable();

   String serviceProvider = "com.sun.jndi.ldap.LdapCtxFactory";
   env.put(Context.INITIAL_CONTEXT_FACTORY, serviceProvider);

6  String jndiUrl = "ldap://localhost:389/o=MyLdapData";
   env.put(Context.PROVIDER_URL,jndiUrl);

   String user = "uid=webdial-admin,ou=webdial,o=canoris";
   env.put(Context.SECURITY_PRINCIPAL, user);
11
   String password = "aPassword";
   env.put(Context.SECURITY_CREDENTIALS, password);

   String authType = "simple";
16 env.put(Context.SECURITY_AUTHENTICATION, authType);

   DirContext dirContext = new InitialDirContext(env);
```

Prog. 6-1: Erzeugung von InitialDirContext

der Benutzername, das Passwort und die Art der Authentifizierung angegeben werden. Im Beispiel wird die Authentifizierungsmethode `simple` eingesetzt (vgl. Zeile 15f.). Bei diesem

[12] Verzeichnisdienst mit LDAP-Unterstützung

Verfahren werden Benutzername und Passwort unverschlüsselt im Klartext übertragen. Mit dem Wert `none` würde ein anonymer Login ohne Authentifizierung versucht. Bei LDAP v3 wurde mit SASL[13] ein dritter Authentifizierungsmechanismus aufgenommen, der den Einsatz verschiedener Authentifizierungsverfahren ermöglicht, die zwischen Client und Server ausgehandelt werden können. Mögliche Varianten nach [J2SE04a@] sind:

- Anonymous (IETF RFC-2245)[14]
- CRAM-MD5 (IETF RFC-2195)
- Digest-MD5 (IETF RFC-2831)
- External (ITEF RFC-2222)
- Kerberos V4 (IETF RFC-2222)
- Kerberos V5 (IETF RFC-2222)
- SecurID (IETF RFC-2808)
- Secure Remote Password (IETF RFC-2945)
- S/Key (IETF RFC-2222)
- X.509 (IETF RFC-2459)

Die Angabe mehrerer Optionen ist möglich. Mit

```
env.put(Context.SECURITY_AUTHENTICATION,
2         "DIGEST-MD5 KERBEROS_V4");
```

würde eine SASL-Authentifizierung zuerst mit `DIGEST-MD5` und dann mit `KERBEROS_V4` versucht.

Wurde der Startkontext erfolgreich erzeugt, können verschiedene Operationen durchgeführt werden. Die Suche nach Objekten stellt dabei die häufigste Anwendung dar. Das Listing 6-2 zeigt eine beispielhafte Suche nach Mitarbeitern, die aus Stuttgart kommen und deren Nachnamen mit dem Buchstaben „M“ beginnt. Bei dieser Suche wird mit einem Filterobjekt zur Einschränkung der Ergebnisse gearbeitet. Eine Suche kann mit einer der acht überladenen `search`-Methoden des Interfaces `javax.naming.directory.DirContext` erfolgen. Die Methode zur filterbasierten Suche erfordert die Parameter

- `nameBase` für den Namenskontext, von dem die Suche ausgehen soll,
- `filter` für den Suchfilter und
- `sc` für die Konfiguration der Sucheinstellungen (hier die Suchtiefe).

```
String nameBase = "ou=People";

SearchControls sc = new SearchControls();
sc.setSearchScope(SearchControls.ONELEVEL_SCOPE);

String filter = "(&(sn=M*)(dcity=Stuttgart))";

NamingEnumeration ne = dirContext.search(nameBase, filter, sc);
```

Prog. 6-2: Filterbasiertes Suchen in LDAP-Verzeichnisdiensten

Die Kombination aus Ausgangspunkt und Tiefe spezifiziert den Bereich der Namenshierarchie, der durchsucht werden soll. Ausgehend vom Startkontext der Suche geben die `static`-Konstanten `OBJECT_SCOPE`, `ONELEVEL_SCOPE` und `SUBTREE_SCOPE` der Klasse `SearchControl` an, ob die Suche sich nur auf das Startobjekt, die tieferliegende Verzeichnisebene oder den ganzen Teilbaum – mit dem Startobjekt als Wurzel – ausdehnen soll. Der Suchfilter wird im Beispiel aus Listing 6-2 in Form des Strings `(&(sn=M*)(dcity=Stuttgart))` spezifiziert. `sn` und `dcity` beziehen sich dabei auf die Objekt-Attribute für den Nachnamen und den Wohnort. Das kaufmännische „und“ drückt eine logische UND-Verknüpfung beider Suchbedingungen aus. Der Wertebereich kann statisch oder über Wildcards wie „*“ eingeschränkt werden. Das RFC-2254 der IETF beschäftigt sich mit der Symbolik bei Suchfiltern in LDAP-Verzeichnissen. Für einfaches Suchen nach konkreten Attribut/Wert-Kombinationen ohne Filter existieren entsprechende Überladungen der Methode `search`.

6.1.2.2 RMI - Remote Method Invocation

Java bietet mit dem Paket `java.net` eine Sammlung von Interfaces und Klassen zur Nutzung von Sockets für Netzwerkverbindungen. Der Aufbau von Netwerkverbindungen sowie das Senden und Empfangen von Daten erfolgt mittels weniger Objekte und Methoden. Die eigentliche Herausforderung besteht in der Implementierung eines Protokolls auf Applikationsebene, das applikationsspezifische Daten kodiert und von Client zu Server beziehungsweise von Peer zu Peer überträgt. Ohne weitere Technologien ist bei der Entwicklung verteilter, Socket-basierter Anwendungen auch immer ein Applikationsprotokoll zu entwerfen und implementieren. Um diese Problematik zu entschärfen, wurden so genannte *Remote Procedure Calls (RPC)* eingeführt, die ein generisches Applikationsprotokoll zum Aufruf entfernter Prozeduren darstellen. Kernkonzept ist die Abstraktion der Netzwerkkommunikation auf einen einfachen lokalen Prozeduraufruf, der die darunter liegende Netzwerkebene verbirgt. Auf diese Weise können Funktionen auf verschiedene Systeme ausgelagert und bei Bedarf über den Aufruf einer entsprechenden Prozedur in Anspruch genommen werden.

In der objektorientierten Programmierung wäre dies gleichbedeutend mit dem Aufruf ei-

[13] Simple Authentication and Security Layer
[14] The Internet Engineering Task Force – Request for Comments

ner Methode. Da jedoch eine Methode Teil eines Objektes ist, ist das Resultat des Aufrufs abhängig vom Objektzustand. Während bei den RPCs nur die Parameter und der Rückgabewert zwischen Server und Client übertragen werden müssen, stellt sich bei einem Methodenaufruf die Frage nach den Zuständigkeiten bei der Objektverwaltung. Wo wird das Objekt gehalten, dessen Methode entfernt aufgerufen wird? Wie werden Objekte behandelt, die als Parameter oder Rückgabewerte dienen? Ändert sich der Objektzustand auch auf Serverseite, wenn er auf dem Client verändert wird? Hinter Letzterem verbirgt sich die Frage nach *Call-by-Value* oder *Call-by-Reference*. In Hinblick auf Java stellt sich zudem die Frage nach einem verteilt arbeitenden Garbage Collector, der Objekte erst löscht, wenn sowohl lokal als auch remote keine Objektreferenzen mehr existieren.

Mit der *Remote Method Invocation (RMI)* wurde das RPC-Konzept auf die Objektebene übertragen. Während bei den RPCs lediglich der Aufruf entfernter Prozeduren transparent ist, kann mit RMI die Ortstransparenz von Objekten realisiert werden, welche auf verschiedenen JVMs verteilt sind. Dabei existiert ein so genanntes *Remote-* oder *Server-Objekt*, das auf Seite der Serveranwendung erzeugt und gehalten wird, zugleich aber von mehreren Clients nebenläufig[15] referenziert und benutzt werden kann. Erreicht wird dies über einen lokalen Stellvertreter (Proxy), der auf Client- wie auf Serverseite erzeugt wird. Diese Proxy-Objekte implementieren die Netzwerkkommunikation zwischen Server und Client. Ein so genanntes *Remote-Interface* definiert dabei die Remote-Methoden des verteilten Objekts, die zum Aufruf über die Grenzen der JVM hinweg zur Verfügung stehen sollen. Das Server-Objekt, das die Methoden des Remote-Interfaces implementiert, enthält die eigentliche Funktionalität. Die Proxy-Objekte, auf Clientseite *Stub* und auf Serverseite *Skeleton* genannt, setzen den Methodenaufruf in eine entsprechende Netzwerkkommunikation um. Der Stub stellt die Methoden des Remote-Interfaces auf dem Client bereit und leitet bei deren Aufruf die Anfrage über das Netzwerk an das Skeleton-Objekt weiter. Dieses bearbeitet die Anfrage unter Nutzung der Implementierung des Server-Objekts und liefert das Ergebnis an den Stub zurück.

Der entscheidende Vorteil dieser Vorgehensweise ist, dass die Proxy-Klassen auf Basis des Server-Objekts automatisch generiert werden. Somit reduziert sich mit RMI der Entwicklungsaufwand eines Applikationsprotokolls auf die Definition des Remote-Interfaces und dessen Implementierung in einer Server-Klasse.

RMI macht intensiv Gebrauch von der Objektserialisierung (Marshalling), um Objekte für die Übermittlung über ein Netzwerk in einen seriellen Datenstrom umzuwandeln. Java kann alle Objekte serialisieren, die das Marker-Interface[16] `Serializable` implementieren. Deshalb müssen Objekte, die über RMI übertragen werden sollen, über dieses Interface verfügen.

Im Rahmen eines Beispielprogramms sollen nun die einzelnen Schritte der Erstellung RMI-basierter Software erläutert werden. Das Beispielprogramm soll ein Server-Objekt mit einer Remote-Methode umfassen, die den String `Hello world!` an einen Client zurückgibt. Zunächst wird das Remote-Interface `Hello` definiert, das vom Interface `java.rmi.Remote` abgeleitet werden muss. In diesem Interface wird eine Remote-Methode definiert, die eine `throws`-Klausel mit der Exception `RemoteException` enthält. Dies ist obligatorisch

[15] RMI ermöglicht den nebenläufigen Zugriff auf Remote-Objekte durch verschiedene Clients. Das Management der Sockets, öffnen und schließen der Ports, übernimmt die RMI-Middleware.

[16] Interface ohne Methoden

für alle Remote-Methoden, da diese Exception bei Fehlern innerhalb der RMI-Middleware geworfen wird. Zumeist sind Netzwerkprobleme hierfür die Ursache. Es können in einem Remote-Interface auch Methoden definiert werden, die zusätzlich zur `RemoteException` benutzerdefinierte Ausnahmen werfen können. Listing 6-3 zeigt das Remote-Interface `Hello` mit der Methode `sayHello()`.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
  public String sayHello() throws RemoteException;
}
```

Prog. 6-3: RMI Remote-Interface

Im zweiten Schritt wird das Server-Objekt erzeugt, welches das eben definierte Remote-Interface implementiert und zudem von der Klasse `UnicastRemoteObject` abzuleiten ist. Neben der Remote-Methode enthält die Klasse einen leeren Konstruktor und zur Vereinfachung die Main-Methode des Server-Threads (vgl. Listing 6-4). Zeile 15 zeigt wie das RMI-Objekt `rmiServer` unter dem Namen `HelloServer` in die RMI-Registry aufgenomen wird. Bei der RMI-Registry handelt es sich um einen Namensdienst, der Clients das Auffinden von RMI-Objekten ermöglicht. Dieser Namensdienst ist im Softwarepaket *Java Runtime Environment* von Sun enthalten. Es handelt sich dabei um das Kommandozeilenprogramm `rmiregistry`, das im Unterverzeichnis `bin/` des Installationsverzeichnisses zu finden ist. Obwohl in diesem Beispiel davon ausgegangen wird, muss dieser Dienst nicht zwangsläufig auf demselben System wie der RMI-Server ausgeführt werden.

Nachdem das Server-Objekt über die Methode `bind` an einen Namenskontext gebunden wurde, kann ein Client dasselbe Objekt über die RMI-Registry mit der Methode `lookup` lokalisieren und benutzen (vgl. Listing 6-5). Mit dem Namen kann dabei sowohl bei `bind` als auch bei `lookup` gemäß dem Schema

```
rmi://<host>:<port>/<object-name>
```

der Server mit angegeben werden, auf dem die RMI-Registry zu finden ist. Werden die Parameter `host` und `port` nicht angegeben (vgl. Listing 6-4) wird der `localhost` mit dem Standardport `1099` angenommen.

Als nächstes können mit Hilfe des RMI-Compilers die Stub- und Skeleton-Klassen erzeugt werden. Der Compiler `rmic` ist ebenfalls im Java SDK unter `bin/` zu finden. Nach dem Aufrufschema

```
rmic [options] <server-implementation>
```

können aus der Server-Klasse die Proxy-Klassen erzeugt werden. Der Aufruf von `rmic Hello` führt zu den `class`-Dateien `Hello_Stub.class` und `Hello_Skel.class`, wobei der RMI-Compiler auf den bereits kompilierten Bytecode der Server-Klasse ausgeführt wird, und nicht, wie man annehmen könnte, auf den Quellcode.

```
import java.rmi.RemoteException;
import java.rmi.Naming;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl extends UnicastRemoteObject
                                    implements Hello {
  public HelloImpl() throws RemoteException {}

  public String sayHello() throws RemoteException {
    return "Hello world!";
  }

  public static void main(String args) throws Exception {
    HelloImpl rmiServer = new HelloImpl();
    Naming.bind("HelloServer", rmiServer);
  }
}
```

Prog. 6-4: Implementierung des RMI-Servers

Sobald der RMI-Stub erzeugt wurde, kann der Client implementiert werden. Listing 6-5 zeigt die Klasse `HelloClient`, die nur aus einer Main-Methode besteht, in der das Remote-Objekt über die RMI-Registry aufgelöst und lokal deklariert wird (vgl. Zeile 5). Man beachte, dass die `lookup`-Methode immer ein Objekt vom Typ `Object` zurückliefert und deshalb über einen Type-Cast in den eigentlichen Typ des Server-Objekts umgewandelt werden muss. Zeile 6 enthält den lokalen Aufruf der Remote-Methode, hinter dem sich die Netzwerkkommunikation verbirgt.

```
import java.rmi.Naming;

public class HelloClient {
  public static void main (String[] args) throws Exception {
    Hello rmiServer = (Hello) Naming.lookup("HelloServer");
    String message = rmiServer.sayHello();
    System.out.println("Server responds: " + message);
  }
}
```

Prog. 6-5: Implementierung des RMI-Clients

Nachdem Interface, Server sowie Client erzeugt wurden, kann das Beispiel ausgeführt werden. Wenn die RMI-Registry aktiv ist, können Server und Client gestartet werden (vgl. Listing 6-6 für Linux).

```
rmiregistry &

java HelloImpl &

java HelloClient

Server responds: Hello world!
```

Prog. 6-6: Start der RMI-Registry sowie Ausführung von Server und Client

Nun wurde gezeigt wie verteilte Objekte erzeugt, registriert und aufgefunden werden können. Wie aber werden diese wieder deregistriert und gelöscht? Die Deregistrierung erfolgt durch die `static`-Methode `Naming.unbind`. Für die Objektzerstörung ist bei Java immer der Garbage Collector zuständig. Da es sich nun aber bei Remote-Objekten um verteilte Objekte handelt, auf die Referenzen in verschiedenen JVMs existieren können, kommt bei RMI ein verteilter Garbage Collector zum Einsatz. Dieser verfügt über ein Kommunikationsprotokoll, das dem Garbage Collector der Server-JVM mitteilt, ob eine Objektreferenz erzeugt oder gelöscht wurde. So kann der serverseitige Garbage Collector entscheiden, ob das Objekt zerstört werden kann oder nicht.

Abschließend soll noch bemerkt werden, dass RMI über Möglichkeiten zum automatischen Laden der Stub-Klasse über einen HTTP-Server verfügt, sodass bei Änderung der Implementierung nur die Stub-Klasse auf dem Code-Server ausgetauscht werden muss. Ansonsten müsste dies auf jedem installierten Client erfolgen. Weiter ermöglicht *RMI-Activation* das automatische Starten des Server-Prozesses und die Registrierung des Remote-Objekts bei Eingang einer Client-Anforderung. Hierfür existiert mit `rmid` ein Dämon, der vergleichbar mit `initd`, auf Anfragen wartet und bei Bedarf den enstprechenden Server startet. Dies kann sich bei einer großen Zahl an Remote-Objekten als effizienter erweisen.

Seit der Java Version 5.0 ist RMI in der Lage die Stub-Klassen dynamisch zur Laufzeit zu generieren, sodass auf den Aufruf des RMI-Compilers ganz verzichtet werden kann, wenn alle beteiligten Java-Komponenten die JRE 5.0 benutzen. Müssen allerdings Clients in älteren JREs ausgeführt werden, sind die Stubs zu erzeugen und zu verteilen. `rmic` des Java SDK 5.0 generiert standardmäßig nur noch die Stub-Klassen, da die Skeleton-Klassen bereits seit dem JDK 1.2 nicht mehr benötigt werden.

Auf die Thematik *Call-by-Value* oder *Call-by-Reference* wird im Kapitel 7.5.1 näher eingegangen. Vorweg soll angemerkt werden, dass Objekte standardmäßig kopiert werden, wenn sie in Form von Übergabeparametern oder Rückgabewerten die Grenzen der lokale JVM überschreiten. Ohne entsprechende Berücksichtigung beim Entwurf betreffen Änderungen nur die Objektkopie und werden nicht über die Grenzen der jeweiligen JVM hinweg propagiert.

6.2 Eclipse 3.1 Integrated Development Environment

Bei der Entwicklung von *webDial* wurde das Eclipse SDK 3.1.0 im Stable-Built I20050219-1500 eingesetzt. Diese Version wurde gewählt, da das enthaltene Plug-in *Java Development*

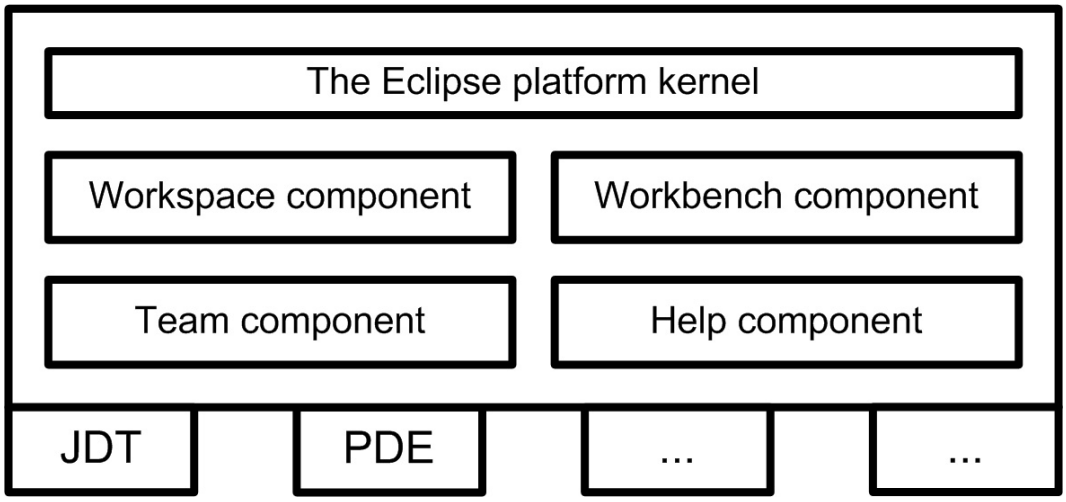

Abb. 6-3: Aufbau der Eclipse-Plattform [HOLZ04]

Toolkit (JDT) in der Version 3.1.0 bereits die neuen Sprachelemente der Java Version 5.0 unterstützt.

Bei Eclipse handelt es sich in erster Linie um eine erweiterbare Plattform, die Grundfunktionen zur Darstellung von Fenstern, zur Projekt- und Dateiverwaltung, zur Koordinierung von Entwicklerteams und zur Integration von Hilfefunktionen bereitstellt. Abbildung 6-3 zeigt die Komponenten der Eclipse-Plattform. Basiskomponenten sind

- *Workspace* für die Projekt- und Ressourcenverwaltung,
- *Workbench* für die Darstellung der Fenster, Menüs und Views,
- *Team* für die Integration von Systemen wie *CVS* und *SVN*[17] sowie
- *Help* für die Integration der Plug-In-Hilfesysteme.

Die Abbildung zeigt wie die Eclipse-Plattform durch die Plug-Ins JDT und *PDE (Plug-In Development Environment)* erweitert wird. Die Mächtigkeit und der Erfolg von Eclipse begründet sich auf diesem Plug-In-Konzept. Ohne weitere Plug-Ins wäre Eclipse im Wesentlichen nur ein Texteditor mit umfangreichen Hilfe- und Konfigurationsfunktionen. Eclipse wird oft fälschlicherweise mit einer Java-IDE gleichgesetzt, obgleich die Java-bezogenen Funktionen über das Plug-In JDT (Java Development Toolkit) bereitgestellt werden. Damit ist Eclipse – entsprechende Plug-Ins vorausgesetzt – als IDE für verschiedenste Programmiersprachen und Frameworks geeignet. Es existieren zahlreiche Plug-Ins unterschiedlichster Art und Funktion. Einige Plug-Ins sind vom Umfang her als eigenständige Programme zu betrachten, die im Rahmen des Eclipse-Frameworks ausgeführt werden. Es ist deshalb nicht verwunderlich, dass sich mittlerweile ein kommerzieller Markt für Entwicklung und Verkauf von Eclipse-Plug-Ins entwickelt hat, wobei die eigentliche Eclipse-Plattform frei von Lizenzgebühren ist.

[17] Concurrent Versions System und Subversion sind Lösungen zur Versions- und Änderungskontrolle zum Zweck der Koordination paralleler Entwicklungsprozesse.

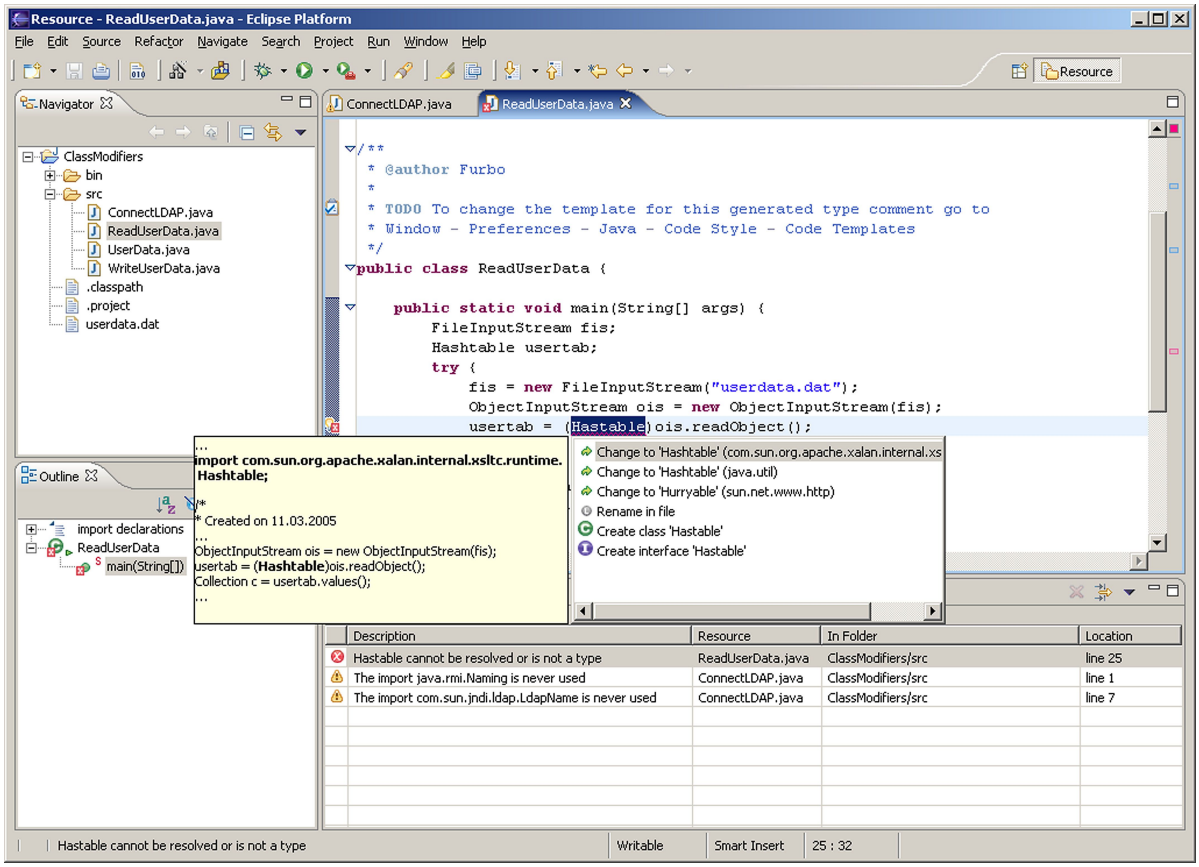

Abb. 6-4: Screenshot der Eclipse-Workbench mit dem Java Development Toolkit

Für den Einsatz von Eclipse als Java-IDE[18] spricht eine Vielzahl von Funktionen zur Erleichterung des Softwareentwicklungsprozesses. Dazu gehören Funktionen zur Codeerzeugung, zum Refactoring und zum Debugging. Mit ihnen ist es beispielsweise möglich, Variablen-, Methoden und Klassennamen an einer Stelle zu verändern, wobei der Rest des Projekts automatisch an die Modifikation angepasst wird. Weiter ist es möglich, Interfaces aus bestehenden Klassen zu extrahieren oder neue Klassen für bestimmte Anforderungen generieren zu lassen. Ergänzt wird dies durch ein umfassendes Hilfesystem und Möglichkeiten zur Änderungsverfolgung sowie zur automatischen Vervollständigung des eingegebenen Codes.

Dies ist nur ein Teil der Möglichkeiten, die die Eclipse-Plattform bietet. Weitere Informationen über Eclipse und der Download der Eclipse SDK sowie einzelner Plug-Ins sind auf der Webseite des Eclipse-Projekts[19] zu erhalten.

[18] In diesem Zusammenhang ist Eclipse in Kombination mit dem Plug-In JDT zu verstehen.
[19] www.eclipse.org

Version	Releasedatum
1.0	November 1996
1.1	Februar 1997
1.2	Dezember 1997
1.3	Juli 1999
1.4	September 2001

Tab. 6-3: Versionsgeschichte von JTAPI

6.3 JTAPI

Wie bereits in den Einführungskapiteln beschrieben, handelt es sich beim *Java Telephony Application Programming Interface (JTAPI)* um eine Schnittstellenspezifikation zur Entwicklung von CT-Anwendungen für die Java-Plattform nach dem *First* und dem *Third Party Call Control Modell*. Im Folgenden wird das Grundkonzept dieses Standards erläutert und ein Gesamtüberblick über die aktuelle Version 1.4 gegeben.

Zentraler Aspekt der JTAPI-Architektur ist die Einführung einer Abstraktionsschicht zwischen CT-Applikation und den Trägern der Telefoniefunktion. Diese kann sowohl von dedizierter Telekommunikationshardware als auch auf Softwarebasis erbracht werden. Egal auf welche Weise die Telefoniedienste implementiert sind – über Nebenstellenanlagen mit Systemtelefonen, Smartphones[20] oder das SIP-Protokoll mit IP-Phones[21] – deren Nutzung geschieht transparent.

Diese Unabhängigkeit wird über einen *JTAPI-Treiber* (auch: *JTAPI-Implementierung* oder *Provider*[22]) ermöglicht, über den die Anwendungsentwickler die Telefoniefunktionen ansprechen können. Da dieser Treiber bestimmten Spezifikationen zu entsprechen hat, also durch JTAPI definierte Schnittstellen implementiert, können die Treiber gegeneinander ausgetauscht werden. Im Idealfall müssen dabei keine Anpassungen vorgenommen werden.

JTAPI ist eine detaillierte Beschreibung eines API, das aus einer Menge von Schnittstellen und einem Zustandsmodell besteht. Eine JTAPI-Implementierung kann vom Hersteller der Telefonielösung oder von Drittanbietern bereitgestellt werden, wobei nicht zwingend alle JTAPI-Schnittstellen unterstützt werden müssen. Vielmehr kann die Auswahl der Schnittstellen von den Fähigkeiten des Telefoniesystems oder anderen Faktoren wie beispielsweise der Produktpolitik abhängen. Dem Anwendungsentwickler bleibt es seinerseits überlassen auszuwählen, welche der angebotenen Schnittstellen einer JTAPI-Implementierung er zur Realisierung seiner Applikation benötigt.

Die Schnittstellen sind aus diesen Gründen in mehrere funktionale Komponenten (vgl. Abbildung 6-5) untergliedert, die unterschiedlichen Aspekten Rechnung tragen, jedoch auf dem *Core-Paket* `javax.telephony` aufbauen. Dieses Hauptpaket bildet den kleinsten gemein-

[20] Ein Smartphone (auch: Mobile Digital Assistant – MDA) vereint den Funktionsumfang eines PDAs mit dem eines Mobiltelefons.

[21] Telefone zum Anschluss an ein IP-basiertes Netz

[22] Der Provider ist das zentrale Objekt im JTAPI-Call-Modell, von dem aus alle relevanten Zustände erfasst und beeinflusst werden können.

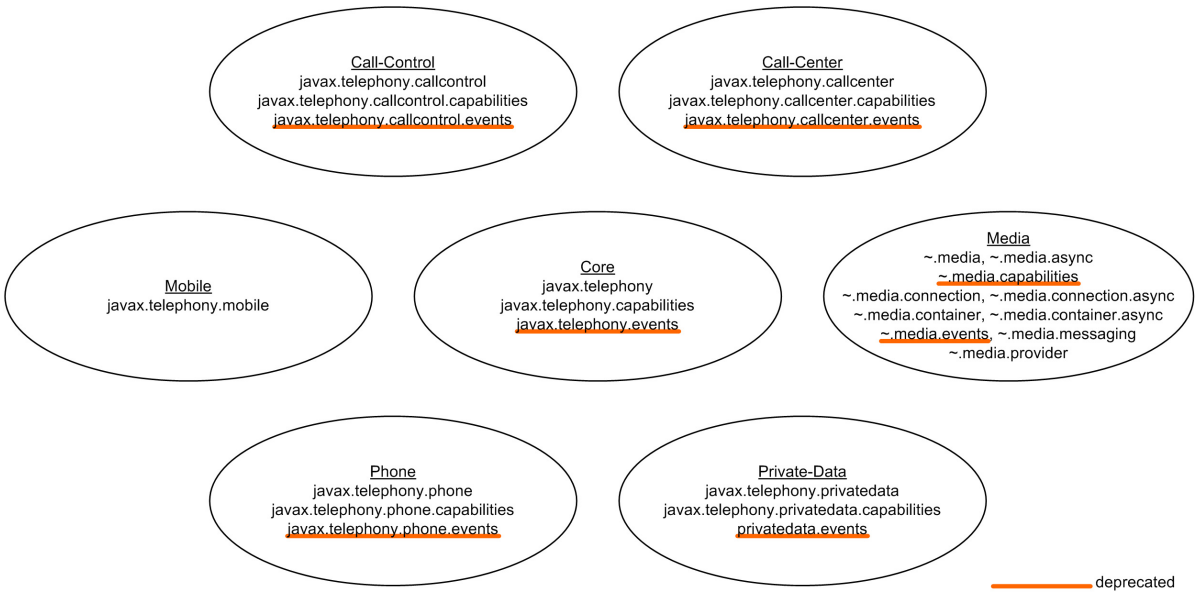

Abb. 6-5: JTAPI-Pakete (nach [JTAP02@])

samen Nenner einer einfachen CT-Anwendung, wobei das zugrunde liegende Call-Modell und darauf aufbauende Grundfunktionen zum Anrufen, Abnehmen und Auflegen definiert werden. Die Erweiterungspakete ergänzen das Core-Paket um neue Objekte oder erweitern bestehende.

Das *Call-Control-Paket* erweitert das Core-Paket um Funktionen zur Beeinflussung und Steuerung von Calls. Es werden Funktionen für das Makeln, für Anrufweiterleitungen und Konferenzschaltungen sowie für das Weiterverbinden ergänzt. Zusätzlich wird das Call-Modell auf ein detaillierteres Zustandsmodell erweitert.

Das *Call-Center-Paket* ergänzt Funktionsaspekte, die bei Call-Center-Applikationen relevant sind. Das Paket ermöglicht die Manipulation des Routings, ACD, Predictive Dialing und die Assoziation von Call und Anwendungsdaten. Letzteres ermöglicht die Zuordnung weiterführender Gesprächsinformationen verschiedenster Art wie zum Beispiel Kundendaten, Fehlerbeschreibungen oder Ticket-IDs.

Das *Mobile-Paket* zielt auf die Entwicklung von Applikationen für Smartphones ab, indem Aspekte der Mobiltelefonie eingeführt werden. Das Paket sieht Funktionen für die Manipulation der Sendeleistung, die Auswahl des Funknetzes sowie für die Deaktivierung und Aktivierung der Funkkomponente vor. Während alle anderen Erweiterungspakete über zusätzliche Unterpakete verfügen, wurde beim Mobile-Paket der Anforderung nach geringstem Speicherbedarf Rechnung getragen und auf erweiterte Funktionen verzichtet.

Das *Media-Paket* ermöglicht die Manipulation von Mediastreams verschiedener Art zur Implementierung von IVR-, Mailbox- und Messagingsystemen. Für die Spracherkennung, Text-to-Speech und das Senden und Empfangen von Faxnachrichten werden ebenfalls Funktionen bereitgestellt. Mit neun Unterpaketen handelt es sich beim Media-Paket um die umfangreichste Erweiterung. Zudem stellt es eine Ausnahme unter sämtlichen JTAPI-Komponenten dar, da es sich um ein selbständiges API handelt, das nicht zwingend auf das Core-Paket angewiesen

ist.[23]

Das *Phone-Paket* ist für die direkte Manipulation von Endgeräten für Telefonie und Fax verantwortlich. Diese Schnittstelle sieht die Beeinflussung der einzelnen Komponenten wie Tasten, Displays, Lampen, Lautsprecher und Mikrophone vor. Weiter wird die Manipulation der Ruftöne und die des Auflegemechanismusses unterstützt.

Das *Private-Data-Paket* kann zur generischen Implementierung weiterer Funktionen benutzt werden. Jedes JTAPI-Objekt, welches das zentrale Interface `PrivateData` implementiert, kann mit der ausführenden Schicht (PBX, Netzwerk, (Mobil)Telefon) implementierungsspezifische Nachrichten austauschen. Dies kann jedoch zu einer Abhängigkeit von einer bestimmten JTAPI-Implementierung führen, die das Private-Data-Interface benutzt. Unter Umständen kann die Kompatibilität zu einem anderen JTAPI-Provider nicht sichergestellt werden.

Mit Ausnahme des Mobile-Pakets verfügen alle JTAPI-Pakete jeweils über ein `Capabilities`- und ein `Events`-Unterpaket. Das Capabilities-Paket ermöglicht einer Applikation die Fähigkeiten eines JTAPI-Providers zu ermitteln. Dabei wird zwischen Abfragen auf statischer und dynamischer Basis unterschieden. Statische Fähigkeiten werden durch die Implementierung des Providers bestimmt, da nicht alle Funktionen des API unterstützt werden müssen. Dynamische Fähigkeiten beziehen zudem die aktuelle Situation des Zustandsmodells mit ein. Somit kann beispielsweise ermittelt werden, ob man von einem bestimmten Terminal aus einen Anruf initiieren kann. Ist der besagte Apparat bereits aktiv, wird eine entsprechende Abfrage negativ ausfallen, ist er frei – positiv.

Die `Events`-Unterpakete definieren im Kontext ihrer übergeordneten Pakete Ereignisse, die über entsprechende Observer-Objekte kommuniziert werden können. Sie sind seit der JTAPI-Version 1.4 im Status *deprecated*, da die Observer durch Listener ersetzt wurden. Im Zusammenhang mit JTAPI beschreibt zwar beides das Publisher-und-Subscriber-Muster, jedoch unterscheiden sie sich in bestimmten Konventionen. Ein Observer-Objekt besitzt *eine* Event-Methode, die zur Benachrichtigung bei Statusänderungen eines bestimmten Objekts, bei dem der Observer registriert wurde, aufgerufen wird. Die Statusinformationen werden mittels Event-Objekten übertragen, die als Parameter der Event-Methode übergeben werden. Dieses Verfahren führte in der Vergangenheit zu einer Flut an Klassen für die Event-Benachrichtigung. Um dieser Entwicklung entgegenzuwirken, wurden Listener-Objekte eingeführt, die für jede Art von Statuswechsel über eine andere Event-Methode verfügen. Die Statusinformation wird nun nicht mehr über den Typ des Event-Objekts, sondern über die Signatur der Event-Methode ausgedrückt. Die Verringerung der Anzahl von Event-Klassen geht somit zu Lasten komplexerer Listener. Auf das Event/Listener-Modell von JTAPI wird in Kapitel 6.3.3 auf Seite 68 eingegangen.

In den folgenden Kapiteln wird nun das Core-Paket von JTAPI beschrieben, wobei auf das Call-Modell und dessen Elemente näher eingegangen wird. Ergänzend hierzu wird das Observer/Listener-Modell erklärt, das zur Überwachung der Zustandsänderungen im Call-Modell angewandt wird. Zuletzt werden die Erweiterungen, die das Call-Control-Paket einführt, erläutert.

[23] Das Media-Paket richtet sich nach der Media-Architektur aus der S.100 Spezifikation der ECTF.

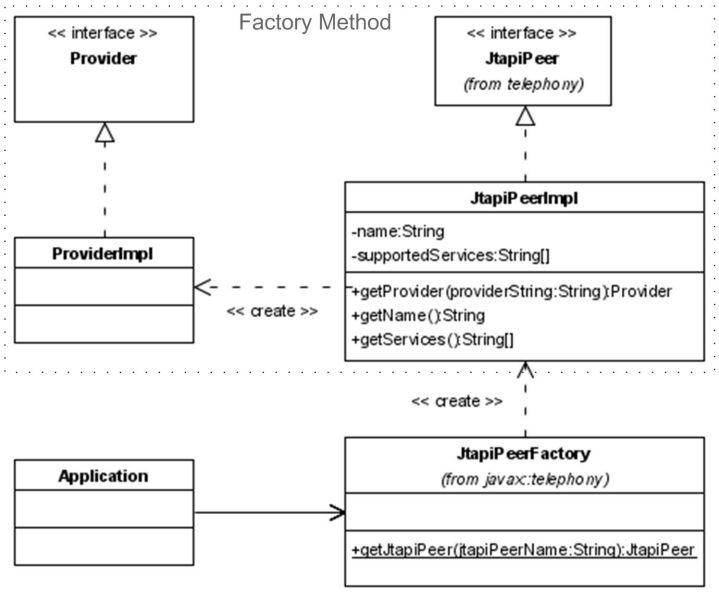

Abb. 6-6: Laden einer JTAPI-Implementierung und Erzeugung eines Providers

6.3.1 Das Core-Paket

Das Core-Paket bildet das Kernstück von JTAPI. In diesem Paket werden die Objekte des zugrunde gelegten Zustandsmodells definiert. Zentrales Element ist das Interface `Provider`. Ein Provider ist die Abstraktion des Telefoniesystems, mit dem die Anwendung interagieren soll. Über dieses Interface erhält eine Applikation Zugriff auf das gesamte Objektmodell (vgl. nächstes Kapitel). Ein Provider kann insofern als austauschbarer Treiber betrachtet werden, als verschiedene Implementierungen des Provider-Interfaces den Zugriff auf andere Telefoniesysteme ermöglichen und aufgrund der einheitlichen Schnittstelle ausgetauscht werden können.

Um den Austausch von JTAPI-Implementierungen zu ermöglichen und die Abhängigkeit der Anwendung zu minimieren, wurde die Erzeugung der Provider-Objekte über ein Erzeugungsmuster vom Typ *Factory Method (Fabrikmethode)* gekapselt (vgl. Abbildung 6-6). Dies ermöglicht einer Anwendung die Erzeugung eines Provider-Objekts, ohne die implementierende Klasse zu kennen. Die Erzeugung eines Providers erfolgt deshalb in zwei Schritten. Das Core-Paket umfasst die Klasse `JtapiPeerFactory`, die nur eine statische Methode `getJtapiPeer` enthält. Dabei handelt es sich um einen virtuellen Konstruktor, der unter Angabe eines Namens eine Implementierung des Interface `JtapiPeer` zurückgibt. Dieses Objekt ist der erste Kontaktpunkt zwischen Anwendung und JTAPI-Implementierung, sodass `JtapiPeerFactory` auch als vorgeschalteter Classloader betrachtet werden kann, dessen Aufgabe das Laden des Treibers ist. Über den JtapiPeer kann ein Anbieter mehrere Provider-Implementierungen für verschiedene Zwecke verfügbar machen. Das JtapiPeer-Objekt enthält dafür die Methoden

- `getName` zur Ermittlung des JtapiPeer-Namens,
- `getServices` zur Auflistung aller verfügbaren Provider-Implementierungen und

- `getProvider` zur Erzeugung des gewünschten Provider-Objekts.

Ausgehend von einem JtapiPeer-Objekt kann nun mit der parametrisierten Fabrikmethode `getProvider` unter Angabe eines Namens ein Provider erzeugt werden. Die Applikation hat nun Zugriff auf das Telefoniesystem. Das Listing 6-7 zeigt die eben geschilderte Vorgehensweise.

```
String peerName = "DefaultJtapiPeer";
JtapiPeer thePeer = JtapiPeerFactory.getJtapiPeer(peerName);
String serviceName = "PBX.Meridian_I; pbxId = BB_SSB";
Provider theProvider = thePeer.getProvider(serviceName);
}
```

Prog. 6-7: Erzeugung des JTAPI-Providers

6.3.2 Call-Modell

Über das Provider-Objekt kann eine Anwendung auf die Objekte des Call-Modells zugreifen, die das Zustandsmodell von JTAPI realisieren. Das Call-Modell setzt sich aus fünf verschiedenen Objekten zusammen, deren Interfaces in der Spezifikation genaustens definiert sind. Die Interfaces `Call`, `Connection`, `Address`, `TerminalConnection` und `Terminal` werden zur Modellierung aller Call-Situationen verwendet.

Ein *Call-Objekt*[24] repräsentiert eines der Gespräche, das innerhalb der Domain des Providers stattfindet. Die Domain ist der Einflussbereich des Providers auf Elemente der Telefonieinfrastruktur. Repräsentiert der Provider beispielsweise eine PBX, umfasst die Domain alle angeschlossenen Terminals und die aktiven Telefongespräche mit Beteiligung lokaler Endgeräte. Ein Provider-Objekt enthält Referenzen auf alle Call-Objekte seiner Domain.

Ein *Address-Objekt* stellt die logische Adresse eines Terminals dar. Die Adresse kann je nach Art des zugrunde liegenden Telefoniesystems einer Telefonnummer, einer SIP-Adresse oder einem beliebigen anderen Schema entsprechen.

Ein *Connection-Objekt* verbindet ein Address-Objekt mit einem Call. Hat ein Call zwei Teilnehmer, sind zwei Address-Objekte über jeweils ein Connection-Objekt mit ihm verbunden.

Ein *Terminal-Objekt* repräsentiert im Gegensatz zum Address-Objekt den physischen Endpunkt einer Verbindung – einen Telefonapparat, ein Fax, eine Interface-Karte, etc. Das *TerminalConnection-Objekt* assoziiert nun ein Terminal-Objekt mit einem Call, indem es die Verbindung zwischen Terminal und Connection-Objekt herstellt. `Terminal` und `Address` bilden in ihrer Einheit ein physisches Gerät samt seiner Adresse ab. Durch die Trennung beider Aspekte können komplexere Konfigurationen abgebildet werden, in denen ein Terminal über mehrere Telefonnummern verfügt oder mehrere Terminals sich eine Rufnummer teilen. Abbildung 6-7 verdeutlicht dieses Modell am Beispiel eines Calls mit zwei Teilnehmern.

[24] genauer: ein Objekt einer Klasse, die das Call-Interface implementiert

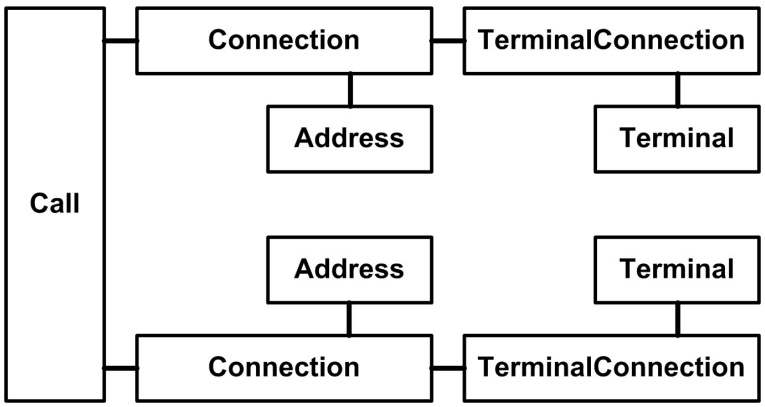

Abb. 6-7: Call-Modell eines Telefongesprächs mit zwei Teilnehmern

Ein Provider ermöglicht den Zugriff auf alle Objekte seiner Domain. Hierfür werden Referenzen auf alle Address-, Terminal- und Call-Objekte gehalten. Der Provider, die Adressen und Terminals sind unveränderliche Objekte, die während der Objektlebensdauer des Providers weder zerstört noch erzeugt werden. Diese statischen[25] Objekte werden über die dynamischen Objekte `Call`, `Connection` und `TerminalConnection` miteinander in Beziehung gebracht.

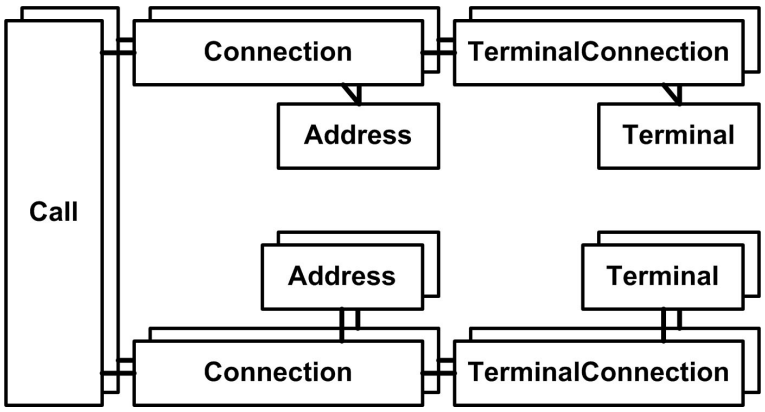

Abb. 6-8: Call-Modell zweier Telefongespräche mit einem gemeinsamen Teilnehmer

Um die Funktionsweise des Call-Modells zu verdeutlichen, folgen zwei weitere Abbildungen komplexerer Situationen. Die Abbildung 6-8 zeigt einen Teilnehmer, der zusätzlich zu einem normalen Telefongespräch noch eine Konsultation durchführt. Das zuerst begonnene Gespräch musste hierfür gehalten werden, um eine freie Leitung für das zweite zu schaffen. Beide Gespräche können dann auf Wunsch zu einem vereint werden, was zu einer Konferenzschaltung führen würde. Eine solche zeigt die Abbildung 6-9.

[25] Damit sind nicht die Objektmodifizierer `static` und `final` gemeint, sondern deren Unveränderlichkeit im Sinne des JTAPI-Modells.

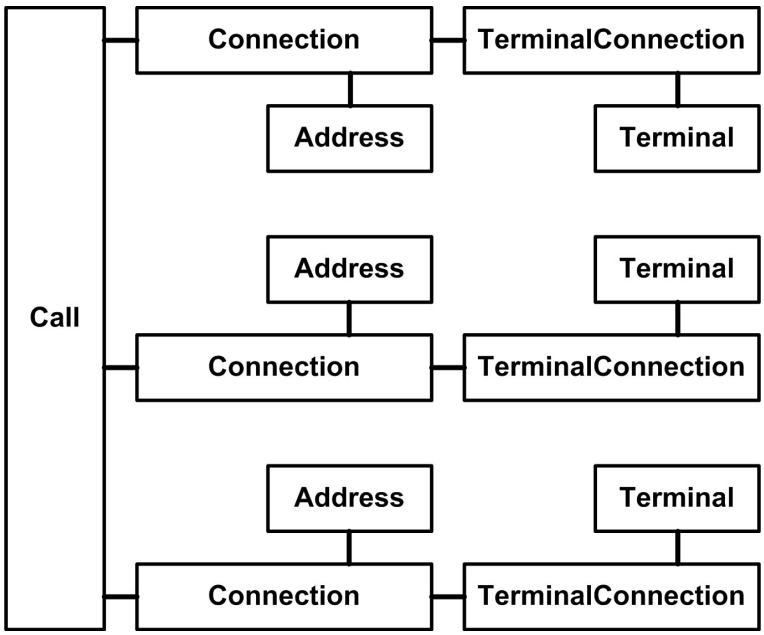

Abb. 6-9: Call-Modell einer Telefonkonferenz mit drei Teilnehmern

6.3.2.1 Provider

Im Folgenden wird zur Vereinfachung davon ausgegangen, dass das darunter liegende Telefoniesystem im Wesentlichen aus einer Telekommunikationsanlage besteht. Gleichwohl lassen sich die nachfolgenden Betrachtungen auch auf andere Rahmenbedingungen übertragen.

Das Provider-Objekt ermöglicht der Anwendung den Zugriff auf die Objekte des Zustandsmodells, welche die aktuelle Situation in der Domain der Telefonanlage abbilden. Zu diesem Zweck sind die Methoden

- `getCalls`,
- `getAddress`,
- `getAddresses`,
- `getTerminal` und
- `getTerminals`

definiert, um Objektreferenzen auf aktive Calls, Adressen und Terminals zurückzugeben. Mit den Methoden `getState` kann der Status des Provider-Objekts in Erfahrung gebracht werden. Es kann sich nur in einem der Zustände `OUT OF SERVICE`, `IN SERVICE` und `SHUTDOWN` befinden. Nach dessen Erzeugung befindet sich ein Provider im Zustand `OUT OF SERVICE`. Wenn alle Initialisierungsaufgaben erfolgreich abgeschlossen wurden (Aufbau der Netzwerkverbindung zur PBX, Erzeugung der Objektstrukturen, etc.), wird dessen Betriebsbereitschaft durch den Zustand `IN SERVICE` bestätigt. Der Zustand kann durch bestimmte interne Ereignisse, meist bei Fehlern, wieder auf `OUT OF SERVICE` zurückfallen.

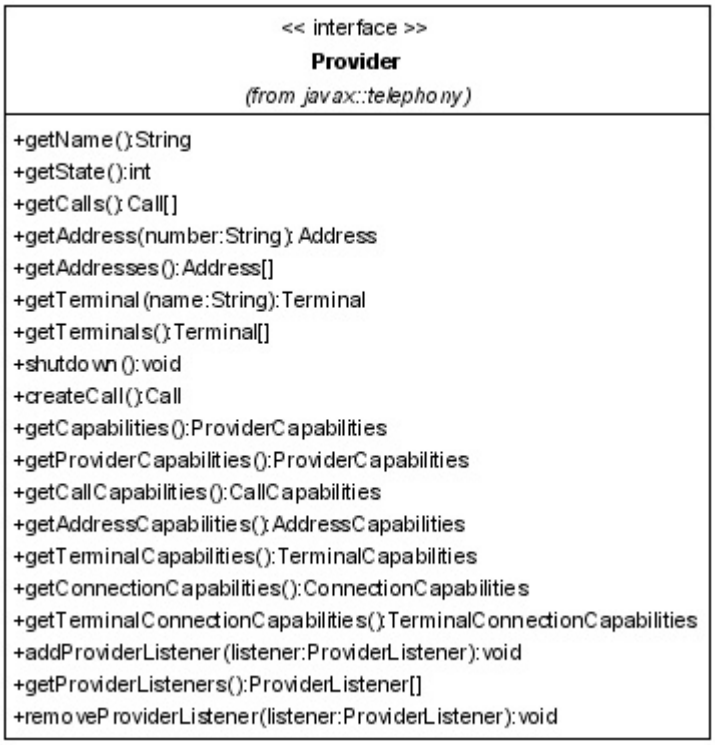

Abb. 6-10: Das Provider-Interface

Wird der Provider deaktiviert, entweder durch Aufruf der Methode `shutdown` oder bei Auftreten fataler Fehler, wechselt sein Status endgültig zu `SHUTDOWN`. Ein Wechsel auf einen der vorherigen Zustände ist von `SHUTDOWN` aus nicht mehr möglich. Abbildung 6-11 verdeutlicht dieses Zustandsmodell. Die möglichen Zustände werden dabei durch die abgerundeten Rechtecke repräsentiert, während gerichtete Kanten die möglichen Zustandsübergänge zeigen.

Neben einer Reihe von Methoden zur Abfrage statischer und dynamischer Capabilities und Methoden zur Registrierung sowie Deregistrierung von Listenern ist die Methode `createCall` Teil der Schnittstelle. Diese erzeugt für die Anwendung neue Call-Objekte, was für den Aufbau neuer Verbindungen erforderlich ist (vgl. Listing 6-8 auf Seite 66).

6.3.2.2 Call

Call-Objekte sind im Zustandsmodell von zentraler Bedeutung, da mit ihrer Hilfe die dynamischen Beziehungen zwischen den statischen Objekten `Provider`, `Address` und `Terminal` hergestellt werden. Hierfür hält ein Call-Objekt Referenzen auf alle Connection-Objekte der beteiligten Parteien sowie auf das Provider-Objekt, in dessen Zuständigkeitsbereich es sich befindet.

Sofort nach dessen Erzeugung ist der Status eines Calls `IDLE`. Sobald das erste Connection-Objekt die Verbindung zu einem Address- und Terminal-Objekt hergestellt hat, wechselt der Status auf `ACTIVE`. Wenn im Gegenzug keine aktiven Connections mehr existieren, wechselt der Status auf `INVALID`.

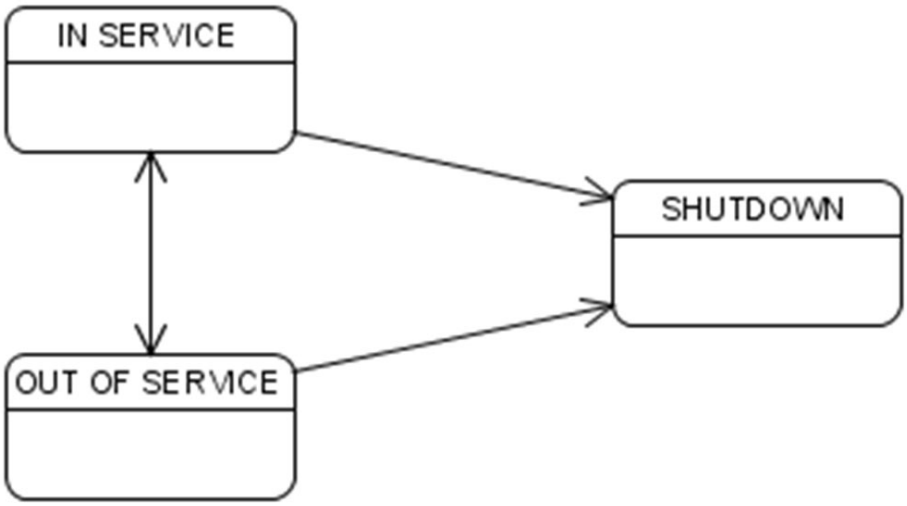

Abb. 6-11: Provider Zustandsdiagramm (nach [JTAP02@])

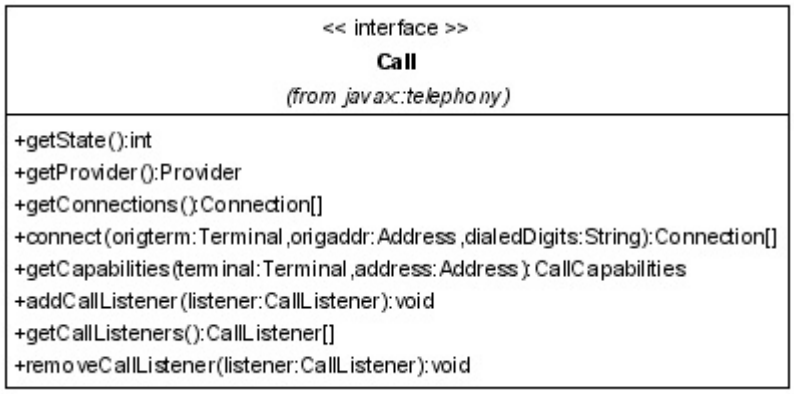

Abb. 6-12: Das Call-Interface

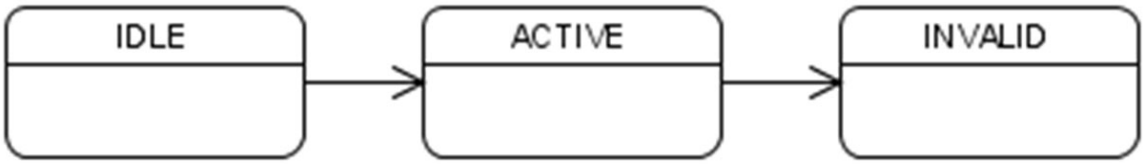

Abb. 6-13: Call Zustandsdiagramm (nach [JTAP02@])

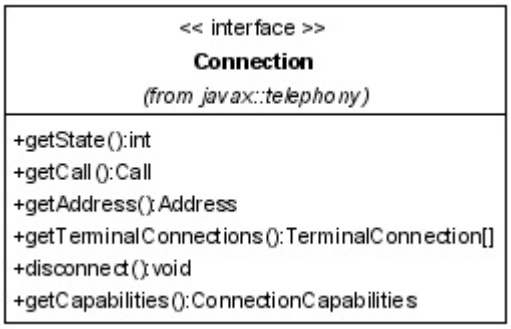

Abb. 6-14: Das Connection-Interface

Neben den get-Methoden für den Objekt-Status, die assoziierten Connections und den Provider verfügt das Call-Objekt über Methoden zur Abwicklung der Listener-Verwaltung und die Abfrage der Capabilities. Aus funktionaler Sicht steht die Methode `connect` im Mittelpunkt, da deren Aufruf, bei einem durch die Methode `Provider.createCall` erzeugten Call-Objekt, zum Verbindungsaufbau zwischen zwei Parteien führt. Als Parameter werden zwei Objekte des Typs `Address` und `Terminal` als Verursacher sowie ein String, der die Adresse des Ziels spezifiziert, akzeptiert. Das Lisiting 6-8 zeigt ein Code-Beispiel zum Aufbau einer Verbindung zwischen zwei Apparaten mit den Rufnummern 4000 und 4001.

```
String numberToDial = "4001";
Address origAddress =
        theProvider.getAddress("4000"); //DN
Terminal origTerminal =
        theProvider.getTerminal("3934223"); //TN
Call newCall = theProvider.createCall();
newCall.connect(origTerminal, origAddress, numberToDial);
}
```

Prog. 6-8: Rufaufbau mit Call.connect()

6.3.2.3 Connection

Das Connection-Objekt verbindet ein Address-Objekt mit einem Call und einem TerminalConnection-Objekt, welches wiederum das beteiligte Terminal referenziert. Unter funktionalen Aspekten ist hier die Methode `disconnect` von Bedeutung. Durch Aufruf von `disconnect` wird die Teilnahme an einem Call beendet, in dessen Folge ein Apparat die Verbindung mit einem Gespräch terminiert. Die Schnittstelle `Connection` verfügt zudem über get-Methoden für den Objektstatus, die Capabilities und die verbundenen Address-, TerminalConnection- und Call-Objekte. Das Zustandsmodell fällt beim Connection-Objekt unter den Objekten des Core-Pakets am umfangreichsten aus. Ein Connection-Objekt kann sich in den Zuständen

Zustand	Bedeutung
IDLE	Initialstatus, keine Aktivität
INPROGRESS	Verbindungsaufbau in Vorbereitung
ALERTING	Terminal wird über eingehendes Gespräch benachrichtigt
CONNECTED	Verbindung aufgebaut
DISCONNECTED	Verbindung beendet
FAILED	Verbindung fehlgeschlagen
UNKNOWN	Zustand momentan unbekannt

Tab. 6-4: Zustände des Connection-Objekts

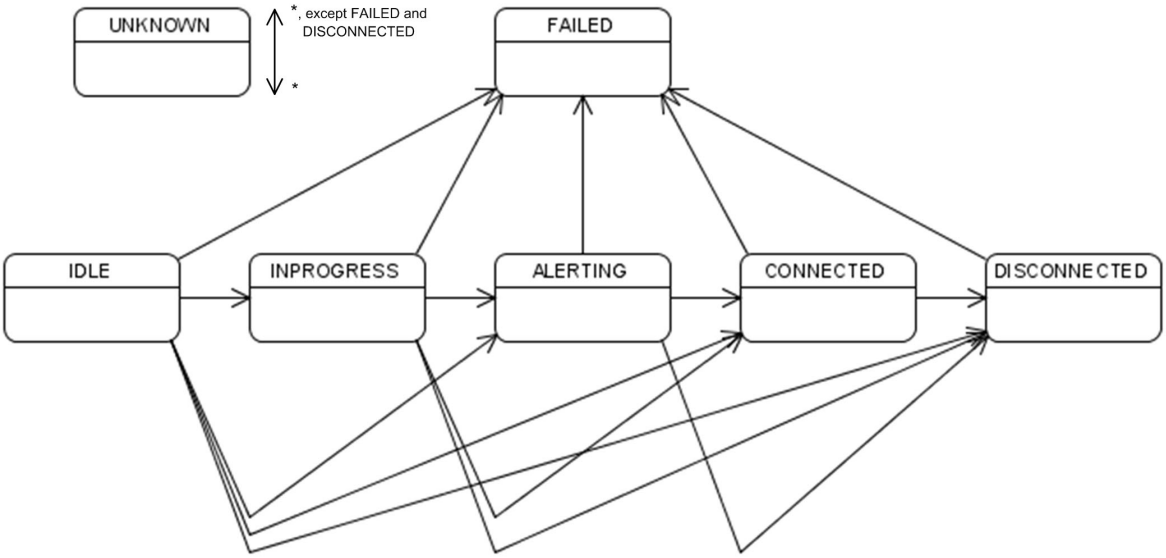

Abb. 6-15: Connection Zustandsdiagramm (nach [JTAP02@])

- `IDLE`,
- `INPROGRESS`,
- `ALERTING`,
- `CONNECTED`,
- `DISCONNECTED`,
- `FAILED` und
- `UNKNOWN`

befinden. Während die folgende Tabelle 6-4 die einzelnen Zustände und ihre Bedeutung zeigt, ergänzt Abbildung 6-15 die möglichen Übergänge.

Mit dem Verweis auf Kapitel 6.3.3 ist anzumerken, dass es nicht vorgesehen ist, Objekte der Typen `Connection` und `TerminalConnection` mittels direkter Registrierung von Listenern zu überwachen. Die Überwachung erfolgt stattdessen indirekt am zugehörigen Call-Objekt. Deshalb verfügen die Schnittstellen `Connection` und `TerminalConnection` über keine entsprechenden Methoden zur Listener-Verwaltung.

6.3.2.4 Address und Terminal

Das Address-Objekt repräsentiert die Adresse eines Teilnehmers, den logischen Endpunkt einer Verbindung. Einer (internen) Adresse ist mindestens ein Terminal als physischer Endpunkt zugeordnet. Aus diesen Gründen verfügen die Schnittstellen `Address` und `Terminal` über entsprechende get-Methoden zur Ermittlung der Zugehörigkeiten. Keine der beiden Schnittstellen verfügt über Methoden zur Umsetzung von Telefoniefunktionen. Ihr Hauptzweck ist es, einer Anwendung die Überwachung bestimmter Teilnehmer zu ermöglichen. Deshalb sind neben den üblichen get-Methoden

- `getName`,
- `getProvider`,
- `getTerminals` beziehungsweise `getAddresses`,
- `getConnections` beziehungsweise `getTerminalConnections` und
- `getCapabilities`

noch Methoden zum Hinzufügen und Entfernen der Listener definiert. Eine Besonderheit ist hierbei die Möglichkeit bei Address- sowie Terminal-Objekten auch Call-Listener registrieren zu können. Dies ermöglicht die Überwachung zukünftiger Calls, an denen das jeweilige Address- beziehungsweise Terminal-Objekt beteiligt sein wird.

6.3.2.5 TerminalConnection

Das TerminalConnection-Objekt stellt die Beziehung eines Terminals zu einem Call her und trägt dabei den Verbindungsstatus. Neben den üblichen get-Methoden enthält das Interface `TerminalConnection` die Methode `answer`. Diese ermöglicht die Annahme eines am Terminal eingehenden Anrufs. Die möglichen Zustände und deren Übergänge werden in Tabelle 6-5 und Abbildung 6-19 dargestellt.

6.3.3 Event/Listener-Modell

Von elementarer Bedeutung für ein Modell, das versucht ein ereignisorientiertes System abzubilden, ist die Benachrichtigung bei Zustandsänderungen. Eine Möglichkeit stellt das *pull-Prinzip* dar. Hierfür rufen Objekte, die am Status anderer interessiert sind, zu bestimmten Zeitpunkten die Statusinformationen ab. Ein Nachteil ergibt sich bei zeitkritischen Anwendungen. Denn treten Statusänderungen kurz nach der letzten Abfrage auf, werden diese erst

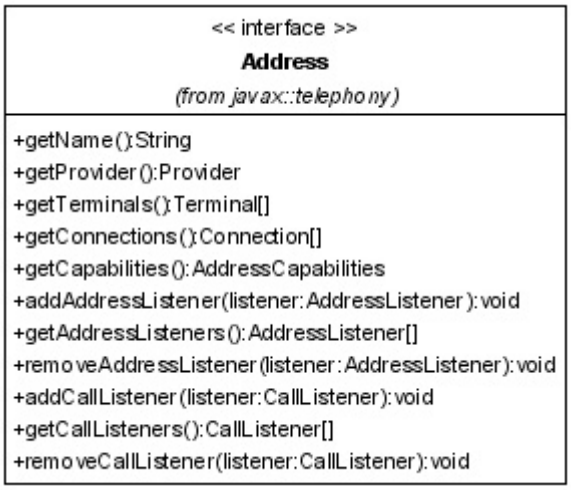

Abb. 6-16: Das Address-Interface

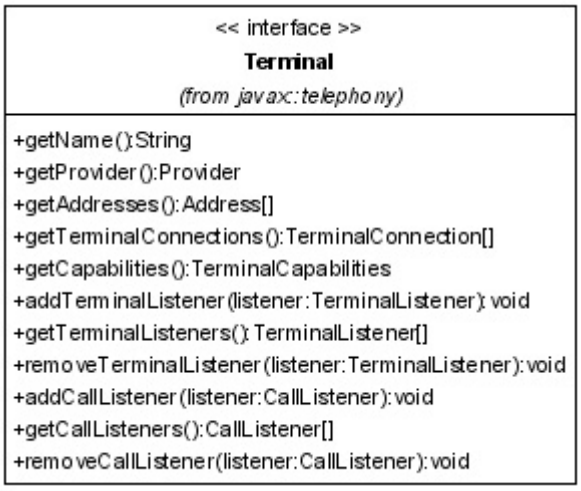

Abb. 6-17: Das Terminal-Interface

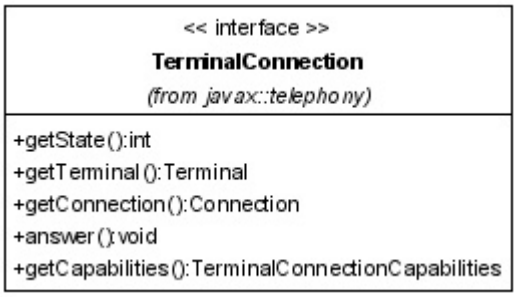

Abb. 6-18: Das TerminalConnection-Interface

Zustand	Bedeutung
IDLE	Initialstatus, keine Aktivität
RINGING	Terminal klingelt (Teilnehmer benachrichtigt)
ACTIVE	Terminal aktiv (Teilnehmer spricht)
PASSIVE	Terminal passiviert
DROPPED	Terminal hat Gespräch verlassen
UNKNOWN	Zustand momentan unbekannt

Tab. 6-5: Zustände des TerminalConnection-Objekts

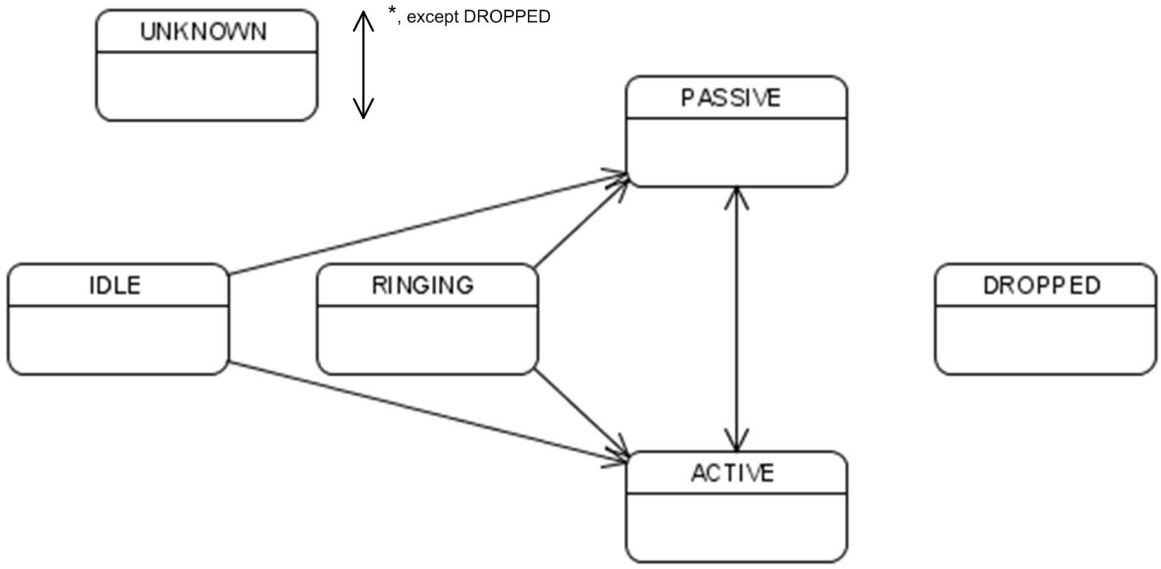

Abb. 6-19: TerminalConnection Zustandsdiagramm (nach [JTAP02@])

beim nächsten Mal erkannt. Überdies finden viele Abfragen unnötigerweise statt, wenn der Status die meiste Zeit unverändert bleibt, sodass sich bei einer hohen Anzahl an Interessenten der Overhead erheblich auf die Systemleistung auswirken kann.

Um diesen Nachteilen entgegenzuwirken, sieht JTAPI zudem die Kommunikation von Statusänderungen gemäß dem Observer-Muster vor. Das nach dem Hollywood-Prinzip[26] aufgebaute Verhaltensmuster geht davon aus, dass sich Interessenten beim betreffenden Objekt als Observer registrieren. Ein Observer verfügt über eine Schnittstelle, die eine oder mehrere Ereignismethoden definiert. Tritt nun ein Ereignis ein, dass den Zustand des überwachten Objekts verändert, wird die entsprechende Ereignismethode jedes einzelnen registrierten Observers aufgerufen. Somit wird ein abhängiges Objekt im Idealfall bei Eintritt über eine Zustandsänderung informiert. Die Zahl der registrierbaren Objekte kann je nach Implementierung und verfolgter Zielsetzung variieren.

Wie bereits beschrieben, wird in der JTAPI-Spezifikation zwischen Observern und Listenern unterschieden. Da das Observer-Prinzip als veraltet definiert wurde, soll im Folgenden deshalb nur auf das Listener-Modell eingegangen werden. Für jedes der Bausteine des Call-Modells existiert jeweils ein Event- und ein Listener-Interface. So sind die Schnittstellen

- `ProviderListener`,
- `CallListener`,
- `ConnectionListener`,
- `TerminalConnectionListener`,
- `AddressListener` und
- `TerminalListener`

definiert, die alle von dem Marker-Interface `EventListener` abgeleitet sind. Während `ProviderListener`, `AddressListener` und `TerminalListener` jeweils nur an Objekten des Typs `Provider`, `Address` und `Terminal` registriert werden können, nehmen die `CallListener` eine Sonderstellung ein (vgl. Abbildung 6-20). Diese lassen sich sowohl an einem Objekt mit der Schnittstelle `Call` als auch an Objekten des Typs `Address` und `Terminal` registrieren. Dies ermöglicht die Überwachung zukünftiger Calls, an denen das jeweilige Address- beziehungsweise Terminal-Objekt beteiligt sein wird. Sobald ein betreffendes Address- oder Terminal-Objekt einem Call beitritt, werden die dort registrierten Call-Listener automatisch an das Call-Objekt durchgereicht.

Die Listener für Connection- und TerminalConnection-Objekte sind hingegen nicht zur direkten Registrierung gedacht. Die Überwachung erfolgt stattdessen auf indirekte Weise am zugehörigen Call-Objekt. Dies wird durch den Umstand ermöglicht, dass es sich bei beiden Listenern um Ableitungen des CallListener-Interface handelt, die dementsprechend ebenfalls an Call-, Address- und Terminal-Objekten registriert werden können. Diese Designentscheidung kann durch die stets enge Beziehung zwischen *einem* Call-Objekt und seinen Connection-

[26] „Don't call us, we call you!"

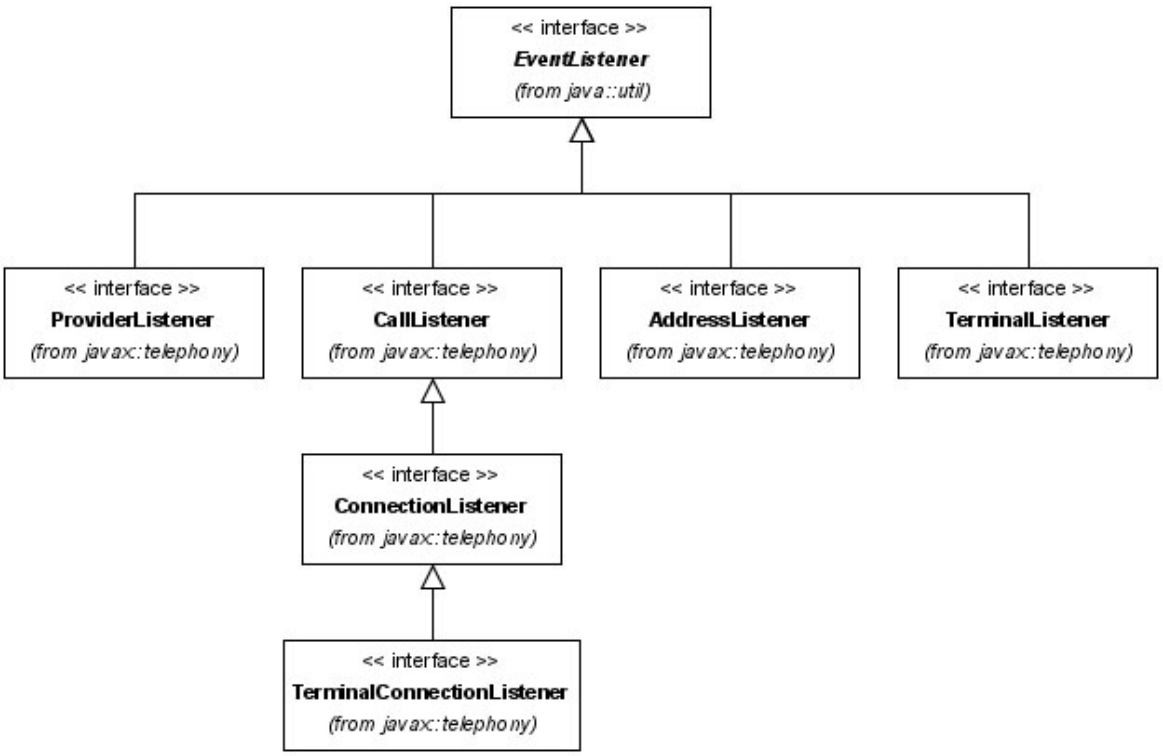

Abb. 6-20: Hierarchie der Listener-Schnittstellen

und TerminalConnection-Objekten begründet werden. Beide werden beim Aufbau eines Calls gemäß der beteiligten Terminals und Adressen erzeugt und überdauern nie dessen Lebenszeit. Da sich der Gesamtzustand eines Calls über die Teilzustände der Verbindungen aller Teilnehmer definiert, werden bei der Call-Überwachung auch Ereignisse mitgeteilt, die Connection- und TerminalConnection-Objekte betreffen. Durch die Wahl des zu registrierenden Listeners wird entschieden, ob

- alle Ereignisse (`TerminalConnectionListener`),
- nur Call- und Connection-Ereignisse (`ConnectionListener`) oder
- nur Call-Ereignisse (`CallListener`)

übermittelt werden sollen. Die Ereignismethoden eines jeden Listeners orientieren sich an den möglichen Objektzuständen. So definiert das Interface `ProviderListener` analog zu den möglichen Zuständen `OUT OF SERVICE`, `IN SERVICE` und `SHUTDOWN` die Methoden

- `providerInService`,
- `providerOutOfService` und
- `providerShutdown`.

Zusätzlich enthalten die Listener noch Methoden für den Fall, dass die Überwachung beendet oder ein Objekt[27] erzeugt wurde. Jede Ereignismethode akzeptiert einen Übergabeparameter, der den entsprechenden Ereignistyp aufnimmt. Hierfür sind wiederum pro Element des Call-Modells die Schnittstellen

- `ProviderEvent`,
- `CallEvent`,
- `ConnectionEvent`,
- `TerminalConnectionEvent`,
- `AddressEvent` und
- `TermnalEvent`

definiert (vgl. Abbildung 6-21). Diese Schnittstellen beschreiben Event-Objekte, die nähere Informationen über die Art des Ereignisses, das betroffene Objekt und die Ursache preisgeben. Neben diesen Ereignissen existieren mit `SingleCallMetaEvent` und `MultiCallMetaEvent` noch gesonderte Meta-Events (vgl. Abbildung 6-22). Diese können dazu benutzt werden, eine Abfolge von Ereignissen, ein oder mehrere Call-Objekte betreffend, in einen Zusammenhang zu stellen.[28]

6.3.4 Das Call-Control-Paket

Nachdem die Aspekte des Core-Pakets behandelt wurden, folgt nun die Beschreibung des Call-Control-Pakets. Das Paket erweitert das Grundmodell um spezialisierte, abgeleitete Varianten der Basisschnittstellen. Mit Ausnahme des Providers wurden alle Elemente des Call-Modells, deren Listener- und Event-Schnittstellen erweitert. Der Name wurde durch das Prefix `CallControl` gebildet. Damit umfasst das Call-Control-Paket die Schnittstellen

- `CallControlCall`, `CallControlCallEvent` und `CallControlCallListener`,
- `CallControlConnection`, `CallControlConnectionEvent` und `CallControlConnectionListener`,
- `CallControlTerminalConnection`, `CallControlTerminalConnectionEvent` und `CallControlTerminalConnectionListener`,
- `CallControlAddress`, `CallControlAddressEvent` und `CallControlAddressListener` sowie

[27] nur bei den Verbindungsobjekten des Typs `Connection` und `TerminalConnection`

[28] Da deren Bedeutung nicht völlig eindeutig spezifiziert ist und einen Spielraum bei der Interpretation lässt, wurde von deren Einsatz Abstand genommen.

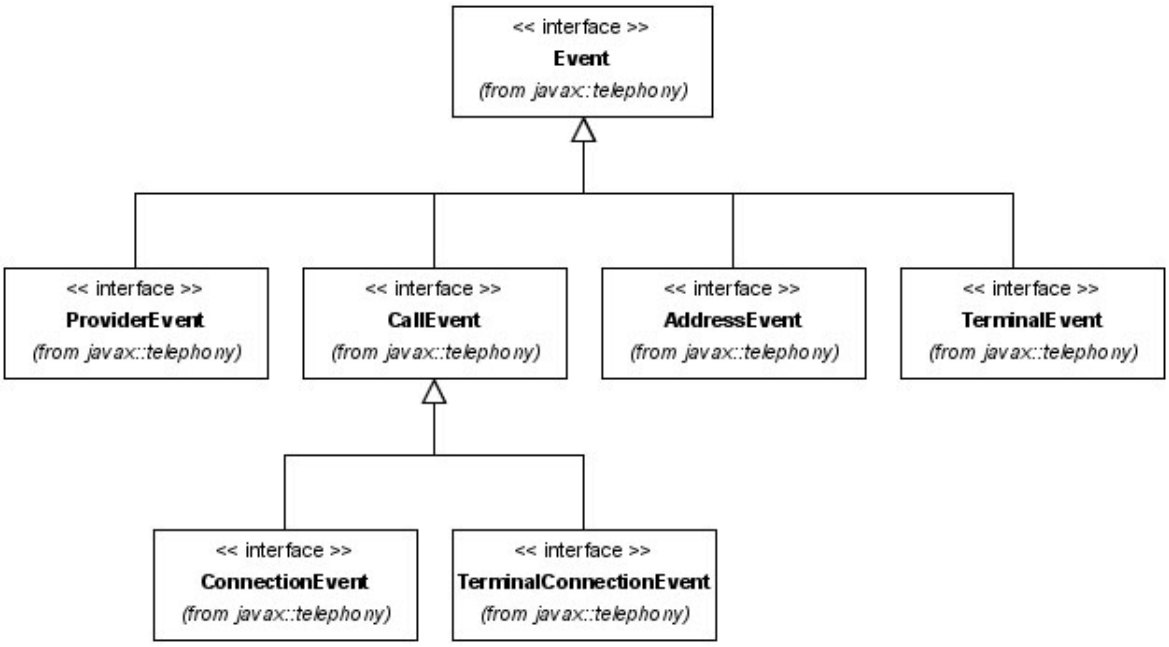

Abb. 6-21: Hierarchie der Event-Schnittstellen

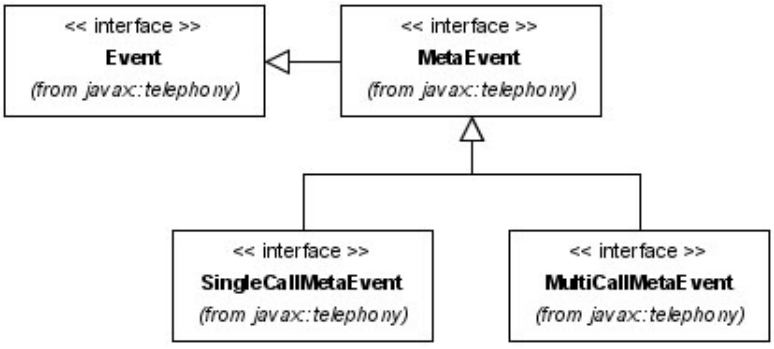

Abb. 6-22: Hierarchie der MetaEvent-Schnittstellen

- `CallControlTerminal`, `CallControlTerminalEvent` und `CallControlTerminalListener`.

Die eingeführten Erweiterungen erstrecken sich auf zwei Bereiche. Zum einen wurden die Schnittstellen des Call-Modells um funktionale Fähigkeiten erweitert, zum anderen wurden die Verbindungsobjekte `CallControlConnection` und `CallControlTerminalConnection` mit einem detaillierteren Zustandsmodell versehen.

Die Erweiterungen der Call-Schnittstelle befassen sich im Wesentlichen mit dem Schalten von Konferenzen und Transferieren von Teilnehmern. So können mit der Methode `consult` Konsultationsgespräche initiiert werden, die später als Transferziel dienen oder einer Konferenzschaltung hinzugefügt werden können. Deshalb sind die Methoden `setTransferEnable` und `setConferenceEnable` definiert, die den Zweck der Konsultation bestimmen. Durch Aufruf der Methoden `conference` respektive `transfer` wird das Konsultationsgespräch mit einem bestehenden Gespräch verschmolzen beziehungsweise vermittelt. Zudem sind mit einer überladenen Variante von `transfer` und der Methode `addParty` so genannte Single-Step-Transfers und -Konferenzen implementierbar. Mittels der Methoden `setTransferController` und `setConferenceController` kann die Partei bestimmt werden, welche die Kontrolle bei der Durchführung von Transfers und Konferenzen übernimmt. Zuletzt ist mit `drop` eine Möglichkeit vorgesehen, einen ganzen Call mitsamt seiner Verbindungen zu den Teilnehmern zu beenden. An den Listener- und Event-Schnittstellen ändert sich aufgrund des gleichbleibenden Zustandsmodells nichts von größerer Tragweite.

Mit `CallControlAddress` werden Methoden hinzugefügt, die das Setzen, Abfragen und Löschen von Weiterleitungsregeln, das Versenden und Empfangen von Nachrichten sowie die *Do-Not-Disturb-Funktion (DND)* ermöglichen. Letztere erlaubt es einem Teilnehmer auf Wunsch zeitweise ungestört zu bleiben, indem eingehende Anrufe nicht mehr zu seinem Apparat durchgestellt werden. Die korrespondierenden Interfaces für Listener und Events beziehen sich gleichsam auf Änderungen in diesen drei Bereichen.

Die Schnittstelle `CallControlTerminal` führt neben einer Terminal-bezogenen DND-Funktion die *Pickup-Funktionalität* ein. Darunter versteht man das Annehmen eines für einen anderen Teilnehmer bestimmten Anrufes. Da Terminals in Gruppen zusammengefasst sein können, die das Pickup-Verhalten reglementieren, wird zwischen den Methoden `pickup` und `pickupFromGroup` unterschieden.

Die Verbindungsobjekte werden durch das Call-Control-Paket zum Zweck genauerer Manipulationsmöglichkeiten um ein feineres Zustandsmodell erweitert. Das Interface `CallControlConnection` führt neue Zustände ein, welche das Unterscheiden verschiedener Entwicklungsverläufe beim Verbindungsaufbau ermöglichen. Neben dem herkömmlichen Weg – Hörer abnehmen, Nummer wählen und auf das Entgegennehmen durch die Gegenstelle warten – wird die Betrachtung auf Netzwerksegmente erweitert. Die Status `NETWORK_REACHED` und `NETWORK_ALERTING` drücken deshalb Ereignisse aus, die ein anderes Netzwerk (außerhalb der Domain der PBX) betreffen. Dies kann beispielsweise bei der Betrachtung von Telefongesprächen mit externen Gesprächsteilnehmern sinnvoll sein, da im Gegensatz zu lokalen Teilnehmern nur eingeschränkt Statusinformationen zur Verfügung stehen. Die Zustände `QUEUED` und `OFFERED` ermöglichen im Zusammenhang mit den Methoden `park`, `accept` und `reject` die Annahme eines Anrufes zu beeinflussen, noch bevor das Termi-

Zustand	Pendant im Core	Bedeutung
IDLE	IDLE	Initialstatus, keine Aktivität
QUEUED	INPROGRESS	Verbindung wurde in Warteschlange eingereiht
OFFERED	INPROGRESS	Verbindungsaufbau wird angeboten
ALERTING	ALERTING	Terminal wird über eingehendes Gespräch benachrichtigt
INITIATED	CONNECTED	Verursacher aktiv, beginnt Verbindungsaufbau zur Gegenstelle
NETWORK_REACHED	CONNECTED	fremdes Netzwerk wurde erreicht
NETWORK_ALERTING	CONNECTED	Gegenstelle im fremden Netzwerk wird über eingehende Verbindung benachrichtigt
ESTABLISHED	CONNECTED	Verbindung aufgebaut
DISCONNECTED	DISCONNECTED	Verbindung beendet
FAILED	FAILED	Verbindung fehlgeschlagen
UNKNOWN	UNKNOWN	Zustand momentan unbekannt

Tab. 6-6: Zustände des CallControlConnection-Objekts

nal benachrichtigt wird. Mit `park` können Anrufe in eine Warteschlange eingereiht werden, wobei das CallcontrolConnection-Objekt in den Zustand `QUEUED` wechselt. Die beiden anderen Methoden ermöglichen das Annehmen beziehungsweise Ablehnen von Verbindungen im Status `OFFERED`. Die Tabelle 6-6 zeigt die neuen Zustände und ihre Entsprechungen im Core-Modell[29].

Zustand	Pendant im Core	Bedeutung
IDLE	IDLE	Initialstatus, keine Aktivität
RINGING	RINGING	Terminal klingelt (Teilnehmer benachrichtigt)
TALKING	ACTIVE	Teilnehmer spricht
HELD	ACTIVE	Verbindung wird gehalten
BRIDGED	PASSIVE	Terminal passiviert, Teilnahme an Calls möglich
INUSE	PASSIVE	Terminal passiviert, keine Teilnahme an Calls
DROPPED	DROPPED	Terminal hat Gespräch verlassen
UNKNOWN	UNKNOWN	Zustand momentan unbekannt

Tab. 6-7: Zustände des CallControlTerminalConnection-Objekts

[29]Dieses Verhältnis ist für die Kompatibilität zum Core-Modell zwingend erforderlich. So kann beispielsweise ein Listener des Typs `ConnectionListener` an einem Objekt des Typs `CallcontrolConnection` registriert werden.

Die Schnittstelle `CallControlTerminalConnection` erweitert das Zustandsmodell um eine genauere Unterscheidung der Zustände `ACTIVE` in `TALKING` und `HELD` sowie `PASSIVE` in `BRIDGED` und `INUSE`. Diese Unterscheidung ermöglicht den Einsatz der Methoden `hold` und `unhold`, die zur Implementierung der Halte-Funktion den Zustand des Objekts zwischen `HELD` und `TALKING` wechseln. Die Methode `leave` wechselt den Status von `TALKING` auf `BRIDGED`, während `join` das Gegenteil bewirkt. Die Tabelle 6-7 zeigt eine Auflistung der Zustände.

Die möglichen Zustandsübergänge der durch die Schnittstellen `CallControl-Connection` und `CallControlTerminalConnection` beschriebenen Objekte sind im Anhang A.1 dargestellt.

7 Design

Dieses Kapitel beschreibt den Aufbau und die Funktionsweise der entwickelten Software. Das erste Unterkapitel schildert zunächst den Gesamtaufbau und die Zusammenarbeit der einzelnen Komponenten. Darauf folgen Kapitel, die sich mit den Komponenten *CTI-Server*, *Meridian-Treiber*, *JTAPI-Implementierung* und dem *webDial*-Dienst beschäftigen.

7.1 Gesamtarchitektur

Die zu entwickelnde Serverkomponente des CTI-Systems kann als Middleware verstanden werden, die als Vermittlungsebene zwischen dem Telefoniesystemen und den Client-Anwendungen auftritt. Auf der einen Seite müssen Befehle und Statusinformationen mit dem CTI-Link der Telefoniehardware ausgetauscht werden, während zur Client-Seite hin Benutzerinteraktionen und Ereignisse verarbeitet werden müssen. Bezug nehmend auf die strukturellen Anforderungen sind dabei die Austauschbarkeit der Telefoniehardware sowie andere Realisierungsmöglichkeiten bei der Client-Entwicklung zu berücksichtigen. Auf funktionaler Ebene sollen zum einen Erweiterungen des *webDial*-Dienstes als auch Ergänzungen um völlig neue, über die Grenzen der Screen-based Telephony hinausgehende, Anwendungsgebiete realisierbar sein.

Aufgrund dieser Einflussbereiche ist eine komponentenorientierte Lösung der Serverimplementierung zweckmäßig. Als zu entwickelnde Komponenten wurden im Einzelnen identifiziert:

- der CTI-Server mit Admin-Interface,
- der Provider,
- der Treiber und
- der *webDial*-Server mit zugehörigem Client-API.

Die Abbildung 7-1 zeigt eine schematische Darstellung der einzelnen Komponenten des Serversystems im Zusammenspiel mit den Client-Anwendungen und Telekommunikationsanlagen.

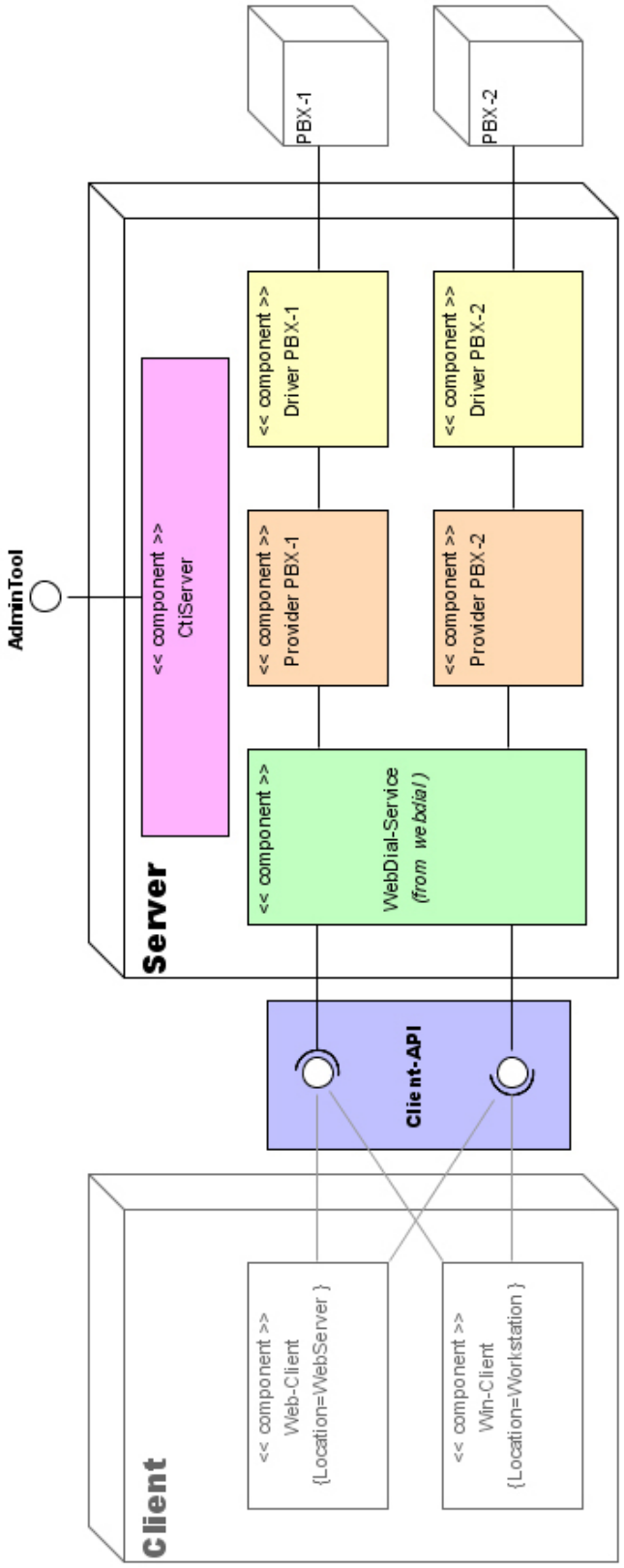

Abb. 7-1: Systemübersicht (implementierte Komponenten farbig hinterlegt)

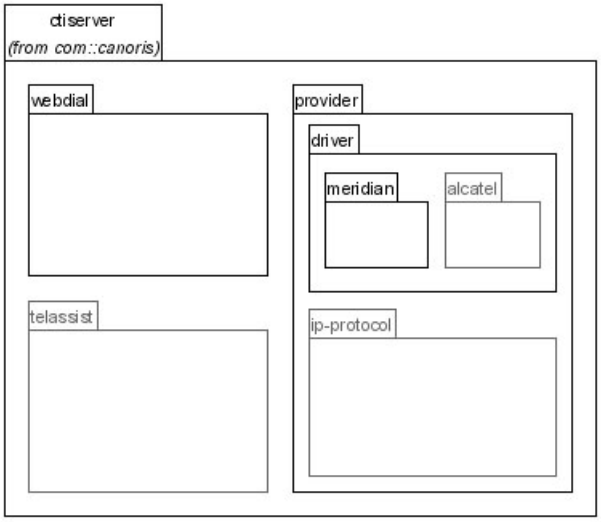

Abb. 7-2: Übersicht über die Paketstruktur unter Andeutung möglicher Erweiterungen (grau)

Die Provider sind für dieses Modell von zentraler Bedeutung. Bei ihnen handelt es sich um Implementierungen eines Call-Modells gemäß der JTAPI-Spezifikation. Ihre Aufgabe ist die Abbildung der aktuellen Situation im Telekommunikationssystem zum Zweck der Überwachung und Manipulation. Jede Provider-Instanz bildet dabei den Zustand einer Telekommunikationsanlage ab. Über die Treiber-Komponente erhält die JTAPI-Implementierung Zugriff auf die CTI-Funktionen der TK-Anlage. Beim Treiber handelt es sich um die Implementierung der Kommunikationslogik auf Basis des in Kapitel 5.3.1 beschriebenen Meridian-Link-Protokolls. Die *webDial*-Komponente implementiert die serverseitige Anwendungslogik des Dienstes und nutzt die JTAPI-Provider für die Umsetzung das Telefoniesystem betreffender Anwendungsfälle. Die Implementierung des *webDial*-API bietet dabei eine Schnittstelle für die Integration von Servlet-basierten[1] und herkömmlichen Clients. Der CTI-Server, die vierte Komponente, startet die einzelnen Dienste, initialisiert die Provider und stellt ein Administrations-Interface für den laufenden Betrieb bereit. Dem CTI-Server ist es möglich, neben *webDial* mehrere andere Dienste, die nebenläufig über die Provider auf Telefoniefunktionen zugreifen können, zu starten und zu administrieren.

Die Implementierung des Servers ist, den Komponenten entsprechend, auf mehrere Java-Pakete verteilt. Das Hauptpaket `com.canoris.ctiserver` enthält die Implementierung des CTI-Servers, das Unterpaket `webdial` die Implementierung des *webDial*-Dienstes und das Client-API. `com.canoris.ctiserver.provider` enthält die JTAPI-Implementierung sowie als Unterpaket den Treiber für die *Meridian I* TK-Anlage (vgl. Abbildung 7-2).

[1] Java-Technologie für die Erstellung dynamischer Web-Anwendungen

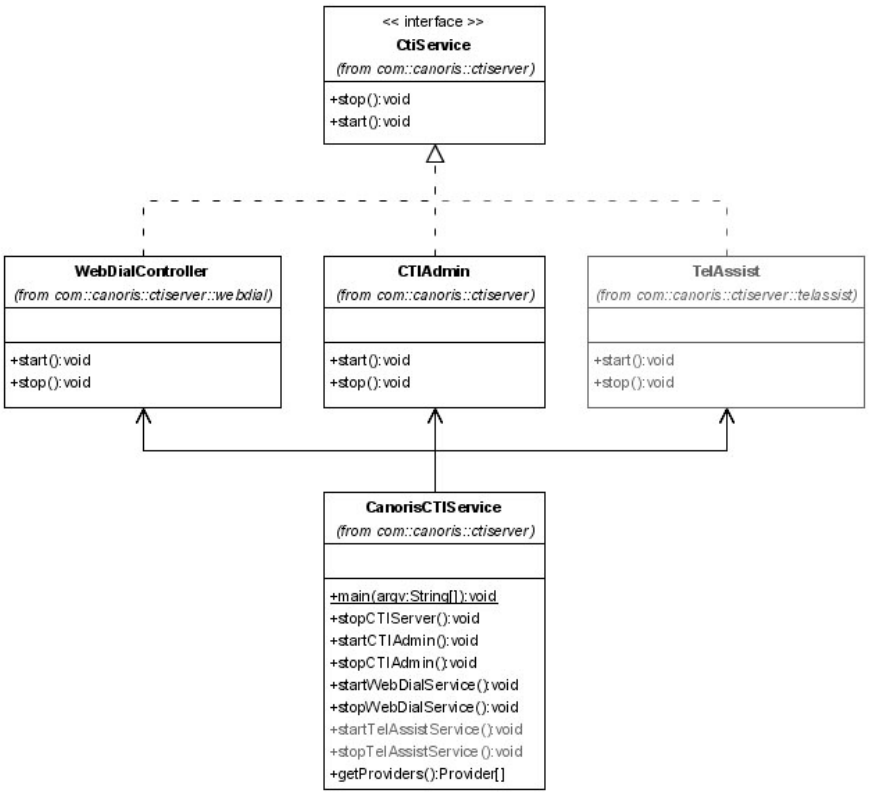

Abb. 7-3: Das Interface CTIService

7.2 Der CTI-Server

Diese Komponente ist für den Start des Server-Prozesses zuständig und enthält aufgrund dessen die Klasse `CanorisCTIService` mit der main-Methode als Einstiegspunkt für den Java-Interpreter. Neben der Initialisierung der Logging-Funktionen (siehe Kapitel 8.3) und dem Einlesen der Konfigurationsdaten werden in dieser Klasse die Provider initialisiert und die verfügbaren Dienste gestartet. Bei den Diensten handelt es sich um eigenständige Threads zur Erfüllung bestimmter Aufgaben. Zur Zeit existieren mit *webDial* und dem Admin-Interface zwei solcher Dienste. Alle Dienste müssen das Interface `CtiService` implementieren. Momentan enthält das Interface lediglich die beiden Methoden `start` und `stop` für das Starten und Beenden eines Dienstes. Erweiterungen der Schnittstelle um Methoden zur Abfrage von Statusinformationen, wie beispielsweise der Anzahl aktiver Benutzer-Sessions, sind realisierbar. Abbildung 7-3 zeigt das Interface und die beiden implementierenden Klassen des Admin-Tools und des *webDial*-Dienstes. Mit `TelAssist` ist eine weitere, zukünftige Implementierung angedeutet, die das Angebot an CTI-Diensten um eine automatisierte, zeitabhängige Konfiguration von Rufumleitungen erweitert.

Die Klasse `CTIAdmin` implementiert die Funktionen der Administrationsschnittstelle `CTIServerAdminInterface` zum Beenden des ganzen Servers, sowie zur Steuerung der einzelnen Dienste. Bei `CTIServerAdminInterface` handelt es sich um ein RMI-

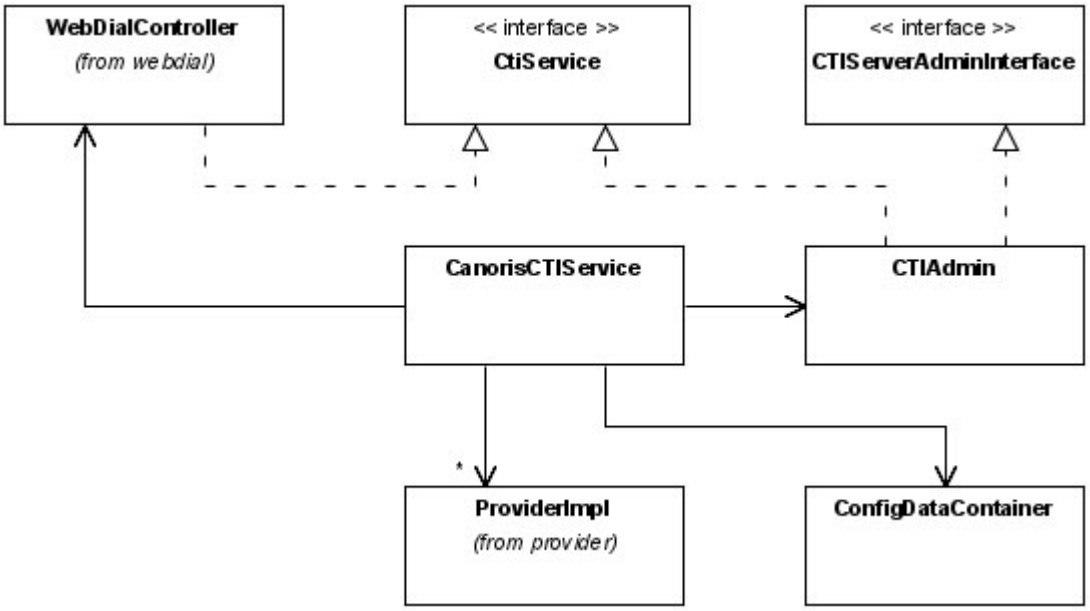

Abb. 7-4: Klassenstruktur des CTI-Servers

Remote-Interface, das von dem Kommandozeilenprogramm `CTIServerAdminTool` für die Steuerung des Serverprozesses verwendet wird.

7.3 Der Meridian-Treiber

Der Treiber für die Telekommunikationsanlage des Typs *Meridian I* stellt das Bindeglied zwischen JTAPI-Implementierung und Telekommunikationshardware dar. Hierfür wird eine Socket-basierte TCP/IP-Verbindung aufgebaut, über welche die Nachrichten des Link-Protokolls ausgetauscht werden.

Die Kommunikation mit dem Provider wird durch zwei Interfaces definiert. Das Interface `PBXDriver` des übergeordneten Pakets `com.canoris.ctiserver.driver` definiert die Methoden für den Aufruf durch den Provider. Die Schnittstelle wird durch die Klasse `MeridianDriver` des Pakets `com.canoris.ctiserver.driver.meridian` implementiert. Die Schnittstelle soll für die Integration weiterer Telefonanlagen die Kompatibilität der JTAPI-Implementierung gewährleisten, wenngleich bei zu sehr voneinander abweichenden CTI-Link-Protokollen eine Anpassung des Providers unvermeidbar wird. Die Schnittstelle definiert Methoden für

- den Auf- und Abbau der Netzwerkverbindung,
- die Registrierung der zu überwachenden Endgeräte,
- das Schalten von einfachen Anrufen sowie Konsultationsgesprächen,
- das Zusammenführen von Gesprächen zu Konferenzschaltungen,

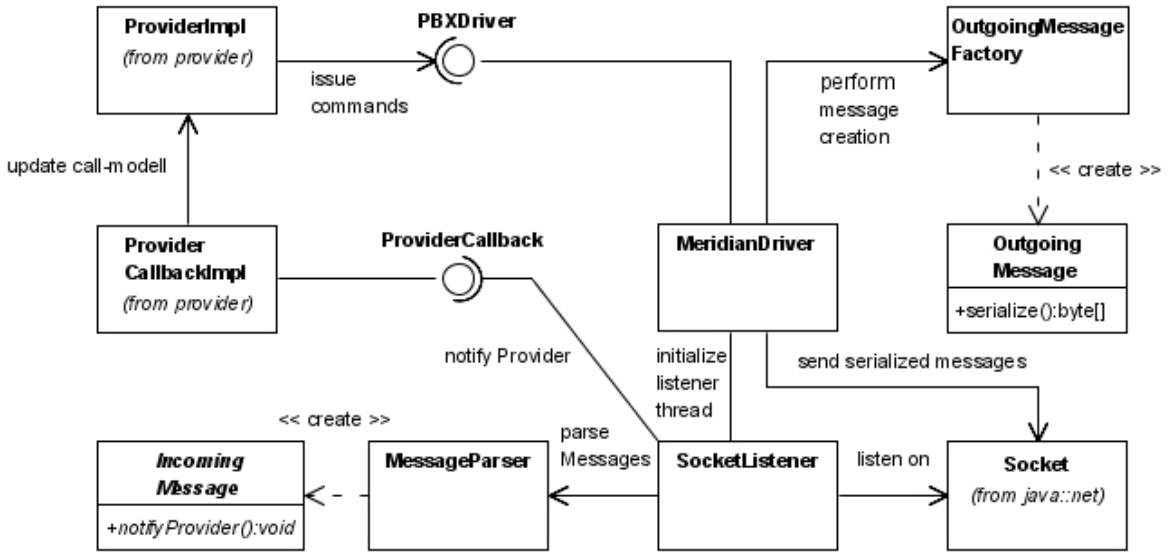

Abb. 7-5: Klassenstruktur des Meridian-Treibers

- das Weiterverbinden von Teilnehmern sowie
- das Wechseln und Beenden aktiver Gespräche.

Das zweite Interface `ProviderCallback` ist durch den Provider zu implementieren. Dabei handelt es sich um einen Listener, der mittels mehrerer Ereignismethoden über Nachrichten von der Telekommunikationsanlage informiert wird. Der Provider muss zu diesem Zweck das Callback-Objekt bei der Initialisierung des Treibers registrieren. Da es sich hier um eine 1-zu-1-Beziehung handelt, ist auch nur die Registrierung eines Listeners möglich. Die Ereignismethoden informieren über den Erfolg- oder Misserfolg bei der Registrierung und Deregistrierung der CTI-Anwendung sowie bei der Registrierung zu überwachender Endgeräte. Statusänderungen registrierter Apparate übermittelt die Methode `deviceStatusChange`.

Mit diesen beiden Interfaces – das eine durch den Treiber, das andere durch den Provider implementiert – wird die bidirektionale Kommunikation bei asynchronem Nachrichtenaustausch ermöglicht. Denn das Ergebnis eines Kommandos wird im Meridian-Link-Protokoll, abgesehen von wenigen Ausnahmen, nicht durch eine explizite Nachricht quittiert, sondern durch eine entsprechende Änderung im Zustand der betroffenen Geräte, die mit zeitlicher Verzögerung eintreten kann.

Die Klasse `MeridianDriver` implementiert das Interface `PBXDriver` und erzeugt je nach aufgerufener Methode eine Nachricht, serialisiert diese zu einem Byte-Vektor und sendet sie über die Socket-Verbindung an die PBX. Abbildung 7-8 zeigt die Oberklasse `OutgoingMessage` und die von ihr abgeleiteten Klassen zur Repräsentation der einzelnen Protokollnachrichten. Die Klasse `OutgoingMessageFactory` wird als Fabrik zur Erzeugung der einzelnen Nachrichtenobjekte eingesetzt. Die Klasse verfügt dafür über eine Sammlung von virtuellen, parametrisierten Konstruktoren zur Erzeugung der verschiedenen Objekte. Die Klassen zur Darstellung der zu versendenden Nachrichten überschreiben die Methode

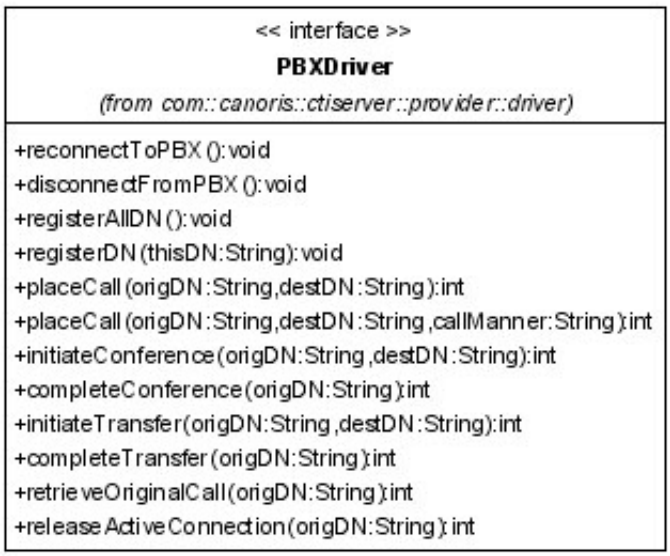

Abb. 7-6: Die Schnittstelle PBXDriver

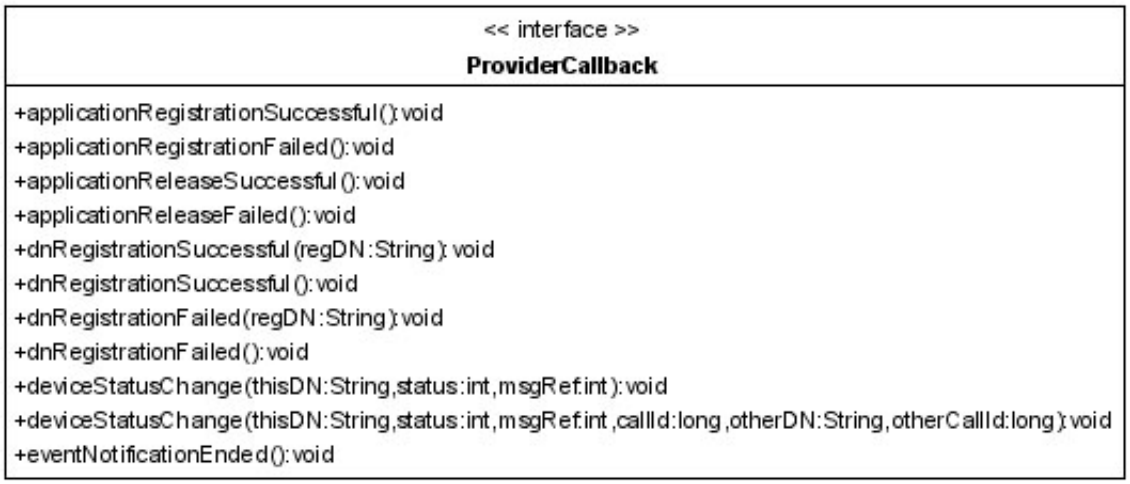

Abb. 7-7: Die Schnittstelle ProviderCallback

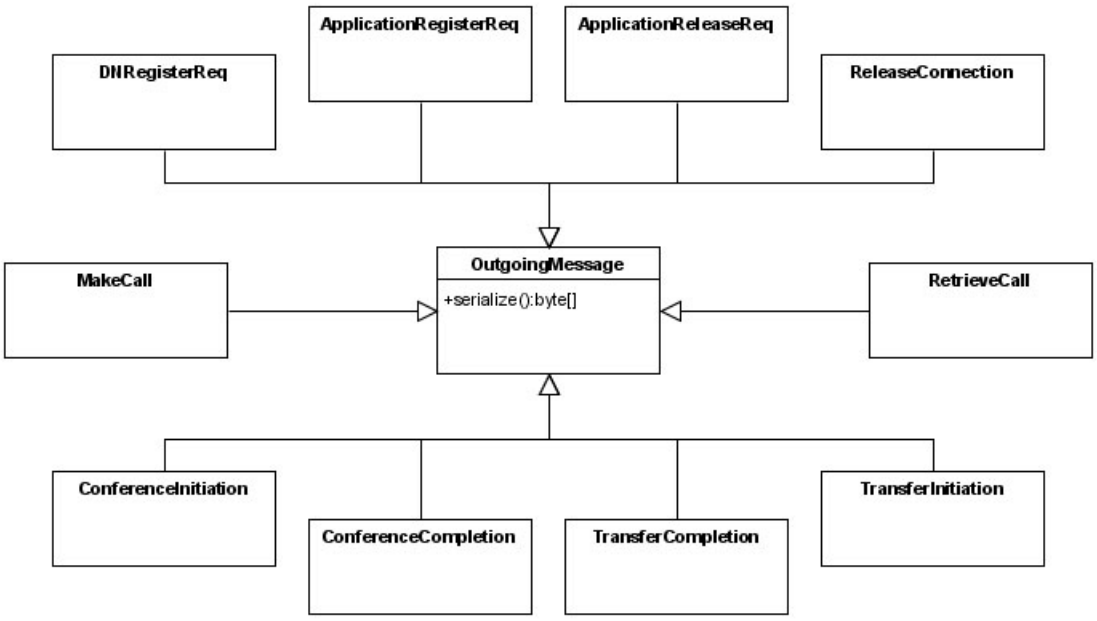

Abb. 7-8: Klassenstruktur ausgehender Nachrichten

`serialize` ihrer Basisklasse, da die Umwandlung in den Byte-Vektor vom Inhalt und Aufbau der jeweiligen Nachricht abhängt. Der durch den Aufruf der Methode erzeugte Vektor wird dann auf den *OutputStream* des Sockets geschrieben.

Der `SocketListener` ist im Gegenzug als eigenständiger Thread implementiert, der auf eingehende Nachrichten von der Telekommunikationsanlage wartet. Empfangene Zeichenfolgen werden an den `MessageParser` weitergereicht, welcher nach dem Erzeugungsmuster *Factory Method* entworfen wurde. Der Parser verfügt hierfür lediglich über eine öffentliche Methode `parseMessage`, welche die zu analysierende Zeichenfolge als Parameter aufnimmt. Je nach Zeichenfolge werden Objekte einer der Nachrichtenklassen

- `DNRegisterRes`,
- `ApplicationRegisterRes`,
- `ApplicationReleaseRes` oder
- `StatusChange`

erzeugt und mit den Werten der Protokollnachricht initialisiert und zurückgegeben. Alle Nachrichtenklassen sind von der abstrakten Klasse `IncomingMessage` abgeleitet, welche die abstrakte Methode `notifyProvider` zur Benachrichtigung des Providers definiert (vgl. Abbildung 7-9). Diese Methode wird von jeder der Nachrichtenklassen auf andere Weise implementiert, je nachdem, welche Ereignismethode des registrierten ProviderCallback-Objekts aufzurufen ist. Diese Vorgehensweise wird in [GAMMA96] als *Strategiemuster* (englisch *Policy* oder *Stategy Pattern*) beschrieben. In der englischen Originalausgabe wird

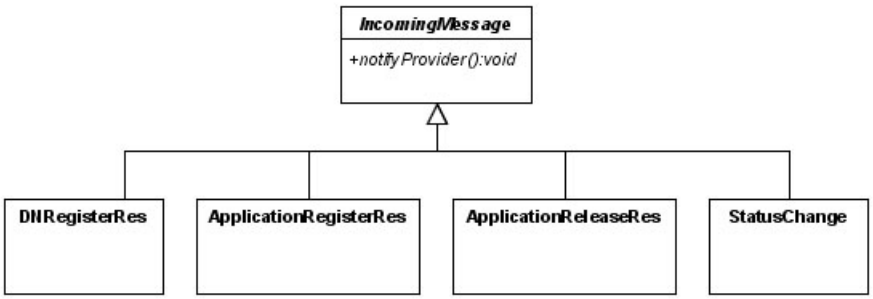

Abb. 7-9: Klassenstruktur eingehender Nachrichten

der Zweck dieses Musters wie folgt zusammengefasst:

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Für den Ablauf der Verarbeitung eingehender Nachrichten bedeutet dies die Unabhängigkeit von der tatsächlichen Nachricht. Der Parser analysiert die über das Netzwerk empfangene Zeichenkette und entscheidet nach der Art der erkannten Nachricht welches Objekt erzeugt werden soll. Der Rückgabetyp der parseMessage-Methode ist jedoch immer das der abstrakten Klasse `IncomingMessage`, sodass der `SocketListener` durch den Aufruf der Methode `notifyProvider` die Benachrichtigung des Callback-Objekts initiiert, ohne wissen zu müssen, welches konkrete Nachrichtenobjekt sich dahinter verbirgt.

Somit wurde beim Entwurf des Treibers darauf geachtet, die Erweiterung um neue Nachrichtentypen möglichst einfach zu gestalten. Durch die Erzeugung der Objekte für eingehende sowie ausgehende Nachrichten über Fabriken und durch den Einsatz des Strategiemusters ist eine Erweiterung der Implementierung des Meridian-Link-Protokolls bei geringem Änderungsaufwand realisierbar.

7.4 Die JTAPI-Implementierung

Der Entwurf der JTAPI-Spezifikation basiert auf dem Erweiterungspaket `javax.telephony.callcontrol`, da das Core-Paket allein für die Umsetzung der funktionalen Anforderungen nicht ausreicht. Denn erst dieses Erweiterungspaket sieht Methoden für Konferenzschaltungen, Transfers und Konsultationsgespräche vor.

Bei der Namensgebung der Klassen wurde zur verbesserten Erfassung der Zusammenhänge das Suffix `Impl` für *Implementierung* an den Namen der wichtigsten, implementierten Schnittstelle angehängt. Beispielsweise implementiert die Klasse `CallImpl` das Interface `CallControlCall`, welches seinerseits von der Schnittstelle `Call` abgeleitet ist. Damit implementiert `CallImpl` ebenfalls indirekt die Schnittstelle des Core-Pakets. Die Tabelle 7-1 zeigt eine Auflistung der Klassen der JTAPI-Implementierung und stellt ihnen die implementierten Interfaces und Basisklassen gegenüber. Auf die Darstellung der im-

Klassenname	Schnittstellen (I) und Basisklassen (C)
DefaultJtapiPeer	JtapiPeer(I)
EventImpl	Event(I)
CallControlEventImp	CallControlEvent(I), EventImpl(C)
ProviderImpl	Provider(I)
ProviderEventImpl	ProviderEvent(I), EventImpl(C)
ProviderCapabilitiesImpl	ProviderCapabilities(I)
CallImpl	CallControlCall(I)
CallEventImpl	CallControlCallEvent(I), CallControlEventImpl(C)
CallCapabilitiesImpl	CallControlCallCapabilities(I)
ConnectionImpl	CallControlConnection(I)
ConnectionEventImpl	CallControlConnectionEvent(I), CallEventImpl(C)
ConnectionCapabilitiesImpl	CallControlConnectionCapabilities(I)
TerminalConnectionImpl	CallControlTerminalConnection(I)
TerminalConnectionEventImpl	CallControlTerminalConnectionEvent(I), CallEventImpl(C)
TerminalConnectionCapabilitiesImpl	CallControlTerminalConnectionCapabilities(I)
TerminalImpl	CallControlTerminal(I)
TerminalEventImpl	CallControlTerminalEvent(I), CallControlEventImpl(C)
TerminalCapabilitiesImpl	CallControlTerminalCapabilities(I)
AddressImpl	CallControlAddress(I)
AddressEventImpl	CallControlAddressEvent(I), CallControlEventImpl(C)
AddressCapabilitiesImpl	CallControlAddressCapabilities(I)

Tab. 7-1: Klassenübersicht der JTAPI-Implementierung mit den korrespondierenden Schnittstellen und Basisklassen

plizit implementierten Schnittstellen des Core-Pakets wird jedoch aufgrund der selbsterklärenden Namensgebung und zur besseren Übersicht verzichtet. Die Implementierung des Call-Modells verteilt sich entsprechend der Vorgaben der JTAPI-Schnittstellen neben der Hilfsklasse `DefaultJtapiPeer` auf die Klassen

- `ProviderImpl`,
- `CallImpl`,
- `ConnectionImpl`,
- `TerminalConnectionImpl`,
- `AddressImpl` und

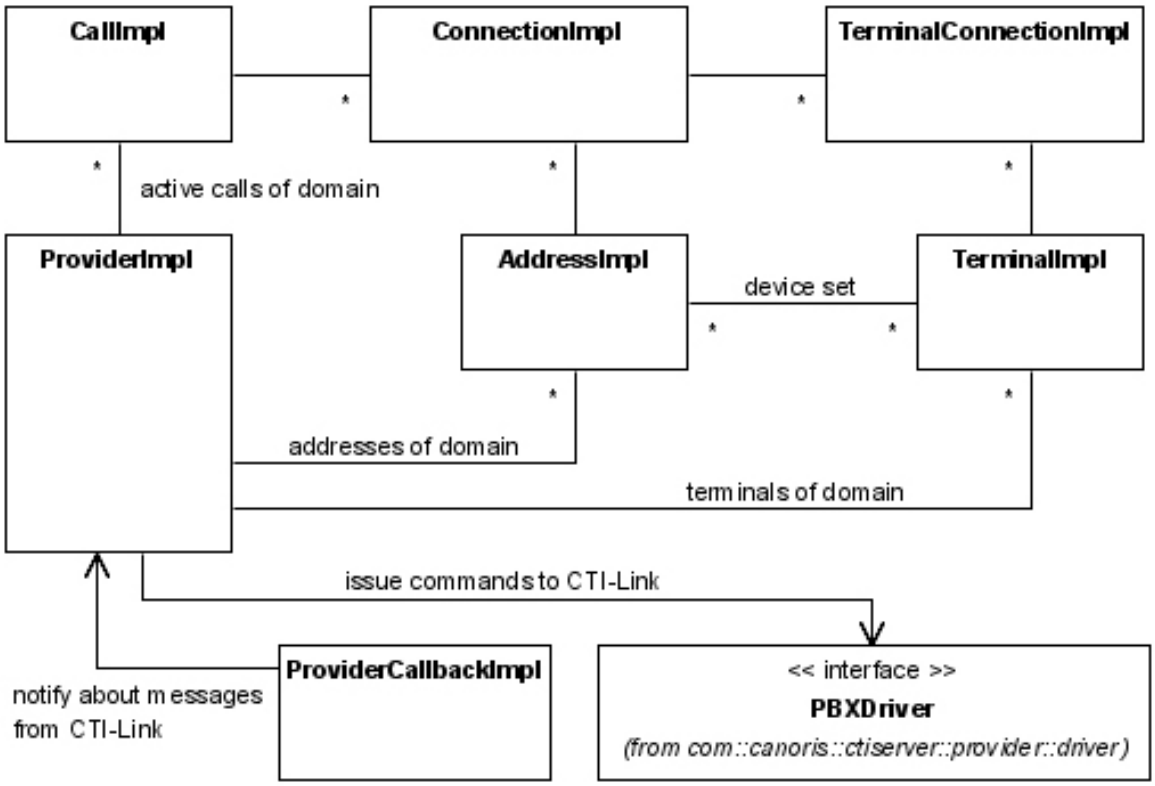

Abb. 7-10: Klassenstruktur des Call-Modells

- `TerminalImpl`.

Sie implementieren die Methoden zur Umsetzung telefoniebezogener Funktionen, ermöglichen die Überwachung der Objekte des Call-Modells mittels Listener und Events und unterstützen die Capabilities-Funktionalität des Core- und Call-Control-Pakets. Eine Übersicht über die Beziehung der Klassen zueinander sowie die Anbindung an den Treiber wird in Abbildung 7-10 gegeben. Im Folgenden sollen die einzelnen Klassen kurz beschrieben werden. Von Interesse ist dabei, wie die Assoziationen realisiert sind und in welchem Umfang das API umgesetzt wurde.

7.4.1 Das Call-Modell

Die folgenden Bemerkungen sind als Grundlage mit Gültigkeit für alle Implementierungsklassen des Call-Modells anzusehen.

Alle als *deprecated* bezeichneten Methoden werden nicht unterstützt, sodass deren Aufruf entweder eine für solche Fälle vorgesehene Ausnahme der Klasse `MethodNotSupportedException` verursacht oder in unkritischen Fällen ohne Fehlermeldung fehlschlägt. Beide Handlungsalternativen sind in der JTAPI-Spezifikation definiert und für jede Methode separat reglementiert. Dies gilt ebenfalls für alle nicht implementierten Methoden des API, die tefoniebezogene Funktionen erfüllen und deren

Unterstützung vom eingesetzten Telefoniesubsystem abhängt. Der Versuch solche Methoden zu nutzen verursacht gleichsam eine `MethodNotSupportedException`.

Implementiert sind hingegen alle für das Event/Listener-Modell notwendigen Methoden zur Abfrage, Registrierung und zum Löschen einzelner Listener. Dies trifft auch auf die Methoden zu, die für die Abfrage der Objektzustände (`getState` respektive `getCallControlState`), der Objektattribute (get-Methoden) sowie für die Ermittlung der statischen und dynamischen Capabilities zuständig sind. Zuletzt soll angemerkt werden, dass Assoziationsbeziehungen innerhalb der JTAPI-Implementierung über Referenzen auf die tatsächlichen Implementierungsklassen und nicht auf die implementierten Schnittstellen erfolgen, wie dies bei den Methoden des API der Fall ist. Dieses Vorgehen erspart häufige und kostenintensive Type-Casts bei der Nutzung implementierungsspezifischer Methoden.

7.4.1.1 DefaultJtapiPeer

Wie in der JTAPI-Einführung in Kapitel 6.3 beschrieben, ermöglicht die Klasse `DefaultJtapiPeer` als Implementierung der Schnittstelle `JtapiPeer` die Erzeugung von Provider-Objekten mit Hilfe der Methode `getProvider`. Die Methode erfordert einen obligatorischen Parameter vom Typ `String`, der den Namen der zu initialisierenden Provider-Implementierung spezifiziert. Da bei Konfigurationen mit mehreren Telekommunikationsanlagen auch mehrere Provider erzeugt, initialisiert und konfiguriert werden müssen, wird jeder PBX eine eindeutige ID zugewiesen. Damit die Provider-Instanzen die zugehörigen Konfigurationsdaten laden können, wird ihnen bei der Erzeugung die ID ihrer zugeordneten PBX übergeben. Für diese Zwecke definiert die JTAPI-Spezifikation eine einfache Syntax zur Erweiterung des Übergabe-Strings. Dieser zufolge können dem Provider-Namen durch Semikolons abgeteilte Schlüssel/Wert-Paare zu Konfigurationszwecken angehängt werden. So wurde für die Übergabe der ID der Schlüssel `pbxId` eingeführt, sodass analog zum Listing 7-1 ein Provider damit initialisiert werden kann. Die ID, im Beispiel `PBX_ONE`, dient dabei als Schlüssel zur Ermittlung der von der Telekommunikationsanlage abhängigen Konfigurationsparameter wie beispielsweise die IP-Adresse des CTI-Links oder die TCP-Portnummer. Diese und andere Parameter müssen dem Treibermodul zum Aufbau der Socket-Verbindung bekannt sein.

```
String thePeer = "com.canoris.ctiserver.provider." +
        "DefaultJtapiPeer";
JtapiPeer thePeer = JtapiPeerFactory.getJtapiPeer(thePeer);
Provider theProvider;
String providerName = "PBX.Meridian_I; pbxId = PBX_ONE";
theProvider = thePeer.getProvider(providerName);
}
```

Prog. 7-1: Erzeugung eines Providers mit DefaultJtapiPeer

Um einer Anwendung die Abfrage der durch einen `JtapiPeer` unterstützten Provider zu ermöglichen, wurde die Methode `getServices` definiert, die ein String-Array

mit allen Namen der verfügbaren Provider-Implementierungen zurückliefert. Die Methode `getServices` der Klasse `DefaultJtapiPeer` liefert dementsprechend ein Array mit lediglich einem String-Element zurück. Der String hält den Wert *PBX.Meridian_I*, den Namen der im Rahmen dieses Projekts entwickelten Implementierung.

7.4.1.2 ProviderImpl

Bei dem Objekt, das von der Methode `DefaultJtapiPeer.getProvider` unter Angabe des Namens *PBX.Meridian_I* zurückgegeben wird, handelt es sich um ein Exemplar der Klasse `ProviderImpl`. Diese Klasse dient als Bindeglied zwischen Call-Modell, Treiber und Anwendung. Der Anwendung ermöglicht die Provider-Klasse Zugriff auf die Objekte des Call-Modells sowie innerhalb der JTAPI-Implementierung Zugriff auf die Schnittstelle des Treibers. Um Letzteres bewerkstelligen zu können, initialisiert der Provider das Treibermodul durch Erzeugung eines Exemplars der Klasse `MeridianDriver` und registriert dort ein Objekt der Klasse `ProviderCallbackImpl` als Listener. Bei `ProviderCallbackImpl` handelt es sich um eine Implementierung der Schnittstelle `ProviderCallback` aus dem Treiber-Paket. Diese Klasse deklariert die Callback-Methoden für die Treiber-zu-Provider-Kommunikation.

Die Klasse `ProviderImpl` muss als Implementierung der Schnittstelle `Provider` einer Anwendung den Zugriff auf sämtliche Erkenntnisobjekte des Call-Modells ermöglichen. Deshalb enthält sie Referenzen auf alle Call-, Address- und Terminal-Objekte der Domain der abgebildeten Telekommunikationsanlage. Die Objekte dieser drei Klassen werden in drei separaten *HashMaps* gespeichert. In dieser Datenstruktur können Objekte in einer mittels Hash-Tabellen realisierten Map abgelegt werden, sodass ein gesuchtes Objekt über seinen Schlüssel bei konstantem Aufwand wiedergefunden werden kann. Der Nachteil des höheren Zeitbedarfs beim Hinzufügen neuer Einträge ist insofern vernachlässigbar, als im Falle von Terminal- und Address-Objekten deren genaue Anzahl und Namen im Vorfeld schon bekannt ist und deren Erzeugung als initialer Startaufwand lediglich einmal am Anfang erbracht werden muss. Auch der höhere Speicherbedarf ist zu vernachlässigen, da sich bei Kenntnis der Anzahl der zu speichernden Objekte optimierte Hash-Tabellen mit idealen Hash-Funktionen realisieren lassen und deshalb kaum Speicher-Overhead verursachen. Als Schlüssel dienen jeweils die Attribute

- `CallImpl.id`,
- `AddressImpl.name` und
- `TerminalImpl.name`,

die sämtlich als `java.lang.String` realisiert sind. Obwohl gemäß der aktuellen Anforderungen Call-IDs, Address- und Teminal-Namen lediglich aus numerischen Zeichen bestehen, würde deren Abbildung mittels Ganzzahlen im Falle von Rufnummern mit führenden Nullen oder dem Ersatzzeichen „+“ einen unnötigen Aufwand verursachen.

Inklusive von `createCall` wurden, unter Vorbehalt der genannten Vorbemerkung, sämtliche durch das Interface `Provider` definierten Methoden implementiert.

7.4.1.3 CallImpl

Die Klasse `CallImpl` stellt als Implementierung der Schnittstelle `Call` die Beziehungen zwischen den Gesprächsteilnehmern her. Hierfür werden die Connection-Objekte in einer Collection des Typs `ArrayList` gespeichert. Für diese Datenstruktur spricht der meist sequenzielle Suchansatz bei kleiner verwalteter Objektzahl mit meist weniger als zehn Elementen. Überdies hält `CallImpl` eine einfache Objekt-Referenz auf das übergeordnete Provider-Objekt.

Neben den Basismethoden des Core-Pakets implementiert die Klasse `CallImpl` Teile der Erweiterungen des Call-Control-Pakets. Für die Implementierung der funktionalen Anforderungen aus Kapitel 4.1 ist diese Klasse von zentraler Bedeutung. `CallImpl` implementiert die Methode `connect`, welche im Interface `Call` als einzige eine telefoniebezogene Funktion erfüllt. Wie diese in Verbindung mit der Methode `Provider.createCall` zur Anrufschaltung eingesetzt wird, wurde bereits im Listing 6-8 auf Seite 66 beschrieben.

Die Methode `consult` wurde zum Zweck des Aufbaus von Konsultationsgesprächen für die Konferenz- und Transferfunktionalität implementiert. Da dem CTI-Link-Modul der *Meridian I* bereits im Vorfeld über die Nachrichten *TransferInitiation* respektive *ConferenceInitiation* (vgl. Kapitel 5.3.1) das Ziel für die Konsultation mitgeteilt werden muss, wurden die Methoden `setTransferEnable` und `setConferenceEnable` implemetiert. Mit ihnen wird spezifiziert, ob das Konsultationsgespräch für eine Konferenz oder eine Weiterleitung aufgebaut werden soll. Die Implementierungen der Methoden `transfer` sowie `conference` schließen dann den Prozess ab, indem das Konsultationsgespräch seiner Bestimmung zugeführt wird. Das Listing 7-2 zeigt das Weiterverbinden eines Teilnehmers und 7-3 den Aufbau einer Konferenzschaltung.

```
//pre-conditions: termConn is a TerminalConnection
//of the Call mainCall; p is a Provider "In Service"
String transDest = "4001";
CallControlCall transConsult = (CallControlCall)p.createCall();
transConsult.setTransferEnable(true);
transConsult.consult(termConn, transDest);
mainCall.transfer(transConsult);
```

Prog. 7-2: Durchführung von Transfers mit der JTAPI-Implementierung

Aufgrund der nicht gegebenen Unterstützung durch die CTI-Link-Schnittstelle konnten die Methoden

- `drop`,
- `offHook`,
- `setTransferController`,
- `setConferenceController` und

```
//pre-conditions: termConn is a TerminalConnection
//of the Call mainCall; p is a Provider "In Service"
String newParty = "4002";
CallControlCall confConsult = (CallControlCall)p.createCall();
confConsult.setConferenceEnable(true);
confConsult.consult(termConn, newParty);
mainCall.conference(consultationCall);
```

Prog. 7-3: Aufbau von Konferenzschaltungen mit der JTAPI-Implementierung

- `addParty`

zusammen mit den Überladungen `transfer(TerminalConnection tc)` und `consult(TerminalConnection tc)` nicht implementiert werden.

7.4.1.4 ConnectionImpl

`ConnectionImpl` hält einfache Referenzen auf jeweils ein Objekt der Klasse `CallImpl` und `AddressImpl`, wohingegen die Referenzen auf die TerminalConnectionImpl-Objekte in einer `ArrayList` abgelegt werden. Von den Methoden mit Telefoniefunktionalität wurde lediglich `disconnect` aus dem Core-Paket zum Beenden von Verbindungen implementiert. Die Methoden `accept`, `addToAddress`, `park`, `redirect` und `reject`, können aufgrund fehlender Unterstützung durch die CTI-Link-Schnittstelle nicht umgesetzt werden.

7.4.1.5 TerminalConnectionImpl

Die Klasse `TerminalConnectionImpl` hält jeweils eine Referenz auf Objekte der Klassen `ConnectionImpl` und `TerminalImpl`. Von den Methoden zur Erfüllung telefoniebezogener Funktionen wurde nur `unhold` zum Zurückholen einer im Rahmen eines Konsultationsgespräches geparkten Verbindung umgesetzt. Die Methoden `answer`, `hold`, `join` und `leave` können auf Basis der bekannten Teile des Link-Protokolls nicht realisiert werden.

7.4.1.6 AddressImpl und TerminalImpl

Dies gilt ebenfalls für die Klasse `AddressImpl` und die Methoden `setDoNotDisturb`, `getDoNotDisturb`, `setForwarding`, `getForwarding`, `cancelForwarding`, `setMessageWaiting` und `getMessageWaiting`. Damit interagiert `AddressImpl` nicht direkt mit dem Treibermodul und der PBX. Die Klasse enthält als Mitglied des Call-Modells dennoch Referenzen auf den Provider sowie auf mehrere Connection- und Terminal-Objekte. Die 1-zu-n-Beziehungen werden über die Collection `ArrayList` abgebildet.

Analog hierzu verhält es sich mit der Klasse `TerminalImpl` und deren Assoziationsbeziehungen mit Address- und TerminalConnection-Objekten. Auch hier kann aus den bekannten Gründen keine der Methoden `pickup`, `pickupFromGroup` und `setDoNotDisturb` zur Bereitstellung weiterer telefoniebezogener Funktionen implementiert werden.

API-Methode	PBXDriver-Methode	Link-Nachricht
CallImpl.connect	placeCall	MakeCall
CallImpl.consult (1)	initiateTransfer	TransferInitiation
CallImpl.consult (2)	initiateConference	ConferenceInitiation
CallImpl.transfer	completeTransfer	TransferCompletion
CallImpl.conference	completeConference	ConferenceCompletion
ConnectionImpl.disconnect	releaseActiveConnection	ReleaseConnection
TerminalConnectionImpl.unhold	retrieveOriginalCall	RetrieveCall

Tab. 7-2: Nutzung der Treiberschnittstelle PBXDriver durch die JTAPI-Implementierung sowie resultierende ausgehende Nachrichten

Für die Abbildung externer Teilnehmer verfügt das Address-Objekt über eine entsprechende Schaltervariable. Im Gegensatz zu den lokalen Address-Objekten, die bei Systemstart initialisiert werden, erfolgt die Erzeugung sobald ein entsprechender externer Teilnehmer registriert wird. Die Schaltervariable wird dabei gesetzt. Ein entsprechendes Terminal-Objekt wird nicht erzeugt, da sich das Endgerät des externen Teilnehmers ohnehin außerhalb der Domain der TK-Anlage befindet und nicht manipuliert werden kann.

7.4.1.7 Interaktion zwischen Call-Modell und Meridian-Treiber

Die Beschreibung des Call-Modells beschließt die Tabelle 7-2. Sie gibt einen Überblick über die Methoden, die zur Kommunikation mit dem Telekommunikationssystem auf die Treiberschnittstelle `PBXDriver` zugreifen und zeigt auf, welche Nachrichten infolgedessen an die PBX gesendet werden. Die zwei Einträge für `CallImpl.consult` beziehen sich auf die beiden Ablaufmöglichkeiten bei Transfers (1) sowie bei Konferenzen (2).

7.4.2 Synchronisation mit Zustandsänderungen im Telefoniesystem

Der Abgleich der Zustände zwischen den Objekten des Call-Modells und der Telekommunikationsanlage ist von großer Bedeutung. Denn nur bei konsistenten Statusinformationen kann das Call-Modell als ein virtuelles Abbild des Telefoniesubsystems dienen, auf dessen Basis festgestellt wird, ob eine Methode zu einem Zeitpunkt ausgeführt werden kann und inwiefern sie das gewünschte Ergebnis erbracht hat. Die Synchronisation der Zustände erfolgt unidirektional vom Telefoniesystem ausgehend in Richtung CTI-Anwendung. Lediglich die API-Methoden mit telefoniebezogener Funktion üben einen indirekten Einfluss auf die Zustände aus. Da JTAPI nicht speziell auf ein Telekommunikationssystem zugeschnitten wurde, um stattdessen vielfältigeren Anforderungen gerecht werden zu können, sind im Call-Modell weitaus mehr und andere Zustände definiert, als durch das CTI-Link-Protokoll kommuniziert werden. Ein weiterer Unterschied besteht in den Objekten deren Status wiedergegeben werden soll. Im Gegensatz zum JTAPI-Call-Modell unterscheidet die Telekommunikationsanlage *Meridian I* nicht zwischen einem Terminal und seiner Adresse, sondern übermittelt

die Statusinformationen eines Endgeräts – im Verständnis der Einheit aus logischem und physischem Endpunkt einer Verbindung. Infolgedessen muss eine Zustandsinformation auf die beiden Verbindungsobjekte der Typen `Connection` und `TerminalConnection` abgebildet werden. Für die Umsetzung der funktionalen Anforderungen sind die Zustände

- `UNKNOWN`,
- `ONHOOK`,
- `OFFHOOK`,
- `RINGING`
- `ACTIVE`,
- `DISCONNECT`,
- `UNRINGING`,
- `RETRIEVE`,
- `TRANSFERCOMPLETEDTO`,
- `CONFERENCECOMPLETEDTO`,
- `CONFERENCETOSIMPLE`,
- `TRANSFERINITIATION` und
- `CONFERENCEINITIATION`

auf das Call-Modell abzubilden. Welche Auswirkungen eine einzelne Zustandsänderung auf die beteiligten Objekte im Einzelnen hat, wird im Kapitel 8.1 im Rahmen der Details zur Implementierung der Methode `deviceStatusChange` der Klasse `ProviderCallbackImpl` näher ausgeführt.

7.4.3 Event-Modell

Die Umsetzung des Ereignismodells erfolgt durch die Implementierungsklassen

- `EventImpl`,
- `CallControlEventImpl`,
- `ProviderEventImpl`,
- `AddressEventImpl`,
- `TerminalEventImpl`,

- `CallEventImpl`,
- `ConnectionEventImpl` und
- `TerminalConnectionEventImpl`.

Bei `EventImpl` und `CallControlEventImpl` handelt es sich um die Implementierung der Basisschnittstellen `Event` und `CallControlEvent`. Da der Provider als einziges Element des Call-Modells keine Erweiterungen durch das Call-Control-Paket erfuhr, wird die zugehörige Ereignisklasse `ProviderEventImpl` lediglich von `EventImpl` abgeleitet. Alle anderen Ereignisobjekte erweitern `CallControlEventImpl` auf direkte oder indirekte Weise. Die Namensgebung folgt der bereits eingeführten Konvention. Der Aufbau aller Implementierungsklassen richtet sich nach dem gleichen Schema, das durch die Basisklassen `EventImpl` und `CallControlEventImpl` vorgegeben wird. So enthält jede Klasse eine Objektreferenz auf das Ereignis verursachende Objekt, welches über eine entsprechende get-Methode (beispielsweise `getProvider`, `getConnection`, usw.) ermittelt werden kann. Auskunft über die Ursache der Statusänderung gibt die Methode `getCause` in Form von ganzzahligen Werten zurück. Die möglichen Ursachen sind als statische Integer-Konstanten in den Schnittstellen `Event` und `CallControlEvent` definiert (vgl. Tabellen 7-3 und 7-4). Dieser Funktionsumfang wird aufgrund der Besonderheiten der einzelnen Ereignis verursachenden Objekte durch die spezifischen Event-Schnittstellen der Call-Modell-Objekte ergänzt. So kann beispielsweise ein Ereignisobjekt mit implementierter Schnittstelle `CallControlAddressEvent` mittels der Methode `getForwarding` Auskunft über die momentan aktiven Weiterleitungsregeln eines Adressobjekts geben.[2]

7.4.4 Capabilities

Die Capabilities-Funktionalität gibt mittels dedizierter Capabilities-Objekte Auskunft darüber, ob eine bestimmte Aktion ausgeführt werden kann oder nicht. Es wird dabei zwischen statischen und dynamischen Capabilities unterschieden. Während die statischen Capabilities Auskunft über die prinzipiell unterstützten Funktionen der JTAPI-Implementierung geben, liefern die dynamischen Capabilities diese Auskunft auf Basis der aktuellen Zustände an der Aktion beteiligter Objekte.

Statische Capabilities werden prinzipiell über den Provider ermittelt. `ProviderImpl` implementiert dementsprechend die Methoden

- `getProviderCapabilities`,
- `getCallCapabilities`,
- `getConnectionCapabilities`,
- `getTerminalConnectionCapabilities`,

[2] `AddressEventImpl` liefert aufgrund fehlender Unterstützung dieser Funktionalität durch das Telekommunikationssubsystem stets `null` zurück.

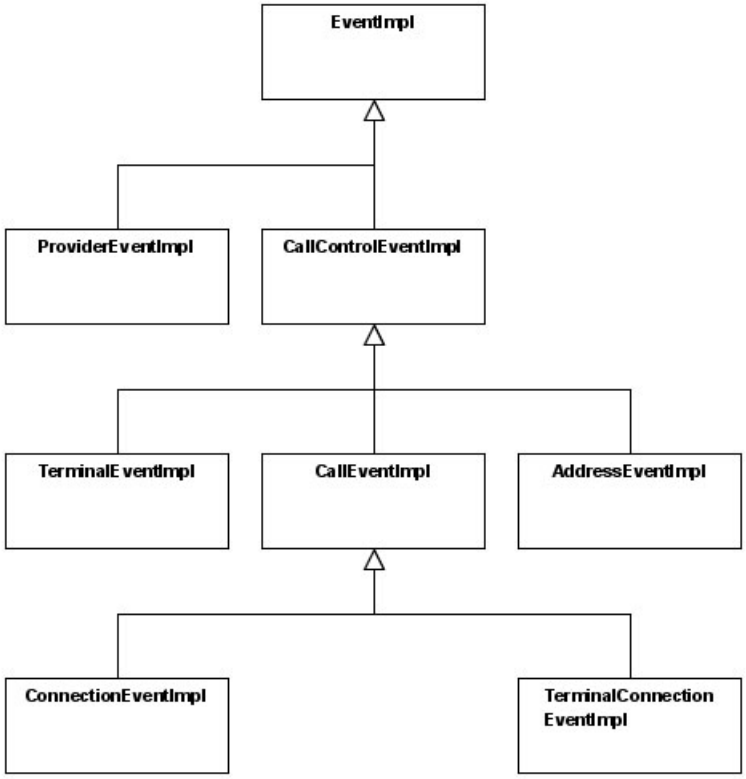

Abb. 7-11: Klassenstruktur der Event-Implementierung

Event-Cause	**Beschreibung**
CAUSE_NORMAL	Ursache liegt im normalen Ablauf
CAUSE_UNKNOWN	Ursache unbekannt
CAUSE_CALL_CANCELLED	Gespräch wurde ohne Zutun des Benutzers beendet
CAUSE_DEST_NOT_OBTAINABLE	Ziel der Verbindung nicht verfügbar
CAUSE_INCOMPATIBLE_DESTINATION	Ziel nicht kompatibel
CAUSE_LOCKOUT	Wählvorgang wegen Time-out nicht abgeschlossen
CAUSE_NEW_CALL	neuer Call
CAUSE_RESOURCES_NOT_AVAILABLE	Ressourcen nicht verfügbar
CAUSE_NETWORK_CONGESTION	Netzwerk blockiert oder überlastet
CAUSE_NETWORK_NOT_OBTAINABLE	Netzwerk nicht verfügbar
CAUSE_SNAPSHOT	Event-Aufbereitung im Rahmen eines Snapshots

Tab. 7-3: Die Cause-Felder von Event

Event-Cause	Beschreibung
CAUSE_ALTERNATE	Gespräch im Zuge der Aktivierung eines anderen geparkt
CAUSE_BUSY	Ziel besetzt
CAUSE_CALL_BACK	durch die Callback-Funktion ausgelöst
CAUSE_CALL_NOT_ANSWERED	Anruf wurde nicht innerhalb eines Zeitlimits entgegen genommen
CAUSE_CALL_PICKUP	durch die Pickup-Funktion ausgelöst
CAUSE_CONFERENCE	durch die Konferenzfunktion ausgelöst
CAUSE_DO_NOT_DISTURB	durch die Do-Not-Disturb-Funktion ausgelöst
CAUSE_PARK	durch die Park-Funktion ausgelöst
CAUSE_REDIRECTED	durch die Redirect-Funktion ausgelöst
CAUSE_REORDER_TONE	Anruf erhielt Besetztzeichen
CAUSE_TRANSFER	durch die Transfer-Funktion ausgelöst
CAUSE_TRUNKS_BUSY	alle Leitungen besetzt
CAUSE_UNHOLD	durch die Unhold-Funktion ausgelöst

Tab. 7-4: Die Cause-Felder von CallControlEvent

- `getAddressCapabilities` und
- `getTerminalCapabilities`.

Diese geben jeweils ein Objekte der Klassen

- `ProviderCapabilitiesImpl`,
- `CallCapabilitiesImpl`,
- `ConnectionCapabilitiesImpl`,
- `TerminalConnectionCapabilitiesImpl`,
- `AddressCapabilitiesImpl`,
- `TerminalCapabilitiesImpl`,

zurück. Auskunft über die konkreten Fähigkeiten der Implementierung kann eine Anwendung über die Methoden dieser Klassen erhalten. Die Methoden können als Fragen aufgefasst werden, die entweder mit `true` oder `false` beantwortet werden. Die Tabelle 7-5 listet die Methoden und deren Rückgabewert auf und gibt damit Auskunft über den Grad der Umsetzung der Core- und Call-Control-Schnittstellen.

Capabilities-Methode	Boolescher Rückgabewert
ProviderCapabilitiesImpl	
isObservable()	true
CallCapabilitiesImpl	
isObservable()	true
canConnect()	true
canDrop()	false
canOffHook()	false
canConsult()	true
canConsult(TerminalConnection tc)	false
canConsult(TerminalConnection tc, String destination)	true
canTransfer()	true
canTransfer(String destination)	false
canTransfer(Call call)	true
canConference()	true
canAddParty()	false
canSetTransferController()	false
canSetConferenceController()	false
canSetTransferEnable()	true
canSetConferenceEnable()	true
ConnectionCapabilitiesImpl	
canDisconnect()	true
canRedirect()	false
canAddToAddress()	false
canAccept()	false
canReject()	false
canPark()	false
TerminalConnectionCapabilitiesImpl	
canAnswer()	false
canHold()	false
canUnhold()	true
canJoin()	false
canLeave()	false
AddressCapabilitiesImpl	
isObservable()	true
canSetForwarding()	false
canGetForwarding()	false
canCancelForwarding()	false
canSetDoNotDisturb()	false
canGetDoNotDisturb()	false
canSetMessageWaiting()	false

canGetMessageWaiting()	false
TerminalCapabilitiesImpl	
isObservable()	true
canGetDoNotDisturb()	false
canSetDoNotDisturb()	false
canPickup()	false
canPickup(Connection connection, Address address)	false
canPickup(TerminalConnection tc, Address address)	false
canPickup(Address address1, Address address2)	false
canPickupFromGroup()	false
canPickupFromGroup(Address address)	false
canPickupFromGroup(String group, Address address)	false

Tab. 7-5: Übersicht der statischen Capabilities der JTAPI-Implementierung

Die dynamischen Capabilities werden mit der Methode `getCapabilities` direkt an den Objekten des Call-Modells abgefragt. Dabei werden nach demselben Schema Capabilities-Objekte erzeugt. Der Unterschied besteht darin, dass bei denjenigen Aktionen, die laut statischer Capabilities unterstützt werden, eine Überprüfung der Durchführbarkeit unter Einbeziehung der Objektzustände stattfindet. Diese kann im Gegensatz zur statischen Überprüfung negativ ausfallen. Der umgekehrte Fall ist ausgeschlossen. Nicht unterstützte Aktionen liefern sowohl bei statischer als auch bei dynamischer Überprüfung negative Ergebnisse.

7.5 Der CTI-Service *webDial*

Mit dem *webDial*-Dienst wird der serverseitige Teil der *webDial*-Anwendung implementiert, der auf Client-Anfragen wartet und sie unter Nutzung der Schnittstellen der Provider-Implementierung bearbeitet. Da entsprechend der Anzahl zu beeinflussender Telekommunikationsanlagen auch Provider-Instanzen existieren, muss der *webDial*-Dienst die voneinander abgegrenzten Zustandsmodelle in einen gemeinsamen Zusammenhang setzen. Der Dienst bietet seine Funktionen den Clients über eine Service-Schnittstelle an, welche durch ein RMI-Remote-Objekt implementiert wird.

Es folgt eine Beschreibung der einzelnen Bestandteile der Klassenstruktur, die zur Verdeutlichung anhand Abbildung 7-12 verfolgt werden kann. Von zentraler Bedeutung ist die Klasse `WebdialController`. Sie implementiert das Interface `CtiService`, welches durch den CTI-Server (`CanorisCTIService`) genutzt wird, um den *webDial*-Dienst zu starten und zu beenden. Der `WebdialController` initialisiert und exportiert das RMI-Server-Objekt `WebDialServer`, das die Remote-Methoden der Schnittstelle `WebDialServerRemoteInterface` implementiert. Dem RMI-Server-Objekt wird jedoch nur die Aufgabe der Überprüfung von Übergabeparametern zuteil, da die Anfragen zur

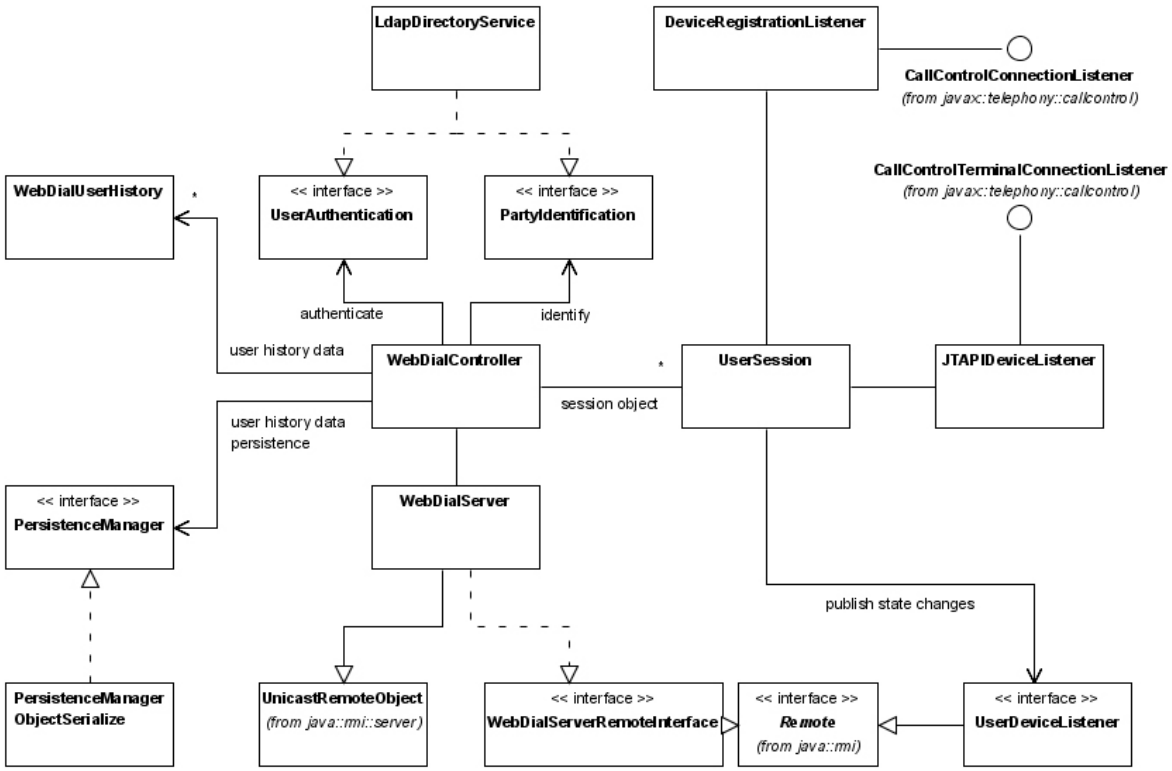

Abb. 7-12: Klassenstruktur des *webDial*-Dienstes

weiteren Verarbeitung an den `WebdialController` weitergeleitet werden, welcher die eigentliche Ablauflogik implementiert.

Für die Benutzerauthentifizierung und die Identifizierung von Gesprächsteilnehmern wurden die Schnittstellen `UserAuthentication` und `PartyIdentification` definiert. Über die Systemkonfiguration wird bestimmt welche Klassen zu deren Implementierung geladen werden sollen. Die einzige Implementierung liegt momentan mit der Klasse `LdapDirectoryService` vor, die beide Schnittstellen implementiert. `LdapDirectoryService` nutzt einen LDAP-fähigen Verzeichnisdienst zur Überprüfung der Login-Daten sowie für die Ermittlung der Gesprächsteilnehmer über deren Rufnummer. Durch Austausch der Implementierungsklasse können verschiedene Verfahren zur Authentifizierung und Identifizierung realisiert werden. Durch Unterscheidung der beiden Schnittstellen können beide Funktionalitäten auf unterschiedliche Weise implementiert werden. So könnte in Zukunft die Benutzerauthentifizierung über eine Passwortdatei und die Teilnehmeridentifizierung über eine SQL-Datenbank erfolgen. Ein vergleichbares Konzept wurde mit der Schnittstelle `PersistenceManager` und der implementierenden Klasse `PersistenceManagerObjectSerialize` verfolgt. Die Schnittstelle definiert Methoden zum Speichern und Laden der Benutzerinformationen, die persistent gehalten werden sollen. Die Implementierung in `PersistenceManagerObjectSerialize` nutzt hierzu die Möglichkeiten der Objektserialisierung von Java (vgl. Kapitel 8.2). Dabei werden die zu speichernden Objekte in eine Bytefolge serialisiert und in einer Datei abgelegt. Beim Einle-

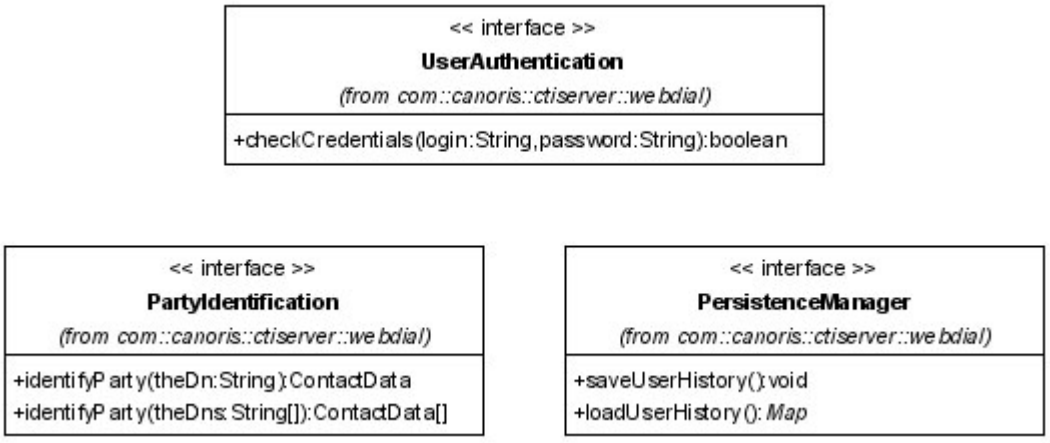

Abb. 7-13: Die Schnittstellen UserAuthentication, PartyIdentification und PersistenceManager

sen wird die Bytefolge aus der Datei ausgelesen und die Objekte daraus wieder rekonstruiert. Auch hier kann durch Austauschen der Implementierungsklasse, beispielsweise unter Einsatz von JDBC, eine datenbankbasierte Lösung eingeführt werden.

Bei den Benutzerdaten, die die Ausführungszeit des Dienstes überdauern sollen, handelt es sich um Objekte der Klasse `WebDialUserHistory`. Diese Klasse verfügt über Attribute für

- den Login-Namen,
- den Zeitpunkt der letzten Anmeldung,
- die Telefonnummer des zuletzt registrierten Apparats,
- den Zeitpunkt der Apparatregistrierung und
- die ID der PBX, an welcher der Apparat angeschlossen ist.

Das Benutzerpasswort wird *nicht* gespeichert. Diese Informationen werden benötigt, um dem Benutzer beim Einloggen die erneute Registrierung seines Telefonapparats ersparen zu können, wenn sie nicht länger als einen bestimmten, konfigurierbaren Zeitraum zurückliegt. Unter der Apparatregistrierung ist ein Sicherheitsmechanismus zu verstehen, der die Manipulation eines Apparats durch Unbefugte verhindert. Ein Benutzer, eindeutig identifiziert durch seinen Login-Namen, kann zu einem Zeitpunkt immer nur *einen* Apparat steuern. Den zu steuernden Apparat legt er über den Registrierungsprozess fest. Hierzu teilt er dem *webDial*-Dienst die Telefonnummer des Apparats mit und ruft von diesem aus manuell eine Registrierungsnummer an. Der *webDial*-Dienst überwacht den Registrierungsapparat auf eingehende Anrufe und vergleicht die Rufnummern der Anrufer mit denen der Apparate deren Registrierung aussteht. Erst wenn ein entsprechender Anruf eingeht, wird die Registrierung abgeschlossen und der Benutzer kann mit der Überwachung und Steuerung seines Apparats beginnen. Da nur

Apparate registriert werden können, auf die der Benutzer Zugriff hat, wird eine wahllose oder gezielte Steuerung der Apparate durch Unbefugte verhindert.

Die Klasse `UserSession` implementiert ein Session-Objekt, das für jeden Benutzer bei dessen Anmeldung erzeugt und nach Beenden der Sitzung zerstört wird. Wichtige Aufgaben dieser Klasse sind Erzeugung und Registrierung der Listener `DeviceRegistrationListener` und `JTAPIDeviceListener` sowie die Speicherung angemeldeter, die Schnittstelle `UserDeviceListener` implementierender Callback-Objekte.

Der `DeviceRegistrationListener` implementiert die Schnittstelle `CallControlConnectionListener` und wird, falls eine Registrierung erforderlich ist, am Address-Objekt des Registrierungsapparats angemeldet. Dieser Listener wartet auf den eingehenden Anruf und schließt die Registrierung gegebenenfalls ab, indem der `DeviceRegistrationListener` abgemeldet und ein `JTAPIDeviceListener` erzeugt und angemeldet wird. Ein `JTAPIDeviceListener` implementiert die Schnittstelle `CallControlTerminalConnectionListener` und wird am Terminal-Objekt des registrierten Apparats als Call-Listener angemeldet. Dadurch wird bewirkt, dass alle Ereignisse übermittelt werden, die Call-, Connection- und TerminalConnection-Objekte zukünftiger Gespräche betreffen, an welchen der registrierte Apparat beteiligt ist.

Nach der erfolgten Anmeldung des Benutzers muss die Client-Anwendung ein Callback-Objekt, das die Schnittstelle `UserDeviceListener` implementiert, am *webDial*-Dienst registrieren. Das Objekt wird im Session-Objekt des Benutzers referenziert und dient der Information des Client-Prozesses über das Eintreten bestimmter Ereignisse (vgl. nächstes Kapitel). Die Implementierung dieses Objekts muss im Rahmen der Entwicklung der Client-Anwendung erfolgen und ist deshalb nicht Bestandteil dieses Projekts. Die Schnittstelle `UserDeviceListener` wird jedoch im Rahmen der Beschreibung der Client-API im nächsten Kapitel erläutert.

7.5.1 *webDial*-Client Application Programming Interface

Das *webDial-Client Application Programming Interface* (kurz: *Client-API*) wird zur Entwicklung von Client-Anwendungen für die Nutzung des *webDial*-Dienstes benötigt. Eine der strukturellen Anforderungen war es, dabei unterschiedliche Realisierungsmöglichkeiten vorzusehen. Zwei Varianten sind dabei in Zukunft von Interesse:

(1) Zur Benutzung des *webDial*-Dienstes setzen die Anwender einen Web-Browser als Thin-Client ein. Aus Sicht des *webDial*-Servers wird die Client-Anwendung im Kontext eines *HTTP-Servers* ausgeführt, der die Web-Seiten der *webDial*-Anwendung erzeugt und dafür mit dem Serverprozess auf dem CTI-Proxy kommuniziert.

(2) Die Benutzer verwenden eine *dedizierte Java-Anwendung* mit Fensterführung. Die Anwendung kommuniziert direkt mit dem Server-Prozess auf dem CTI-Proxy. Die plattformunabhängige Realisierung ermöglicht eine Ausführung unter den gängigen Betriebssystemen *Windows*, *Linux*, *Mac OS* und *Solaris*.

Die zentrale Problemstellung beim Entwurf des *webDial*-Servers ist die Implementierung einer Schnittstelle, die von beiden Client-Varianten gleichermaßen genutzt werden kann. Hinderlich sind dabei der zustandslose Charakter des HTTP-Protokolls und die Request/Response-basierte Kommunikation, die keinen vom Server ausgehenden Kontrollfluss erlaubt. All diese Einschränkungen können bei einer dedizierten Client-Software durch Spezifikation eines entsprechenden Applikationsprotokolls umgangen werden.

Wie bereits zu Beginn des Design-Kapitels erwähnt, wird für die Implementierung des Server-Dienstes auf die RMI-Technologie zurückgegriffen, sodass die Entwicklung eines eigenständigen Protokolls auf Applikationsebene entfällt. Für eine Client-Anwendung besteht damit die Server-Schnittstelle aus mindestens einem Remote-Interface, das die angebotenen Dienstelemente definiert. Zudem können Callback-Abläufe implementiert werden, die eine Umkehrung des Kontrollflusses ermöglichen. Ein Client übergibt dem Server mittels einer dafür bestimmten Registrierungsmethode ein Callback-Objekt, das der Server-Prozess für die Client-gerichtete Kommunikation nutzt. Da das übergebene Objekt das Interface `java.rmi.Remote` implementiert, wird es für die Übermittlung an den Server nicht, wie sonst bei RMI üblich, kopiert. Stattdessen wird eine über die Systemgrenzen hinaus gültige Speicheradresse übertragen. Das Ergebnis ist ein verteiltes Objekt, das auf der Clientseite gehalten wird, aber zugleich im Serverprozess manipuliert werden kann. Der Unterschied zu einem exportierten Remote-Objekt liegt zum einen in der fehlenden Ableitung von der Klasse `UnicastRemoteObject` und zum anderen an der nicht stattgefundenen Exportierung als RMI-Server-Objekt. Letzteres würde das RMI-Framework zur Initialisierung eines Socket-basierten Servers zur Multi-Threading-fähigen Verarbeitung der Client-Anfragen veranlassen.

`WebDialServerRemoteInterface` und `UserDeviceListener` definieren nach diesem Prinzip die Schnittstellen für den RMI-Server und das Callback-Objekt. Während die Klasse `WebDialServer` die Server-Schnittstelle implementiert, muss das Callback-Objekt entsprechend den Anforderungen der Client-Anwendung implementiert werden. Diese beiden Schnittstellen ermöglichen die bidirektionale Kommunikation bei entgegengesetztem Kontrollfluss, sodass ein Client die Statusinformationen nicht zyklisch abfragen muss, sondern durch den Server-Prozess über Zustandsänderungen zeitnah informiert wird. Der Server wird damit zum einen vor übermäßiger Last bewahrt, und zum anderen verfügt der Client stets über aktuelle Zustandsinformationen. Für die Entwicklung der Client-Systeme hat dies die folgenden Konsequenzen:

(1) Die Clients *müssen* mit Java entwickelt werden, da RMI nur die Kommunikation zwischen verschiedenen Java-Laufzeitumgebungen ermöglicht.

(2) Eine HTTP-basierte Client-Implementierung muss die *Servlet-Technologie* einsetzten, um lokale Stellvertreterobjekte des RMI-Server-Objekts zu erzeugen und sie für die Kommunikation nutzen zu können.

Durch dieses Modell wird eine zustandsbehaftete Kommunikation bei bidirektionalem Kontrollfluss für sowohl dedizierte als auch HTTP-basierte Clients möglich. Davon bleibt jedoch die Request/Response-gesteuerte Kommunikation zwischen Browser und HTTP-Server unberührt. Auch das Session-Management muss auf der Seite des HTTP-Servers implementiert werden. Die für die Ausführung von *Servlets* ohnehin benötigten

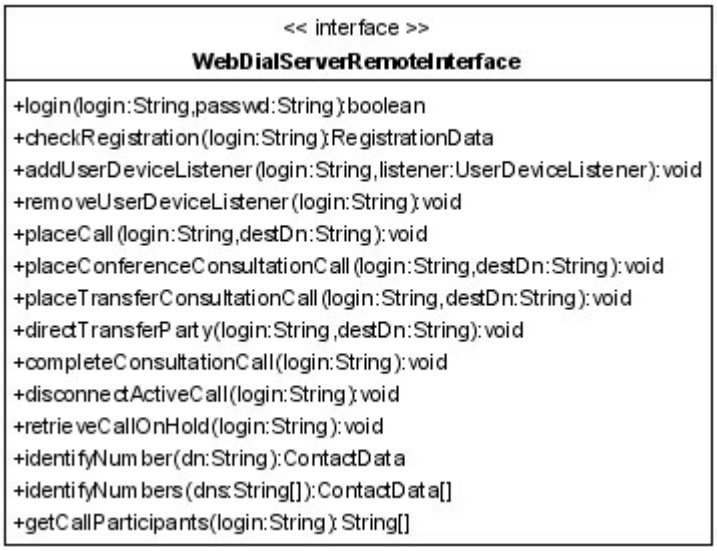

Abb. 7-14: Das Remote-Interface WebDialServerRemoteInterface

Container-Implementierungen[3] bieten bereits Mechanismen zur Unterstützung des Session-Managements an.

7.5.1.1 Die Service-Schnittstelle WebDialServerRemoteInterface

Die Schnittstelle `WebDialServerRemoteInterface` definiert die Remote-Methoden, die einem *webDial*-Client zur Nutzung der Server-Dienste zur Verfügung stehen. Beim Entwurf dieser Schnittstelle steht das Ziel im Vordergrund, mit möglichst wenigen Remote-Methoden auszukommen, um eine unnötige Netzwerkkommunikation zu vermeiden. Es folgt eine Beschreibung der Schnittstelle (vgl. Abbildung 7-14).

Da das RMI-Server-Objekt für eine nebenläufige Bearbeitung der Client-Anfragen verantwortlich ist, muss allen Remote-Methoden, die sitzungsabhängige Funktionen erfüllen, als erster Parameter der Login-Name des angemeldeten Benutzers übergeben werden. Bei Fehlern, die die Netzwerkkommunikation oder das RMI-Framework betreffen, kann jede dieser Methoden eine Ausnahme der Klasse `RemoteException` werfen. Die Methoden lassen sich den Bereichen

- Benutzerverwaltung,
- Telekommunikation und
- Teilnehmeridentifizierung

[3] Eine bekannte Implementierung ist der Tomcat Servlet- und JSP-Container. Er wird auf http://jakarta.apache.org/tomcat/ zum Download angeboten und darf gemäß der Apache Software License frei von Lizenzgebühren eingesetzt werden.

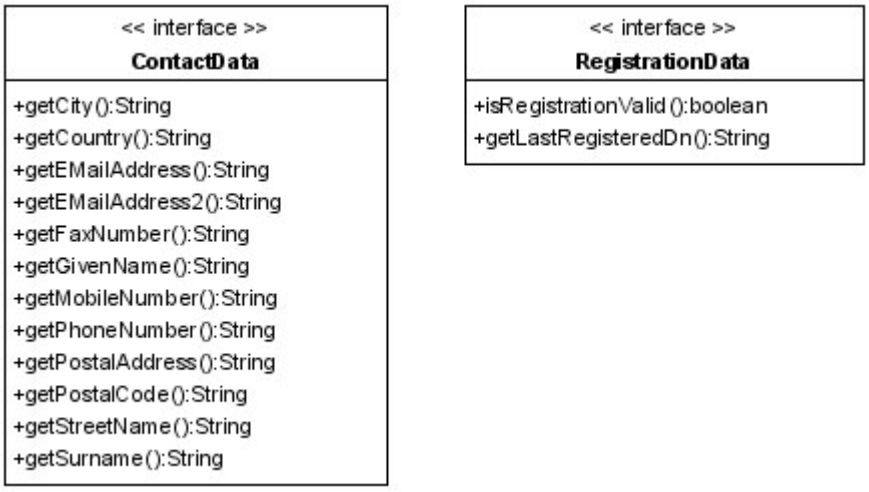

Abb. 7-15: Die Container ContactData und RegistrationData

zuordnen. Die Methoden `login`, `checkRegistration`, `addUserDeviceListener` und `removeUserDeviceListener` sind für die Benutzerverwaltung zuständig. Um die Methoden zur Ausführung von Telekommunikationsfunktionen verwenden zu können, muss der Anmelde- und Registrierungsprozess abgeschlossen sein.

login	
1. Parameter	`String` – Login
2. Parameter	`String` – Passwort
Rückgabetyp	`boolean` – Ergebnis der Anmeldung, `true` ⇒ erfolgreich, `false` ⇒ fehlgeschlagen
Exceptions	`RemoteException`
Beschreibung	Diese Methode meldet den durch den Login-Namen und Passwort spezifizierten Benutzer an. Bei Erfolg wird ein positiver, andernfalls ein negativer Boolescher Wert zurückgegeben.

addUserDeviceListener	
1. Parameter	`String` – Login
2. Parameter	`UserDeviceListener` – Callback-Objekt
Rückgabetyp	`void`
Exceptions	`RemoteException`, `UserUnknownException`

Beschreibung	Mit dieser Methode wird das Callback-Objekt für die Übermittlung der Statusinformationen registriert. Da das Callback-Objekt gleichzeitig die Telefonnummer des zu steuernden Apparats definiert, entscheidet der Serverprozess in diesem Schritt über die Notwendigkeit einer Registrierung. Dies wird dem Client im Anschluss über das eben registrierte Callback-Objekt mitgeteilt (vgl. nächstes Kapitel). Ist der spezifizierte Benutzer nicht angemeldet oder unbekannt, wird die Ausnahme `UserUnknownException` geworfen.
removeUserDeviceListener	
1. Parameter	`String` – Login
Rückgabetyp	`void`
Exceptions	`RemoteException`, `UserUnknownException`
Beschreibung	Mit dieser Methode wird die Sitzung des Benutzers beendet und das Callback-Objekt gelöscht. Ist der spezifizierte Benutzer nicht angemeldet oder unbekannt, wird die Ausnahme `UserUnknownException` geworfen.
checkRegistration	
1. Parameter	`String` – Login
Rückgabetyp	`RegistrationData` – Registrierungsinformationen
Exceptions	`RemoteException`, `UserUnknownException`
Beschreibung	Mit dieser Methode kann der aktuelle Status der Apparatregistrierung abgefragt werden. Der Rückgabewert ist vom Typ `RegistrationData`. Dabei handelt es sich um ein Containerobjekt, das über die Rufnummer des zuletzt registrierten Apparats und den Status der Registrierung Auskunft gibt. Ist der spezifizierte Benutzer nicht angemeldet oder unbekannt, wird die Ausnahme `UserUnknownException` geworfen.

Tab. 7-6: Remote-Methoden der Kategorie Benutzerverwaltung

Wurde der Benutzer korrekt authentifiziert, das Callback-Objekt angemeldet und liegt eine gültige Apparatregistrierung vor, können die Methoden für die Telekommunikationsaufgaben ausgeführt werden. Wird eine der Methoden aufgerufen, obwohl eine dieser Bedingungen nicht erfüllt ist, hat dies eine `DeviceRegistrationException` zur Folge. Die Ausnahme `InvalidDeviceStateException` tritt hingegen auf, wenn eine Methode zwar prinzipiell aufgerufen werden kann, jedoch der aktuelle Zustand des Call-Modells dies nicht zulässt. Unterstützt das zugrunde liegende Telekommunikationssystem die angeforderte Funktion nicht, tritt eine `FeatureNotSupportedException` auf. Methoden, die eine Zielrufnummer als Übergabeparameter erfordern, können zudem die Ausnahme `InvalidDnFormatException` verursachen, wenn die angegebene Rufnummer ungültige Zeichen enthält.

placeCall	
1. Parameter	`String` – Login
2. Parameter	`String` – Zielrufnummer
Rückgabetyp	`void`
Exceptions	`RemoteException`, `DeviceRegistrationException`, `InvalidDeviceStateException`, `FeatureNotSupportedException`, `InvalidDnFormatException`
Beschreibung	Diese Methode schaltet einen Telefonanruf zwischen dem registrierten Apparat und der angegebenen Rufnummer. Ist der Apparat, von dem der Anruf ausgehen soll, nicht frei, wird eine Ausnahme des Typs `InvalidDeviceStateException` ausgelöst.

placeConferenceConsultationCall	
1. Parameter	`String` – Login
2. Parameter	`String` – Zielrufnummer
Rückgabetyp	`void`
Exceptions	`RemoteException`, `DeviceRegistrationException`, `InvalidDeviceStateException`, `FeatureNotSupportedException`, `InvalidDnFormatException`
Beschreibung	Für den Aufbau oder die Erweiterung einer Konferenz schaltet diese Methode einen Consultation Call zwischen dem registrierten Apparat und der angegebenen Rufnummer. Ist der verursachende Teilnehmer nicht bereits an einem Main Call beteiligt, wird die Ausnahme `InvalidDeviceStateException` geworfen.

placeTransferConsultationCall	
1. Parameter	`String` – Login
2. Parameter	`String` – Zielrufnummer
Rückgabetyp	`void`
Exceptions	`RemoteException`, `DeviceRegistrationException`, `InvalidDeviceStateException`, `FeatureNotSupportedException`, `InvalidDnFormatException`
Beschreibung	Für die Vermittlung eines Gesprächspartners an einen anderen Teilnehmer schaltet diese Methode einen Consultation Call zwischen dem registrierten Apparat und der angegebenen Rufnummer. Ist der verursachende Teilnehmer nicht bereits an einem Main Call beteiligt, wird die Ausnahme `InvalidDeviceStateException` geworfen.

completeConsultationCall	
1. Parameter	`String` – Login
Rückgabetyp	`void`
Exceptions	`RemoteException`, `DeviceRegistrationException`, `InvalidDeviceStateException`, `FeatureNotSupportedException`

Beschreibung	Diese Methode schließt das durch `placeConferenceConsultationCall` respektive `placeTransferConsultationCall` aufgebaute Konsultationsgespräch in Hinblick auf die Weitervermittlung eines Teilnehmers oder die Aufnahme eines Teilnehmers in eine Konferenzschaltung ab. Das Konsultationsgespräch ist danach beendet und im Falle eines Transfers wird der registrierte Apparat aufgelegt. Ist der Apparat des Teilnehmers nicht an einem Main sowie Consultation Call beteiligt, wird eine Ausnahme des Typs `InvalidDeviceStateException` geworfen.

directTransferParty	
1. Parameter	`String` – Login
2. Parameter	`String` – Zielrufnummer
Rückgabetyp	`void`
Exceptions	`RemoteException`, `DeviceRegistrationException`, `InvalidDeviceStateException`, `FeatureNotSupportedException`, `InvalidDnFormatException`
Beschreibung	Für die Vermittlung eines Gesprächspartners an einen anderen Teilnehmer ahmt diese Methode einen Single-Step-Transfer nach, indem die Aktionen `placeTransferConsultationCall` und `completeConsultationCall` direkt nacheinander ausgeführt werden. Der Benutzer muss dabei nicht auf den Aufbau der Konsultationsverbindung warten. Ist der verursachende Teilnehmer nicht bereits an einem Main Call beteiligt, wird die Ausnahme `InvalidDeviceStateException` geworfen.

disconnectActiveCall	
1. Parameter	`String` – Login
Rückgabetyp	`void`
Exceptions	`RemoteException`, `DeviceRegistrationException`, `InvalidDeviceStateException`, `FeatureNotSupportedException`
Beschreibung	Diese Methode veranlasst den registrierten Apparat das aktive Gespräch zu verlassen. Bei zwei Gesprächen wird der Consultation Call beendet. Bei nur einem Gespräch wird der Apparat aufgelegt. Liegt kein aktives Gespräch vor, wird eine Ausnahme des Typs `InvalidDeviceStateException` geworfen.

retrieveCallOnHold	
1. Parameter	`String` – Login
Rückgabetyp	`void`
Exceptions	`RemoteException`, `DeviceRegistrationException`, `InvalidDeviceStateException`, `FeatureNotSupportedException`

Beschreibung	Diese Methode holt ein gehaltenes Gespräch zurück. Da eine dedizierte Hold-Funktion nicht verfügbar ist, ist dies nur bei Gesprächen sinnvoll, die im Zuge des Aufbaus von Konsultationsverbindungen gehalten wurden. Liegt kein gehaltenes Gespräch (im Status `TerminalConnection.HOLD`) vor, wird eine Ausnahme des Typs `InvalidDeviceStateException` geworfen.

Tab. 7-7: Remote-Methoden der Kategorie Telekommunikation

Die Methoden der Kategorie Teilnehmeridentifizierung versuchen mittels der Implementierung der Schnittstelle `PartyIdentification` über die Rufnummer den zugehörigen Teilnehmer zu identifizieren. Die Klasse `ContactData` dient diesen Methoden als Container für die Kontaktdaten.

identifyNumber	
1. Parameter	`String` – Rufnummer
Rückgabetyp	`ContactData` oder `null` – Kontaktinformationen
Exceptions	`RemoteException`
Beschreibung	Diese Methode versucht gemäß der übergebenen Rufnummer ein entsprechendes Objekt der Klasse `ContactData` zurückzuliefern. Dies kann fehlschlagen, wenn die Rufnummer nicht ermittelt werden oder der Dienst, der die Kontaktdaten anbieten soll, nicht erreicht werden kann. In diesen Fällen wird `null` zurückgeliefert.
identifyNumbers	
1. Parameter	`String[]` – Array mit Rufnummern
Rückgabetyp	`ContactData[]` – Array mit Kontaktinformationen
Exceptions	`RemoteException`
Beschreibung	Diese Methode dient der Identifizierung mehrerer Rufnummern in einem Vorgang. Die Rufnummern werden in einem String-Array übergeben. Die Position im Array definiert die Position im Rückgabe-Array der ContactData-Objekte. Können Rufnummern nicht identifiziert werden, steht an der korrespondierenden Position im Ergebnis-Array `null`.

getCallParticipants	
1. Parameter Rückgabetyp Exceptions	`String` – Login `String[]` – Array mit Rufnummern `RemoteException`, `DeviceRegistrationException`, `FeatureNotSupportedException`
Beschreibung	Diese Methode ermittelt die Rufnummern aller beteiligten Parteien, die an Gesprächen des registrierten Apparats teilnehmen. Deren Identifizierung kann mit `getCallParticipants` erfolgen.

Tab. 7-8: Remote-Methoden der Kategorie Teilnehmeridentifizierung

Bei der Implementierung einer Client-Anwendung sollte darauf geachtet werden, dass beim Beenden der Sitzung der `UserDeviceListener` deregistriert wird, um ein unnötiges Vorhalten der Sitzungsdaten bis zur nächsten Anmeldung zu vermeiden.

7.5.1.2 Die Callback-Schnittstelle UserDeviceListener

Die Schnittstelle `UserDeviceListener` definiert die Remote-Methoden, über die eine Client-Anwendung durch den *webDial*-Dienst über Ereignisse informiert wird. Beim Entwurf dieser Schnittstelle stand ebenfalls das Ziel im Vordergrund, mit möglichst wenigen Remote-Methoden auszukommen. Es folgt eine Beschreibung der einzelnen Ereignismethoden und deren Verhalten.

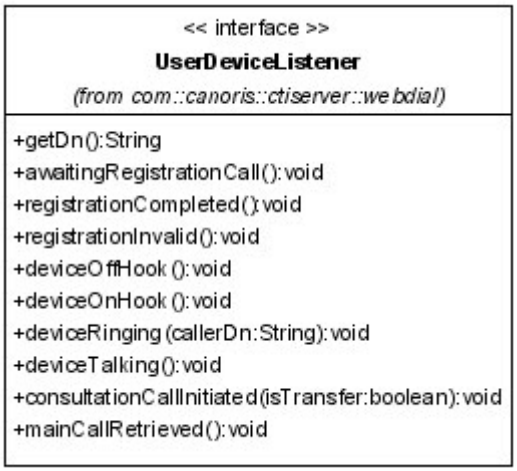

Abb. 7-16: Die Listener-Schnittstelle UserDeviceListener

getDN	
Rückgabetyp	`String` – Rufnummer des zu steuernden Apparats
Exceptions	`RemoteException`
Beschreibung	Diese Methode ermöglicht dem *webDial*-Dienst das Ermitteln der Rufnummer des zu steuernden Apparats zur Bestimmung des Status der Registrierung sowie die Erzeugung und Anmeldung der Listener `DeviceRegistrationListener` und `JTAPIDeviceListener`.

awaitingRegistrationCall	
Rückgabetyp	`void`
Exceptions	`RemoteException`
Beschreibung	Diese Methode signalisiert dem Client, dass eine Registrierung des Apparats notwendig ist und deshalb ein `DeviceRegistrationListener` am Registrierungsapparat angemeldet worden ist und auf den Registrierungsanruf wartet.

registrationCompleted	
Rückgabetyp	`void`
Exceptions	`RemoteException`
Beschreibung	Diese Methode teilt dem Client mit, dass der Registrierungsanruf eingegangen ist und der Registrierungsprozess erfolgreich abgeschlossen wurde.

registrationInvalid	
Rückgabetyp	`void`
Exceptions	`RemoteException`
Beschreibung	Es können Ereignisse innerhalb des *webDial*-Dienstes auftreten, die eine bisher gültige Registrierung verfallen lassen. Dies wird dem Client über diese Methode mitgeteilt, sodass über die Methode `addUserDeviceListener` der Server-Schnittstelle `WebDialServerRemoteInterface` ein erneuter Registrierungsprozess gestartet werden kann.

deviceOffHook	
Rückgabetyp	`void`
Exceptions	`RemoteException`
Beschreibung	Diese Nachricht teilt dem Client mit, dass der registrierte Apparat abgehoben wurde. Dies kann im Rahmen einer der Telekommunikationsfunktionen der Server-Schnittstelle, durch manuelles Abheben des Hörers oder durch das Betätigen einer der Funktionstasten des Systemtelefons geschehen.

deviceOnHook	
Rückgabetyp	`void`
Exceptions	`RemoteException`

Beschreibung	Diese Nachricht signalisiert dem Client, dass der registrierte Apparat aufgelegt wurde. Dies kann im Rahmen der Methode `disconnectActiveCall` der Server-Schnittstelle, durch manuelles Auflegen des Hörers sowie durch die Terminierung des letzten Gesprächs ausgelöst werden.
deviceRinging	
1. Parameter	`String` oder `null` – Rufnummer der anrufenden Partei
Rückgabetyp	`void`
Exceptions	`RemoteException`
Beschreibung	Diese Methode informiert den Client über einen ankommenden Anruf und die Rufnummer des Anrufers. Meist zeigt der registrierte Telefonapparat dies über ein akustisches Signal an. Kann die Rufnummer nicht ermittelt werden, ist der Rückgabewert `null`.
deviceTalking	
Rückgabetyp	`void`
Exceptions	`RemoteException`
Beschreibung	Diese Methode gibt dem Client zu verstehen, dass das registrierte Gespräch über eine aktive Verbindung verfügt. Ab diesem Zeitpunkt sind die Teilnehmer miteinander verbunden und können sprechen.
consultationCallInitiated	
1. Parameter	`boolean` – Zweck der Konsultation, `true` ⇒ Transfer, `false` ⇒ Konferenz
Rückgabetyp	`void`
Exceptions	`RemoteException`
Beschreibung	Diese Methode informiert den Client über den erfolgreichen Aufbau eines Konsultationsgespräches. Der Parameter gibt an, ob die Konsultation im Rahmen eines Transfers oder einer Konferenz stattfindet.
mainCallRetrieved	
Rückgabetyp	`void`
Exceptions	`RemoteException`
Beschreibung	Diese Methode zeigt dem Client an, dass das Zurückholen des Main Calls erfolgt ist und bestehende Consultation Calls beendet wurden. Trat dieses Ereignis im Zuge eines Transfer-Prozesses auf, folgt eine weitere Ereignisbenachrichtigung mittels `deviceOnHook`. Im Fall von Konferenzschaltungen kann die neue Teilnehmerliste mit der Methode `getCallParticipants` der Server-Schnittstelle ermittelt werden.

Tab. 7-9: Remote-Methoden des UserDeviceListeners

8 Implementierungsdetails

In diesem Kapitel wird auf Teile der Implementierung eingegangen. Dabei wird zunächst gezeigt wie in der Klasse `ProviderCallbackImpl` die Synchronisation der Zustände des JTAPI-Objektmodells erfolgt. Des Weiteren wird beschrieben wie die Objektserialisierung bei der Speicherung der Benutzerdaten angewandt wird. Abschließend wird die Umsetzung der Log-Funktionalität zur Protokollierung systeminterner Ereignissen erläutert.

8.1 Statusverarbeitung und Synchronisation des Call-Modells

Wie bereits festgestellt wurde, basiert die TK-Anlage *Meridian I* auf einem anderen Zustandsmodell als JTAPI. Deshalb können die durch den CTI-Link übermittelten Zustandsinformationen nicht direkt in das Call-Modell übernommen werden, sondern müssen zuvor in ein kompatibles Modell überführt werden. Zum einen verfügt JTAPI über ein detaillierteres Modell mit mehr Zuständen als die TK-Anlage und zum anderen deckt sich deren Aussagegehalt aufgrund der anders gelagerten Bedeutung nicht vollständig. Die Differenzierung zwischen physischer und logischer Adresse führt bei der *Meridian I*, im Gegensatz zu JTAPI, nicht zu einer konsequenten Unterscheidung ihrer Zustände. Ändert sich der Zustand eines Endgeräts der TK-Anlage, muss der übermittelte neue Status meist auf die zwei korrespondierenden Terminal- sowie Address-Objekte abgebildet werden. Wird ein neuer Zustand signalisiert, erfolgt daher zuerst auf Basis der DN die Ermittlung des betroffenen Address- und Terminal-Objekts. Betrifft das Ereignis zudem Teilnehmer außerhalb der Domain der TK-Anlage, gilt es zudem zu bewerten, inwieweit deren Status ebenso verändert werden muss. Für die Abbildung von Gesprächen mit externen Teilnehmern verfügt die Klasse `AddressImpl` über eine Schaltervariable. Diese zeigt an, ob es sich um eine interne oder externe Adresse handelt. Terminal-Objekte werden für solche Teilnehmer nicht erzeugt, da zum einen ohnehin kein Einfluss auf das Endgerät genommen werden könnte und zum anderen keine Informationen über die tatsächlichen Verhältnisse an diesem Endpunkt der Verbindung verfügbar sind.

Im Folgenden wird die Ereignisverarbeitung nach dem übermittelten Status aufgeschlüsselt und die Auswirkungen auf die Objekte des Call-Modells werden beschrieben. Es soll an dieser Stelle daran erinnert werden, dass sich eine Statusänderung immer auf ein Device bezieht und je nach Ausprägung des Status andere mit beeinflussen kann, jedoch nicht muss. Ein Kriterium stellt dabei stets die Beteiligung externer Teilnehmer dar, für die keine expliziten Statusinformationen übermittelt werden.

UNKNOWN Erfährt ein Device einen Wechsel in diesen Status, wird das Connection- sowie das TerminalConnection-Objekt in den Status `UNKNOWN` versetzt.

ONHOOK Dieser Zustand zeigt das Ende der Beteiligung an sämtlichen Gesprächen an. Deshalb werden alle Connection-Objekte der betroffenen Adresse in den Status `DISCONNECTED` und alle TerminalConnection-Objekte des betroffenen Terminals in den Status `DROPPED` versetzt. Verlässt damit der letzte interne Teilnehmer ein Gespräch wird der Status des Calls auf `INVALID` gesetzt. Die Connection-Objekte externer Teilnehmer wechseln in diesem Fall ebenso in den Status `DISCONNECTED`.

OFFHOOK Dieser Zustand impliziert den Beginn eines Gesprächs. Deshalb wird ein neues Call-Objekt erzeugt, dem für das betroffene Device ein neues Connection-Objekt im Status `INITIATED` und ein TerminalConnection-Objekt im Status `TALKING` hinzugefügt wird.

RINGING Eine Änderung in diesen Status bedeutet, dass das Device über den Eingang eines Anrufs benachrichtigt wird und dies dem Benutzer signalisiert. Dementsprechend wird das Connection-Objekt in den Status `ALERTING` und das TerminalConnection-Objekt in den Status `RINGING` versetzt.

ACTIVE Dieser Zustand zeigt den erfolgreichen Verbindungsaufbau an. Das Device ist nun Teil des Calls. Deshalb wird dessen Connection-Objekt in den Status `ESTABLISHED` versetzt. Falls nicht schon im Vorfeld geschehen, wird auch dessen TerminalConnection-Objekt in den Status `TALKING` versetzt.

DISCONNECT Dieser Status signalisiert, dass das Device ein Gespräch verlassen hat. Daher wird der Status des Connection-Objekts auf `DISCONNECTED` und der Status des TerminalConnection-Objekts auf `DROPPED` geändert. Hat mit diesem Device der letzte interne Teilnehmer das Gespräch verlassen, wird der Call in den Status `INVALID` versetzt. Die Connection-Objekte externer Teilnehmer wechseln in diesem Fall ebenso in den Status `DISCONNECTED`.

UNRINGING Diese Statusinformation besagt, dass ein eingehender Anruf vom Verursacher beendet wurde, bevor er angenommen werden konnte. Die Benachrichtigung des Benutzers durch ein akustisches oder visuelles Signal wird dabei beendet. Der Status des Connection-Objekts wird zu `DISCONNECTED` und der des TerminalConnection-Objekts zu `DROPPED` gewechselt. Handelt es sich beim Anrufer um einen externen Teilnehmer, wird auch dessen Connection-Objekt in den Status `DISCONNECTED` versetzt.

RETRIEVE Diese Zustandsänderung zeigt den Wechsel zu einer zuvor gehaltenen Verbindung an. Die Ausgangsverbindung wird dabei verlassen und beendet. Deshalb wird das TerminalConnection-Objekt des jetzt aktiven Gesprächs in den Status `TALKING` und, sofern dies noch nicht der Fall ist, das Connection-Objekt in den Status `ESTABLISHED` versetzt.

TRANSFERINITIATION Dieser Status signalisiert den Aufbau einer Konsultationsverbindung zum Zweck des anschließenden Weiterverbindens. Dafür wird ein neues Call-Objekt erzeugt und für das Device ein neues Connection-Objekt im Status `INITIATED` und ein TerminalConnection-Objekt im Status `TALKING` hinzugefügt. Da dieses Ereignis impliziert, dass das vorherige Basisgespräch gehalten wird, wird das TerminalConnection-Objekt zu diesem Call-Objekt in den Status `HELD` versetzt. Das entsprechende Connection-Objekt verbleibt im Status `ESTABLISHED`, da die Verbindung nicht beendet wurde, sondern nur gehalten wird.

CONFERENCEINITIATION Dieser Status signalisiert den Aufbau einer Konsultationsverbindung zum Zweck der anschließenden Zusammenschaltung zu einer Konferenz. Dafür wird ein neues Call-Objekt erzeugt und für das Device ein neues Connection-Objekt im Status `INITIATED` und ein TerminalConnection-Objekt im Status `TALKING` hinzugefügt. Da dieses Ereignis impliziert, dass das vorherige Basisgespräch gehalten wird, wird das TerminalConnection-Objekt zu diesem Call-Objekt in den Status `HELD` versetzt. Das entsprechende Connection-Objekt verbleibt im Status `ESTABLISHED`, da die Verbindung nicht beendet wurde, sondern nur gehalten wird.

TRANSFERCOMPLETEDTO Diese Zustandsänderung zeigt an, dass der Gesprächsparter des Konsultationsgesprächs mit dem Teilnehmer des Basisgesprächs verbunden wurde. Da infolgedessen das Konsultationsgespräch beendet wurde, werden das Connection-Objekt des betroffenen Devices in den Status `DISCONNECTED` und das entsprechende TerminalConnection-Objekt in den Status `DROPPED` versetzt. Handelt es sich dabei um den letzten der beiden Teilnehmer, der das Gespräch verlässt, wird der Status des Call-Objekts auf `INVALID` gesetzt.

CONFERENCECOMPLETEDTO Diese Statusänderung signalisiert die erfolgreiche Vereinigung der Teilnehmer des Konsultationsgesprächs mit denen des Basisgesprächs. Da infolgedessen das Konsultationsgespräch beendet wurde, werden das Connection-Objekt des betroffenen Devices in den Status `DISCONNECTED` und das entsprechende TerminalConnection-Objekt in den Status `DROPPED` versetzt. Handelt es sich dabei um den letzten der beiden Teilnehmer, der das Gespräch verlässt, wird der Status des Call-Objekts auf `INVALID` gesetzt.

CONFERENCETOSIMPLE Dieser Zustand zeigt an, dass der drittletzte Teilnehmer eine Konferenzschaltung verlassen hat, sodass diese zu einem normalen Gespräch mit zwei Teilnehmern wird. Diese Statusänderung wird nur im Fall externer Teilnehmer berücksichtigt, um trotz fehlender Statusinformationen feststellen zu können, dass eine Konferenzschaltung aufgrund zu weniger Teilnehmer beendet wurde. Trifft das eben genannte zu, wird der Status des Connection-Objekts des externen Teilnehmers auf `DISCONNECTED` gesetzt.

Über das eben genannte hinaus, spielt es bei der Synchronisation der Objektzustände eine Rolle, wann der CTI-Server gestartet wurde. Denn es ist äußerst wahrscheinlich, dass zu diesem Zeitpunkt bereits zahlreiche Gespräche aktiv sind. Für die entsprechende Anpassung

des Zustandsmodells bedeutet dies, dass Statusmeldungen bestehender Gespräche übermittelt werden, ohne zuvor die gesamte Entwicklung der Call-Situation erfasst haben zu können. Deshalb muss die JTAPI-Implementierung in der Lage sein die fehlenden Informationen adaptiv aus dem Inhalt der erhaltenen Statusinformationen zu erschließen. Im konkreten Fall bedeutet dies, dass für ein Device eine Statusmeldung mit dem Status `ACTIVE` übermittelt wird, obwohl dem Provider kein Call mit der entsprechenden Call-ID bekannt ist. Aus dieser Situation muss damit deduktiv abgeleitet werden, dass der Call-Aufbau verpasst wurde und entsprechend nachträglich Objekte zu erzeugen und Zustände anzupassen sind.

Eine weitere Problemstellung ergibt sich bei der Vergabe der Call-IDs. Diese erfolgt zwar in jedem Fall durch die TK-Anlage und wird stets zusammen mit der DN und der TN des betroffenen Endgeräts übertragen. Wird jedoch die Erzeugung eines Calls auf der Seite des Call-Modells durch die Methoden *Call.connect* und *CallControlCall.consult* verursacht, ist die Call-ID nicht bekannt, da die TK-Anlage diese erst im Zuge der Verarbeitung des Kommandos vergeben wird. Aus diesem Grund wird jedem Call-Objekt, dessen Erzeugung nicht von Statusmeldungen der TK-Anlage ausgelöst wird, eine temporäre ID zugewiesen. Diese unterscheidet sich in ihrer Integer-Repräsentation schlicht durch ein umgekehrtes Vorzeichen. Wird nun im Zuge der Statusübermittlung ein Call erfasst, dessen ID dem Provider unbekannt ist, wird zunächst überprüft, ob ein passendes Objekt mit temporärer ID existiert. Diese Suche erfolgt auf Basis der von der Statusänderung betroffenen Address- und Terminal-Objekte. Ist die Suche erfolgreich und wird ein entsprechendes Call-Objekt mit temporärer ID identifiziert, wird sie durch die richtige ID ersetzt. Andernfalls wird ein neues Call-Objekt erzeugt.

Unter Berücksichtigung der eben genannten Aspekte soll der Ablauf bei der Statusverarbeitung in der Methode `ProviderCallbackImpl.deviceStatusChange` beschrieben werden. Diese Methode wird im Rahmen der Schnittstelle `ProviderCallback` zur Übertragung der Statusinformationen verwendet und ist auf Provider-Seite für die entsprechende Aktualisierung des Call-Modells verantwortlich. Der Aufruf der Methode erfolgt im Treiber-Modul, wenn eine eingehende Nachricht eine Statusänderung mitteilt, wobei in Form von Übergabeparametern die folgenden Informationen übergeben werden:

- die DN des betroffenen Endgeräts,
- der neue Status,
- die Nachrichtenreferenznummer,
- die Call-ID,
- die DN anderer betroffener Endgeräte (optional) sowie
- die Call-ID in Beziehung stehender Calls (optional).

Auf Basis dieser Daten kann die Anpassung des Call-Modells erfolgen (vgl. Abbildung 8-1). Zu Beginn werden mit der DN als Schlüssel das Address- und das Terminal-Objekt des Apparats ermittelt, was in jedem Fall erfolgreich ist, da bei der Initialisierung des Providers sichergestellt wurde, dass für jede Rufnummer ein Address- und ein verbundenes Terminal-Objekt erzeugt wird. Danach wird mit der Call-ID versucht, das Call-Objekt zu ermitteln. An dieser Stelle ergeben sich drei mögliche Ablaufpfade.

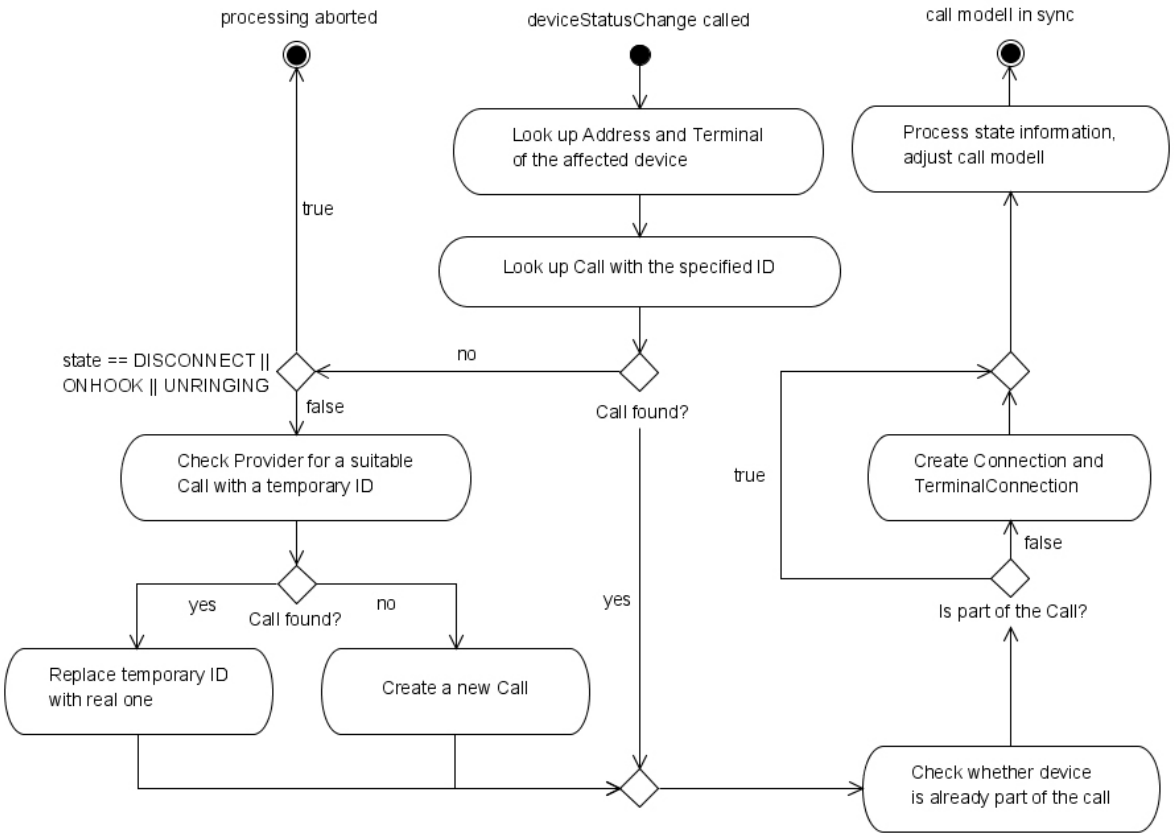

Abb. 8-1: Ablaufplan der Statusverarbeitung in der Methode deviceStatusChange

(1) Es wird kein Call-Objekt mit dieser ID gefunden und die Suche nach einem passenden Objekt mit temporärer ID liefert ebenfalls kein Ergebnis. Es wird ein neues Call-Objekt erzeugt.

(2) Es wird ebenfalls kein Call-Objekt gefunden. Jedoch kann ein Objekt mit temporärer ID ermittelt werden. Die temporäre ID wird mit der aktuellen ersetzt.

(3) Es wird ein passendes Call-Objekt gefunden und mit dem nachfolgenden Schritt fortgefahren.

Nun ist in jedem Fall ein Call-Objekt verfügbar. Sofern nicht bereits geschehen, werden jetzt die Connection- und TerminalConnection-Objekte erzeugt, die das Address- und das Terminal-Objekt mit dem Call-Objekt verbinden. Nach dem eingangs erläuterten Schema erfolgt nun abschließend die Aktualisierung der Objekt-Zustände.

8.2 Objektserialisierung zur Speicherung der Benutzerdaten

Es werden im Rahmen von *webDial* momentan nur Benutzerinformationen gespeichert, die für den Apparat-Registrierungsprozess notwendig sind. Die wichtigsten Inhalte sind dabei der Zeitpunkt der letzten erfolgreichen Registrierung und die Rufnummer des registrierten Apparats. Aufgrund dieser Daten wird entschieden, ob eine erneute Registrierung notwendig ist. Damit diese Informationen nicht bei einem Systemneustart verloren gehen, werden sie in eine Datei gespeichert. Die Nutzung eines Datenbanksystems war in Anbetracht der wenigen zu speichernden Informationen nicht praktikabel. Aus diesem Grund wurde auf die Objektserialisierung zurückgegriffen. Dabei werden Objekte auf eine wohl definierte Weise in eine Byte-Sequenz umgewandelt, um sie auf ein serielles Medium schreiben zu können. Die Byte-Sequenz kann danach wieder eingelesen und das ursprüngliche Objekt erzeugt werden.[1]

Die Benutzerinformationen werden in einer Collection des Typs `Hashtable` gespeichert. In wohl definierten Zyklen schreibt ein Thread mittels Objektserialisierung die ganze Collection in eine Datei. Das Listing 8-1 zeigt diesen Vorgang. Zunächst wird mit der Erzeugung eines Objekts des Typs `FileOutputStream` ein Ausgabestrom zum Schreiben in eine Datei geöffnet. Der Konstruktorparameter `saveFile` definiert dabei die zu öffnende Datei, während der zweite Parameter mit `false` dafür sorgt, dass die Datei bei jeder Speicherung überschrieben wird. Im zweiten Schritt wird mit `ObjectOutputStream` ein weiterer Ausgabestrom zum serialisierten Schreiben beliebiger Java-Objekte initialisiert. Als Parameter wird dem Konstruktor der zuvor erzeugte Ausgabestrom für das Schreiben in die Sicherungsdatei übergeben. Damit ergibt sich eine Verkettung zweier Ausgabeströme, die bewirkt, dass mit dem Aufruf der Methode `writeObject` des Objektausgabestroms ein übergebenes Objekt serialisiert und in die Datei geschrieben wird (vgl. Zeile 8).

```
//pre-condition: userHistories is an instance of
//Hashtable<String,WebDialUserHistory>
String saveFile = "/var/webdial/data/user-data.save";
FileOutputStream fileOutStream =
        new FileOutputStream(saveFile, false);
ObjectOutputStream objectOutStream =
        new ObjectOutputStream(fileOutStream);
objectOutStream.writeObject(userHistories);
```

Prog. 8-1: Speicherung der Benutzerdaten mit Hilfe der Objektserialisierung

Analog verläuft der Ladeprozess, der bei jedem Neustart des *webDial*-Dienstes durchlaufen wird. Zunächst wird durch die Erzeugung eines Objekts der Klasse `FileInputStream` ein Eingabestrom zum Einlesen von Daten aus der Sicherungsdatei geöffnet (vgl. Listing 8-2). Darauf wird mit `ObjectInputStream` der zweite Eingabestrom zum Einlesen serialisierter Java-Objekte erzeugt. In der Zeile 6ff. wird durch Aufruf der Methode `readObject`

[1] Tatsächlich handelt es sich dabei um ein neues Objekt, das sich lediglich im gleichen Zustand befindet wie das zuvor gespeicherte.

das Einlesen der Byte-Folge ausgelöst. Da die Methode ein Objekt der Basisklasse `Object` zurückgibt, muss dieses noch mittels eines expliziten Type-Casts in den gewünschten Typ `Hashtable<String,WebDialUserHistory>` umgewandelt werden.

```
String saveFile = "/var/webdial/data/user-data.save";
FileInputStream fileInStream =
        new FileInputStream(saveFile);
ObjectInputStream objectInStream =
        new ObjectInputStream(fileInStream);
userHistories.putAll(
  (Hashtable<String,WebDialUserHistory>)
        objectInStream.readObject()
        );
```

Prog. 8-2: Laden der Benutzerdaten mit Hilfe der Objektserialisierung

8.3 Protokollierung mit java.util.logging

Die Ereignisprotokollierung ist für das Erheben statistischer Daten und die Fehlersuche von zentraler Bedeutung, deshalb wurde im gesamten System davon Gebrauch gemacht. Die Protokollierung eines Systems, das aus mehreren Modulen besteht und mit mehreren Threads arbeitet, stellt dabei besondere Anforderungen. Wenn Module ausgetauscht oder neu hinzugefügt werden, soll die Protokollierung homogen fortgeführt werden können. Zudem sollte eine Implementierung Thread-sicher sein, um die Benutzung der Protokollierungsressourcen mehreren Threads ermöglichen zu können. Außerdem müssen die Ressourcen an jeder Stelle des Codes verfügbar sein.

Das Paket `java.util.logging` bietet eine praktische, Thread-sichere und zudem einfach zu handhabende Lösung. Das Paket enthält die Klasse `Logger`, die über eine statische Methode `getLogger` zur Erzeugung eines Logger-Objekts verfügt. Diese Methode akzeptiert einen String-Parameter, der den Namen des Logger-Objekts definiert, das zurückgegeben werden soll. Wird ein bestimmter Logger zum ersten Mal angefordert, wird er erzeugt und am `LogManager` registriert. Diese Klasse verwaltete die erzeugten Logger-Instanzen, sodass jede nachfolgende Abfrage mit `getLogger` die Referenz auf das bestehende Logger-Objekt zurückliefert. Die Klasse Logger implementiert für die Protokollierung alle notwendigen Funktionen. Die Erzeugung kann nur über die static-Methode `getLogger` erfolgen. Der Konstruktor ist `protected` und kann deshalb nicht zur Objekterzeugung eingesetzt werden. Bei diesem Verfahren handelt es sich um das *Singleton*-Erzeugungsmuster mit einem parametrisierten static-Konstruktor [GAMMA96]. Der statische Charakter der Methode `getLogger` verbirgt die dynamische Objektverwaltung des `LogManager` und macht zugleich das Logger-Objekt an jeder Codestelle verfügbar, ohne dass ständig Variablen für die Referenz auf das Logger-Objekt mitgeführt werden müssen.

Das Paket definiert so genannte Log-Levels, die die Kategorisierung der Log-Nachrichten

ermöglichen. Dabei wird jeder zu protokollierenden Nachricht eine Dringlichkeitsstufe zugewiesen. Die folgende Auflistung zeigt die Dringlichkeitsstufen in aufstrebender Rangordnung:

- `FINEST`,
- `FINER`,
- `FINE`,
- `INFO`,
- `WARNING` und
- `SEVERE`.

Ein Logger-Objekt kann auf eine bestimmte Stufe konfiguriert werden, sodass alle Nachrichten mit der entsprechenden oder höheren Dringlichkeitsstufe aufgezeichnet werden. Nachrichten mit niedrigerer Stufe werden ignoriert. Zudem existieren mit `ALL` und `OFF` zwei Sonderfälle. `All` ermöglicht das Protokollieren aller Nachrichten, während `OFF` die Protokollierung vollständig deaktiviert.

Nach diesem Konzept wurden die Log-Nachrichten implementiert. Alle schwerwiegenden Fehler, vor allem diejenigen, die einen Programmabbruch verursachen, werden mit der Stufe `SEVERE` protokolliert. Ereignisse, die keinen fatalen Programmabbruch verursachen, aber dennoch die Funktionsfähigkeit des Systems beeinträchtigen können, werden mit `WARNING` erfasst. Die Stufe `INFO` wird zur Protokollierung interner Abläufe verwendet. Für die Fehlersuche wurden Log-Nachrichten in den Stufen `FINE` und `FINER` implementiert, die in unterschiedlicher Granularität viele Status- und Ablaufinformationen liefern. Das System definiert die Logger-Objekte

- `CommandLine` für die Ausgabe aller Nachrichten auf `stdout`,
- `CTIMain` für die Nachrichten den CTI-Server betreffend,
- `Provider` für die Nachrichten alle Provider-Instanzen betreffend,
- `MeridianDriver` für die Nachrichten alle Treiber-Instanzen betreffend und
- `WebDialServer` für die Nachrichten, die den *webDial*-Dienst betreffen.

Für jedes dieser Objekte kann in den Systemeinstellungen der Log-Level und der Ausgabepfad für die Log-File (ausgenommen `CommandLine`) separat spezifiziert werden. Listing 8-3 zeigt, wie das Logger-Objekt `CTIMain` erzeugt wird.

```
1 //pre-condition: ConfigDataContainer is a data structure
  //containing configuration information
  Logger aLogger = Logger.getLogger("CTIMain");
```

```
//send all loggable messages to the parent logger
aLogger.setParent(commandLine);
aLogger.setUseParentHandlers(true);

//create FileHandler for logging
String logFilePath = ConfigDataContainer.getLogCtiMainOutput();
FileHandler theFileHandler;
theFileHandler = new FileHandler(logFilePath,true);

//attach handler for simple non-xml output
theFileHandler.setFormatter(new SimpleFormatter());

//set loglevel
theFileHandler.setLevel(
        ConfigDataContainer.getLogCtiMainLevel());

aLogger.setLevel(ConfigDataContainer.getLogCtiMainLevel());

//add the FileHandler to logger
aLogger.addHandler(theFileHandler);

//test logger by sending messages of levels SEVERE and INFO
aLogger.severe("This is a severe message.");
aLogger.info("This is only an informational message.");
```

Prog. 8-3: Ereignisprotokollierung mit java.util.logging

9 Test

Ursprünglich war es Teil der Zielsetzung die Testphase während der Projektlaufzeit durchzuführen und abzuschließen. Es traten jedoch innerbetriebliche Umstände ein, die sich negativ auf den Projektverlauf auswirkten. Allem voran ist das Fehlen der Spezifikation für das Meridian-Link-Protokoll zu nennen, die zu Beginn des Projektes angekündigt wurde, mir jedoch nie zur Verfügung stand. Dies machte meinerseits eine nicht eingeplante zeitaufwändige Analyse der vorherigen Implementierung der *webDial*-Software notwendig, während derer das Link-Protokoll systematisch abgeleitet werden musste. Zum Teil erfolgte dies auf Basis des Quelltextes und zum Teil durch genaue Analyse des Netzwerkverkehrs zwischen Software und PBX. Zudem wirkte sich eine finanzielle Engpasssituation innerhalb des Unternehmens ungünstig auf die Arbeitsbedingungen aus. In Verbindung mit dem unterschätzten Entwicklungsaufwand war eine rechtzeitige Fertigstellung der Testphase nicht mehr möglich. Mein Versäumnis war es, nicht rechtzeitig mit Nachdruck auf die verzögernden Faktoren hinzuweisen.

Trotz des geschilderten Sachverhalts soll an dieser Stelle auf die Testabläufe eingegangen werden. Dabei wird zwischen den Testkategorien

- Komponententest,
- Funktionstest und
- Integrationstest

unterschieden. Die Komponententests befassen sich mit der Funktionsprüfung der Bestandteile des entwickelten Systems. Im Einzelnen werden die Implementierungsklassen dahingehend getestet, inwiefern sie sich gemäß ihrer Schnittstellenspezifikation verhalten. Dabei können logische Programmierfehler und in begrenztem Maße auch semantische Fehler entdeckt werden. Wurde mit Hilfe der Komponententests die Funktionsfähigkeit der einzelnen Teile sichergestellt, kann im Rahmen der Funktionstest überprüft werden, ob das System den funktionalen Anforderungen gerecht wird. Abschließend wird mit Hilfe der Integrationstests die korrekte Funktionsweise innerhalb der Systemumgebung verifiziert. Eine Akzeptanztestreihe ist im Rahmen der Entwicklung des CTI-Servers nicht sinnvoll, ohne dass vorher eine Client-Implementierung fertiggestellt wurde.

Mit den Komponententests werden die Methoden mit der Sichtbarkeit `public` und `protected` der Implementierungsklassen getestet. Hierfür werden Testfälle aufgestellt, die das Verhalten der Methoden in Standard- und Sonderfällen untersuchen. Um einen schnellen Testablauf zu ermöglichen, der bei Erweiterungen und Anpassungen des Systems beliebig

```
package com.canoris.ctiserver.webdial;
import com.canoris.ctiserver.ServiceException;
import junit.framework.TestCase;

public class WebDialControllerTest extends TestCase {

  protected WebDialController webDialController;

  public WebDialControllerTest(String name) {
    super(name);
    webDialController = new WebDialController();
  }

  public final void testLoginUser() {

    assertEquals(
      "Login with right password",
      webDialController.loginUser("userXY","rightPassword"),
      true
      );

    assertEquals(
      "Login with wrong password",
      webDialController.loginUser("userXY","wrongPassword"),
      false
      );

    assertEquals(
      "Login with unknown user",
      webDialController.loginUser("unknownUser","aPassword"),
      false
      );
  }
}
```

Prog. 9-1: Beispielhafter JUNIT-Testfall

wiederholt werden kann, werden die Tests, so weit möglich und sinnvoll, automatisiert. Mit *JUNIT* steht ein Test-Framework zur Verfügung, das Werkzeuge und Methoden zur Erstellung, Ausführung sowie Auswertung von automatisierten Tests bereitstellt.

Anhand der Methode `testLoginUser` der Klasse `WebDialController` zeigt das Listing 9-1 wie ein JUNIT-Testfall aussehen kann. Die Methode akzeptiert zwei Parameter für den Login und das Passwort. Werden Login und Passwort eines gültigen Benutzeraccounts

übergeben, liefert die Methode den booleschen Wert `true` zurück. Ist das Passwort falsch oder unter dem Login kein gültiger Useraccount zu finden, gibt die Methode `false` zurück. Diese drei Ablaufmöglichkeiten werden durch jeweils eine `assertEquals`-Anweisung abgedeckt. Als erster Parameter wird dieser Methode die Beschreibung des Testfalls übergeben, während die Parameter zwei und drei logisch miteinander verglichen werden. Ergibt die Evaluierung beider Parameter den gleichen Wahrheitswert, so gilt der Teiltest als bestanden. Ein solcher Test kann beliebig oft wiederholt werden, sodass bei Änderungen an der Implementierung ein erneuter Testlauf sofort sicherstellen kann, dass die neue Version ebenfalls korrekt funktioniert.

Die Funktionstests stellen sicher, dass die in Kapitel 4.1 aufgeführten funktionalen Anforderungen erfüllt werden. Da das Überprüfen der einzelnen Funktionen am einfachsten in einem realen System mit einer angeschlossenen TK-Anlage erfolgen kann, ist eine Automatisierung weniger sinnvoll. Deshalb wird ein Praxistest mit zwei Versuchspersonen durchgeführt. Diese gehen Anhand einer Anleitung vor, die spezifiziert, welche Tätigkeiten ausgeführt werden sollen und welches Ergebnis zu erwarten ist. Jede Person verfügt über zwei Telefonapparate, die jeweils an einer anderen TK-Anlage des Anlagenverbunds angeschlossen sind. Zudem werden zwei Mobiltelefone benötigt, die für den Aufbau externer Verbindungen eingesetzt werden. Mit dieser Ausgangssituation ist es nun möglich alle Call-Situationen durchzuspielen und die korrekte Verhaltensweise des Systems zu überprüfen. Zum einen wird getestet, ob das Ziel einer bestimmten Aktion erreicht werden kann, zum anderen wird festgestellt, ob die JTAPI-Implementierung jeweils ein korrektes Objekt-Modell zur Abbildung der einzelnen Situation erzeugt. Letzteres wird mit Hilfe der Protokollierungsfunktionen festgehalten. Es folgt eine Auflistung der identifizierten Testfälle.

Testfälle	
Rufannahme	
1	Manuelle Annahme eines Anrufs von Intern
2	Manuelle Annahme eines Anrufs von einer anderen TK-Anlage
3	Manuelle Annahme eines Anrufs von Extern
Make Call	
4	Anruf eines internen Teilnehmers
5	Anruf eines internen Teilnehmers einer anderen TK-Anlage
6	Anruf eines externen Teilnehmers
Transfer	
7	Weiterleiten eines internen Teilnehmers an einen internen Teilnehmer
8	Weiterleiten eines internen Teilnehmers an einen Teilnehmer einer anderen TK-Anlage
9	Weiterleiten eines internen Teilnehmers an einen externen Teilnehmer
10	Weiterleiten eines Teilnehmers einer anderen TK-Anlage an einen internen Teilnehmer
11	Weiterleiten eines Teilnehmers einer anderen TK-Anlage an einen Teilnehmer einer anderen TK-Anlage

12	Weiterleiten eines Teilnehmers einer anderen TK-Anlage an einen externen Teilnehmer
13	Weiterleiten eines externen Teilnehmers an einen internen Teilnehmer
14	Weiterleiten eines externen Teilnehmers an einen Teilnehmer einer anderen TK-Anlage
Conference	
15	Aufbau einer Konferenz mit drei internen Teilnehmern
16	Aufbau einer Konferenz mit einem internen Teilnehmer und zwei Teilnehmern einer anderen TK-Anlage
17	Aufbau einer Konferenz mit einem internen und zwei externen Teilnehmern
18	Aufbau einer Konferenz mit einem internen, einem externen Teilnehmer und einem Teilnehmer an einer anderen TK-Anlage
Retrieve Main Call	
19	Zurückholen des gehaltenen Basisgesprächs
20	Zurückholen des gehaltenen Basisgesprächs aus einem Konsultationsgespräch heraus
Release Active Connection	
21	Beenden eines Gesprächs mit zwei Teilnehmern
22	Verlassen einer Konferenzschaltung mit verbleibendem internen Teilnehmer
23	Verlassen einer Konferenzschaltung als letzter interner Teilnehmer

Tab. 9-1: Auflistung der Testfälle der Funktionstests

Die Überprüfung der Funktion zur Identifizierung anderer Teilnehmer erfolgt im Rahmen der Integrationstests, da der verwendete Verzeichnisdienst ein externes System darstellt. Mit den Integrationstests soll im Wesentlichen das Performanceverhalten unter realen Bedingungen ermittelt werden. Dazu wird in einer ersten Phase das System während normaler Arbeitszeiten aktiviert. Dabei soll nur die Zustandssynchronisation mit den TK-Anlagen und keine Bearbeitung von Client-Anfragen stattfinden. Über entsprechende Monitoring-Tools werden Faktoren wie die CPU- und Speicherauslastung festgehalten. In einer zweiten Phase wird unter verschiedenen Lastsituationen (bspw. nach Feierabend, am Wochende und vormittags an einem Werktag) das Antwortverhalten bei Client-Anfragen (unter Nutzung der Client-API) untersucht. Auf Basis der hierdurch erhobenen Werte, können genauere Aussagen bezüglich der Hardwareanforderungen getroffen werden. Überdies wird bei den Integrationstests, die korrekte Interaktion mit externen Systemen getestet. In diesem Bereich erfolgt lediglich der Test der LDAP-Anbindung im Rahmen der Benutzerauthentifizierung und der Teilnehmeridentifizierung.

Wie eingangs bereits erwähnt wurde, konnte die Testphase nicht vollständig abgeschlossen werden. Während die Funktionstests zu 90% abgeschlossen wurden und eine Erfüllung der funktionalen Anforderungen nachgewiesen ist, sind die Komponententests lediglich zu 30 % fertiggestellt. Mit den Integrationstests konnte noch nicht begonnen werden.

10 Systemanforderungen, Installation und Konfiguration

Dieses Kapitel bietet einen Überblick über die Systemanforderungen, die Installation sowie über die Konfiguration des CTI-Servers. Die Systemanforderungen sind Tabelle 10-1 zu entnehmen, wobei die angegebenen Hardwareanforderungen auf Erfahrungen basierende Annahmen darstellen. Das bisher beobachtete Performanceverhalten des Systems deutet jedoch auf geringere Mindestanforderungen hin. Dennoch muss bei der Wahl der Hardware berücksichtigt werden, wie viele TK-Anlagen angesteuert werden sollen. Je größer die Anzahl, desto mehr steigt die CPU-Last im Zuge des erhöhten Aufwands bei der Synchronisierung der Zustandsmodelle mit den TK-Anlagen. Der Synchronisierungsprozess ist unter Berücksichtigung der anderen Aufgaben als maßgebend für die Gesamtlast zu erachten.

Das Programm besteht aus einem Jar-Archiv, in dem alle class-Dateien enthalten sind, einer Konfigurationsdatei und einem Shell-Skript. Die Installation erfolgt lediglich durch das Kopieren des Installationsordners an eine geeignete Stelle in der Verzeichnishierarchie. Zudem muss darauf geachtet werden, dass der Prozess über die nötigen Zugriffsrechte auf das Installationsverzeichnis, die Log-Files und die Ausgabedatei für die Speicherung der Benutzerdaten verfügt. In welchem Benutzerkontext der Server auszuführen ist, wird nicht vorgeschrieben. Das Installationsverzeichnis enthält die Unterverzeichnisse

- `bin` mit dem Shellskript `ctiserver`,
- `lib` mit dem jar-Archiv `CanorisCTIService.jar`,
- `etc` mit der Konfigurationsdatei `CanorisCTIService.conf`,
- `log` für die Ausgabe der Log-File (konfigurierbar) und
- `data` für die Ausgabe der Datei zur Speicherung der Benutzerdaten (konfigurierbar).

Mit dem Shell-Skript `ctiserver` kann der CTI-Server gestartet und gestoppt werden. Für den Start wird die JVM mit dem *Java Application Launcher* `java` unter Angabe des jar-Archivs und der Klasse `CanorisCTIService` gestartet. Falls notwendig wird dabei ebenfalls die RMI-Registry gestartet. Auf Fehler in der Konfigurationsdatei wird an dieser Stelle mit einem Abbruch des Startvorgangs und einer entsprechenden Meldung hingewiesen. Wichtig ist dabei, dass die Programme `rmiregistry` und `java` im Suchpfad zu finden sind. Alternativ kann die Umgebungsvariable `JAVA_HOME` gesetzt werden. Für das Stoppen des CTI-Servers greift das Shell-Skript auf das Admin-Tool zurück. Dieses ermöglicht neben dem Beenden des gesamten Prozesses auch das Starten und Stoppen der einzelnen Teildienste. Das

Systemanforderungen	
Betriebssystem	Linux (prinzipiell alle, für die eine JVM zur Verfügung steht; das Start-Skript liegt jedoch nur in einer Variante für die Bash-Shell vor)
Java Runtime Environment	Version 5.0 Update 1 oder höher
Hardware Arbeitsspeicher	CPU mit 1 GHz oder höher 512 MB oder höher

Tab. 10-1: Systemanforderungen

Shellskript `ctiserver` akzeptiert dementsprechend die folgenden Parameter, die einzeln anzugeben sind:

- `start` startet den CTI-Server und die vorhandenen CTI-Dienste,
- `stop` stoppt den CTI-Server,
- `stopwebdial` stoppt den *webDial*-Dienst,
- `startwebdial` startet den *webDial*-Dienst.

Beim Start des Servers kann mit der Option `--config <log file>` zudem eine andere Konfigurationsdatei spezifiziert werden. Bei der Konfigurationsdatei `CanorisCTIService.conf` handelt es sich um eine Textdatei, die im ASCII-Zeichensatz die Konfigurationsdaten in Form von Java-Properties (`java.util.Properties`) enthält. Das Listing 10-1 zeigt die Konfigurationsdatei mit den Standardeinstellungen. Nähere Informationen können den Kommentaren entnommen werden. Dabei ist zu beachten, dass die Zeilenumbrüche innerhalb einzelner Konfigurationseinträge für das Layout aufgrund zu langer Zeilen eingefügt werden mussten. Denn in Property-Konfigurationsdateien werden einzelne Einträge durch Zeilenumbrüche abgeschlossen.

```
1 //This is the configuration file for the CanorisCTIService.
  //It is a java properties file. All properties are compulsory.
  //values are divided by comma; one property per line;

  //section -- logging
6 logLevelCommandLine = WARNING
  logXMLCommandLine = false
  logfileCTIMain = ../log/ctiMain.log
  logLevelCTIMain = WARNING
  logXMLCTIMain = false
11 logfileProvider = ../log/provider.log
  logLevelProvider = WARNING
```

```
logXMLProvider = false
logfileMeridianDriver = ../log/meridianDriver.log
logLevelMeridianDriver = WARNING
logXMLMeridianDriver = false
logfileWebDialServer = ../log/webDialServer.log
logLevelWebDialServer = WARNING
logXMLWebDialServer = false

//section -- JTAPI and TK/CTI-Link
//order defines affected pbx
tkIdList = PBX_ID_1,PBX_ID_2
ctiLinkIP = 192.168.100.12,192.168.100.13
ctiLinkPort = 3000,3000
tkHostName = Lanlink,Lanlink
tkApplicationId = api_tool,api_tool
tkMeridianName = SL16,SL16
providerName = PBX.Meridian_I,PBX.Meridian_I
jtapiPeerImpl =
        com.canoris.ctiserver.provider.DefaultJtapiPeer,
        com.canoris.ctiserver.provider.DefaultJtapiPeer

//section -- site-specific configuration data
maxDnLength = 4
dnRanges = 4000-4010,5002,5090-5100
countryPrefix = 0049
areaPrefix = 0711
sitePrefix = 972
dialoutPrefix = 0

//section -- webDial-specific configuration data
numberOfRegistrationDevice = 6000;
persistenceManager =
        com.canoris.ctiserver.webdial.
                PersistenceManagerObjectSerialize
userDataSaveFile = ../data/userData

userAuthenticationImpl =
        com.canoris.ctiserver.webdial.LdapDirectoryService
partyIdentificationImpl =
        com.canoris.ctiserver.webdial.LdapDirectoryService
ldapInitialContextFactory =
        com.sun.jndi.ldap.LdapCtxFactory
ldapServerIp = 172.16.1.199
ldapServerPort = 389
```

```
ldapUser = cn=webdial,dc=canoris,dc=com
ldapPassword = secret
ldapAuthType = simple
ldapUserAccountContext = ou=users,ou=groups,dc=canoris,dc=com
ldapUserAccountObjectClass = simpleSecurityObject
ldapUserAccountLoginAttribute = uid
ldapUserAccountPasswordAttribute = userPassword
ldapUserAccountSearchDepth = subtree //or onelevel
ldapContactsContext = ou=contacts,ou=groups,dc=canoris,dc=com
ldapContactsObjectClass = inetOrgPerson
ldapContactsAttribute =
        telephoneNumber,mobile,facsimileTelephoneNumber
ldapContactsSearchDepth = subtree //or onelevel

registrationValidityPeriod = 43545600
```

Prog. 10-1: Die Konfigurationsdatei CanorisCTIService.conf

11 Ausblick und Resümee

11.1 Mögliche Weiterentwicklungen

Im weiteren Projektverlauf ist zunächst die Testphase abzuschließen, um ein stabiles System gewährleisten zu können. Im Einzelnen betrifft dies die Komponenten- und Integrationstests. Dafür ist ein Zeitaufwand von einem Monat zu veranschlagen. Zeitgleich könnte die Entwicklung des HTTP-Dienstes für die Client-Komponente stattfinden, um möglichst bald einen produktiven Status zu erreichen. Nach Abschluss dieser Phase kann die Weiterentwicklung beginnen. Es ist sowohl eine Erweiterung des Funktionsumfanges des *webDial*-Dienstes denkbar, als auch die Entwicklung völlig neuer CTI-Dienste. Im Folgenden werden einige der Erweiterungmöglichkeiten erläutert.

(1) Im Rahmen einer selbständigen, fensterbasierten Client-Software kann die Nutzung des *webDial*-Dienstes über eine GUI erfolgen, die dem dynamischen Verhalten eines Telekommunikationssystems besser gerecht wird, als eine Browser-gestützte HTML-Oberfläche. Die durch Java erreichte Plattformunabhängigkeit würde einen *dedizierten webDial-Client* ermöglichen, der in den gängigen Betriebssystemumgebungen ausgeführt werden kann.

(2) Eine Accounting- und Billingkomponente kann Informationen über das Verhalten der Anwender zur Berechnung nutzungsabhängiger Entgelte heranziehen. Diese Gebühreninformationen können dann in das bestehende Abrechnungssystem *TelAccount* exportiert werden.

(3) Eine detailliertere Erfassung des Benutzerverhaltens kann dazu genutzt werden, noch besser auf die Bedürfnisse der Anwender einzugehen.

(4) Die Unterstützung von Benutzerprofilen kann dazu dienen, die Benutzeroberfläche des *webDial*-Dienstes an die Wünsche und Anforderungen des Benutzers anzupassen. Unter anderem kann ein eigener Bereich für die Speicherung persönlicher Kontakte eingerichtet werden, der es dem Anwender erlaubt, Daten über Personen und Organisationen abzulegen, die nicht im zentralen Unternehmensverzeichnis verfügbar sind.

(5) Für den Fall, dass ein Benutzer mehrere Telefone gleichzeitig steuern möchte, kann der Registrierungsprozess für die Unterstützung mehrere Apparate angepasst werden.

(6) Um die Client/Server-Kommunikation in restriktiveren Umgebungen sicherzustellen, können mit Hilfe der Konfigurationsmöglichkeiten des RMI-Frameworks die zu verwendenden TCP-Ports eingestellt werden – Ports, die im betreffenden Unternehmensnetz nicht blockiert werden.

(7) Eine verbesserte Absicherung der Netzwerkkommunikation kann durch den optionalen, auf die Lastsituation des Servers anzupassenden Einsatz der *SSL/TLS-Socket-Factory-Klassen* `javax.rmi.ssl.SslRMIServerSocketFactory` und `javax.rmi.ssl.SslRMIClientSocketFactory` erreicht werden. Diese ermöglichen Socket-Verbindungen auf Basis der Kryptographieprotokolle SSL und TLS zur Sicherung der RMI-Netzwerkkommunikation zu Lasten eines höheren Rechenaufwands.

(8) Die Konfiguration des CTI-Servers kann in Zukunft auf Basis von XML-Dokumenten erfolgen. Dadurch können Fehler bei der Anpassung der Software durch die Überprüfung auf Gültigkeit der Konfiguration gegen ein W3C-Schema minimiert werden. Dabei kann das *Java API for XML Processing (JAXP)* eingesetzt werden, um mit Hilfe des XML-Parsers *Xerces 2* die Konfigurationsdaten in Form einer DOM[1]-Objektstruktur innerhalb des Systems zur Verfügung zu stellen.

(9) Das Zustandsmodell der JTAPI-Implementierung kann zur Realisierung eines eigenständigen Überwachungsdienstes eingesetzt werden. Dieser würde die Schnittstelle `CTIService` implementieren und zusammen mit *webDial* als gleichberechtigter CTI-Dienst ausgeführt werden. Die Überwachungsfunktionen könnten eine Übersicht über den Zustand bestimmter Terminals liefern oder die aktuelle Auslastung des System visualisieren.

(10) Durch Implementierung des JTAPI-Erweiterungspakets `javax.telephony.phone` kann ein Dienst zur Konfiguration der Systemtelefone realisiert werden. Über eine Oberfläche, die ein detailgetreues Abbild des Telefons wiedergibt, kann ein Benutzer die zu konfigurierenden Komponenten auswählen und manipulieren. Auf diese Weise können Tasten bestimmte Funktionen zugewiesen, die Lautstärke des Lautsprechers angepasst und die Helligkeit des Displays eingestellt werden. Der Dienst kann separat oder im Rahmen der *webDial*-Oberfläche zugänglich sein.

(11) Das Produkt *TelAssist* kann auf die Java-Technologie portiert werden, um als selbständiger CTI-Dienst neben *webDial* angeboten zu werden. Die zu implementierende zeitgesteuerte Konfiguration von Rufumleitungen kann über einen zyklisch ausgeführten Thread implementiert werden. Dieser setzt die Weiterleitungsregeln in Form von Objekten der Klasse `CallControlForwarding` (aus dem Call-Control-Paket) und hinterlegt sie bei den entsprechenden Address-Objekten.

(12) Der Funktionsumfang des *webDial*-Dienstes kann um ACD-Funktionen erweitert werden, sodass ein Benutzer Regeln für die Annahme eingehender Gespräche definieren kann. Eine Regel könnte beispielsweise externe Anrufer auf eine Sprachmailbox weiterleiten, während Anrufe interner Teilnehmer auf den Apparat durchgestellt werden. Auch die Konfiguration von Umleitungen auf andere Rufnummern könnte in diesem Rahmen angeboten werden.

[1] Document Object Model

(13) Eine andere funktionale Erweiterung ist die Erfassung entgangener Anrufe und die Visualisierung in einer übersichtlichen Darstellung, mit der Möglichkeit durch Anklicken eines Eintrags den Rückruf zu initiieren.

(14) Die Verwaltung der persönlichen Sprachmailbox kann ebenfalls in Form eines Dienstes umgesetzt werden. Neben dem Abrufen, Löschen oder Speichern von Nachrichten kann der Benutzer den Ansagetext oder die Vorhaltezeit konfigurieren. Es ist sogar denkbar hierfür ein eigenes Mailboxsystem auf Softwarebasis zu entwickeln.

Einige der eben genannten Erweiterungen sind direkt von den Fähigkeiten der CTI-Schnittstelle abhängig. Deshalb stellt ein erster Schritt zu deren Umsetzung das Ausschöpfen der Möglichkeiten dar, die das CTI-Link-Modul der *Meridian I* bereitstellt, wofür allerdings die Protokollspezifikation verfügbar sein muss. Die durch Reverse Engineering ermittelten Informationen über das Protokoll sind für weiterführende Funktionen unzureichend. Weitere Schritte betreffen die Unterstützung anderer Telekommunikationsanlagen, wie der *Alcatel OmniPCX 4400*, durch die Eigenentwicklung einer CSTA-Protokollimplementierung für ein neues Treiber-Modul. In Abhängigkeit von den verfügbaren Treibern und APIs kann die Adaption anderer Telekommunikationsanlagen auf verschiedene Weisen erfolgen.

(1) Im Idealfall wird für das Telefoniesystem eine Provider-Implementierung angeboten, die über die JtapiPeerFactory benutzt werden kann.

(2) Steht ein in Java realisierter Treiber zur Verfügung, kann dieser über eine angepasste Variante der vorhandenen JTAPI-Implementierung integriert werden.

(3) Wird der Treiber hingegen nur in Form einer plattformabhängigen Library angeboten, bleibt die Möglichkeit der Integration über das *Java Native Interface (JNI)*.

(4) Der Nutzung von JNI ist die Eigenentwicklung der Treiberkomponente auf Basis der Protokollspezifikationen (sofern verfügbar) vorzuziehen.

Die flexible Komponentenarchitektur des entwickelten Systems wird dabei auch in Zukunft die Anpassung an Telekommunikationssysteme anderer Anbieter ermöglichen und damit die Grundlage für neue Dienste und Funktionen schaffen.

11.2 Abschlussbemerkungen

Abschließend bleibt festzuhalten, dass es im Rahmen dieser Arbeit möglich war die gesetzten strukturellen und funktionalen Anforderungen zu erfüllen. Auch wenn die Testphase als letzter vorgesehener Projektabschnitt nicht vollständig abgeschlossen werden konnte, ist zumindest auf Basis der bisherigen Testergebnisse zu prognostizieren, dass in zeitnaher Zukunft ein produktiver Status erreicht werden kann.

Mit dieser Ausarbeitung wurde dargelegt wie – ausgehend von den Anforderungen an Struktur und Funktionalität – das Produktkonzept des monolithischen Vorgängers auf eine flexible Basis überführt wurde, die die Realisierung weiterer CTI-Dienste ermöglicht. Das vorherige

Kapitel zeigte bereits, welche zahlreichen Erweiterungsmöglichkeiten auf der Grundlage der entwickelten Software implementiert werden können. Der Einsatz zeitgemäßer Technologien wie die Java Plattform und das Java Telephony API tragen neben der modularen Gesamtkonzeption zu einem Produkt bei, das den zukünftigen Anforderungen gerecht wird.

Rückblickend auf den gesamten Projektverlauf kann ich für mich ein überwiegend positives Fazit ziehen. Es war mir durch die Arbeit möglich meine fachlichen Fähigkeiten zu verfeinern und wertvolle Erfahrungen zu sammeln. Es bestätigte sich zudem meine Ansicht, den Softwareentwicklungsprozess als einen komplexen, anspruchsvollen jedoch lohnenden Vorgang des Lernens und der ständigen Weiterentwicklung persönlicher wie fachlicher Kompetenzen aufzufassen. Es ergeben sich ständig andere Anforderungen und Aufgaben, deren Bewältigung stets neue Herausforderungen birgt. Dies zeichnet ein Berufsbild, das mich in meinem Bestreben bestätigt, meine berufliche Zukunft im Tätigkeitsbereich des Software-Engineerings zu suchen.

An dieser Stelle möchte ich mich noch bei allen herzlich bedanken, die mich in der vergangenen Zeit unterstützt und mir Rückhalt gegeben haben. Mein besonderer Dank gilt meinen Betreuern Herrn Dr. Ralf Leibscher und Herrn Dipl.-Phys. Thomas Jaekel, die wesentlich zum Entstehen dieser Arbeit beigetragen haben sowie meinen Eltern, die mir dieses Studium ermöglicht haben.

Literaturverzeichnis

[CHAN05] *Chandrasekhara V., Eberhart A., Hellbrück H., Kraus S., Walther U.*: JAVA 5.0 - KONZEPTE, GRUNDLAGEN UND ERWEITERUNGEN IN 5.0. Hanser 2005

[ECMAa94] *European Computer Manufacturer Association International*: ECMA-217 SERVICES FOR COMPUTER SUPPORTED TELECOMMUNICATIONS (CSTA) PHASE II. Dezember 1994

[ECMAb94] *European Computer Manufacturer Association International*: ECMA-218 PROCTOCOL FOR COMPUTER SUPPORTED TELECOMMUNICATIONS APPLICATIONS (CSTA) PHASE II. Dezember 1994

[GAMMA96] *Gamma E., Helm R., Johnson R., Vlissides J.*: ENTWURFSMUSTER – ELEMENTE WIEDERVERWENDBARER OBJEKTORIENTIERTER SOFTWARE. deutsche Ausgabe, Addison-Wesley 1996

[GRIG00] *Grigonis R.*: COMPUTER TELEPHONY ENCYCLOPEDIA. CMP Books 2000

[HOLZ04] *Holzner S.*: ECLIPSE COOKBOOK. O'Reilly 2004

[J2SE04a@] *Sun Microsystems*: JAVA 2 PLATFORM STANDARD EDITION 5.0 API SPECIFICATION. http://java.sun.com/j2se/1.5.0/docs/api/, 2004

[J2SE04b@] *Sun Microsystems*: J2SE PLATFORM AT A GLANCE. http://java.sun.com/j2se/1.5.0/docs/, 2004

[J2SE05@] *Sun Microsystems*: J2SE 5.0 PERFORMANCE WHITE PAPER. http://java.sun.com/performance/reference/whitepapers/5.0_performance.html, 2005

[JDBC05@] *Sun Microsystems*: INDUSTRY SUPPORT OF JDBC. http://java.sun.com/products/jdbc/reference/industrysupport/index.html, April 2005

[JNDI02@] *Sun Microsystems*: THE JNDI TUTORIAL. http://java.sun.com/products/jndi/tutorial/index.html, November 2002

[JNDI05@] *Sun Microsystems*: INDUSTRY SUPPORT OF JNDI. http://java.sun.com/products/jndi/reference/industrysupport/index.html, April 2005

[JTAP02@] *Sun Microsystems*: JAVA TELEPHONY API (JTAPI) SPECIFICATION VERSION 1.4 (FINAL RELEASE). http://jcp.org/aboutJava/communityprocess/final/jsr043/index.html, 2002

[NEWT04] *Newton H.*: NEWTON'S TELECOM DICTIONARY. 20. Auflage, CMP Books 2004

[NOTE96] *Northern Telecom*: MERIDIAN LINK / CUSTOMER CONTROLLED ROUTING – INSTALLATION AND UPGRADE GUIDE. Version 1.0, 1996

[OEST01] *Oestereich B., Hruschka P., Josuttis N., Kocher H., Krasemann H., Reinhold M.*: ERFOLGREICH MIT OBJEKTORIENTIERUNG – VORGEHENSMODELLE UND MANAGEMENTPRAKTIKEN FÜR DIE OBJEKTORIENTIERTE SOFTWAREENTWICKLUNG. 2. Auflage, Oldenbourg 2001

[OEST04] *Oestereich B.*: OBJEKTORIENTIERTE SOFTWAREENTWICKLUNG – ANALYSE UND DESIGN MIT DER UML 2.0. 6. Auflage, Oldenbourg 2004

[SCHU03] *Schäpers A., Huttary R.*: DANIEL DÜSENTRIEB – C#, JAVA, C++ UND DELPHI IM EFFIZIENZTEST, TEIL 2. c't-Magazin, Heft 21, Seite 222

[THOM02] *Thomas T.*: JAVA DATA ACCESS – JDBC, JNDI, AND JAXP. M&T Books 2002

Abbildungsverzeichnis

Tabellenverzeichnis

Programmverzeichnis

A Anhang

A.1 Zustandsübergänge von CallcontrolConnection und CallcontrolTerminalConnection

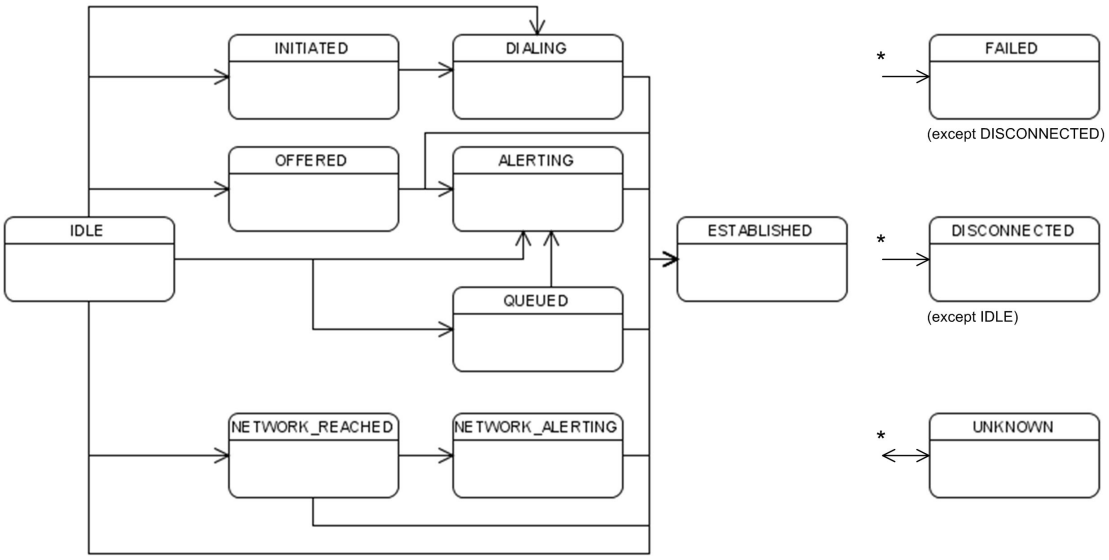

Abb. A-1: CallcontrolConnection-Zustandsdiagramm (nach [JTAP02@])

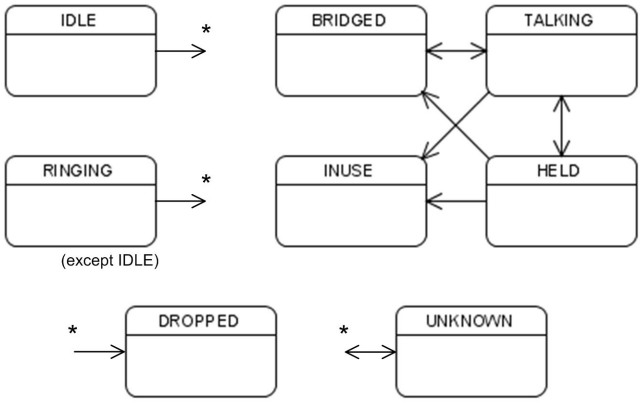

Abb. A-2: CallcontrolTerminalConnection-Zustandsdiagramm (nach [JTAP02@])

A.2 Vor- und Nachbedingungen implementierter JTAPI-Methoden mit Telefoniefunktion

Der folgende Pseudo-Code ist aus der JTAPI-Spezifikation [JTAP02@] entnommen und beschreibt die Vor- und Nachbedingungen für die Ausführung der Methoden. Existieren für eine Methode mehrere Überladungen, sind in der Überschrift die Übergabeparameter zur eindeutigen Identifizierung aufgeführt.

Die Bedingungen wurden seit der Umstellung von Observer auf Listener nicht mehr aktualisiert, sodass die veralteten Event-Objekte erwähnt werden. Dennoch lässt sich der Zusammenhang gut erkennen, da an deren Stelle pro Objekt des Call-Modells nur ein einzelnes Event-Objekt tritt.

A.2.1 Provider.createCall()

Vorbedingungen

```
(1) this.getState() == Provider.IN_SERVICE
```

Nachbedingungen

```
(1) this.getState() == Provider.IN_SERVICE
(2) Let Call call = this.createCall()
(3) call.getState() == Call.IDLE
(4) call.getConnections() == null
(5) call element of this.getCalls()
```

A.2.2 Call.connect()

Vorbedingungen

```
(1) (this.getProvider()).getState() == Provider.IN_SERVICE
(2) this.getState() == Call.IDLE
```

Nachbedingungen

```
(1) (this.getProvider()).getState() == Provider.IN_SERVICE
(2) this.getState() == Call.ACTIVE
(3) Let Connection c[] = this.getConnections()
(4) c.length == 2
(5) c[0].getState() == Connection.IDLE (at least)
(6) c[1].getState() == Connection.IDLE (at least)
```

A.2.3 CallControlCall.conference()

Vorbedingungen

```
(1) Let tc1 be the conference controller on this Call.
(2) Let connection1 = tc1.getConnection()
(3) Let tc2 be the conference controller on otherCall.
(4) (this.getProvider()).getState() == Provider.IN_SERVICE
(5) this.getState() == Call.ACTIVE
(6) tc1.getTerminal() == tc2.getTerminal()
(7) tc1.getCallControlState() ==
       CallControlTerminalConnection.TALKING
(8) tc2.getCallControlState() ==
       CallControlTerminalConnection.HELD
(9) this != otherCall
```

Nachbedingungen

```
(1)  (this.getProvider()).getState() == Provider.IN_SERVICE
(2)  this.getState() == Call.ACTIVE
(3)  otherCall.getState() == INVALID
(4)  Let c[] be the Connections to be merged from otherCall.
(5)  Let tc[] be the TerminalConnections to be merged from
     otherCall.
(6)  Let new(c) be the set of new Connections created on this
     Call.
(7)  Let new(tc) be the set of new TerminalConnections
     created on this Call.
(8)  new(c) element of this.getConnections()
(9)  new(c).getCallState() == c.getCallState()
(10) new(tc) element of
     (this.getConnections()).getTerminalConnections()
(11) new(tc).getCallState() == tc.getCallState()
(12) c[i].getCallControlState() ==
       CallControlConnection.DISCONNECTED for all i
(13) tc[i].getCallControlState() ==
       CallControlTerminalConnection.DROPPED for all i
(14) CallInvalidEv is delivered for otherCall.
(15) CallCtlConnDisconnectedEv/ConnDisconnectedEv is
     delivered for all c[i].
(16) CallCtlTermConnDroppedEv/TermConnDroppedEv is delivered
     for all tc[i].
(17) ConnCreatedEv is delivered for all new(c).
(18) TermConnCreatedEv is delivered for all new(tc).
(19) Appropriate events are delivered for all new(c) and
     new(tc).
```

A.2.4 CallControlCall.consult(TerminalConnection tc, String dialedDigits)

Vorbedingungen

```
(1) this.getProvider().getState() == Provider.IN_SERVICE
(2) this.getState() == Call.IDLE
(3) tc.getCallControlState() ==
        CallControlTerminalConnection.TALKING
(4) (tc.getCall()).getState() == Call.ACTIVE
```

Nachbedingungen

```
(1)  Let connections[] be the two Connections created and
     returned.
(2)  this.getProvider().getState() == Provider.IN_SERVICE
(3)  this.getState() == Call.ACTIVE
(4)  tc.getCallControlState() ==
        CallControlTerminalConnection.HELD
(5)  (tc.getCall()).getState() == Call.ACTIVE
(6)  Let c[] = this.getConnections() such that c.length == 2
(7)  c[0].getCallControlState() at least
     CallControlConnection.IDLE
(8)  c[1].getCallControlState() at least
     CallControlConnection.IDLE
(9)  c == connections
(10) CallActiveEv is delivered for this Call.
(11) ConnCreatedEv are delivered for both connections[i].
(12) CallCtlTermConnHeldEv is delivered for tc.
```

A.2.5 CallControlCall.setConferenceEnable()

Vorbedingungen

```
(1) (this.getProvider()).getState() == Provider.IN_SERVICE
(2) this.getState() == Call.IDLE
```

Nachbedingungen

```
(1) (this.getProvider()).getState() == Provider.IN_SERVICE
(2) this.getState() == Call.IDLE
(3) enable = this.getConferenceEnable()
```

A.2.6 CallControlCall.setTransferEnable()

Vorbedingungen

```
(1) (this.getProvider()).getState() == Provider.IN_SERVICE
(2) this.getState() == Call.IDLE
```

Nachbedingungen

```
(1) (this.getProvider()).getState() == Provider.IN_SERVICE
(2) this.getState() == Call.IDLE
(3) enable = this.getConferenceEnable()
```

A.2.7 CallControlCall.transfer(Call otherCall)

Vorbedingungen

```
(1) Let tc1 be the transfer controller on this Call.
(2) Let tc2 be the transfer controller on otherCall.
(3) this != otherCall
(4) (this.getProvider()).getState() == Provider.IN_SERVICE
(5) this.getState() == Call.ACTIVE
(6) otherCall.getState() == Call.ACTIVE
(7) tc1.getCallControlState() ==
        CallControlTerminalConnection.TALKING or
        CallControlTerminalConnection.HELD
(8) tc2.getCallControlState() ==
        CallControlTerminalConnection.TALKING or
        CallControlTerminalConnection.HELD
```

Nachbedingungen

```
(1)  (this.getProvider()).getState() == Provider.IN_SERVICE
(2)  this.getState() == Call.ACTIVE
(3)  otherCall.getState() == Call.INVALID
(4)  tc1.getCallControlState() ==
        CallControlTerminalConnection.DROPPED
(5)  tc2.getCallControlState() ==
        CallControlTerminalConnection.DROPPED
(6)  Let c[] be the Connections to be transferred from
     otherCall.
(7)  Let tc[] be the TerminalConnections to be transferred
     from otherCall.
(8)  Let new(c) be the set of new Connections created on this
     Call.
(9)  Let new(tc) be the set of new TerminalConnections
     created on this Call.
```

```
(10) new(c) element of this.getConnections()
(11) new(c).getCallState() == c.getCallState()
(12) new(tc) element of
     (this.getConnections()).getTerminalConnections()
(13) new(tc).getCallState() == tc.getCallState()
(14) c[i].getCallControlState() ==
        CallControlConnection.DISCONNECTED for all i
(15) tc[i].getCallControlState() ==
        CallControlTerminalConnection.DROPPED for all i
(16) CallInvalidEv is delivered for otherCall.
(17) CallCtlTermConnDroppedEv/TermConnDroppedEv is delivered
     for tc1, tc2.
(18) CallCtlConnDisconnectedEv/ConnDisconnectedEv is
     delivered for all c[i].
(19) CallCtlTermConnDroppedEv/TermConnDroppedEv is delivered
     for all tc[i].
(20) ConnCreatedEv is delivered for all new(c).
(21) TermConnCreatedEv is delivered for all new(tc).
(22) Appropriate events are delivered for all new(c) and
     new(tc).
```

A.2.8 Connection.disconnect()

Vorbedingungen

```
(1) ((this.getCall()).getProvider()).getState() ==
        Provider.IN_SERVICE
(2) this.getState() == Connection.CONNECTED or
        Connection.ALERTING or
        Connection.INPROGRESS or
        Connection.FAILED
(3) Let TerminalConnection tc[] =
        this.getTerminalConnections (see post-conditions)
```

Nachbedingungen

```
(1) ((this.getCall()).getProvider()).getState() ==
        Provider.IN_SERVICE
(2) this.getState() == Connection.DISCONNECTED
(3) For all i, tc[i].getState() == TerminalConnection.DROPPED
(4) this.getTerminalConnections() == null
(5) this is not an element of
        (this.getCall()).getConnections()
(6) ConnectionDisconnected is delivered for this Connection.
(7) TerminalConnectionDropped is delivered for all
```

```
    TerminalConnections associated with this Connection.
(8) ConnectionDisconnected/TerminalConnectionDropped are
    delivered for all other Connections and TerminalConnections
    dropped indirectly as a result of this method.
(9) CallInvalid if all of the Connections are dropped
    indirectly as a result of this method.
```

A.2.9 CallControlTerminalConnection.unhold()

Vorbedingungen

```
(1) ((this.getTerminal()).getProvider()).getState() ==
        Provider.IN_SERVICE
(2) this.getCallControlState() ==
        CallControlTerminalConnection.HELD
```

Nachbedingungen

```
(1) ((this.getTerminal()).getProvider()).getState() ==
        Provider.IN_SERVICE
(2) this.getCallControlState() ==
        CallControlTerminalConnection.TALKING
(3) CallCtlTermConnTalkingEv is delivered for this
    TerminalConnection.
```

Ehrenwörtliche Erklärung

Hiermit erkläre ich, Jan Hasert, geboren am 18.12.1978 in Böblingen,

(1) dass ich meine Diplomarbeit mit dem Titel:

„Entwicklung einer Anwendung zur computergestützten Telefonie (CTI) auf Basis des Java Telephony API (JTAPI)“

bei der Canoris GmbH & Co. KG IT-Dienstleistungen unter Anleitung von Professor Dr. Leibscher selbständig und ohne fremde Hilfe angefertigt habe und keine anderen als die angeführten Hilfen benutzt habe;

(2) dass ich die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 31. Mai 2005

Jan Hasert