Diplomarbeit zur Erlangung des Grades *Diplom-Ingenieur (FH)*

Fachhochschule Bielefeld
Erstprüfer: Prof. Dr. rer. nat. Christian Schröder
Zweitprüfer: Prof. Dr. math. Wolfgang Bunse

Integration des Troubleticketsystems OTRS bei einem mittelständischen Unternehmen

Felix J. Ogris (fjo@ogris.de)
Matr.-Nr.: 203583

Datum der Abgabe: 15. Januar 2008

Erklärung entsprechend der ADPO vom 25.06.1982 § 26 Abs. 1

Ich versichere, dass ich die Diplomarbeit selbstständig verfasst und keine, als die von mir angegebenen Hilfsmittel benutzt und bei Zitaten die Quellen kenntlich gemacht habe.

Mir ist bekannt, dass ich meine Diplomarbeit nach Ablauf der Aufbewahrungsfrist von 5 Jahren zurückbekommen kann.

Herford, 15.01.2008

Unterschrift

In dieser Diplomarbeit wird ein System entwickelt, welches mehrere Instanzen des Open Ticket Request Systems OTRS auf einer Plattform bereitstellt. Die Instanzen sind bis auf Betriebssystemebene gegeneinander isoliert, so dass das System mehrere OTRS-Instanzen für Kunden bereitstellen kann. Mit Hilfe einer eigens entwickelten Erweiterung können die Instanzen jedoch lose per Email gekoppelt werden. Zudem wird jede Instanz mit einer SOAP-Schnittstelle ergänzt, mit der Projekte und Kundendaten aus anderen Systemen automatisiert angelegt werden können. Für die Selbstverwaltung durch einen Kunden wurden ausserdem zwei Erweiterungen erstellt, die es ermöglichen, eine Instanz unter verschiedenen Webadressen erreichbar zu machen und die Benutzerschnittstelle zu verändern. Vor der Darstellung der entwickelten Erweiterungen werden die vielfältigen theoretischen Grundlagen diskutiert.

This diploma thesis presents a system which makes it possible to setup multiple instances of the Open Ticket Request System OTRS on one platform. As these instances are isolated against each other at the operating system level, one hardware platform can be used for many customers. By using a self-developed extension those instances can loosely be coupled by email. Each instance can be equipped with a self-developed SOAP interface which allows the user to import customer data and projects from other systems. Two additional extensions allow the customers to modify the user interface to fit their needs and to make each instance reachable under different hostnames. Prior to the presentation of each self-developed extension a discussion of the theoretical background is given.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ACL	Access Control List
AD	Active Directory
AES	Advanced Encryption Standard
AMI	Asterisk Manager Interface
AT&T	American Telephone & Telegraph Corporation
BNF	Backus-Naur-Form
BSD	Berkeley Software Distribution
CA	Certificate Authority
CAPI	Common Application Programming Interface
CGI	Common Gateway Interface
CPU	Central Processing Unit
DAV	Distributed Authoring and Versioning
DES	Data Encryption Standard
DMZ	Demilitarisierte Zone
DSA	Digital Signature Algorithm
DTD	Document Type Definition
EBNF	Erweiterte Backus-Naur-Form
FTP	File Transfer Protocol
GB	Gigabyte; $1GB = 10^9 Byte$
GiB	Gibibyte; $1GiB = 2^{30} Byte$
GPL	General Public License
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ISDN	Integrated Services Digital Network
ISP	Internet Service Provider
LDAP	Lightweight Directory Access Protocol
LIFO	Last In, First Out
MiB	Mebibyte; $1MiB = 2^{20} Byte$
MTA	Mail Transfer Agent
NFS	Network File System
NSA	National Security Agency
OTRS	Open Ticket Request System
Perl	Practical Extraction and Report Language
PHP	PHP Hypertext Preprocessor
PI	Processing Instruction
POD	Plain Old Documentation
RADIUS	Remote Authentication Dial-In User Service
RAID	Redundant Array of Independent Disks
regex	regular expression
RPC	Remote Procedure Call
RSA	Rivest-Shamir-Adleman
SCSI	Small Computer System Interface
SHA	Secure Hash Algorithmus
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSH	Secure Shell

SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
UC Berkeley	University of California, Berkeley
URI	Uniform Resource Identifier
URL	Unified Resource Locator
W3C	World Wide Web Consortium
WAL	Write Ahead Logging
WS-I	Web Services Interoperability Organization
WSDL	Web Services Description Language
WWW	World Wide Web
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language
XSL-FO	Extensible Stylesheet Language Formatting Objects
XSLT	Extensible Stylesheet Language Transformations

Abbildungsverzeichnis

Tabellenverzeichnis

1. Prolog

Die vorliegende Arbeit beschreibt die technische Integration des *Open Ticket Request System* (OTRS) bei der DTS Firmengruppe in Herford. Im Laufe der Arbeit wurden die Abteilungen *Security*, *Data Center* und *Internet Service*, welche die betrieblichen Resourcen für diese Arbeit bereitstellte, zum 1. Januar 2008 in die DTS Systeme GmbH integriert. Der Bedarf nach einem funktional gekoppelten, aber dennoch in eigenständige Bereiche unterteiltem Ticketsystemen verlor hierdurch an Bedeutung. Vielmehr musste ein System geschaffen werden, welches auch Kunden bereitgestellt werden kann. Die ursprüngliche Anforderung, das OTRS an das vorhandene Projektsystem *Work@Web* anzubinden, blieb erhalten. Hier musste eine HTTP-GET-Schnittstelle geschaffen werden und für zukünftige Erweiterungen eine SOAP-Schnittstelle implementiert werden. Letzteres nimmt zusammen mit den notwendigen theoretischen Erläuterungen in Kapitel 6 einen grossen Teil dieser Arbeit ein. Eine besondere Herausforderung bestand darin, schon während der Einarbeitungs- und Implementierungsphasen in den Dialog mit Kunden zu treten, die inhaltliche und terminliche Zusagen erwarteten. Als zeitintensive Fehlentscheidung hat sich das Festhalten am ursprünglichen Design erwiesen, welches ein mandantenfähiges OTRS vorsah. Die Idee war, die Sicht auf die Datenbank anhand des Hostnamens zu beschränken, über den der Benutzer auf das OTRS zugegriffen hat. Jedoch besitzt jede OTRS-Instanz einen eindeutigen Hostnamen. Dieser wird an vielen Stellen wie z.B. in Emailtexten verwendet. Im webbasierten Teil von OTRS ist dies durch eine neue Datenbankschicht lösbar. Automatisierte Wartungsarbeiten z.B. durch periodisch ausgeführte Skripte greifen jedoch auf den fest konfigurierten Hostnamen zurück. Diese hätten neu gestaltet werden müssen, was ein Update des Systems unnötig erschwert. Zudem ist das Berechtigungssystem von OTRS nicht für eine derartige mandatenbasierte Lösung ausgelegt. Empirisch ist ausserdem belegt, dass selbst technisch nicht versierte Kunden ein System selbsttätig administrieren möchten. Dies ist nur mit getrennten OTRS-Instanzen möglich, wie sie letztendlich realisiert wurden. Die bis dato entwickelte Mandantenlösung wurde in ein Modul übernommen, mit dem jeder Administrator seine Instanz unter verschiedenen Hostnamen erreichbar machen kann.

Als weitere Herausforderung stellten sich einige Fehler in den verwendeten Softwarepaketen heraus. So hat der Autor der vorliegenden Arbeit drei Fehler in der OTRS Version 2.2.3 aufgezeigt[1][2][3] und ein Fehlverhalten der Funktionsbibliothek *SOAP::Lite* aufgezeigt[4]. Da die verwendeten Softwarepakete als *Open Source* verfügbar sind und die jeweiligen Projekte über Mailinglisten verfügen, wurden diese Fehler entsprechend veröffentlicht. Positiv fiel hierbei die Reaktionszeit der Entwickler auf. Eine Antwort bzw. ein Lösungsvorschlag war i.d.R. innerhalb von einigen Stunden oder wenigen Tagen verfügbar.

Diese Arbeit ist inhaltlich logisch strukturiert. Nach einer kurzen Erläuterung der zur Verfügung stehenden Hardware wird das verwendete Betriebssystem *FreeBSD* vorgestellt. Darauf aufbauend werden die eingesetzten Programme bzw. Dienste erörtert. Da ein grosser Teil dieser Arbeit auf der Programmierung eigener Module für das OTRS beruht, folgt eine Darstellung der Programmiersprache *Perl*. Ausserdem wird das unter unixartigen Betriebssystemen typische *Shellscripting* erklärt. Den Abschluss dieser Grundlagendiskussion bilden zwei Kapitel über die Auszeichnungssprache *XML* und des-

[1] http://lists.otrs.org/pipermail/dev/2007-September/001709.html
[2] http://lists.otrs.org/pipermail/dev/2007-September/001712.html
[3] http://lists.otrs.org/pipermail/dev/2007-September/001714.html
[4] http://sourceforge.net/mailarchive/message.php?msg_name=C372633E.B9C9B%25fjo-lists%40ogris.de

sen Ableitungen sowie die Datenbankabfragesprache *SQL*. Die Darstellung des OTRS in Kapitel 8.1 erfolgt mehr aus der Sicht eines Administrators oder Programmierers denn aus der eines Anwenders. Abschliessend werden die entwickelten Module präsentiert und deren Programmierung in Auszügen erklärt.

2. Hardware

2.1. Entwicklungssystem

Als Entwicklungssystem stand ein älterer Server vom Typ Compaq Proliant DL360 (s. Abbildung 1) zur Verfügung. Er besitzt 2 Intel Pentium 3 CPUs mit je 866 MHz sowie

Abbildung 1: Compaq Proliant DL360 (Quelle: Hewlett Packard)

1 GB Arbeitsspeicher. Zwei SCSI-Festplatten mit einer Kapazität von je 18,2 GB bilden mittels eines RAID-Controllers einen Festplattenverbund im Level RAID-1, was den Ausfall einer Festplatte ohne betriebliche Einbußen oder Datenverlust verkraftet. Der Server wurde per Firewall geschützt im internen Techniknetz der DTS Service GmbH platziert. Als Hostname wurde `fjo-otrs.dts-online.net` vergeben.

2.2. Produktivsystem

Für die produktive Installation für OTRS wurde von der DTS Systeme GmbH ein Server vom Typ Hewlett Packard Proliant DL360 G4 (s. Abbildung 2) bereitgestellt. Er verfügt

Abbildung 2: HP Proliant DL360 G4 (Quelle: Hewlett Packard)

über 2 Intel Xeon Prozessoren mit je 3,6 GHz Taktfrequenz. Dies sind CPUs mit je 2 logischen Kernen pro physikalischem Prozessor (*dual core*). Als Hauptspeicher stehen 4 GB zur Verfügung. Zwei 146 GB fassende SCSI-Festplatten sind wie beim Entwicklungssystem per RAID-1 zu einem logischen Datenträger mit einer Nettokapazität von 146 GB verbunden. Ferner stehen zwei Ethernetschnittstellen mit einer Geschwindigkeit von 1 GBit/s bereit. Der Server wurde in einer demilitarisierten Zone (*DMZ*) platziert, so dass zwar Schutz durch eine Firewall gegeben ist, jedoch der Zugriff aus dem Internet per

- Email (SMTP per *Transmission Control Protocol* (TCP), Port 25)
- Web (HTTP per TCP, Port 80)

- SSL geschütztem Web (HTTPS per TCP, Port 443)

möglich ist.

3. Betriebssystem

3.1. Einführung in FreeBSD

Das eingesetzte Betriebssystem *FreeBSD* in der Version 6.2 kann als Ur-Ur-Urenkel (vgl. Éric Lévénez (2007)) des 1969 von Ken Thompson und Dennis Ritchie entwickelten Unix angesehen werden. Der Name *FreeBSD* setzt sich aus den Teilen *free* im Sinne von *frei von lizenzrechtlich geschütztem Code* und *Berkeley Software Distribution* (BSD) zusammen. Die damals zur US-amerikanischen *American Telephone & Telegraph Corporation* (AT&T) gehörenden *Bell Laboratories*, unter dessen Obhut Thompson und Ritchie ihrerzeit Unix entwickelten, gab zwar den Sourcecode des Betriebssystems unentgeltlich an Forschungseinrichtungen wie die *University of California, Berkeley* (UC Berkeley) weiter, die entscheidende Entwicklungen wie den TCP/IP-Stack oder die Trennung des Quellcodes in CPU-spezifische und generische Teile vornahmen, diese verbesserten Versionen von Unix aber nur weitergeben durften, wenn der jeweilige Interessent oder Hersteller eine Lizenz von AT&T erwarb. Die Rechtsstreitigkeiten zwischen AT&T und der UC Berkeley ergaben, dass aus dem Quellcode von BSD 3 von ca. 18000 Dateien entfernt werden mussten und dass AT&T in die von BSD übernommenen Codefragmente "vergessene"Copyright-Hinweise wieder hinzufügen musste. Aus der bereinigten Interimsversion *4.3BSD Lite* ging im Dezember 1993 FreeBSD 1.0 hervor, welches seitdem vom *FreeBSD Project*, einer Gruppe von freiwilligen Entwicklern, gepflegt und als Quellcode frei zur Verfügung gestellt wird. FreeBSD läuft vornehmlich auf der x86-Architektur vom Intel 386 Prozessor bis zu aktuellen Pentium- und AMD Athlon-CPUs und auf deren 64bittigen Nachfolgern wie neusten Intel Xeon- und AMD Opteron-CPUs, die ob ihrer ähnlichen Befehlssätze unter FreeBSD als amd64-Architektur zusammengefasst sind. Ferner existieren Portierungen auf die UltraSPARC- und ARM-Architektur. Bis auf den Betriebssystemkern, den *Kernel*, und systemnahe Programme existieren keine Unterschiede zwischen den einzelnen Versionen, so dass sich ein FreeBSD 6.2 auf einem älteren Intel Pentium 3 genauso administrieren lässt wie auf einem AMD Opteron. FreeBSD zeichnet sich vor allem durch die folgenden Eigenschaften aus:

Stabilität Laufzeiten (*uptime*) von mehreren hundert Tagen sind selbst bei belasteten Systemen keine Seltenheit. Oftmals werden Systeme nur aufgrund von Hardwarefehlern, nach dem Einspielen sicherheitskritscher Updates oder physikalischer Relokation neu gestartet oder heruntergefahren.

Kontinuität Ein FreeBSD-System verhält sich über den Laufzeitraum wie vom Administrator vorgegeben. Ein dynamisches Verhalten wie z.B. selbständiges Anpassen von Konfigurationsparametern findet nicht statt.

Transparenz FreeBSD liegt vollständig als Quelltext vor und verfügt über eine umfangreiche und gute Dokumentation (u.a. die sog. *manpages*), die vom Hilfsprogramm bis zu internen Funktionen des Kernels reicht

Aktualität FreeBSD kann entweder per vorkompilierter Programmpakete oder über das *Portssystem* aktuell gehalten werden. Das Portssystem oder kurz die *Ports* stellen eine Metadatenbank dar, die Regeln zum Herunterladen, Übersetzen und Installieren von i.d.R. quelloffenen Programmen beinhalten. Obwohl dies grösstmögliche Aktualität der jeweiligen Programme bedeutet, können Programme auch als fertige Pakete eingespielt werden. Beide Arten (Ports und Pakete) können gleichzeitig auf einem Rechner verwendet werden. Zudem können aus den über das Portssystem installierten Programmen eigene Pakete erstellt werden, um sie z.B. auf weiteren lokalen Rechnern einzuspielen.

BSD-Lizenz FreeBSD unterliegt dem allgemeinhin als *BSD-Lizenz* bekannten Copyright. Änderungen am Quellcode oder an der Zusammensetzung der Programme müssen nicht veröffentlicht werden, solange der Copyright-Hinweis[5] übernommen wird. Dies steht im Gegensatz zur *General Public License* (GPL), der z.B. Linux und viele andere Opensource-Programme unterliegen. Ausgenommen von der BSD-Lizenz sind jedoch Programme, die z.B. über das Portssystem eingespielt wurden, aber einer anderen Lizenz unterliegen.

Flexibilität Eine Minimalinstallation von FreeBSD benötigt ca. 140 MiB (Mebibyte; $1MiB = 2^{20}Byte$). Festplattenkapazität und 32 MiB Arbeitsspeicher und beinhaltet diverse kommandozeilenbasierte Programme wie

- einen C-Compiler samt Linker und Assembler
- die *Secure Shell* (SSH), eine sichere Methode zur entfernten Anmeldung
- Client- und Serverprogramme für das *File Transfer Protocol* (FTP)
- Sendmail zum Versenden und Empfangen von Emails über das *Simple Mail Transfer Protocol* (SMTP)
- diverse Hilfsprogramme zum Einrichten von Netzwerkschnittstellen, Formatieren von Festplatten, usw.

Wegen der aufgezählten Gründe verwenden *Internet Service Provider* (ISP) neben Linux bevorzugt FreeBSD.

3.2. Installation

FreeBSD wird i.d.R. per CD-ROM installiert. Hierfür nötige CD-Abbilder, sprich *ISO-Files*, sind auf dem FTP-Server des FreeBSD-Projektes[6] oder einem Spiegel[7] zum Download verfügbar. Alternativ kann die Installation komplett über das Netzwerk erfolgen. Hierzu ist jedoch hardwareseitige Unterstützung der Netzwerkkarte sowie ein entsprechender Installationsserver notwendig. Daher wird meist per CD-ROM gebootet, und dann entweder von jener CD das System aufgespielt oder aus dem Installationsmenü (s. Abbildung 3) eine Netzwerkinstallation über FTP, *Network File System* (NFS), o.ä. ausgewählt. Die eigentliche Installation ähnelt sehr der von anderen Betriebssystemen. Unterschiede betreffen lediglich die Art der Partitionierung und die Möglichkeit, vorab eine Paketauswahl zu treffen, um so z.B. keine grafische Oberfläche und keine Spiele zu installieren. FreeBSD verwendet eine herkömmliche Festplattenpartition, die *Slice* genannt wird, und richtet erst innerhalb dieses Slices verschiedene Partitionen für z.B. das Betriebssystem, die Nutzdaten und den Swapbereich ein. Andere Betriebssysteme sehen nur den Slice, nicht aber die in ihm enthaltenen Partitionen. Auf der Entwicklungsmaschine wurde wegen des beschränkten Festplattenplatzes von 18,2 GB (Gigabyte; $1GB = 10^{9}Byte$) eine System- und Datenpartition von 15 GiB (Gibibyte; $1GiB = 2^{30}Byte$) und eine Swappartiton mit 2 GiB eingerichtet. Die Produktivmaschine verfügt über 146 GB Nettokapazität. Daher wurde eine sogenannte *Root-Partition* für das Betriebssystem mit einer Grösse von 16 GiB eingerichtet. Die Swappartition wurde mit 4 GiB so gross wie der verfügbare Hauptspeicher gewählt. Der verbleibende Festplattenplatz von 113 GiB wurde einer eigenen Partition für Nutzdaten zugewiesen und unterhalb des Verzeichnisses `/var` eingebunden. Um die wiederkehrende Aufgabe

[5] http://www.freebsd.org/copyright/index.html
[6] ftp://ftp.freebsd.org/
[7] http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/mirrors-ftp.html

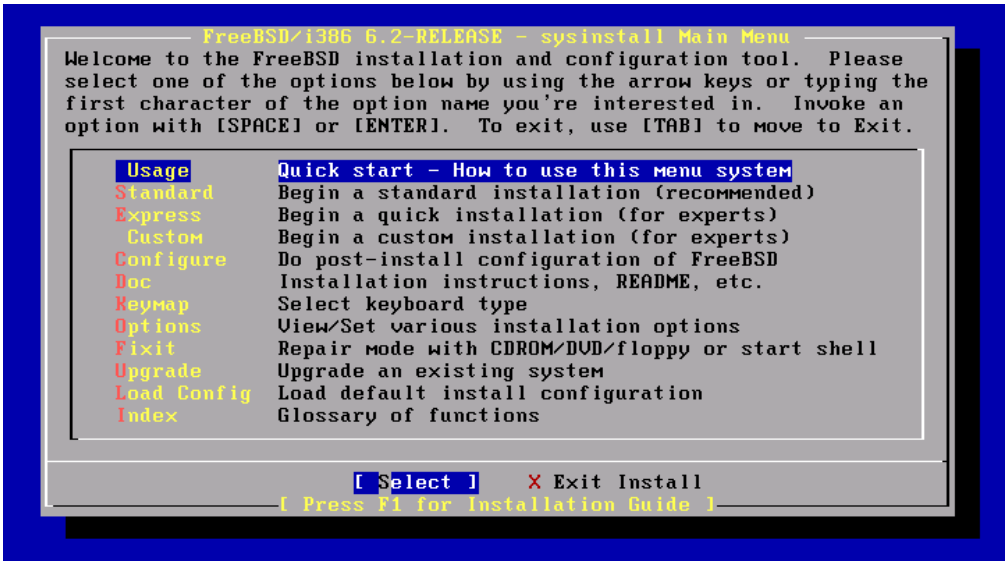

Abbildung 3: Das Installationsmenü von FreeBSD

der Installation von FreeBSD auf einem neuen Rechner zu vereinfachen, hat der Autor der vorliegenden Diplomarbeit im Januar 2006 ein nicht veröffentlichtes Skript- und Konfigurationsbundle namens *DTS-BSD* begonnen, welches nach der Installation neuer Systeme eine einheitliche Basiskonfiguration schreibt. Dies umfasst vor allem:

- Herabsetzen der Wartezeit des Bootloaders am Prompt zur Betriebssystemauswahl
- Setzen der lokalen Zeitzone auf `Europe/Berlin`
- Umleiten aller vom System generierten Statusemails an `hostmaster@dts-online.net`
- Anpassen der zentralen Konfigurationsdatei `/etc/rc.conf`, so dass
  - ein deutsches Tastaturlayout geladen wird
  - der SSH-Server zur sicheren Anmeldung auf diesem Server über das Netzwerk gestartet wird
  - der Systemlogdienst nur lokale Meldungen annimmt und nicht von anderen Rechnern im Netzwerk
  - der Zeitserver gestartet wird, falls dieser installiert wurde
  - der Webserver (*Apache*) gestartet wird, falls dieser installiert wurde
  - der Server für das *Simple Network Management Protocol* (SNMP) zur Überwachung des Rechners gestartet wird, falls dieser installiert wurde
  - der PortgreSQL-Datenbankserver gestartet wird, falls dieser installiert wurde
  - der MySQL-Datenbankserver gestartet wird, falls dieser installiert wurde
  - zur vollständigen Konfiguration nur noch Hostname, IP-Adresse und Defaultgateway eingetragen werden müssen
- Eintragen der Standarddomain `dts-online.net` und der beiden Nameserver in die Datei `/etc/resolv.conf`

- Heraufsetzen von Kernelparametern in der Datei `/etc/sysctl.conf` zur Steigerung des Netzwerk- und Festplattendurchsatzes
- Anpassen der Konfigurationsdatei `/etc/ssh/sshd_config`, um dem Administrator (*root*) die entfernte Anmeldung zu ermöglichen
- Eintragen der Zeitserver `ntp1.dts-online.net` und `ntp2.dts-online.net` in die Konfigurationsdatei `/usr/local/etc/ntpd.conf` des Zeitservers
- Eintragen des Servers `cvsup.dts-online.net` zur Aktualisierung des Systems in die Dateien `/usr/local/etc/cvsup-ports.conf` und `/usr/local/etc/cvsup-src.conf`
- Anpassen der Konfigurationsdatei `/usr/local/etc/snmp/snmpd.conf` des SNMP-Dienstes

Zudem aktualisiert ein Skript das Portssystem und installiert dann vorab ausgewählte Programme.

3.3. Starten von Systemdiensten

Dienste wie z.B. der Systemlogserver (`syslogd`) oder ein Webserver werden beim Systemstart von Shellskripten (s. Kapitel 5.2) aktiviert. Diese sogenannten *RC-Skripte* liegen im Verzeichnis `/etc/rc.d`, falls es sich um einen Dienst des Basissystems handelt, oder in `/usr/local/etc/rc.d`, falls es sich um einen über das Portssystem (oder per Programmpaket) installierten Server handelt. Da das Starten, Stoppen, usw. eines Dienstes oft gleichartige Schritte umfasst, wurden diese als separate Funktionen in der Datei `/etc/rc.subr` zusammengefasst, welche jedes RC-Skript einbindet bzw. einbinden sollte. Ob und mit welchen Parametern ein Dienst gestartet werden sollte, muss in der zentralen Konfigurationsdatei `/etc/rc.conf` hinterlegt werden. So muss dort z.B. für den Webserver *Apache* in der Version 2.2 das Flag `apache22_enable` auf `YES` gesetzt werden. Etwaige Parameter werden in der Variablen `apache22_flags` übergeben. Soll der Webserver nicht mit den Rechten des Administrators *root* gestartet werden, so muss in `apache22_user` der gewünschte Benutzername stehen. Das Namensprefix (hier: `apache22`) bestimmt jedes RC-Skript selbst, indem es eine Variable `name` entsprechend initialisiert.

3.4. Portssystem

Das Portssystem wird i.d.R im Verzeichnis `/usr/ports` installiert. Es handelt sich dabei vorwiegend um eine thematisch sortierte Sammlung von sogenannten *Makefiles*. Diese Makefiles sind Textdateien, die Vorschriften zum automatischen Herunterladen, Übersetzen und Installieren von Softwarepakten enthalten. Makefiles werden üblicherweise in der modularen Programmierung eingesetzt, um beim erneuten Übersetzen eines aus mehreren Objekten bestehenden Programmes nur diejenigen zu kompilieren, deren zugrunde liegender Quellcode geändert wurde. Formal beschreibt ein Makefile, welche Schritte notwendig sind, um aus einer Anzahl von Quelldateien eine oder mehrere Zieldateien zu erhalten. Im Portssystem sind über 17000 Programme verfügbar (Dezember 2007). Eine Suchfunktion hilft, die gewünschte Anwendung zu finden, z.B. den Webserver Apache (s. Kapitel 4.1) in der Version 2.2. Hierfür wechselt man zunächst in das Verzeichnis `/usr/ports` und setzt den Befehl `make search key=<Suchwort>` ab:

```
$ cd /usr/ports
$ make search key=apache22
```

Man erhält eine Ausgabe wie in Abbildung 4. Neben den Angaben, welche genaue Version

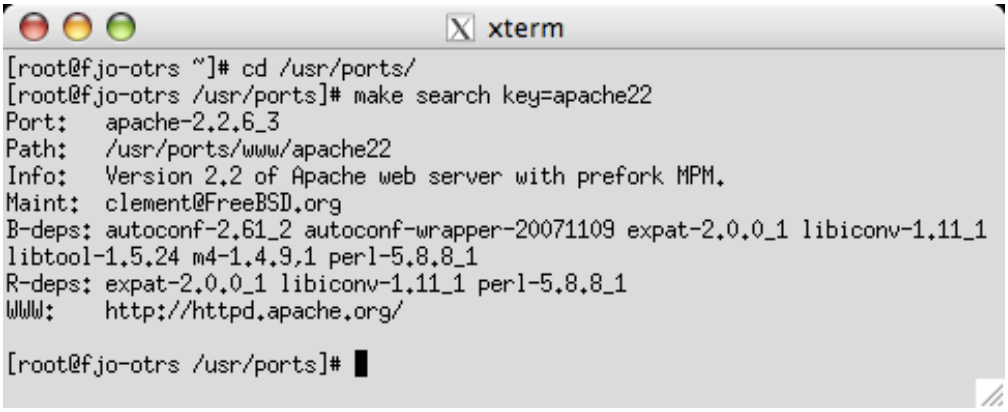

Abbildung 4: Die Suche nach einem Programm im Portssystem

der Software verfügbar ist und welche Pakete zum Übersetzen (`B-deps`) bzw. zum Betrieb (`R-deps`) notwendig sind, ist vor allem das lokale Verzeichnis (`Path`) wichtig, in dem sich die gewünschte Anwendung befindet (hier: `/usr/ports/www/apache22`). Man wechselt in jenes Verzeichnis (`cd /usr/port/www/apache22`) und setzt den Befehl `make install clean` ab, um das Softwarepaket zu installieren und um anschliessend alle während des Kompilierens erzeugten Dateien zu löschen. Das Portssystem wird alle benötigten Pakete (B-deps, R-deps) automatisch installieren, sofern sie noch nicht im System vorhanden sind. Wird ein Port zum allerersten Mal installiert und bietet er verschiedene Optionen zur Installation an wie z.B. Optimierungsflags oder zusätzliche Module, so wird dem Administrator ein entsprechendes Auswahlmenü präsentiert. Manuell lässt sich ein derartiges Optionsmenü per `make config` im Verzeichnis des entsprechenden Paketes aufrufen.

4. Anwendungsprogramme

4.1. Apache

4.1.1. Einleitung

Der Webserver *Apache* ist eine von der Apache Software Foundation[8] im Quellcode zur Verfügung gestellte Serversoftware. Üblicherweise wird er verwendet, um Dateien per *Hypertext Transfer Protocol* (HTTP) auszugeben, welches allgemeinhin die Grundlage des *World Wide Web* (WWW) bildet. Er ist modular aufgebaut. Zum Standardumfang gehören vor allem Module, die

- die sichere Variante *HTTPS* des Hypertext Transport Protocols implementieren
- die Generierung von Webseiten per externer Programme ermöglichen (das sogenannte *Common Gateway Interface* (CGI)
- den Betrieb des Servers als Proxy für HTTP und FTP erlauben.

4.1.2. Installation

Der Apache Webserver wurde über das Portssystem installiert (s. Abbildung 5). Hierbei

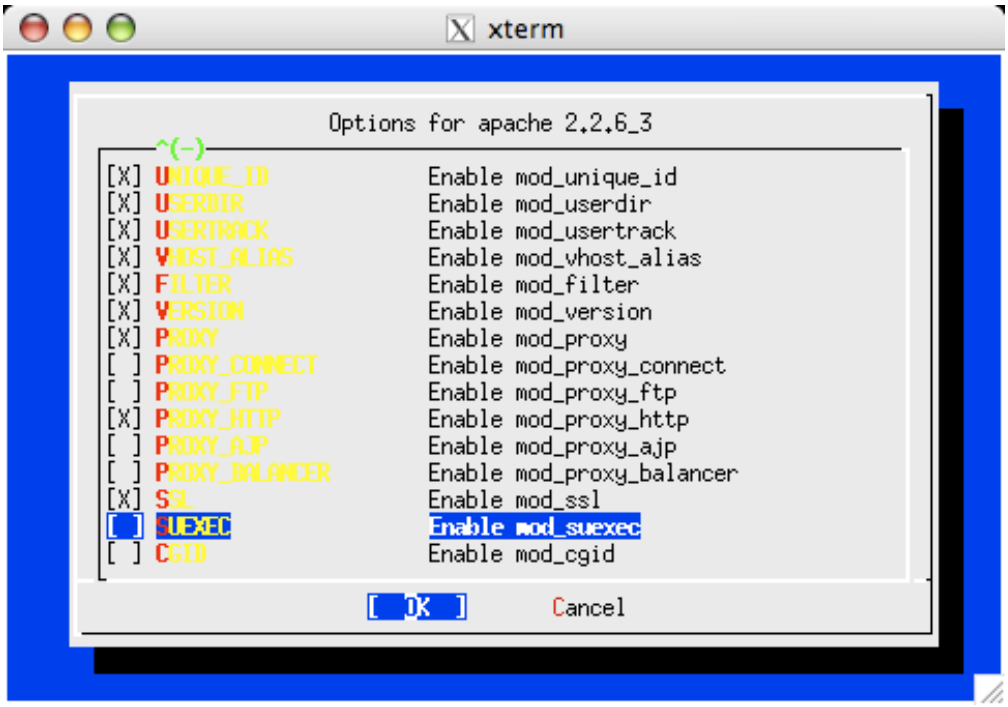

Abbildung 5: Das Optionsmenü zur Installation des Apache Webservers

wurden neben der Defaultauswahl die folgenden Module installiert:

mod_proxy Dieses Modul stellt allgemeine Proxyfunktionen bereit.

mod_proxy_http Dieses Modul stellt Proxyfunktionen für das Hypertext Transfer Protocol bereit.

[8] http://www.apache.org

mod_ssl Dieses Modul ermöglicht den Einsatz von HTTPS, der sicheren Variante des Hypertext Transfer Protocols.

Die Module `mod_dav` und `mod_dav_fs`, welche das HTTP-basierte Verfahren *Distributed Authoring and Versioning* (DAV) zum Bearbeiten von Dateien auf einem Webserver implementieren, wurden aus Sicherheits- und Performancegründen abgewählt.

4.1.3. Konfiguration

Damit der Apache beim Booten des Systems gestartet wird, muss in der Datei `/etc/rc.conf` folgender Eintrag gesetzt sein:

```
apache22_enable="YES"
```

FreeBSD verfügt über einen Filter, der neue Verbindungen nur dann an eine Anwendung signalisiert, wenn der Kernel eine gültige HTTP-Anfrage erkannt hat. Da dies die Anzahl der Kontextwechsel zwischen Kernel- und Userspace mindert, wurde die `/etc/rc.conf` um folgende Option erweitert:

```
apache22_http_accept_enable="YES"
```

Standardmässig stellt die Datei `/usr/local/etc/apache22/httpd.conf` die Konfiguration des Apache dar. Auf der Entwicklungsmaschine (s. Kapitel 2.1) wurde diese bis zum Einsatz des Moduls DTSMaster (s. Kapitel 9.6) verwendet. Auf der Produktivmaschine hingegen werden automatisch generierte Webserverkonfigurationen verwendet. Tabelle 1 erläutert die wichtigsten Parameter einer Konfigurationsdatei für den Apache.

Tabelle 1: Ausgewählte Konfigurationsparameter des Apache Webservers

Parameter	Beschreibung
`ServerRoot` *VERZEICHNIS*	legt das Stammverzeichnis des Servers fest; alle relativen Pfadangaben beziehen sich hierauf
`User` *NAME*	weist den Apache an, nach dem Start mit den Rechten des (unprivilegierten) Benutzers *NAME* zu arbeiten
`Group` *NAME*	weist den Apache an, nach dem Start mit den Rechten der (unprivilegierten) Gruppe *NAME* zu arbeiten
`LoadModule` *NAME MODUL*	lädt das Modul mit dem Namen *NAME* aus der Bibliothek *MODUL*
`AcceptMutex` *TYPE*	Apache kreiert beim Start mehrere Arbeitsprozesse, die beim Zustandekommen einer neuen Clientverbindung serialisiert werden müssen, damit nur ein Arbeitsprozess den neuen Client bedient; üblich ist der Typ `sysvsem`, der einen Semaphore als Mutex verwendet
`Listen` *ADRESSE*	gibt die IP-Adresse und optional die Portnummer an, auf der der Apache Verbindungen akzeptiert; als Wildcard für alle Adressen des Systems kann ein Stern `*` verwendet werden

Tabelle 1: Ausgewählte Konfigurationsparameter des Apache Webservers (Forts.)

Parameter	**Beschreibung**
`NameVirtualHost` *ADRESSE*	das Hypertext Transfer Protocol ermöglicht es, dass ein Server mehrere, anhand ihrer Hostnamen unterschiedene Webauftritte beherbergt; jener Parameter gibt an, auf welchen Netzwerkschnittstellen der Apache dieses Verhalten unterstützt
`ServerAdmin` *EMAIL*	gibt die Emailadresse des Administrators an, die z.B. auf Fehlerseiten angegeben wird
`PidFile` *DATEI*	gibt an, in welcher Datei die Prozessnummer des Servers gespeichert werden soll, damit z.B. die RC-Skripte des Betriebssystems den Apachen stoppen können
`ErrorLog` *DATEI*	spezifiert die Datei, in der Fehlermeldungen protokolliert werden
`CustomLog` *DATEI FORMAT*	gibt an, in welcher Datei und mit welchen Format Zugriffe protokolliert werden; hierfür muss das Modul `log_config_module` aus der Bibliothek `mod_log_config.so` geladen worden sein
`<VirtualHost` *ADRESSE*`>`	leitet einen Abschnitt für einen anhand des Hostnamen unterschiedenen Webauftritt ein; dieser muss mit `</VirtualHost>` wieder geschlossen werden
`ServerName` *HOSTNAME*	gibt den Namen eines VirtualHost an
`DocumentRoot` *VERZEICHNIS*	gibt das Wurzelverzeichnis für einen VirtualHost an; per default werden abgerufene Dateien aus diesem Verzeichnis ausgeliefert
`RedirectPermanent` *URL1 URL2*	Anfragen von Clients nach Dokumenten unter dem Pfad *URL1* werden zum Pfad *URL2* weitergeleitet; hierfür muss das Modul `alias_module` aus der Bibliothek `mod_alias.so` geladen worden sein
`<LocationMatch` *REGEX*`>`	leitet einen Abschnitt für Anfragen nach Dokumenten, auf deren Pfad der reguläre Ausdruck *REGEX* zutrifft, ein; innerhalb eines solchen Abschnittes können gesonderte Regeln definiert werden; ein solcher Abschnitt muss per `</LocationMatch>` wieder geschlossen werden

Tabelle 1: Ausgewählte Konfigurationsparameter des Apache Webservers (Forts.)

Parameter	**Beschreibung**
`ProxyPass` *URL HOSTNAME*	Anfragen nach Dokumenten unterhalb des Pfades *URL* werden an den Rechner mit dem Namen *HOSTNAME* durchgeleitet; hierfür müssen die Module `proxy_module` aus der Bibliothek `mod_proxy.so` und – falls der entfernte Rechner per HTTP angesprochen werden soll – `proxy_http_module` aus `mod_proxy_http.so` geladen worden sein
`SSLEngine On`	gibt an, dass Dokumente per HTTPS geschützt übertragen werden sollen; hierfür muss das Modul `ssl_module` aus der Bibliothek `mod_ssl` geladen worden sein
`SSLCertificateFile` *DATEI*	gibt den Pfad zum SSL-Zertifikat an
`SSLCertificateKeyFile` *DATEI*	gibt den Pfad zum SSL-Schlüssel an

4.1.4. mod_perl2

Aktuelle Webserver ermöglichen es, Webseiten und andere Inhalte dynamisch zu generieren. Hierzu wird ein externes Programm oder Skript aufgerufen, das z.B. Anfragen an eine Datenbank stellt und diese Daten dem Besucher bzw. Client darstellt. Ein solches Programm kann in nahezu jeder Programmiersprache erstellt werden. Es muss lediglich in der Lage sein,

- etwaige übergebene Parameter auf der Standardeingabe lesen zu können
- Hilfswerte wie z.B. die IP-Adresse des Clients als Umgebungsvariable (*environment*) einlesen zu können
- den dynamisch erzeugten Inhalt, z.B. eine Webseite, aber auch ein Bild, per Standardausgabe an den Webserver zurückliefern zu können.

Eine typische Anwendung ist z.B. ein Gästebuch, bei dem der Besucher zunächst in eine statische Webseite seinen Namen, seine Emailadresse und einen Kommentar eingibt. Per Mausklick auf einen meist mit *Absenden* oder *Eintragen* betitelten Knopf werden die Werte an eine dynamische Webseite gesendet, hinter der sich ein externes Programm verbirgt. Dieses nimmt die Daten des Besuchers entgegen und speichert sie i.d.R. zusammen mit Datum und der IP-Adresse des Clients in einer Datenbank ab. Dieses *Common Gateway Interface*, oder kurz *CGI* genannte Verfahren ist zwar flexibel, beim Einsatz einer interpretierten Sprache wie Perl jedoch muss der Webserver bei jedem Aufruf den Perlinterpreter aufrufen. Dieser parst und kompiliert zunächst das gewünschte Skript und evtl. zusätzliche Module. Anschliessend kann dieses Skript mit der eigentlichen Aufgabe beginnen, z.B. eine Datenbank befragen und die Ausgabe generieren. Diese Schritte sind zum einen sehr aufwendig und zum anderen redundant, da die Anzahl der Webseitenaufrufe auf einem produktiven Server i.d.R. die Häufigkeit von Änderungen an der Perlinstallation oder an den CGI-Skripten übersteigt. Daher wird der Perlinterpreter in den Webserver geladen. Das Modul *mod_perl* bzw. dessen Nachfolger *mod_perl2* sind als eigene Pakete im Portssystem von FreeBSD verfügbar. Die Installation von *mod_perl2* ist relativ einfach:

```
$ cd /usr/ports/www/mod_perl2
$ make install clean
```

Die Konfigurationsdatei des Webservers muss um folgende Zeile ergänzt werden:

```
LoadModule    perl_module    libexec/apache22/mod_perl.so
```

Die relative Pfadangabe `libexec/apache22/mod_perl.so` setzt voraus, dass der Parameter `ServerRoot` als `/usr/local` konfiguriert ist. Damit Perlskripte nicht vom (externen) Interpreter ausgeführt werden, sondern von *mod_perl2*, verwendet man die Anweisung `SetHandler perl-script`, z.B. innerhalb eines per `<LocationMatch>` definierten Abschnittes:

```
LoadModule    perl_module    libexec/apache22/mod_perl.so
<VirtualHost *>
    ServerName      www.fh-bielefeld.de
    DocumentRoot    /var/www/www.fh-bielefeld.de
    <LocationMatch ^/gaestebuch/>
        SetHandler    perl-script
    </LocationMatch>
</VirtualHost>
```

Ruft ein Besucher in seinem Browser z.B. die URL
`http://www.fh-bielefeld.de/gaestebuch/script.pl` auf, so wird der Webserver die lokale Datei `/var/www/www.fh-bielefeld.de/gaestebuch/script.pl` per *mod_perl2* ausführen. Kapselt man ein Skript als Perlmodul, so kann dieses Modul beim Starten des Webservers von *mod_perl2* interpretiert und als kompilierter Bytecode im Speicher gehalten werden. Ein solches Modul muss wie folgt strukturiert sein:

```
#!/usr/bin/perl

package FHBielefeld::Gaestebuch;

sub handler ()
{
    # ehemaliger Skriptcode...
}

# evtl. lokale Subroutinen...

# erfolgreiches Einbinden melden
1;
```

Der Name des Package ist frei wählbar, darf aber nicht mit anderen Paketen kollidieren und sollte generell passend zur Anwendung gewählt werden. Das eigentliche Skript bzw. dessen Code auf der Hauptebene muss in eine Funktion namens `handler` kopiert werden. In der Konfiguration des Webservers muss der Name des Packages als `PerlResponseHandler` hinterlegt werden:

```
    <LocationMatch ^/gaestebuch/>
        SetHandler             perl-script
        PerlResponseHandler    FHBielefeld::Gaestebuch
        PerlOptions            +SetupEnv
    </LocationMatch>
```

Somit wird der Apache Anfragen nach jeglichen Dokumenten, deren URL mit `http://www.fh-bielefeld.de/gaestebuch/` beginnt, an die Funktion `handler` im

Package `FHBielefeld::Gaestebuch` weitergeben. Ist wie im Beispiel die zusätzliche Option `+SetupEnv` gesetzt, so kann im Package auf den Hash `%ENV` zugegriffen werden, der z.B. unter dem Schlüssel `$ENV{REQUEST_URI}` den Pfad des ursprünglich gewünschten Dokumentes beinhaltet. Ruft ein Besucher die Seite `http://www.fh-bielefeld.de/gaestebuch/script.pl` auf, ist in `$ENV{REQUEST_URI}` der String `/gaestebuch/script.pl` hinterlegt. Mit der Anweisung `PerlSetVar` können der Funktion `handler` zusätzliche Optionen übergeben werden:

```
<LocationMatch ^/gaestebuch/>
    SetHandler              perl-script
    PerlResponseHandler     FHBielefeld::Gaestebuch
    PerlOptions             +SetupEnv
    PerlSetVar    Ausgabe   hello, world
</LocationMatch>
```

Die Funktion `handler` bekommt als einzigen Parameter eine Instanz der Klasse `Apache2::RequestRec` übergeben, welche von `Apache2::RequestUtil` erbt. Die Memberfunktion `dir_config` der Klasse `Apache2::RequestUtil` wird genutzt, um die per `PerlSetVar` definierten Optionen abzufragen:

```
#!/usr/bin/perl

package FHBielefeld::Gaestebuch;

sub handler ()
{
    my $Request = shift;
    my $Ausgabe = $Request->dir_config("Ausgabe");

    # ...
}
```

Der Scalar `$Ausgabe` enthält nun den String `hello, world`, wie in der Konfiguration des Webservers definiert. Damit beim Starten von Apache alle verwendeten Perlmodule geladen werden, verwendet man den Parameter `PerlRequire` gefolgt von einer Pfadangabe zu einem Perlskript:

```
PerlRequire /usr/local/etc/preload.pl
```

Das Skript `preload.pl` muss lediglich alle verwendeten Module per `use` laden. Durch den Einsatz von mod_perl werden sie in kompilierter Form im Speicher gehalten.

4.2. PostgreSQL

4.2.1. Einleitung

Der Datenbankserver *PostgreSQL* ist eine Weiterentwicklung des *POSTGRES* Projektes. Dieses wurde im Jahre 1986 an der UC Berkeley initiiert und stellte ein Modell zur objekt-relationalen Datenhaltung dar. Es verfügte über eine eigene Abfragesprache namens *PostQUEL*, deren Semantik an die heute übliche *Structured Query Language* (SQL) erinnert. PostgreSQL verwendet jedoch ausschliesslich SQL als Abfragesprache (s. Kapitel 7.1). PostgreSQL hält Daten in Datenbanken vor, die wiederum sogenannte *Relationen* beinhalten. Die Menge aller Datenbanken wird im PostgreSQL-Umfeld *Datenbankcluster* genannt. Ein Datenbankserver verfügt i.d.R. über genau einen solchen Cluster und stellt für jede Anwendung eine eigene Datenbank bereit. Analog werden Relationen so modelliert, dass sie möglichst ein Abbild real existierender Datensammlungen

entsprechen. Bildlich hat sich für eine Relation der Begriff *Tabelle* durchgesetzt. So kann z.B. die Relation `Person` je nach Anwendungszweck die Attribute `Vorname` und `Nachname` umfassen. Die Datenhaltung erfolgt dann in einer Tabelle mit dem Namen `Person` oder auch `Personen`. Die Spalten dieser Tabelle haben dann die Bezeichnung `Vorname` und `Nachname`. Jeder Reihe dieser Tabelle stellt dann eine Person dar. Die Spalten müssen zudem von einem bestimmten Datentyp sein, hier z.B. bieten sich zwei Zeichenketten an. PostgreSQL ermöglicht es, dass Tabellen im Sinne der objektorientieren Programmierung voneinander erben. Die Tabelle `Mitarbeiter` könnte z.B. von `Person` erben, so dass Mitarbeiter Vor- und Nachnamen haben (Attributvererbung) und dass jeder Mitarbeiter automatisch eine Person ist (Typvererbung). Die physikalische Speicherung der Daten erfolgt in binären Dateien. PostgreSQL schreibt veränderte Daten jedoch nicht unmittelbar in den eigentlichen Datenbereich, sondern zunächst in ein sequentielles Logfile. Dieses *Write Ahead Logging* (WAL) genannte Verfahren bietet zwei Vorteile. Zum einen ist sequentielles Schreiben in eine Datei schneller als wahlfreier Zugriff innerhalb einer Datei. Zum anderen kann somit relativ einfach ein Spiegel der Datenbank betrieben werden. Der Zugriff auf die Daten erfolgt i.d.R über ein Netzwerk mittels binärem Protokoll. Clientbibliotheken sind für Sprachen wie Perl, PHP, C, u.v.m. verfügbar.

4.2.2. Installation

Die Installation von PostgreSQL gestaltet sich unter FreeBSD aufgrund des Portssystems sehr einfach:

```
$ cd /usr/ports/databases/postgresql82-server
$ make config install clean
```

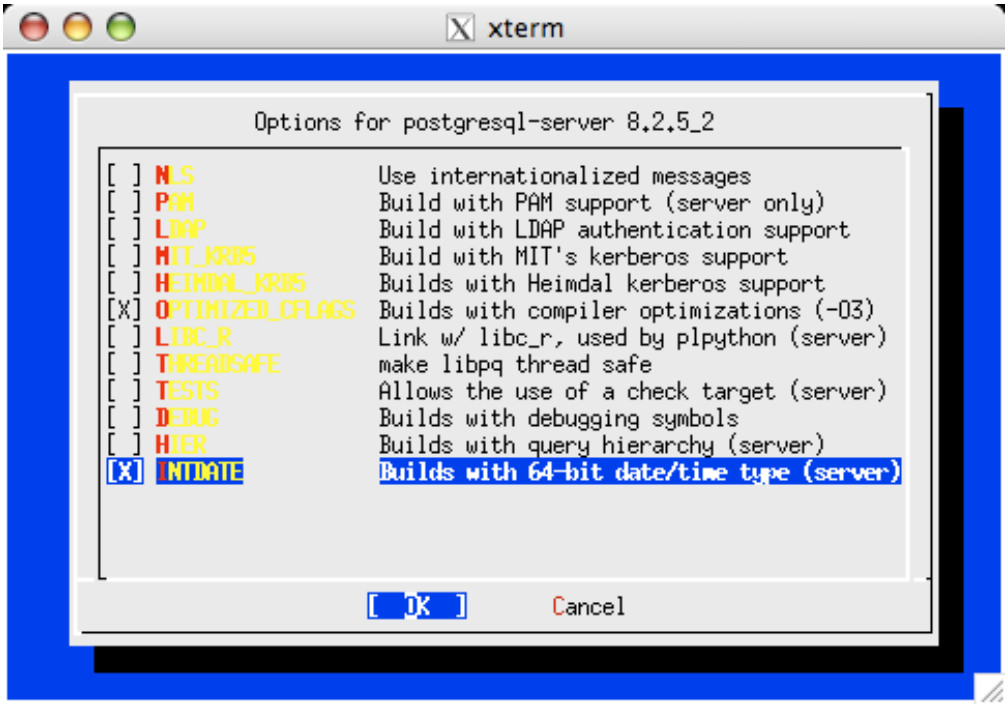

Abbildung 6: Das Optionsmenü zur Installation des PostgreSQL Datenbankservers

Im Optionsdialog (s. Abbildung 6) wurden lediglich die Parameter `OPTIMIZED_CFLAGS` und `INTDATE` ausgewählt. Ersterer übersetzt den Server mit dem Compilerflag `-O3`, welches zugunsten der Ausführungsgeschwindigkeit grösseren Code erzeugt. Der Parameter

`INTDATE` bewirkt, dass der PostgreSQL-Server für Datum- und Zeitwerte 8 Byte grosse Integerwerte statt Fließkommazahlen verwendet. Dies schränkt zwar den absoluten Bereich darstellbarer Daten von einem Zeitraum zwischen 4713 v.Chr. und 5874897 n.Chr. auf einen Zeitraum zwischen 4713 v.Chr. und 294276 n.Chr. ein, garantiert jedoch eine Auflösung von einer Mikrosekunde über den gesamten Zeitraum. Ausserdem wird in Diverse (2006c)[9] darauf hingewiesen, dass im Fließkommacode noch immer Fehler gefunden werden. Vor dem ersten Start des PostgreSQL-Servers wurden in der zentralen Konfigurationsdatei `/etc/rc.conf` folgende Variablen gesetzt:

```
postgresql_enable="YES"
postgresql_data="/var/db/pgsql/data"
```

Der Parameter `postgresql_data` gibt das Verzeichnis an, in dem die Dateien der Datenbank gespeichert werden. Auf dem Produktionsserver ist diese Angabe wichtig, da der unter `/var` eingebundenen Partition der meiste Speicherplatz zugewiesen wurde. Per Aufruf von `/usr/local/etc/rc.d/postgresql start` wird der Datenserver manuell gestartet. Existiert das Datenverzeichnis `/var/db/pgsql/data` noch nicht, so legt es das Startskript automatisch an und initialisiert den Datenbankserver per Aufruf des Programmes `initdb`.

4.2.3. Konfiguration

Per default ist für den Administrationsbenutzer `pgsql` des PostgreSQL-Servers kein Passwort vergeben. Dieses kann mit dem Kommandozeilenprogramm `psql` vergeben werden. Hierzu verbindet man sich lokal auf die durch die Installation angelegte Testdatenbank `postgres` und vergibt mit dem SQL-Befehl `ALTER USER` ein Passwort für den Benutzer `pgsql`. Die einzelnen Schritte zeigt Abbildung 7. Die Textdatei

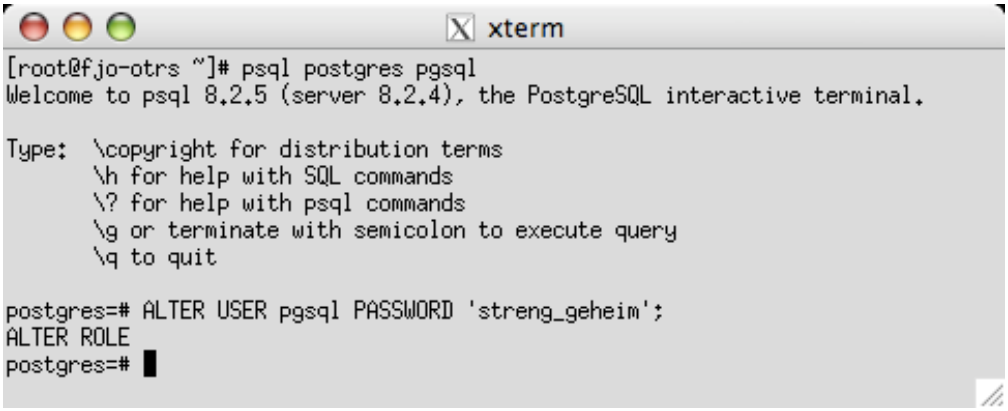

Abbildung 7: Passwortänderung für den Administrator des PostgreSQL Datenbankservers

`/var/db/pgsql/data/pg_hba.conf` stellt einen Filter dar, der Benutzern Verbindungen auf die Datenbank gestattet oder verwehrt. Die Datei besteht aus 5 per Whitespace getrennten Spalten. Jede Zeile beschreibt eine Zugriffserlaubnis:

```
host    db1    user1    10.0.0.0/8    md5
```

[9] http://www.postgresql.org/docs/8.2/static/install-procedure.html

Diese Zeile gibt dem Benutzer `user1` Zugriff auf die Datenbank `db1`, wenn die Verbindung vom einem Rechner aus dem IP-Netz `10.0.0.0/8` kommt und eine Passwortauthentifizierung per MD5-Hash stattfindet. Statt expliziter Datenbank- und/oder Benutzernamen kann der Wildcard `all` verwendet werden:

```
host    all    all    10.0.0.0/8    md5
```

Somit wird allen Benutzern Zugriff auf jede Datenbank gestattet, wenn sie aus dem Netz `10.0.0.0/8` kommen und sich per MD5-Hash authentifizieren. Obwohl eine Anmeldung per MD5-Schlüssel sehr sicher ist, existieren ferner die Authentifizierungsmethoden `trust`, welche kein Passwort verlangt, und `password`, welche ein Klartextpassword erwartet. Verwendet man `local` statt des Schlüsselwortes `host`, so definiert man Zugriffsrechte für Benutzer, die sich lokal über einen *Unix-Socket* auf die Datenbank verbinden möchten. Die Angabe eines IP-Netzes oder einer IP-Adresse ist hierbei auszulassen:

```
local    all    all        md5
```

Diese Zeile gestattet allen Benutzern, die sich erfolgreich per MD5-Hash authentifiziert haben, Zugriff auf jede Datenbank, sofern die Verbindung lokal erfolgt. Die Datei `pg_hba.conf` definiert lediglich Verbindungsrechte. Jeder Datenbank ist eine Liste zugeordnet, die einzelnen Benutzern entsprechende Rechte zum eigentlichen Auslesen und Bearbeiten der Daten gestattet. Diese Liste wird mit dem SQL-Kommando `GRANT` modifiziert (s. Kapitel 7.1). Soll dem Benutzer `otrs1` nur Zugriff auf die Datenbank `otrs1`, dem Benutzer `otrs2` nur Zugriff auf `otrs2`, usw. gegeben werden, kann für den Datenbanknamen der Bezeichner `sameuser` verwendet werden:

```
local    sameuser    all        md5
```

Der Administrator `pgsql` soll natürlich weiterhin auf alle Datenbanken zugreifen dürfen:

```
local    all         pgsql        md5
local    sameuser    all          md5
```

Da Datenbanknamen und Benutzernamen der einzelnen OTRS-Instanzen i.d.R. gleichlautend sind, wurde auf dem Entwicklungs- und dem Produktionsserver die oben gezeigte Konfiguration in die Datei `pg_hba.conf` übernommen. Die Textdatei `/var/db/pgsql/data/postgresql.conf` stellt die Hauptkonfiguration des PostgreSQL-Servers dar. Sie wurde um die folgenden Parameter erweitert bzw. entsprechend angepasst:

`log_destination = 'syslog'` leitet alle Logausgaben der Datenbank an den Systemlogdienst weiter

`autovacuum = on` startet einen internen Hilfsprozess, der die Datenbank bereinigt und nicht belegten Speicherplatz freigibt; hierfür müssen ferner die Optionen `stats_block_level = on` und `stats_row_level = on` aktiviert sein.

Damit die Meldungen des PostgreSQL-Servers vom Systemlogdienst in die separate Datei `/var/log/pgsql.log` geschrieben werden, wurde die Konfigurationsdatei `/etc/syslog.conf` um folgende Zeile erweitert:

```
local0.*    /var/log/pgsql.log
```

4.3. Postfix

4.3.1. Einleitung

Auf einem FreeBSD-System ist per default *sendmail* installiert. Dies ist ein sogenannter *Mail Transfer Agent* (MTA), der für die Annahme, das Routing und die Weiterleitung von Emails zuständig ist. Weil sendmail eine auffällige (Un-)Sicherheitshistorie[10] vorzuweisen hat, verarbeitet es unter FreeBSD standardmässig nur Emails, die von lokalen Benutzern des Systems versendet werden. Es akzeptiert keine per Netzwerk eingelieferten Emails. Obschon ein- als auch ausgehende Emails über separate, von der DTS Service betreute und daher aus Sicht interner Systeme vertrauenswürdige Mailserver geleitet werden können, wurde sendmail deaktiviert und durch *Postfix* ersetzt. Postfix wurde von Wietse Venema als sichere Alternative zu sendmail entwickelt. Während sendmail ein monolithisches Programm ist, besteht Postfix aus mehreren Teilen, die mit unterschiedlichen Benutzerrechten laufen und über definierte Schnittstellen miteinander kommunizieren. Dennoch wird Postfix zentral mittels einiger Textdateien konfiguriert. Im Gegensatz zu sendmail sind diese Konfigurationsdateien verständlicher und gut strukturiert. Da pro Maschine mehrere OTRS-Instanzen mit unterschiedlichen Benutzerrechten laufen, muss das jeweilige Perlskript, welches Emails in die Datenbank übernimmt, ebenfalls mit den Rechten des entsprechenden Benutzers ausgeführt werden. In Postfix wird dies mit 3 trivialen Textzeilen erreicht. Mit sendmail würde dies entweder die manuelle Bearbeitung der `sendmail.cf`, der zentralen und komplexen Konfigurationsdatei, oder die Erweiterung des Präprozessors bedeuten, welcher die `sendmail.cf` erzeugt.

4.3.2. Installation

Die Metadaten zum Installieren von Postfix über das Portssystem liegen im Verzeichnis `/usr/ports/mail/postfix`. Nach dem Wechsel dorthin ruft man den Befehl `make config install clean` auf, um den Optionsdialog aufzurufen, Postfix zu installieren, und um temporäre Dateien nach der Kompilierung zu löschen. Im Optionsdialog (s. Abbildung 8) sind alle Parameter abzuwählen. Postfix kann verschiedene SQL-Datenbanken, *Lightweight Directory Access Protocol* (LDAP)-Verzeichnisse
und In-Memory-Datenbanken abfragen, um so Routinggentscheidungen für Emails zu treffen. Diese Möglichkeiten werden nicht benötigt. Als Datenquelle werden auf dem OTRS-Server automatisch erzeugte Hashdateien verwendet, die von einem Perlskript angelegt werden. Die Unterstützung für derartige Hashdateien gehört zum Standardumfang von Postfix.

4.3.3. Konfiguration

Noch während der Installation wird der Administrator gefragt, ob Postfix als Standardmailserver des Systems in die Datei `/etc/mail/mailer.conf` eingetragen werden soll. Dies ist zu bejahen. Nach der Installation wird sendmail per Aufruf von
`/etc/rc.d/sendmail stop` beendet. Anschliessend muss die zentrale Konfigurationsdatei `/etc/rc.conf` des FreeBSD-Systems angepasst werden:

```
sendmail_enable="NONE"
postfix_enable="YES"
```

Somit wird beim Systemstart nicht mehr sendmail, sondern Postfix aktiviert. Manuell kann man Postfix per Aufruf von `/usr/local/etc/rc.d/postfix` starten, stoppen

[10] s.a. http://cr.yp.to/maildisasters/sendmail.html

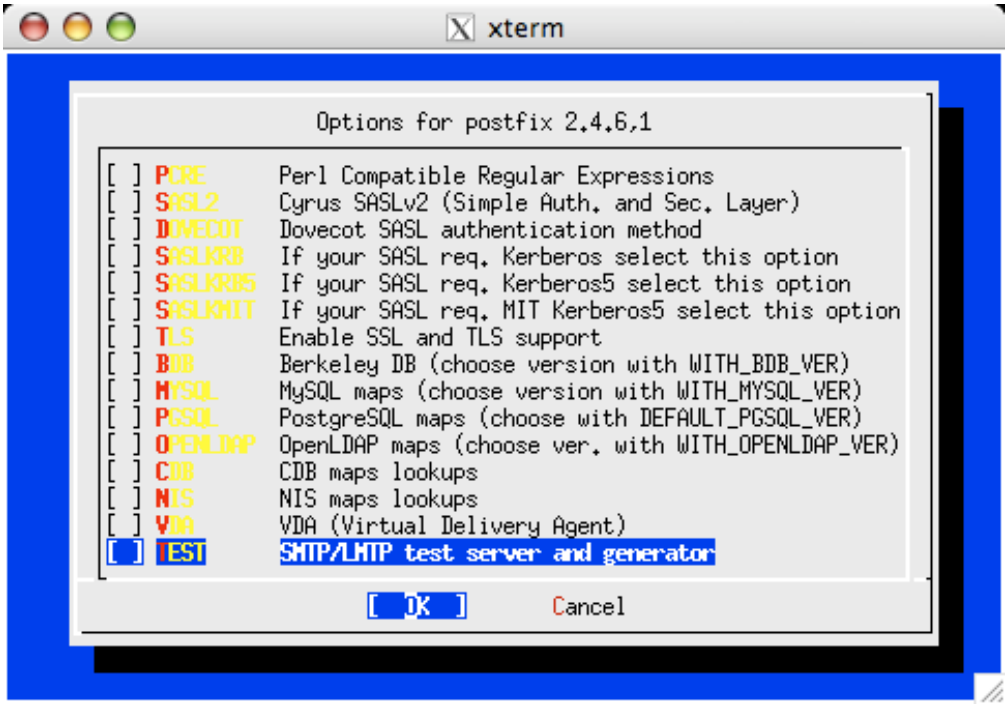

Abbildung 8: Das Optionsmenü zur Installation des Postfix Mailservers

oder zum erneuten Einlesen seiner Konfiguration veranlassen, wenn man den Parameter `start`, `stop` respektive `reload` verwendet. Die Konfigurationsdateien liegen im Verzeichnis `/usr/local/etc/postfix`. Die Datei `master.cf` definiert alle innerhalb des Mailservers verfügbaren Dienste. Jede Zeile besteht aus 8, per Whitespace getrennten Spalten und stellt einen Service dar, z.B.:

```
smtp    inet    n    -    n    -    -    smtpd
```

Die einzelnen Felder haben folgende Bedeutung:

1. **Name (`smtp`)** Das erste Feld definiert den frei wählbaren Namen des Services. Der Name zusammen mit dem Typ (s.u.) müssen eindeutig sein.

2. **Typ (`inet`)** Das zweite Feld definiert den Typ des Services. Der Wert `inet` gibt an, dass dieser Dienst bzw. das Programm (s. Punkt 8.) so gestartet werden soll, dass er über das Netzwerk konnektierbar ist. Der Wert `unix` gibt an, dass dieser Dienst so gestartet werden soll, dass er nur per lokalem Unix-Socket angesprochen werden kann.

3. **Privatflag (`n`)** Das dritte Feld akzeptiert nur die boolschen Werte `n` für *no*, `y` für *yes* oder `-` für den Defaultwert. Es gibt an, ob der Zugriff auf diesen Dienst nur durch Postfix selbst gestattet sein soll. Für Dienste vom Typ `inet` muss hier `n` angegeben werden. Der Standardwert (`-`) ist gleichbedeutend mit `y`.

4. **Privilegflag (`-`)** Das vierte Feld ist ebenfalls ein boolsches Feld. Der Defaultwert ist `y` und gibt an, dass der Dienst nicht mit den Rechten des Administrators `root` arbeiten darf.

5. **Chroot-Flag (`n`)** Das fünfte Feld bestimmt, ob der Dienst in einer vom restlichen System abgeschotteten Umgebung, einer sogenannten *Chroot-Umgebung*, laufen soll. Der Standardwert ist `y`.

6. **Interval (`-`)** Das sechste Feld gibt die Zeit in Sekunden an, nach der ein Dienst erneut gestartet wird. Der Standardwert von `0` kann wie bei boolschen Feldern (s.o.) per `-` referenziert werden. Ein Zeitwert von null Sekunden deaktiviert das erneute Starten eines Services. Das Feld wird i.d.R. nur für einige interne Postfixprozesse gesetzt, z.B. für den Dienst, der die Mailqueue verwaltet.

7. **Prozesslimit (`-`)** Das siebte Feld gibt an, wie viele Instanzen von diesem Service gleichzeitig laufen dürfen. Auch hier stellt der Bindestrich `-` den Defaultwert dar, der gleich dem Wert der Variablen `default_process_limit` in der Datei `main.cf` (s.u.) ist. Standardmässig ist `default_process_limit` auf `100` gesetzt. Ein Prozesslimit von `0` erlaubt beliebig viele gleichzeitige Instanzen eines Dienstes.

8. **Programm und optionale Argumente (`smtpd`)** Im letzten Feld wird der Name des Programmes erwartet, welches bei Verbindungen auf diesen Service ausgeführt wird. Wird wie im Beispiel kein absoluter Pfad angegeben, so geht Postfix davon aus, dass das Programm in dem Verzeichnis liegt, welches durch den Parameter `daemon_directory` in der Datei `main.cf` spezifiziert ist. Optionale Kommandozeilenargumente werden meist per Whitespace eingerückt in der nächsten Zeile angegeben.

Das Skript `PostMaster.pl` einer OTRS-Instanz liest eine Email von der Standardeingabe und schreibt sie in die Datenbank. Das Programm `pipe` des Postfixservers kann Emails auf die Standardeingabe von beliebigen externen Programmen weiterleiten. Der notwendige Eintrag in der `master.cf` sieht wie folgt aus:

```
otrs2     unix    -    n    n    -    -    pipe
        user=otrs2 argv=/var/otrs/otrs2/bin/PostMaster.pl
```

Der Postfixservice `otrs2` wird über einen lokalen Unix-Socket angesprochen. Er darf nur vom Postfixserver angesprochen werden, läuft mit Administratorrechten, wird nicht in einer Chroot-Umgebung gekapselt, unterliegt keinem Reaktivierungsinterval, und darf höchstens 100 Mal gleichzeitig gestartet werden. Das Programm `pipe` wird mit den Rechten des Benutzers `otrs2` das externe Programm `/var/otrs/otrs2/bin/PostMaster.pl` starten, sobald durch das Routing eine Mail an den Service `otrs2` weitergeleitet wird. Die Datei `main.cf` stellt die Hauptkonfiguration eines Postfixservers dar, in der neben über 500 Betriebsparametern auch Datenbanken oder Tabellen konfiguriert werden, die das Mailrouting bestimmen. Postfix teilt Empfängeremailadressen bzw. deren Domains in Klassen ein. Eine Klasse bestimmt die weitere Verarbeitung einer Email. Gehört eine Domain z.B. zur *Relayklasse*, so werden Emails angenommen und weitergeleitet, wenn die Empfängeradresse zu jener Domain gehört. Der Parameter `relay_domains` gibt eine externe Tabelle an, in der alle Domains aufgeführt werden, die zur Relayklasse gehören:

```
relay_domains = hash:/usr/local/etc/postfix/relay_domains
```

Die Datei `/usr/local/etc/postfix/relay_domains` ist eine Textdatei, die in jeder Zeile den Namen einer Domain aufführt, die zur Relayklasse gehört, z.B.:

```
fh-bielefeld.de
dts.de
```

Das Prefix `hash:` gibt an, dass nicht diese Textdatei direkt, sondern ein daraus erzeugtes Datenbankfile mit dem Namen `relay_domains.db` verwendet werden soll. Dies ist besonders bei sehr vielen Einträgen sinnvoll, da indizierte Textdateien effizienter zu durchsuchen sind. Das Programm `postmap` erzeugt ein solches Datenbankfile, welches automatisch das Suffix `db` erhält:

```
$ postmap /usr/local/etc/postfix/relay_domains
```

Ohne zusätzliche Transporteinträge werden zwar Mails für die Domains der Relayklasse angenommen, aber nicht an die OTRS-Instanzen weitergeleitet. Im schlimmsten Falle würde man sogar eine Mailschleife konstruieren. Daher schränkt man zunächst die möglichen Empfängeradressen aller Domains in der Relayklasse ein:

```
relay_recipient_maps = hash:/usr/local/etc/postfix/relay_recipient_maps
```

Der Parameter `relay_recipient_maps` in der `main.cf` gibt an, welche Emailadressen aus der Relayklasse angenommen werden können. Die Textdatei `relay_recipient_maps` enthält pro Zeile eine Empfängeradresse, z.B.:

```
webmaster@fh-bielefeld.de
student1@fh-bielefeld.de
info@dts.de
abuse@dts.de
```

Auch diese Datei muss per `postmap` indiziert werden:

```
$ postmap /usr/local/etc/postfix/relay_recipient_maps
```

Mit einer weiteren Tabelle, die Transportziele enthält, werden nun diese Empfängeradressen an verschiedene Ziele wie z.B. OTRS-Instanzen weitergeleitet. Die Datei `/usr/local/etc/postfix/transport_maps` hat folgenden Inhalt:

```
webmaster@fh-bielefeld.de    otrs1:[127.0.0.1]
student1@fh-bielefeld.de     otrs2:[127.0.0.1]
info@dts.de                  otrs2:[127.0.0.1]
abuse@dts.de                 otrs1:[127.0.0.1]
```

Die Emailadressen `webmaster@fh-bielefeld.de` und `abuse@dts.de` werden an den Service `otrs1`, Mails an `student1@fh-bielefeld.de` oder `info@dts.de` an den Service `otrs2` weitergeleitet. Sind beide Dienste wie oben gezeigt in der Datei `master.cf` eingetragen, werden Mails über das Skript `PostMaster.pl` in die jeweilige OTRS-Datenbank eingetragen. Die IP-Adresse `127.0.0.1` ist hier ein Platzhalter, der das erwartete Format der `transport_maps` kompletiert.

4.4. OpenSSL

OpenSSL ist ein Softwarepaket zum Umgang mit kryptographischen Schlüsseln und Zertifikaten. Es ist frei als Opensource-Software[11] erhältlich und gehört zum Standardumfang eines FreeBSD-Systems. OpenSSL besteht im wesentlichen aus einer C-Biblibothek für *Secure Sockets Layer* (SSL) und dem darauf basierenden Kommandozeilentool `openssl`. Die Bibliothek wird meist in Programmen verwendet, die über einen unsicheren Kommunikationskanal wie dem Internet Nachrichten austauschen müssen. So greift der Apache Webserver (s. Kapitel 4.1) mit dem Modul `mod_ssl` auf die OpenSSL-Library zu.

[11] http://www.openssl.org

Das Kommandozeilenprogramm wird eingesetzt, um die für eine sichere Kommunikation notwendigen Schlüssel und Zertifikate zu erstellen. Sicherheit bedeutet:

Vertraulichkeit Die übertragenen Daten können nur von den an der Kommunikation teilnehmenden und dazu authorisierten Parteien eingesehen werden, z.B. dem Browser des Anwenders und dem HTTPS-Webserver.

Integrität Die Daten werden auf ihrem Weg zwischen den Parteien nicht verändert. Insbesondere muss Datenmanipulation erkannt und als ungültige Kommunikation von allen Teilnehmern verworfen werden.

Authentizität Die Kommunikationspartner müssen untereinander vertrauen. So sind Daten, die vertraulich und integer übermittelt wurden, wertlos, wenn der Kommunikationspartner nicht derjenige ist, der er vorgibt zu sein.

Vertraulichkeit wird i.d.R. mittels Blockchiffren wie dem *Advanced Encryption Standard* (AES) oder dem älteren *Data Encryption Standard* (DES) erreicht. Neben einer sicheren, sprich *starken* Verschlüsselung soll ein Chiffrierungsalgorithmus einen hohen Datendurchsatz bieten. Integrität lässt sich mit Prüfsummen oder Hash-Funktionen erreichen. Hash-Funktionen bilden beliebig grosse Urmengen auf eine beschränkte Bildmenge ab. So erzeugt die Hash-Funktion *Secure Hash Algorithmus* (SHA) unabhängig von den eingegebenen Daten immer eine 20 Byte grosse Ausgabe, den *Hash-Wert* dieser Daten. Man fordert von einer Hash-Funktion Bijektivität und Nicht-Existenz einer Umkehrfunktion. Obgleich es wegen der beschränkten Bildmenge theoretisch unmöglich ist, sollen keine Eingabedaten existieren, die den gleichen Hash-Wert liefern. Liefern zwei unterschiedliche Eingabedaten den gleichen Hash-Wert, so liegt eine *Kollision* vor. Auf keinen Fall darf es möglich sein, aus einem Hash-Wert das ursprüngliche Datum zu ermitteln. Ferner sollen alle möglichen Hash-Werte tatsächlich erreicht werden können. Eine Vertrauensstellung zwischen den Kommunikationsteilnehmern erreicht man mittels *asymetrischer Verschlüsselung* bei Verwendung einer Zertifizierungsinstanz. Hierzu erzeugen die Kommunikationsteilnehmer, deren Authentizität bestätigt werden soll, ein sogenanntes *Schlüsselpaar*, bestehend aus einem privaten (*private key*) und einem öffentlichen Schlüssel (*public key*). Beide Schlüssel müssen folgenden Anforderungen genügen:

- Der private Schlüssel darf nicht aus dem öffentlichen Schlüssel berechnet werden können
- Mit dem privaten Schlüssel können Daten digital signiert werden. Diese Signatur und somit die Integrität der Daten kann mit dem öffentlichen Schlüssel verifiziert werden
- Mit dem öffentlichen Schlüssel können Daten vertraulich verschlüsselt werden. Die verschlüsselten Daten dürfen nur mit Hilfe des privaten Schlüssels wieder in die ursprünglichen Klartextdaten überführt werden.
- Es muss überprüft werden können, ob privater und öffentlicher Schlüssel zusammen gehören. Praktisch bedeutet dies, dass der öffentliche Schlüssel aus dem privaten erzeugt werden kann.

Der öffentliche Schlüssel sollte publiziert werden oder zumindest auf Anfrage abrufbar sein. Der private Schlüssel darf jedoch auf keinen Fall veröffentlich werden. Die Vertrauensstellung wird erst mit Hilfe einer dritten, an der eigentlichen Datenübertragung unbeteiligten Partei erreicht, der sogenannten *Certificate Authority* (CA). Diese signiert

mit ihrem privaten Schlüssel die öffentlichen Schlüssel aller Kommunikationsteilnehmer und deren Metadaten wie Name, Emailadresse, Wohnort, etc. Mit diesen *Zertifikaten* kann sich nun jeder Kommunikationspartner gegenüber anderen ausweisen. Jeder Teilnehmer kann mit Hilfe des öffentlichen Schlüssels der CA ermitteln, ob das Zertifikat seines Gegenübers tatsächlich von der CA ausgestellt wurde. Vertrauen alle Teilnehmer der CA, so vertrauen sie automatisch ihrem jeweiligen Kommunikationspartner. Wie und ob ein Teilnehmer der CA vertraut, kann allerdings nicht technisch gelöst werden[12]. In der Praxis wird die gesamte Situation vereinfacht. Zum einen kommunizieren nur zwei Teilnehmer miteinander, entweder ein Client und ein Server (z.B. per HTTPS) oder zwei natürliche Personen per sicherer Email. Zum anderen wird auf eine Authentizitätsüberprüfung des Clients meist verzichtet (nicht jedoch in Sicherheitsarchitekturen wie Kerberos oder dem davon abgeleiteten *Active Directory* (AD)). Um ein Zertifikat zu erhalten, sind vier Schritte notwendig:

1. Ein Schlüsselpaar muss vom Antragsteller des Zertifikates generiert werden
2. Der öffentliche Schlüssel wird in einem Zertifikatsrequest zusammen mit Angaben über den Antragsteller wie Name, Emailadresse, etc. an eine CA gesendet
3. Mit ihrem privaten Schlüssel signiert die CA nach Überprüfung die im Antrag enthaltenen Daten und sendet das erzeugte Zertifikat an den Antragsteller zurück
4. Zertifikat und privater Schlüssel müssen in die Anwendung, z.B. einen Webserver, eingebunden werden.

Schritt 3 ist i.d.R. mit finanziellem Aufwand für den Antragsteller verbunden und wird daher hier nicht angewendet. Stattdessen wird der Zertifikatsrequest mit dem privaten Schlüssel des Antragstellers unterschrieben. Man spricht dann von einem *selbst-signierten Zertifikat*. Setzt man dieses und den privaten Schlüssel in einem Webserver ein, so werden nur Vertraulichkeit und Integrität der Daten erzielt. Für den Einsatz in den Webservern der OTRS-Instanzen bietet dies hinreichende Sicherheit. Natürlich kann ein selbstsigniertes gegen ein von einer anerkannten CA unterschriebenes Zertifikat ausgetauscht werden. Für die Schritte 1 bis 3 kann OpenSSL verwendet werden. Mit folgendem Befehl wird ein privater Schlüssel nach dem *Rivest-Shamir-Adleman* (RSA)-Algorithmus erzeugt:

```
$ openssl genrsa -out private_key.txt
```

OpenSSL unterstützt neben RSA auch den *Digital Signature Algorithm* (DSA). Dieser erfordert eine Parameterdatei, die jedoch an dieser Stelle nicht weiter erläutert wird. Der öffentliche Schlüssel kann mit der Option `-pubout` ausgegeben werden:

```
$ openssl rsa -in private_key.txt -pubout
```

Einen Zertifikatsrequest erzeugt man mit dem folgenden Befehl:

```
$ openssl req -new -config req_config.txt -key private_key.txt \
    -out cert_req.txt
```

Die Datei `cert_req.txt` enthält den öffentlichen Schlüssel sowie die Angaben über den Antragsteller aus der Datei `req_config.txt` und kann daher zur Zertifikatserstellung an eine CA gesendet werden. Verwendet man zusätzlich den Parameter `-x509`, so erzeugt OpenSSL ein selbst-signiertes Zertifikat:

[12] Vgl. die Situation im alltäglichen Leben, in der die Echtheit einer Person festgestellt werden muss, was i.a. durch Vertrauen in hoheitlich ausgestellte Dokumente wie Personalausweis, Geburtsurkunde, etc. gelöst wird.

```
$ openssl req -new -x509 -config req_config.txt -key private_key.txt \
    -out self_signed_cert.txt
```

Der private Schlüssel `private_key.txt` und das selbst-signierte Zertifikat `self_signed_cert.txt` sind geeignet, um z.B. per `mod_ssl` in einen Apache Webserver eingebunden zu werden. Die Datei `req_config.txt` besteht aus mindestens zwei Abschnitten:

```
[ req ]
distinguished_name = request_options
prompt = no

[ request_options ]
C            = DE
ST           = NRW
L            = Herford
O            = DTS
OU           = ISP
CN           = www.dts.de
emailAddress = info@dts.de
```

Der Abschnitt `[ req ]` wird beim Erzeugen eines neuen Zertifikatsrequestes oder eines selbst-signierten Zertifikates ausgewertet. Der Parameter `prompt` gibt an, ob die Angaben über den Antragsteller manuell beim Aufruf von `openssl` eingegeben werden sollen oder ob die Werte aus dieser Konfiguration verwendet werden sollen. Wird `prompt` auf `no` gesetzt, werden jegliche interaktiven Rückfragen unterdrückt. Der Parameter `distinguished_name` definiert den Abschnitt, der die Angaben zum Antragsteller enthält, hier `request_options`. In diesem Abschnitt werden folgende Daten erwartet:

Country (C) definiert das Heimatland des Antragstellers bzw. das Land, in dem der Webserver steht

State (ST) definiert das Bundesland

Locality (L) definiert die Heimatstadt des Antragstellers bzw. die Stadt, in der der Webserver steht

Organization (O) definiert die Firma bzw. Einrichtung, für die der Antragsteller arbeitet oder die den Webserver besitzt

Organizational Unit (OU) definiert die Abteilung, in der der Antragsteller arbeitet bzw. die den Webserver betreut

Common Name (CN) definiert den allgemeinhin bekannten Namen des Webservers

emailAddress definiert die Emailadresse des Antragstellers bzw. des für den Webserver zuständigen Administrators

Pro Datei können mehrere solcher Abschnitte angelegt werden; sie müssen sich lediglich im Namen unterscheiden (z.B. `request_options1`, `request_options2`, usw.). Somit ist es möglich, mit einer Konfigurationsdatei unterschiedliche Zertifikatsanträge zu erzeugen.

5. Programmiersprachen

5.1. Perl

5.1.1. Aufruf

Die *Practical Extraction and Report Language* (Perl) wurde 1987 vom damals bei der *National Security Agency* (NSA) angestellten Linguisten Larry Wall entwickelt, um ein einfaches, aber dennoch mächtiges Werkzeug zur Textverarbeitung zu erhalten. Perl ist eine Interpretersprache, d.h. in Perl geschriebene Programme, sogenannte *Skripte*, liegen als Klartext vor und werden erst zur Laufzeit in einen Bytecode übersetzt. Der Perl-Interpreter selbst ist in C geschrieben und wurde auf eine Vielzahl von Systemen portiert, darunter die meisten Unix-Derivate wie FreeBSD und Linux, aber auch Microsoft Windows. Eine lauffähige Installation von Perl umfasst neben dem Interpreter mehrere hundert Bibliotheken, von denen die betriebssystemnahen und -spezifischen in C, die übrigen in Perl programmiert sind. Ein Perlskript kann auf unterschiedliche Weise aufgerufen werden. Trivial und nur für sehr wenige Codezeilen geeignet ist der parameterlose Aufruf des Interpreters aus der Shell des Betriebssystems. Das Skript erwartet der Interpreter hierbei auf seiner Standardeingabe, sprich, der Anwender kann es direkt auf der Tastatur eingeben:

```
$ perl
print "hello, world\n";
```

Die Eingabe wird mittels der Tastenkombination `STRG+D` beendet, und die Zeichenkette *hello, world* ausgegeben. Gleichwertig ist die Übergabe des Skriptinhaltes per Kommandozeilenparamter `-e`:

```
$ perl -e 'print "hello, world\n";'
```

In der Regel werden Perlskripte jedoch nicht flüchtig in (Klartext-)Dateien gespeichert, deren Name auf den Zusatz `.pl` enden sollte. Zum Erstellen der Programmtexte wird keine spezielle Entwicklungsumgebung benötigt. Ein einfacher Texteditor wie z.B. *vi* unter Unix oder *Notepad* unter Windows genügen. Für grössere Projekte sollte jedoch auf komfortablere Entwicklungswerkzeuge wie *Eclipse*[13] oder *OpenKomodo*[14] zurückgeriffen werden, die über Hilfsmittel wie Syntaxhighlighting, Code-Faltung, Templates, etc. verfügen. Ein (auf der Festplatte gespeichertes) Skript kann dem Perl-Interpreter als Kommandozeilenparameter zur Ausführung übergeben werden:

```
$ perl hello.pl
```

Eleganter ist es unter unixartigen Betriebssystemen jedoch, die sogenannte *Shebang*-Zeile zu verwenden. Dabei wird die erste Zeile des Perlskripts mit den beiden Zeichen `#!` und der genauen Pfadangabe des Perlinterpreters eingeleitet. Für das Skript `hello.pl` ergibt sich somit:

```
#!/usr/bin/perl

print "hello, world\n";
```

Die Shebang-Zeile dient dem Betriebssystemkernel als Hinweis, mit welchem Programm das Skript auszuführen ist. Der Kernel wird in diesem Fall das Programm `/usr/bin/perl` mit dem Parameter `hello.pl` starten. Ferner leitet die Raute in Perl einen Kommentar ein, so dass die Shebang-Zeile das Skript nicht beeinflusst. Gibt man dem Skript `hello.pl` noch per Aufruf von `chmod a+x hello.pl` Ausführungsrechte, kann es wie ein gewöhnliches Programm gestartet werden:

[13] http://www.eclipse.org
[14] http://www.openkomodo.com

```
$ ./hello.pl
```

5.1.2. Variablen

Perl ist eine schwach typisierte Sprache, die keine explizite Variablendeklaration erfordert und bei der sich der eigentliche Typ einer Variablen erst aus dem Kontext ergibt. Der Name einer Variablen darf sich aus beliebigen alphanumerischen Zeichen inklusive des Unterstrichs zusammensetzen mit der Bedingung, dass das erste Zeichen keine Zahl ist. Zudem wird zwischen Gross- und Kleinschreibung unterschieden (*case-sensitive*), so dass die Variablen `$test` und `$Test` nicht dieselbe Speicherstelle darstellen. Es existieren folgende 5 Typen:

Scalare Ein Scalar ist ein eindimensionaler Datentyp, der

- Zeichenketten nahezu beliebiger Länge (nur begrenzt durch den virtuellen Hauptspeicher)
- Ganzzahlwerte mindestens im Wertebereich des C-Typs *int*, also in der Regel von -2^{31} bis $2^{31} - 1$
- Gleitkommazahlen im Wertebereich des C-Typs *double*, also mit 53 Bit grosser Mantisse und 11 Bit grossem Exponenten

speichern kann. Scalare werden durch ein dem Bezeichner vorangestelltes Dollarzeichen identifiziert, wie z.B. in `$test`.

Arrays Ein Array ist eine Liste oder Stack von Scalaren. Der Zugriff auf einzelne Werte kann über einen impliziten numerischen Index erfolgen, welcher bei 0 beginnt und bis zur Anzahl der Elemente minus 1 läuft. Ein Array wird per vorangestelltem At-Zeichen identifiziert, wie z.B. in `@liste`. Einzelne Elemente, welche ja einen Scalar darstellen, werden jedoch per Dollarzeichen und nachgestellter Elementnummer in eckigen Klammern angesprochen. So stellt `$liste[1]` das zweite Element jenes Arrays dar.

Hashes Ein Hash ist vergleichbar mit einem Array, bei dem die Indizierung jedoch über beliebige Scalare erfolgt. Bildlich gesprochen besteht ein Hash aus einer Anzahl von Schlüssel-/Wertepaaren. Hashes werden per vorangestelltem Prozentzeichen angesprochen, einzelne Elemente (Scalare) jedoch per Dollarzeichen und nachgestelltem Schlüssel in geschweiften Klammern. So spricht `$kunde{"Strasse"}` den Wert an, der im Hash `%kunde` unter dem Schlüssel `Strasse` abgelegt ist.

Filehandles Ein Filehandle stellt einen Input- oder Outputstream dar, der z.B. mit der Funktion `print` beschrieben oder von der Funktion `open` zum Lesen aus einer Datei geöffnet werden kann. Filehandles werden als einzige Ausnahme ohne Prefix angesprochen.

Typeglobs Mit einem Typeglob wird ein Alias für einen anderen Bezeichner oder einen Funktionsnamen in der internen Symboltabelle von Perl angelegt. Ein Typeglob ist somit vergleichbar mit Referenzen in C++ oder Hardlinks in Dateisystemen. Das Prefixzeichnen für Typeglobs ist das Sternchen. In folgendem Beispiel wird für den Scalar `$a` der Typeglob `*b` angelegt. Danach kann `$b` wie `$a` verwendet werden:

```
$a = "hello, world";
*b = *a;
$b = "hallo, welt";
print $a; # Gibt "hallo, welt" aus, weil b ein Typeglob für a ist
```

Zu beachten ist, dass ein Typeglob über den Namen gebildet wird, so dass nach obigem Beispiel auch die Arrays `@a` und `@b`, die Hashes `%a` und `%b`, usw. identisch sind.

Jeder Variablentyp besitzt in Perl seinen eigenen Namensraum, so dass man z.B. einen Scalar `$test`, einen Hash `%test` und eine Funktion (s.u.) namens `test` anlegen kann. Zwecks Verständlichkeit sollte man davon jedoch selten bis gar nicht Gebrauch machen. Wertzuweisungen erfolgen wie in anderen Programmiersprachen per Gleichheitszeichen. Nicht deklarierte oder nicht initialisierte Variablen haben den Metawert `undef`. Boolsche Ausdrücke werden wie in C behandelt. Zu logisch falsch evaluieren:

- Variablen im Zustand `undef`
- ein Scalar mit dem numerischen Wert 0
- ein Scalar, der eine Zeichenkette der Länge 0 darstellt
- ein leeres Array
- ein leerer Hash

Zeichenketten werden in einfachen oder doppelten Anführungsstrichen notiert; innerhalb doppelter Anführungsstriche werden Variablen expandiert:

```
$a = "test";
$b = "this is a $a"; # $a wird expandiert
$c = 'this is a $a'; # $a wird nicht expandiert
```

Der Scalar `$a` enthält die Zeichenkette `test`, `$b` ist `this is a test`, `$c` hingegen `this is a $a`. Möchte man innerhalb doppelter Anführungsstriche reservierte Zeichen wie Variablenprefixe verwenden, müssen diese mit einem vorangestellten Backslash maskiert werden, was gemeinhin *escapen* genannt wird:

```
$d = "this is a \$a"; # $a wird nicht als Variable erkannt
```

Numerische Werte können als ganze Zahl, in Kommaschreibweise oder als Exponentendarstellung zur Basis 10 eingegeben werden:

```
$year = 2007;
$pi = 3.1415;
$million = 1e6;
$zahl = -4.711e-43;
```

Arrays werden als Liste von Scalaren innerhalb runder Klammern zugewiesen:

```
@namen = ("Meier", "Schmidt", "Schuster");
@liste = ("Tisch", 5, 1.5);
@elemente = ();
```

Dem Array `@elemente` wird eine leere Liste zugewiesen. Dies ist ein typisches Konstrukt, um eine nicht initialisierte Variable zu vermeiden. Ein Array kann auch aus einem schon definierten Array Werte erhalten:

```
@angestellte = @namen;
@mitarbeiter = ("Chef1", "Chef2", @namen);
```

Ferner kann man aus einem Array nur bestimmte Elemente in eine neue Liste übernehmen. Dies wird *(Array-)Slice* genannt:

```
@a = @mitarbeiter[1..3];
@b = @mitarbeiter[0,0,4];
@c = @mitarbeiter[99];
```

Das Array `@a` enthält somit den zweiten bis vierten Mitarbeiter, `@b` zweimal den ersten und einmal den fünften Mitarbeiter. Eine offensichtliche Arraygrenzüberschreitung wie in der Zuweisung von Array `@c` erzeugt keinen Fehler oder Warnung, sondern (in diesem Fall) lediglich ein leeres Array. Hashes können wie Arrays als Liste initialisiert werden, bei der alle Elemente an ungeraden Indizes als Schlüssel und alle Elemente an geraden Indizes als Werte betrachtet werden:

```
%gehalt = ("meier", 2200, "schulze", 2500);
```

Verwendet man zwischen den Elementen mit ungeradem und geradem Index statt eines Kommas jeweils den äquivalenten Operator `=>`, wird obiges Beispiel verständlicher:

```
%gehalt = ("meier" => 2200, "schulze" => 2500);
```

Im Hash `%gehalt` wird somit dem Eintrag `meier` der Wert `2200`, dem Eintrag `schulze` der Wert `2500` zugeordnet. Umgekehrt lässt sich auch ein Hash einem Array zuweisen:

```
@liste = %gehalt;
```

Das Array `@liste` enthält hiernach alle Schlüssel-/Wertepaare als flache Liste.
In Perl gibt es ferner die Möglichkeit, von fast allen aufgeführten Datentypen sogenannte *Referenzen* zu bilden. Dies sind Zeiger oder auch Pointer, wie man sie von anderen Sprachen kennt. Eine Referenz ist selbst ein Scalar, der den Typ der referenzierten Variable (Scalar, Array, Hash, usw.) und deren Speicheraddresse beinhaltet. Die Referenz einer Variablen wird mit dem Backslash gebildet:

```
$array_ref = \@array;
```

Der Scalar `$array_ref` ist nun eine Referenz (ein Zeiger) auf das Array `@array`. Der Zugriff auf die sich hinter einer Referenz befindlichen Variablen, sprich die Dereferenzierung, erreicht man, indem man vor die Referenz das Prefix des ursprünglichen Datentyps stellt:

```
@liste = @{$array_ref};
```

Das Array `@liste` ist nun eine inhaltliche Kopie von `@array`. Die Dereferenzierung weist syntaktische und inhaltliche Ähnlichkeit zum Typecasting in C und artverwandten Sprachen auf. So erzeugt das Dereferenzieren von `$array_ref` in einen Hash einen Laufzeitfehler ("Can't coerce array into hash"):

```
%hash = @{$array_ref};
```

Wegen der Operatorpräzedenz kann man hier auf die geschweiften Klammern verzichten:

```
%hash = @$array_ref;
```

Der Zugriff auf ein einzelnes Element des Arrays über seine Referenz geschieht nicht über das At-Zeichen, sondern über Dollarzeichen, da man ja einen Scalar erhalten möchte. Hierbei sind geschweifte Klammern jedoch obligatorisch:

```
$elem = ${$array_ref}[0];
```

Für C-Programmierer vertrauter und eleganter ist der Zugriff über den Operator `->`:

```
$elem = $array_ref->[0];
```

Analog werden Referenzen von Scalaren behandelt:

```
$string = "hello, world\n";
$scalar_ref = \$string;
print ${$scalar_ref}; # Gibt "hello, world" aus
```

Der Umgang mit Hashreferenzen ähnelt dem von Arrayreferenzen. Auch hierbei kann zwischen dem Dereferenzierungsoperator und der Dereferenzierung per geschweifter Klammern gewählt werden:

```
%farben = ("red" => "rot", "blue" => "blau", "green" => "grün");
$farben_ref = \%farben;
print $farben_ref->{"blue"}; # gibt blau aus
print ${$farben_ref}{"red"}; # gibt rot aus
```

Eine Ausnahme bilden lediglich Filehandles. Sie können nicht referenziert werden. Referenzen können jedoch nicht nur von Variablen gebildet werden. Perl gestattet es, Datenstrukturen direkt als Referenz im Speicher anzulegen (ähnlich einer auf dem Stack initialisierten Struktur in C). Man spricht dann von einem *anonymen Array*, *anonymen Scalar*, usw. Von Vorteil ist hierbei, dass man einen (oftmals nicht wieder benötigten) Bezeichner spart. Ausserdem werden mehrdimensionale oder komplexe Datenstrukturen über derartige Referenzen auf anonyme Speicherbereiche gebildet. Ein anonymer Scalar wird erzeugt, indem man vor den eigentlichen Wert den Referenzoperator \ schreibt:

```
$string_ref = \"hello, world";
$integer_ref = \23;
$real_ref = \42.23;
```

Die eigentlichen Werte erhält man nun wieder per Dereferenzierung:

```
print $$string_ref;  # gibt "hello, world" aus
print $$integer_ref; # gibt 23 aus
print $$real_ref;    # gibt 42.23 aus
```

Für anonyme Arrays und Hashes existieren jeweils eigene Operatoren, da es keinen prinzipiellen Unterschied in deren Initialisierung gibt und Perl daher nicht unterscheiden könnte, ob eine anonyme Liste ein Array oder einen Hash darstellt. Die Werte für anonyme Arrays werden in eckige Klammern gefasst:

```
$mitarbeiter_ref = [ "Meier", "Schmidt", "Schulze" ];
print $mitarbeiter_ref->[2]; # gibt "Schulze" aus
```

Anonyme Hashes werden hingegen per geschweifter Klammern deklariert:

```
$farben_ref = {
    "yellow" => "gelb",
    "gray" => "grau",
    "black" => "schwarz"
};
print $farben_ref->{"yellow"}; # gibt "gelb" aus
```

Mehrdimensionale Datentypen können prinzipiell auf zwei Arten gebildet werden. So ist es möglich, als Datentyp erster Dimension ein benanntes Array oder einen Hash zu werden, deren Werte dann jeweils Referenzen auf weitere Arrays oder Hashes sind:

```
@rechteck = ( [0,0], [1,0], [1,1], [0,1] );
```

Das Array `@rechteck` enthält vier Werte, nämlich 4 Referenzen auf je ein anonymes Array, welche wiederum je zwei Scalare enthalten. Einzelne Werte dieses Arrays kann man nun über Dereferenzierung oder über die aus anderen Programmiersprachen bekannte Matrixschreibweise ansprechen:

```
$oben_links_y = $rechteck[3]->[1];
$unten_links_x = $rechteck[0][0];
```

Analog kann man einen Hash mit anonymen Arrays definieren:

```
%rechteck = (
    links_unten  => [0,0],
    rechts_unten => [1,0],
    rechts_oben  => [1,1],
    links_oben   => [0,1]
);
```

Auch hierbei sind wieder beide Zugriffsarten möglich:

```
$oben_rechts_x = $rechteck{"rechts_oben"}[0];
$unten_rechts_y = $rechteck{"rechts_unten"}->[1];
```

Weitaus häufiger wird man jedoch komplexe Strukturen komplett aus anonymen Daten konstruieren, z.B. als Referenz auf einen Hash aus Referenzen auf Hashes:

```
$rechteck = {
    links_unten => {
        x => 0,
        y => 0
    },
    rechts_unten => {
        x => 1,
        y => 0
    },
    rechts_oben => {
        x => 1,
        y => 1
    },
    links_oben => {
        x => 0,
        y => 1
    }
};
```

Der Zugriff sollte hierbei zwecks Lesbarkeit über den Dereferenzierungsoperator erfolgen:

```
$unten_rechts_x = $rechteck->{"rechts_unten"}->{"x"};
```

5.1.3. Gültigkeitsbereich

In Perl müssen Variablen nicht explizit deklariert werden. Sie existieren ab ihrer ersten Verwendung. Ebenso werden nicht mehr referenzierte Speicherbereiche automatisch durch einen *garbage collector* freigegeben. Dennoch sollte ein Programmierer von den Möglichkeiten Gebrauch machen, die Perl zur strukturierten Programmierung bereit hält. Lokale Variablen innerhalb eines Blocks oder einer Funktion (s. Kapitel 5.1.7) werden i.d.R. per `my` erzeugt, so dass sie ausserhalb jenes Blocks und über Funktionsaufrufe hinweg nicht sichtbar ist. Datei-globale Bezeichner können ebenfalls per `my` angelegt werden, werden jedoch meistens mit `our` deklariert. Dies hat vor allem stilistische Gründe: Eine per `our` global deklarierte Variable kann innerhalb eines Blocks oder einer Funktion ebenfalls per `our` deklariert werden und hat dann den Wert der globalen Variablen. Mittels `my` angelegte Veränderliche hingegen haben grundsätzlich einen neu initialisierten Speicherbereich zur Folge: Sie verdecken globale Variablen. Ferner kann auf Variablen, die in einem Modul (s. Kapitel 5.1.8) mit `my` angelegt wurden, nicht von ausserhalb zugegriffen werden.

5.1.4. Operatoren

Da Perl eine schwach typisierte Sprache ist, kommen den Operatoren eine besondere Bedeutung zu. Variabeln werden anhand der Argumenttypen ausgewertet, die ein Operator erwartet. So stellt z.B. der folgende Ausdruck eine korrekte Addition dar:

```
$ergebnis = 10 + 0.5 + "10" + "2.5";
```

Der Scalar `$ergebnis` enthält nun den numerischen Wert `23`. Die in Perl zur Verfügung stehenden Operatoren sind weitestgehend von C übernommen:

Tabelle 2: Ausgewählte Operatoren in Perl

Operator	Beschreibung
`++`	inkrementiert einen Scalar numerisch
`--`	dekrementiert einen Scalar numerisch
`+`, `-`	addiert bzw. subtrahiert zwei Scalare numerisch
`/`	dividiert zwei Scalare, Ergebnis kann ein Integer- oder Real-wert sein
`*`	multipliziert zwei Scalare
`%`	Rest einer Ganzzahldivision
`!`	negiert einen Ausdruck
`**`	exponentiert einen Scalar
`$#`	liefert den Index des letzten Elementes eines Arrays
`x`	wiederholt einen Scalar n-Mal: `"a" x 3` liefert `"aaa"`
`.`	verbindet zwei Scalare als Zeichenkette
`<`*FILEHANDLE*`>`	liest die nächste Zeile von *FILEHANDLE*
`qw(`*$SCALAR1 $SCALAR2 ...*`)`	erzeugt aus den aufgeführten Scalaren ein Array, ohne die Notwendigkeit, Zeichenketten in Hochkommata zu setzen und Kommata zwischen den Scalaren zu verwenden
`<<`, `>>`	bitweises Schieben eines Scalars nach links bzw. rechts
`<`, `>`, `<=`, `>=`	prüft zwei Scalare numerisch auf kleiner, grösser, kleiner-gleich bzw. grösser-gleich
`lt`, `gt`, `le`, `ge`	prüft zwei Scalare textuell auf kleiner, grösser, kleiner-gleich bzw. grösser-gleich
`==`, `!=`	prüft zwei Scalare numerisch auf Gleichheit bzw. Ungleichheit
`<=>`	vergleicht zwei Scalare numerisch: `1 <=> 2` liefert `-1`, `1 <=> 1` liefert `0`, `2 <=> 1` liefert `1`
`eq`, `ne`	vergleicht zwei Scalare textuell
`cmp`	vergleicht zwei Scalare textuell: `"a" cmp "b"` liefert `-1`, `"a" cmp "a"` liefert `0`, `"b" cmp "a"` liefert `1`
`&`, `|`, `^`	bitweise Und-, Oder- bzw. Xor-Verknüpfung zweier Scalare
`&&`, `||`	logische Und- bzw. Oder-Verknüpfung
`=`	Wertzuweisung
`+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `.=`, `&&=`, `||=`	wendet die vor dem Gleichheitszeichen stehende Operation auf die Zielvariable an und weist ihr das Ergebnis zu: `$a *= 3` multipliziert `$a` mit `3`, `$b .= "hallo, welt"` fügt an das Ende von `$b` die Zeichenkette `hallo, welt` hinzu
`=~`, `!~`	prüft einen Scalar gegen ein Textmuster (s. Kapitel 5.1.5): `=~` gibt bei Übereinstimmung `1` bzw. logisch wahr, bei Nichtüber-einstimmung `0` bzw. logisch falsch zurück, `!~` ist die Negation von `=~`

5.1.5. Reguläre Ausdrücke

Ein *Regulärer Ausdruck* (auch *regular expression* (regex)) stellt eine Vorschrift dar, gegen die ein Scalar bzw. eine Zeichenkette geprüft werden kann. Ein Regex ist eine vereinfachte Alternative zu einem lexikalischen Parser. Man kann ihn sich als eine Art Schablone vorstellen, die über einen zu prüfenden Text gelegt wird. Neben dem prinzipiellen Test, ob ein regulärer Ausdruck auf einen Text überhaupt zutrifft, kann Perl bestimmte Teile des

Regex's bzw. des Textes zurückliefern oder auch ersetzen. Derartige Tests werden durch die in Kapitel 5.1.4 aufgeführten Operatoren `=~` bzw. `!~` eingeleitet. Der eigentliche Regex wird per Konvention in Schrägstriche gefasst. Im folgenden Beispiel wird überprüft, ob der Scalar `$name` die Zeichenkette `schulze` enthält:

```
$name =~ /schulze/
```

Möchte man innerhalb des regulären Ausdrucks Schrägstriche verwenden, müssen diese maskiert werden:

```
$preis =~ /Euro\/kg/
```

Alternativ kann der Regex mit einem kleinen `m` (für *match*) eingeleitet werden. Somit können statt der Schrägstriche Zeichen verwendet werden, die nicht im Regex vorkommen:

```
$preis =~ m@Euro/kg@
```

Soll ein Ausdruck unabhängig von Gross- und Kleinschreibung zutreffen (*case-insensitive*), so muss ein kleines `i` angehängt werden:

```
$name =~ /schulze/i
```

Dieser Ausdruck wird wahr, sofern die Variable `$name` eine der Zeichenketten `Schulze`, `schulze`, `scHulzE` o.ä. enthält. Die Sonderzeichen `^` und `$` treffen auf den Anfang bzw. das Ende einer Zeichenkette zu:

```
$name =~ /^Schulze$/
```

Hier müsste `$name` genau die Zeichenkette `Schulze` enthalten. Innerhalb eines regulären Ausdrucks lassen sich bedingt logische Operatoren verwenden:

```
$name =~ /Sch|ber/
```

Dieser Ausdruck trifft auf Namen zu, die `Sch` oder `ber` enthalten, also z.B. Schmidt, Obermann, Schober, etc. Möchte man lediglich an einer Position eine Variation zulassen, so müssen eckige Klammern eingesetzt werden:

```
$artikel =~ /T[ea]ster/
```

Mit diesem Regex werden alle Zeichenketten erkannt, die `Tester` oder `Taster` enthalten. Leitet man eine solche in eckigen Klammern gefasste Zeichenklasse mit dem Negationsoperator `^` ein, so trifft der Ausdruck nur dann zu, wenn an der entsprechenden Stelle die aufgeführten Zeichen nicht stehen:

```
$artikel =~ /T[^ea]ster/
```

Hier treffen alle Zeichenketten wie z.B. `Tqster`, `T8ster`, `Tister` zu, nicht aber `Tester` und `Taster`. Innerhalb einer Zeichenklasse kann zur Vereinfachung auch ein Bereich von Zeichen angegeben werden:

```
$artikel =~ /T[e-o]ster/
```

Dieser Ausdruck wird nur wahr, wenn `$artikel` den String `Tester`, `Tfster`, ... , `Tnster` oder `Toster` enthält. Zusätzlich stehen innerhalb regulärer Ausdrücke die in Tabelle 3 vordefinierten Zeichenklassen zur Verfügung.

Tabelle 3: Vordefinierte Zeichenklassen in regulären Ausdrücken

Zeichenklasse	**Beschreibung**
`.`	trifft auf jedes Zeichen zu
`\d`	trifft auf eine Ziffer zu

Tabelle 3: Vordefinierte Zeichenklassen in regulären Ausdrücken (Forts.)

Zeichenklasse	**Beschreibung**
\D	trifft auf jedes Zeichen ausser einer Ziffer zu
\w	trifft auf alphanumerische Zeichen und den Unterstrich zu
\W	trifft auf jedes Zeichen ausser alphanumerischen und den Unterstrich zu
\s	trifft auf Leerzeichen und Tabulatoren (*Whitespaces*) zu
\S	trifft auf jedes Zeichen ausser Whitespaces zu

Ferner können Quantifizierungsoperatoren eingesetzt werden, mit denen angegeben wird, wie oft ein Zeichen im zu prüfenden Text vorkommen muss. Das Fragezeichen steht hierbei für ein null- oder einmaliges Vorkommen, das Pluszeichen für ein mindestens einmaliges und das Sternchen für ein beliebig häufiges Vorkommen. So ist die folgende Aussage wahr, wenn der Scalar `$name` `"Ana"`, `"Anna"`, `"Annna"` usw. enthält:

```
$name =~ /An+a/
```

Das Fragezeichen steht für null- oder einmaliges Vorkommen:

```
$reise =~ /Schifff?ahrt/
```

Dieser Ausdruck lässt das Wort `Schiffahrt` sowohl in alter als auch in neuer Rechtschreibung gelten. Mithilfe von geschweiften Klammern lassen sich Wiederholungen frei definieren:

```
$einkommen =~ /\d{3}/
```

Der Scalar `$einkommen` muss hier aus mindestens 3 Ziffern bestehen, damit der Ausdruck wahr wird.

```
$preis =~ /\d{3,6}/
$password =~ /\w{4,}/
```

Hier muss `$preis` aus mindestens 3, aber höchstens 6 Ziffern bestehen, während für `$password` 4 oder mehr alphanumerische Zeichen gefordert sind. Reguläre Ausdrücke können nicht zum einfachen Testen eines Scalars dienen, sondern auch, um erkannte Passagen zur weiteren Verarbeitung in Variablen zu speichern. Hierzu werden die gewünschten Passagen in runde Klammern gefasst. Trifft der Regex zu, stehen die erkannten Textstücke in den Variablen `$1`, `$2`, `$3` usw. zur Verfügung:

```
$name =~ /(John) (Smith)/
```

Enthält der Scalar `$name` tatsächlich den Namen `John Smith`, so steht nach dem Regex-Test der Vorname in der Variablen `$1`, der Nachname in `$2` zur Verfügung. In der Regel wird man einen solchen Test mit Zeichenklassen definieren, um z.B. einen Parser für Konfigurationsdateien zu konstruieren:

```
$zeile =~ /^([^\:])+:\s+(.*)$/
```

Dieser Ausdruck trifft auf Zeilen zu, die mit einem oder mehreren Zeichen ausser dem Doppelpunkt beginnen, dann einen Doppelpunkt und mindestens ein Whitespace-Zeichen aufweisen, und mit einer beliebigen Anzahl beliebiger Zeichen enden. Somit werden z.B. `Email: fjo@ogris.de` oder `Option: Wert` erkannt. Anschliessend steht in `$1` der Wert `Email` bzw. `Option` und in `$2` der String `fjo@ogris.de` bzw. `Wert`.
Ferner lassen sich mit regulären Ausdrücken auch Texte ersetzen. Hierzu muss der Regex mit einem kleinen `s` (für *substitute*) eingeleitet werden:

```
$zeile =~ s/alter Text/neuer Text/
```

Dieser Ausdruck ersetzt das erste Vorkommen von `alter Text` im Scalar `$zeile` durch `neuer Text`. Möchte man hingegen jedes Vorkommen von `alter Text` ersetzen, muss man den Ausdruck als *gierig* bzw. *greedy* markieren:

```
$zeile =~ s/alter Text/neuer Text/g
```

Fasst man im Suchmuster einzelne Textpassagen in runde Klammern, so stehen diese im Ersatztext über die Variablen `$1`, `$2` usw. zur Verfügung:

```
$preis =~ s/(\d+) DM/$1 EUR/g
```

Mit diesem Ausdruck werden alle Preisangaben wie `19 DM` in `19 EUR` umgesetzt.

5.1.6. Kontrollstrukturen

Sieht man von einer mehrfachen Fallunterscheidung wie *switch/case* ab, so bietet Perl alle aus C bekannten Kontrollstrukturen. Zusätzlich stehen einige Schlüsselwörte für perltypische Datenstrukturen (Arrays, Hashes) bereit. Tabelle 4 listet die gebräuchlichsten Kontrollstrukturen auf.

Tabelle 4: Ausgewählte Anweisungen in Perl

Anweisung	**Beschreibung**

`for (`*INIT*`;` *CHECK*`;` *LOOP*`)`
`{`
BLOCK
`}`
	führt die durch *BLOCK* gegebenen Anweisungen aus, bis *CHECK* logisch falsch wird; bei Schleifenbeginn wird *INIT* ausgeführt, bei jedem Schleifendurchgang *LOOP*

`foreach` *$SCALAR*`(`*@ARRAY*`)`
`{`
BLOCK
`}`
	führt die durch *BLOCK* gegebenen Anweisungen für jedes Element des Arrays *@ARRAY* aus, das jeweils aktuelle Element steht – falls angegeben – in *$SCALAR* zur Verfügung, sonst in der impliziten Variablen `$_`

`while (`*CHECK*`) {`
BLOCK
`}`
	führt die durch *BLOCK* gegebenen Anweisungen aus, solange *CHECK* logisch wahr ist

`until (`*CHECK*`) {`
BLOCK
`}`
	führt die durch *BLOCK* gegebenen Anweisungen aus, solange *CHECK* logisch falsch ist

`do {`
BLOCK
`} while (`*CHECK*`)`
	führt die durch *BLOCK* gegebenen Anweisungen aus, solange *CHECK* logisch wahr ist

`do {`
BLOCK
`} until (`*CHECK*`)`
	führt die durch *BLOCK* gegebenen Anweisungen aus, solange *CHECK* logisch falsch ist

`while (($key, $value) =`
`each (`*%HASH*`)) {`
BLOCK
`}`
	`each` ist ein zustandsbehafteter Operator, der bei jedem Zugriff das jeweils nächste Schlüssel-/Wertepaar des Hashes liefert; hier werden die durch *BLOCK* gegebenen Anweisungen über alle Elemente von *HASH* iteriert

`if (`*CHECK*`) {`
BLOCK
`}`
	führt die durch *BLOCK* gegebenen Anweisungen aus, falls *CHECK* logisch wahr ist

Tabelle 4: Ausgewählte Anweisungen in Perl (Forts.)

Anweisung	**Beschreibung**

`if (`*CHECK*`) {`
BLOCK1
`} else {`
BLOCK2
`}`
	führt die durch *BLOCK1* gegebenen Anweisungen aus, falls *CHECK* logisch wahr ist, sonst die durch *BLOCK2* dargestellten Anweisungen

`if (`*CHECK1*`) {`
BLOCK1
`} elsif (`*CHECK2*`) {`
BLOCK2
`} ...`
`} elsif (`*CHECKn*`) {`
BLOCKn
`} else {`
BLOCK
`}`
	führt die durch *BLOCK1* gegebenen Anweisungen aus, falls *CHECK1* logisch wahr ist, sonst die durch *BLOCK2* gegebenen Anweisungen aus, falls *CHECK2* logisch wahr ist, ..., sonst die durch *BLOCK* dargestellten Anweisungen

`unless (`*CHECK*`) {`
BLOCK
`}`
	führt die durch *BLOCK* gegebenen Anweisungen aus, falls *CHECK* logisch falsch ist; `unless` stellt die Negation von `if` dar, so dass alle o.g. Darstellungen für `if` auch für `unless` gültig sind

STATM `if (`*CHECK*`)`	führt die durch *STATM* gegebene Anweisung aus, falls *CHECK* logisch wahr ist
STATM `unless (`*CHECK*`)`	führt die durch *STATM* gegebene Anweisung aus, falls *CHECK* logisch falsch ist

Funktionale Blöcke werden wie in C durch geschweifte Klammern gebildet. Ebenso müssen Anweisungen mit einem Semikolon abgeschlossen werden. Im Gegensatz zu anderen Programmiersprachen gibt es keine explizite Hauptroutine wie z.B. `main()` in C oder den Zwang, einen Einsprungspunkt wie in Assembler definieren zu müssen. Stattdessen werden die sich nicht in Funktionen (s. Kapitel 5.1.7) befindlichen Anweisungen sequentiell abgearbeitet (sofern nicht durch Schleifen, Verzweigungen, etc. anders vorgegeben). Ein triviales, dennoch vollständiges Perlskript sieht also wie folgt aus:

```
#!/usr/bin/perl

$ausgabe = "hello, world\n";
print $ausgabe;
```

Den Pfadnamen, unter dem das Skript aufgerufen wurde, hinterlegt der Perlinterpreter in der Variablen `$0`. Kommandozeilenargumente stehen im Array `@ARGV` bereit. Vom Betriebssystem definierte Umgebungsvariablen finden sich im Hash `%ENV`. Standardmässig sind die Filehandles `STDOUT` und `STDERR` zum Schreiben sowie `STDIN` zum Lesen geöffnet.

5.1.7. Funktionen

Eigene Funktionen werden mit dem Schlüsselwort `sub` definiert. Der eigentliche Funktionsblock wird in geschweifte Klammern gefasst. Verschachtelte Funktionen wie in Pascal werden nicht unterstützt. Wie in C werden Funktionsparameter in runden Klammern übergeben. Anzahl und Typ der Funktionsargumente können deklariert werden:

```
sub test ($$)
{
    ...
}

sub test2 (%@)
{
    ...
}
```

Die Funktion `test` erwartet pro forma zwei Scalare (oder Referenzen, die ja ebenfalls Scalare sind). `test2` erwartet einen Hash und ein Array, oder Referenzen auf derartige Datentypen. Die tatsächliche Parameterübergabe findet jedoch immer per Liste statt. Dieses steht jeder Funktion unter dem Array `@_` zur Verfügung. Werden Arrays oder Hashes einer Funktion nicht als Referenzen übergeben, werden diese linearisiert und ihre Elemente in das Array `@_` kopiert. Sofern man also *call-by-value* verwendet, sind innerhalb einer Funktion die übergebenen Werte nicht mehr eindeutig zuordbar. Zudem entscheidet die Art des Funktionsaufrufes, ob ungültige Funktionsparameter zu einem Fehler führen:

```
test "hallo", "welt";   # ok
test "hallo";           # Fehler
test("hallo", "welt");  # ok
test("hallo");          # Fehler
&test("hallo", "welt"); # ok
&test("hallo");         # ok!
```

Daher wird in Perl meist auf eine Parameterdeklaration verzichtet und das Array `@_` als Stack von Funktionsparametern ausgewertet:

```
sub test3 ()
{
    $string1 = $_[0];
    $string2 = $_[1];

    ...
}
```

Oftmals werden Funktionsparameter per Hash übergeben:

```
&drucke_zeile(
    text => "hello, world",
    farbe => "rot"
);
```

Der entsprechende Funktionsrumpf beginnt dann wie folgt:

```
sub drucke_zeile ()
{
    %param = @_;
    $text = $param{"text"};
    $farbe = $param{"farbe"};
}
```

Dies ist möglich, da Hashes und Arrays aufgrund ihres Listencharakters ineinander umgewandelt werden können. Die Vorteile dieser Art der Parameterübergabe sind Flexibilität und Transparenz: Weitere Parameter können durch Erweiterung des übergebenen Hash hinzugefügt werden, und sowohl beim Aufruf als auch in der Funktion ist erkennbar, welche Argumente zu übergeben sind. Rückgabewerte müssen ebenso nicht deklariert werden. Jede Funktion gibt implizit ein Array zurück, dass vom Aufrufer als Scalar, Hash oder einfach als Array interpretiert werden kann. Es ist generell Aufgabe des Programmierers sicherzustellen, dass sowohl aufrufende als auch aufgerufene Funktion die

gleiche Signatur erwarten. Verzichtet man bei einem Funktionsaufruf auf explizite Übergabe jeglicher Parameter, so wird der aufgerufenen Funktion automatisch das Array `@_` der aufrufenden Funktion übergegeben:

```
&test; # bekommt mein @_ übergeben
```

Analog zu Datentypen können auch Referenzen auf Funktionen gebildet werden:

```
sub test ()
{
    ...
}

$test_ref = \&test
```

Der Aufruf muss per Dereferenzierung erfolgen:

```
&{$test_ref}();
$test_ref->();
```

Ebenso ist es möglich, anonyme Funktionen anzulegen:

```
$test_ref = sub {
    ...
};

&{$test_ref}();
```

Der Scalar `$test_ref` ist somit eine Referenz auf eine Funktion und kann dementsprechend eingesetzt werden.
Im Gegensatz zu C bietet Perl eine umfangreiche Anzahl von internen Funktionen, von denen einige in Tabelle 5 aufgeführt sind.

Tabelle 5: Ausgewählte Funktionen in Perl

Funktion	**Beschreibung**
`chr(`*$SCALAR*`)`	lieft das ASCII-Zeichen mit der Nummer *$SCALAR*
`hex(`*$SCALAR*`)`	liefert die hexadezimale Zahl *$SCALAR* als Dezimalzahl
`index(`*$SCALAR1*`, `*$SCALAR2*`, `*$SCALAR3*`)`	sucht *$SCALAR2* in *$SCALAR1* (Vorwärtssuche), optional ab Position *$SCALAR3*, liefert -1, falls *$SCALAR2* nicht in *$SCALAR1* enthalten ist
`rindex(`*$SCALAR1*`, `*$SCALAR2*`, `*$SCALAR3*`)`	sucht *$SCALAR2* in *$SCALAR1* (Rückwärtssuche), optional ab Position *$SCALAR3*, liefert -1, falls *$SCALAR2* nicht in *$SCALAR1* enthalten ist
`lc(`*$SCALAR*`)`	liefert *$SCALAR* in Kleinbuchstaben
`uc(`*$SCALAR*`)`	liefert *$SCALAR* in Grossbuchstaben
`length(`*$SCALAR*`)`	liefert die Länge von *$SCALAR*
`reverse(`*$SCALAR*`)`, `reverse(`*$ARRAY*`)`	liefert *$SCALAR* bzw. *$ARRAY* in umgekehrter Reihenfolge
`substr(`*$SCALAR1*`, `*$SCALAR2*`, `*$SCALAR3*`)`	liefert oder setzt den Teilstring von *$SCALAR1* ab Position *$SCALAR2*, optional mit maximaler Längenangabe *$SCALAR3*; `substr("test", 1)` liefert `est`, `substr("test", 1, 2)` liefert `es`, `substr($name, 0, 3) = "Abc"` ersetzt die ersten drei Zeichen in `$name` durch `Abc`

Tabelle 5: Ausgewählte Funktionen in Perl (Forts.)

Funktion	**Beschreibung**
`split(`*$SCALAR1*`,` *$SCALAR2*`,` *$SCALAR3*`)`	teilt *$SCALAR2* an den durch den Regex in *$SCALAR1* gegebenen Stellen auf (optional limitiert durch die Anzahl in *$SCALAR3*) und liefert ein Array mit den verbleibenden Teilstrings
`pop` *@ARRAY*	entfernt das letzte Element von *@ARRAY* und liefert es zurück
`push` *@ARRAY*`,` *$SCALAR*	fügt *$SCALAR* an das Ende von *@ARRAY* hinzu
`shift` *@ARRAY*	entfernt das erste Element von *@ARRAY* und liefert es zurück; fehlt die Angabe eines Arrays, wird das erste Element vom Array `@_` (s. Kapitel 5.1.7) entfernt
`unshift` *@ARRAY*`,` *$SCALAR*	fügt *$SCALAR* an den Anfang von *@ARRAY* hinzu
`join(`*$SCALAR*`,` *@ARRAY*`)`	liefert *@ARRAY* als Zeichenkette, wobei alle Elemente mit *$SCALAR* verbunden sind
`map {` *BLOCK* `}` *@ARRAY*	führt *BLOCK* für jedes Element von *@ARRAY* aus; innerhalb von *BLOCK* kann auf das aktuelle Arrayelement lesend und schreibend per Variable `$_` zugegriffen werden; Rückgabewert ist das evtl. modifizierte Array
`sort {` *BLOCK* `}` *@ARRAY*	liefert *@ARRAY* textuell sortiert; optional wird für jeden Vergleich (Quicksort-Algorithmus) *BLOCK* ausgeführt, der jeweils zwei zu vergleichende Elemente in den Variablen `$a` und `$b` erhält und -1, 0 oder 1 zurückliefern soll, wenn `$a` kleiner, gleich bzw. grösser `$b` ist
`keys` *%HASH*	liefert alle Schlüssel des Hashes als (unsortiertes) Array
`values` *%HASH*	liefert alle Werte des Hashes als (unsortiertes) Array
`delete` *$HASH*`{`*$ELEM*`}`	löscht das durch den Schlüssel *$ELEM* dargestellte Schlüssel-/Wertepaar aus dem Hash
`exists` *$HASH*`{`*$ELEM*`}`	liefert logisch wahr, falls der Schlüssel *$ELEM* im Hash *%HASH* existiert
`binmode(`*FILEHANDLE*`)`	zeigt an, dass die durch *FILEHANDLE* dargestellte Datei Binärdaten enthält
`close(`*FILEHANDLE*`)`	schliesst die durch *FILEHANDLE* dargestellte Datei
`flock(`*FILEHANDLE*`,` *$SCALAR*`)`	sperrt die durch *FILEHANDLE* dargestellte Datei • exklusiv, wenn *$SCALAR* 2 ist (pro Datei darf nur ein Prozess gleichzeitig eine exklusive Sperre halten) (Write-Lock) • zum Schreiben, wenn *$SCALAR* 1 ist (pro Datei dürfen unendlich viele Prozesse gleichzeitig ein solches *Read-Lock* halten, solange keine exklusive Sperre vorliegt); hebt eine Dateisperre auf, wenn *$SCALAR* den Wert 8 hat

Tabelle 5: Ausgewählte Funktionen in Perl (Forts.)

Funktion	**Beschreibung**
`die(`*$SCALAR*`)`	beendet die Ausführung des Skripts mit einem Fehler, der optional per *$SCALAR* erläutert werden kann
`print(`*@ARRAY*`)`	gibt alle Elemente von *@ARRAY* aus
`open(`*FILEHANDLE*`,` *$SCALAR*`)`	öffnet die durch *$SCALAR* angegebene Datei zum Lesen
`open(`*FILEHANDLE*`, ">",` *$SCALAR*`)`	öffnet die durch *$SCALAR* angegebene Datei zum Beschreiben; alternativ kann die Syntax `open(FILEHANDLE, ">DATEINAME")` verwendet werden
`caller($SCALAR)`	liefert Informationen über die aufrufende Funktion, optional über die aufrufende Funktion, falls *$SCALAR* 1 ist, über die Grosselternfunktion, falls *$SCALAR* 2 ist, usw.
`eval(`*BLOCK*`)`	führt die durch *BLOCK* gegebenen Anweisungen aus, bricht jedoch bei Fehlern nicht das Skript ab, sondern stellt die Fehlerbeschreibung in der Variablen `$@` bereit
`exit(`*$SCALAR*`)`	beendet das Skript mit dem optionalen, durch *$SCALAR* gegebenen numerischen Fehlercode
`next`	springt innerhalb einer Schleife (`for`, `while`, usw.) zum nächsten Durchgang
`last`	verlässt eine Schleife (wie `break` in C)
`system(`*$SCALAR*`)`	führt den durch *$SCALAR* gegebenen Systembefehl aus und liefert dessen Ausgabe
`time()`	liefert die Sekunden seit dem 1. Januar 1970 00:00 Uhr (der sogenannten *Epoch*)
`localtime(`*$SCALAR*`)`	liefert den übergebenen Epoch-Zeitwert als lokale Zeit in Form eines Arrays, das Sekunde, Minute, Stunde, Tag, Monat, Jahr, Wochentag, Jahrestag und ein Flag für die Sommerzeit darstellt
`-r` *$SCALAR*	liefert logisch wahr, falls die Datei *$SCALAR* lesbar ist
`-w` *$SCALAR*	liefert logisch wahr, falls die Datei *$SCALAR* schreibbar ist
`-x` *$SCALAR*	liefert logisch wahr, falls die Datei *$SCALAR* ausführbar ist
`-e` *$SCALAR*	liefert logisch wahr, falls die Datei *$SCALAR* existiert
`-z` *$SCALAR*	liefert logisch wahr, falls die Datei *$SCALAR* 0 Byte gross ist
`-s` *$SCALAR*	liefert logisch wahr, falls die Datei *$SCALAR* nicht 0 Byte gross ist
`-d` *$SCALAR*	liefert logisch wahr, falls *$SCALAR* ein Verzeichnis ist
`-f` *$SCALAR*	liefert logisch wahr, falls *$SCALAR* eine Datei ist

5.1.8. Module und Packages

Perlmodule stellen Funktionsbibliotheken dar. Sie werden wie Perlskripte in Textdateien gepspeichert, jedoch mit der Dateiendung `.pm`. Ein Modul enthält i.d.R. keinen Programmcode auf der Hauptebene, sondern lediglich Funktionen und ggf. globale Variablen. Der Name eines Moduls und somit der Namensraum wird über das Schlüsselwort `package` festgelegt.

```
#!/usr/bin/perl

package Fahrzeug;
```

Die Shebangzeile am Anfang mag unnütz erscheinen, jedoch erkennt man so unmittelbar, dass es sich um eine Perldatei handelt. Zudem kann man versehentlich ein Modul aufrufen (z.B. weil man es für ein Skript hält). Bei fehlender Shebangzeile jedoch (und da es sich um keine binäre, vom Kernel als ausführbares Programm erkannte Datei handelt), wird die Shell (s. Kapitel 5.2) versuchen, die enthaltenen Befehle zu interpretieren, was meistens nicht erwünscht ist und zudem zu Nebeneffekten führen kann. Wie in Java kann eine Namensraumhierarchie aufgebaut werden, um Module namentlich zu bündeln:

```
#!/usr/bin/perl

package Fahrzeug::Auto;
```

Module müssen als letzte Anweisung einen logisch wahren Wert liefern (z.B. "1"), um ihr fehlerfreies Einbinden in das aufrufende Skript zu signalisieren. Würde im Modul ein Fehler festgestellt werden, bricht die Importfunktion ab und beendet das Skript. Ein Modul hat daher immer den folgenden Aufbau:

```
#!/usr/bin/perl

package Fahrzeug::Auto;

1;
```

Es ist möglich, in einer Moduldatei mehrere solcher Packages zu definieren. Dieses sollte man jedoch vermeiden, da das Schlüsselwort `use` zum Einbinden eines Moduls nicht den bei `package` angegebenen Namen anspricht, sondern den Dateipfad. Im folgenden Beispiel wird Perl versuchen, eine Datei `Auto.pm` im Verzeichnis `Fahrzeug` einzubinden:

```
#!/usr/bin/perl

use Fahrzeug::Auto;

...
```

Perl sucht per default Module in allen Verzeichnissen, die im Array `@INC` angegeben sind. Hierzu zählen u.a. das aktuelle Verzeichnis und unter unixartigen Betriebssystemen die Verzeichnisse `/usr/lib/perl` und `/usr/local/lib/perl`. Daher muss `Auto.pm` im Unterverzeichnis `Fahrzeug` des aktuellen Verzeichnisses liegen, oder unterhalb eines anderen, in `@INC` aufgeführten Verzeichnisses. Ist dies nicht möglich, und liegt das Modul `Auto.pm` z.B. im Verzeichniss `/home/felix/lib`, so muss dieses dem Array `@INC` hinzugefügt werden. Ein simples `push @INC, "/home/felix/lib"` oder auch ein `unshift @INC, "/home/felix/lib"` führen nicht zum Erfolg, da zu deren Zeitpunkt das Skript schon in Bytecode kompiliert wurde und keine fehlenden Module aufweisen darf. Daher wird das perleigene Modul `lib` verwendet, dem zusätzliche, von `@INC` abweichende Pfade beim Einbinden als Liste übergeben werden:

```
#!/usr/bin/perl
```

```
use lib "/home/felix/lib";
use Fahrzeug::Auto;

...
```

Somit kann die Datei `Auto.pm` im Verzeichnis `/home/felix/lib/Fahrzeug` gefunden und eingebunden werden. Wegen dieser Namensabhängigkeiten zwischen Dateipfad, Modul und Package sollte ein Modul immer nur ein Package enthalten, dessen Name gleich dem Modelnamen ist. So sollte in der Datei bzw. im Modul `Fahrzeug/Auto.pm` nur das Package `Fahrzeug::Auto` vorliegen. Der Zugriff auf Funktionen und Variablen in Modulen erfolgt über den Namen ihres Packages:

```
use Fahrzeug::Auto;  # Modul einbinden

$Fahrzeug::Auto::meldung = "hello, world";
&Fahrzeug::Auto::ausgabe();
```

5.1.9. Objektorientiertes Programmieren

Die in Kapitel 5.1.8 dargestellten Module dienen in Perl als Grundlage für objektorientiertes Programmieren. Jedes *Package* kann eine Klasse darstellen, sofern es einen Konstruktor besitzt und seine Funktionen als Memberfunktionen programmiert sind. Im Gegensatz zu anderen Programmiersprachen ist der Name des Konstruktors frei wählbar. Üblich ist jedoch `new`:

```
use Fahrzeug::Auto;

$auto = Fahrzeug::Auto->new(Farbe => "rot", PS => 150);
```

Hier wird zunächst das Modul `Fahrzeug::Auto` bzw. die Datei `Fahrzeug/Auto.pm`, welche in einem in `@INC` aufgeführten Verzeichnis liegen muss, eingebunden. Dieses Modul enthält das Package `Fahrzeug::Auto`, in dem wiederum die Funktion `new` definiert ist. Diese Funktion dient als Konstruktor. Ihr Rückgabewert ist ein Objekt vom Typ `Fahrzeug::Auto`. Intern wandelt Perl den Aufruf von `new` in folgende Anweisung um:

```
$auto = Fahrzeug::Auto::new(
    "Fahrzeug::Auto",
    Farbe => "rot",
    PS => 150
);
```

Jeder Konstruktor bekommt den Klassennamen als ersten Parameter übergeben. Das zurückgelieferte, neue Objekt ist eine Hashreferenz, die durch die Funktion `bless` als eine Instanz von `Fahrzeug::Auto` deklariert wurde:

```
package Fahrzeug::Auto;

sub new ()
{
    my $Klasse = shift;

    my %Object = ();

    my $Object_ref = bless(\%Object, $Klasse);

    return $Object_ref;
}
```

Der Konstruktor `new` im Package `Fahrzeug::Auto` übernimmt zunächst per Aufruf von `shift`, welches ohne Parameter auf das Array `@_` wirkt, den Namen der Klasse. Anschliessend wird ein leerer Hash namens `%Object` angelegt, der durch den Aufruf von `bless` als Instanz von `Fahrzeug::Auto` markiert wird. Die Funktion `bless` (engl. für *segnen*, man erkennt den subtilen Humor im Sprachdesign von Perl) erwartet zwei Parameter, nämlich eine Referenz auf die zu segnende bzw. zu klassifizierende Variable und den Namen der Klasse. Der Rückgabewert von `bless` und des Konstruktors ist die klassifizierte Referenz. Membervariablen wie z.B. `Farbe` oder `PS` werden i.d.R. mittels eines temporären Hashes initialisiert:

```
package Fahrzeug::Auto;

sub new ()
{
    my $Klasse = shift;

    my %Object = ();

    my %Param = @_;
    foreach my $Variable("Farbe", "PS") {
        $Object{$Variable} = $Param{$Variable};
    }

    my $Object_ref = bless(\%Object, $Klasse);

    return $Object_ref;
}
```

Dies ist möglich, da das Array `@_` nach dem Aufruf von `shift` nur noch Schlüssel-/Wertepaare wie `Farbe => rot` enthält. Durch die mit `foreach` gebildete Schleife werden nur gewünschte Variablen in das Objekt übernommen. Memberfunktionen werden analog zum Konstruktor über das Objekt aufgerufen und erhalten dieses als ersten Parameter:

```
$auto->lackieren(Farbe => "blau");
```

Da `$auto` eine klassifizierte Variable ist, wandelt Perl den Aufruf um:

```
Fahrzeug::Auto::lackieren($auto, Farbe => "blau");
```

Daher sollte die Funktion `lackieren` (ein typischer *Setter*) wie folgt aussehen:

```
sub lackieren ()
{
    my $Object_ref = shift;

    my %Param = @_;
    foreach my $Variable("Farbe") {
        $Object_ref->{$Variable} = $Param{$Variable};
    }
}
```

Der Destruktor einer Klasse muss zwingend den Namen `DESTROY` haben, da er vom Garbagecollector aufgerufen wird, sobald das Objekt nicht mehr verwendet wird. Der genaue Zeitpunkt, zu dem ein Destruktor aufgerufen wird, ist nicht vorhersagbar. Daher sollte er keine laufzeitkritischen Aufgaben erfüllen. Sieht man von diesen Besonderheiten ab, stellt sich ein Konstruktor wie eine normale Klassenfunktion dar, die bis auf die Objektreferenz parameterlos aufgerufen wird:

```
sub DESTROY ()
{
    my $Object_ref = shift;
```

```
    # aufräumen ....
}
```

Im Gegensatz zu anderen objektorientierten Sprachen kennt Perl keinen strengen Vererbungsmechanismus. Stattdessen teilt man dem Interpreter mit, in welchen Packages zu suchen ist, falls eine Methode nicht im aktuellen Paket definiert ist. Hierzu hinterlegt man im globalen Array `@ISA` (lies: *is a* $\Rightarrow$ *ist ein*) die Namen aller Superklassen:

```
package Fahrzeug::Auto;

our @ISA = ("Fahrzeug");
```

Ruft man nun eine nicht im Package `Fahrzeug::Auto` definierte Methode auf, so versucht Perl, sie im Package `Fahrzeug` zu finden. Natürlich kann das Array `@ISA` mehrere Superklassen enthalten, wodurch man Mehrfachvererbung implementiert. Hierbei wird der Reihe nach in jedem aufgeführten Package die gewünschte Funktion gesucht, bis diese gefunden ist. Um auch die Membervariablen der Superklasse zu erhalten, sollte der Konstruktor von `Fahrzeug::Auto` angepasst werden. Hierzu wird die Pseudoklasse `SUPER` verwendet, über die der Konstruktor der Superklasse angesprochen wird:

```
package Fahrzeug::Auto;

our @ISA = ("Fahrzeug");

sub new ()
{
    my $Klasse = shift;

    my %Param = @_;

    my $Object_ref = SUPER->new(%Param);

    foreach my $Variable("Farbe", "PS") {
        $Object_ref->{$Variable} = $Param{$Variable};
    }

    $Object_ref = bless($Object_ref, $Klasse);

    return $Object_ref;
}
```

Dem Konstruktor von `Fahrzeug` werden alle Parameter übergeben, damit dieser die Membervariablen der Instanz von `Fahrzeug` initialisieren kann. Der Aufruf von `bless` im oben gezeigten Konstruktor von `Fahrzeug::Auto` ist notwendig, damit `$Object_ref` nicht als Typ `Fahrzeug`, sondern `Fahrzeug::Auto` klassifiziert wird.

5.1.10. Pragmatisches Perl

Mit den Modulen `strict` und `warnings` lässt sich ein sauberer Programmierstil erzwingen. Verwendet man `warnings`, so muss jeder Variablen vor ihrem ersten lesenden Zugriff ein Wert zugewiesen sein. Ausserdem müssen interne Perlfunktionen wie z.B. `time()` oder `split()` ihre Rückgabewerte an eine Variable liefern. So beinhaltet folgendes Beispiel gleich drei Fehler:

```
#!/usr/bin/perl

use warnings;
```

```
localtime($jetzt);
```

Erstens wird `localtime()` ohne Rückgabe an eine Variable verwendet. Zweitens wurde `$jetzt` kein Wert zugewiesen. Drittens wird `$jetzt` nur ein einziges Mal verwendet. Die durch das Modul `warnings` erkannten Fehler produzieren nur Warnungen auf der Standardfehlerausgabe. Verwendet man ein derart fehlerhaftes Perlskript jedoch als CGI-Programm oder als per `mod_perl` aufgerufenes Modul in einem Webserver, so werden diese Warnungen an den Browser des Besuchers geschickt und können u.U. die Ausgabe der Webseite beeinflussen. Setzt man hingegen `strict` ein, so führen die durch dieses Modul erkannten Fehler zum Abbruch des Skriptes. Hierzu zählen globale Variablen, die nicht per `my` oder `our` deklariert wurden, und die Zuweisung von Zeichenketten, die weder in Anführungsstrichen notiert sind noch Funktionsnamen darstellen:

```
#!/usr/bin/perl

use strict;

sub HelloUniverse ()
{
    return "hello, universe";
}

$jetzt = time();

my $var1 = HelloWorld;
my $var2 = HelloUniverse;
```

Das gezeigte Skript erzeugt zwei Fehler: Zum einen wird der Scalar `$jetzt` nicht deklariert. Zum anderen stellt `HelloWorld` keinen Funktionsnamen dar. Die Variable `$var2` hingegen enthält die Zeichenkette `hello, universe`, da `HelloUniverse` eine gültige Funktion darstellt und eben jenen String liefert. Für Perlskripte im produktiven Einsatz sollten immer beide Module `strict` und `warnings` eingebunden werden, um unsauberen Programmierstil und somit "versteckte"Fehler von vornherein zu unterbinden.

5.1.11. Plain Old Documentation

Mit dem *Plain Old Documentation* (POD)-Format werden speziell gekennzeichnete Abschnitte in Perl-Skripten und -Modulen als Dokumentationstext interpretiert. Mit externen Programmen wie `pod2html` oder `pod2text` werden aus jenen Abschnitten Quellcodedokumentationen als HTML-Seiten, Textdateien, o.ä. ausgegeben[15]. POD-Schlüsselwörter müssen mit einem Gleichheitszeichen und am Anfang einer Zeile beginnen. Zum Abschluss einer POD-Anweisung muss eine Leerzeile folgen. Ein Dokumentationsabschnitt wird explizit mit `=pod` oder implizit mit jedem POD-Schlüsselwort eingeleitet. Zum Verlassen eines POD-Blocks muss `=cut` verwendet werden. Mit `=head1` bis `=head4` stehen unterschiedlich markante Formatierungsanweisungen für Kopfzeilen bereit. Ein einfaches Beispiel sieht wie folgt aus:

```
#!/usr/bin/perl

=head1 helloworld.pl

Dieses Programm gibt "hello, world" aus.

=cut

print "hello, world\n";
```

[15] Vgl. ähnliche Systeme wie z.B. javadoc

Aufzählungen werden per `=over` und `=back` begonnen bzw. beendet. Die einzelnen Punkte einer Liste werden mit `=item` angeführt. Zusätzlich stehen folgende Formatierungsanweisungen zur Verfügung:

`I<TEXT>` druckt *TEXT* kursiv

`B<TEXT>` druckt *TEXT* fett

`C<TEXT>` druckt *TEXT* in einer Proportionalschriftart

`S<TEXT>` verhindet, dass *TEXT* umgebrochen wird

`F<DATEINAME>` zur einheitlichen Darstellung von Dateinamen

`L<LINK>` falls *LINK* eine URL darstellt wie z.B. `http://www.fh-bielefeld.de/`, wird ein entsprechender externer Link erzeugt; ansonsten wird ein Link zu der angegebenen lokalen Unix-Manpage erzeugt

Mit `=begin` und `=end` können Abschnitte definitert werden, die nur eine Klasse von POD-Parsern auswertet. So wird folgendes Beispiel nur von `pod2text` dargestellt:

```
=begin text

Dies erscheint nur in einer Textdatei.

=end text
```

Abbildung 9 zeigt die per `pod2html` erzeugte HTML-Dokumentation in einem Webbrowser, die aus folgendem Perl-Skript erzeugt wurde:

```
#!/usr/bin/perl

=head1 helloworld.pl

=head2 Aufruf

F</helloworld.pl>

=head2 Eingabe

I<keine>

=head2 Ausgabe

I<hello, world>

=head2 Bugs

=over

=item Perl wird benötigt.

=item Nicht lauffähig unter DOS.

=back

=head2 Literatur

L<http://de.wikipedia.org/wiki/Hallo-Welt-Programm>

=begin text
```

```
Benutze pod2text, damit dieser Abschnitt in der Dokumentation erscheint.

=end text

=cut

print "hello, world\n";
```

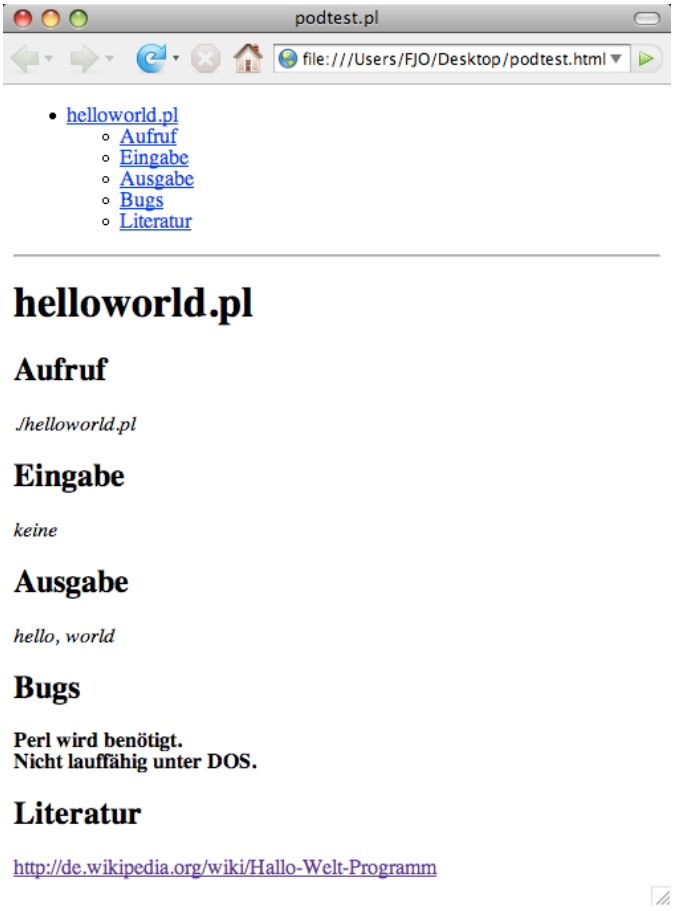

Abbildung 9: Per pod2html erzeugte HTML-Dokumentation des Quellcodes

5.2. Shellscripting

Die Shell ist nicht nur die primäre Schnittstelle zwischen Mensch und (unixartigem) Betriebssystem, sondern auch eine einfache Programmiersprache, die je nach Shell mehr oder weniger weit über herkömmliche Stapelverarbeitung hinaus geht. Auf einem FreeBSD-System sind standardmässig die *tcsh*, eine leicht an die Syntax von C angelehnte Shell, und die *sh* installiert. Letztere wird als Interpreter in Systemskripten verwendet, die beim Starten und Stoppen des Betriebssystems wichtige Aufgaben übernehmen, z.B. das Konfigurieren von Netzwerkschnittstellen oder das kontrollierte Herunterfahren von Serverprozessen. Shellskripte und in der Shell (als interaktives Programm) eingegebene Befehle sind äquivalent. Beide stellen Befehlsfolgen dar, die interpretiert und ggf. ausgeführt werden, z.B. wenn es sich um den Aufruf eines (externen) Programmes handelt. Da Perl einige Konzepte aus der Shellprogrammierung übernommen hat, erscheinen di-

verse Elemente vertraut. Wie jedes andere nicht-binäre Programm unter Unix sollte auch ein Shellskript mit einer Shebang-Zeile eingeleitet werden:

```
#!/bin/sh

# Dies ist ein Kommentar
```

Kommentare werden wie gezeigt per `#` eingeleitet. Analog zu Perl existiert keine Hauptroutine. Ansonsten müsste man diese auch im interaktiven Betrieb definieren. Befehle werden entweder per Semikolon oder per Zeilenvorschub voneinander getrennt. Die Shell `/bin/sh` unterstützt lediglich einen einfachen Datentyp, der Zeichenketten oder numerische Werte aufnehmen kann (vgl. Scalare in Perl). Sie müssen nicht deklariert werden und sind ab ihrer ersten Verwendung global definiert. Ausnahme bilden per `local` markierte Bezeichner innerhalb von Funktionen. Variablen, denen ein Wert zugewiesen wird, werden ohne Prefix notiert. Soll der Wert abgerufen werden, so muss ein Dollarzeichen vorangestellt werden:

```
#!/bin/sh

meldung="hello, world"
echo $meldung
```

Zusätzlich kann der Variablenname in geschweifte Klammern gefasst werden. Dies ist sinnvoll, wenn Zeichen folgen, die auch als Teil des Variablennamens interpretiert werden könnten:

```
#!/bin/sh

meldung="This is just a test"
echo ${meldung}script
```

Innerhalb einfacher Anführungsstriche findet keine Variablenexpansion statt. So gibt folgendes einfach nur `$meldung` aus (statt `hello, world`):

```
#!/bin/sh

meldung="hello, world"
echo '$meldung'
```

Der Backslash dient zum Maskieren von Metazeichen:

```
#!/bin/sh

meldung="Perl is \"funny\""
echo $meldung
```

Dieses Beispiel gibt wie erwartet `Perl is "funny"` aus. Vorbelegte Variablen sind `$0`, `$1`, `$2`, usw. In `$0` ist der Name des Skriptes hinterlegt, in den Variablen ab `$1` etwaige Kommandozeilenparameter bzw. innerhalb einer Funktion die ihr übergebenen Werte. Per Aufruf von `unset meldung` (ohne Dollarzeichen vor dem Namen der Variablen) wird die Variable in den nicht initialisierten Zustand zurückversetzt. Zusätzlich existieren die in Tabelle 6 aufgeführten Arten von Wertezuweisungen.

Tabelle 6: Bedingte Wertezuweisungen in der Shellprogrammierung

Zuweisung	Beschreibung
`ziel=$quelle:-$default`	Weist der Variablen `$ziel` den Wert von `$quelle` zu, falls `quelle` nicht leer ist und nicht die leere Zeichenkette darstellt; sonst wird der Wert von `$default` zugewiesen
`ziel=$quelle:=$default`	Weist der Variablen `$ziel` den Wert von `$quelle` zu, falls `quelle` nicht leer ist und nicht die leere Zeichenkette darstellt; sonst wird der Wert von `$default` zunächst `$quelle` zugewiesen und dieser schliesslich an `$ziel` geliefert
`ziel=$quelle:?$fehler`	Weist der Variablen `$ziel` den Wert von `$quelle` zu, falls `quelle` nicht leer ist und nicht die leere Zeichenkette darstellt; sonst bricht die Ausführung des Skriptes ab, optional mit dem in `$fehler` hinterlegten Text
`ziel=$quelle:+$default`	Gegenteil zu `ziel=$quelle:-$default`: weist der Variablen `$ziel` den Wert von `$quelle` zu, falls `quelle` leer ist oder die leere Zeichenkette darstellt; ansonsten (wenn `$quelle` also einen String länger 0 Buchstaben enthält) wird der Wert von `$default` zugewiesen

Die Syntax von Kontrollstrukturen ist entfernt an Pascal angelehnt:

```
#!/bin/sh

name="Hans"

if [ $name = "Peter" ]; then
    echo "Hallo Peter"
elif [ $name = "Hans"
    echo "Hallo Hans"
else
    echo "Hallo Unbekannter"
fi
```

Dieses Konstrukt prüft, ob der Name `Peter` oder `Hans` lautet und gibt bei keiner Übereinstimmung `Hallo Unbekannter` aus. Wesentlich eleganter ist eine Fallunterscheidung. Auch hierbei wird das einleitende Schlüsselwort `case` in umgekehrter Schreibweise (`esac`) als Abschluss des Befehls erwartet:

```
#!/bin/sh

name="Hans"

case $name in
    Peter)
        echo "Hallo Peter"
        ;;
    Hans)
        echo "Hallo Hans"
        ;;
    *)
        echo "Hallo Unbekannter"
```

```
        ;;
esac
```

Schleifen werden per `while` oder `for` gebildet. Letztere ähnelt jedoch einer mit `foreach` programmierten Schleife in Perl:

```
#!/bin/sh

for i in "Hans" "Peter" "Felix"; do
    echo "Hallo $i"
done
```

Der Befehl `for` iteriert über jedes Element der Liste, auf die er angewendet wird, so dass obiges Beispiel der Reihe nach `Hallo Hans`, `Hallo Peter` und `Hallo Felix` ausgibt. Das folgende Beispiel gibt die Zahlen von 0 bis 9 aus. Der hierzu notwendige arithmetische Ausdruck wird per zweifacher runder Klammern gebildet:

```
#!/bin/sh

i=0

while [ $i -lt 10 ]; do
    echo "$i"
    i=$((i+1))
done
```

Die Befehle `if` und `while` werten generell die Rückgabe eines Programmes aus. Beendet sich ein Programm mit dem Status `0`, wird dies als logisch wahr gewertet, jeder andere Rückgabewert als falsch. Tatsächlich war der Operator `[` in alten Unixversionen ein externes Programm. Die Shell in aktuellen FreeBSD-Versionen verfügt hingegen über diesen Operator und muss für Vergleiche wie oben gezeigt kein externes Programm aufrufen. Aus Kompatibilitätsgründen liegt im Verzeichnis `/bin` ein Programm namens `[`, welches identisch mit dem ebenfalls dort befindlichen Programm `test` ist. Die von diesen Programmen bzw. Operatoren angebotenen Tests zeigt Tabelle 7. Sie ähneln den Funktionen `-r` bis `-f` in Perl (s. Tabelle 5).

Tabelle 7: Ausgewählte Tests des Shelloperators `[` bzw. des Programmes `test`

Test	**Beschreibung**
`-d` *VERZEICHNIS*	*VERZEICHNIS* existiert und ist ein Verzeichnis
`-e` *DATEI*	*DATEI* existiert
`-f` *DATEI*	*DATEI* existiert und ist eine Datei
`-r` *DATEI*	*DATEI* existiert und ist lesbar
`-s` *DATEI*	*DATEI* existiert und ist grösser als 0 Byte
`-w` *DATEI*	*DATEI* existiert und ist schreibar
`-x` *DATEI*	*DATEI* existiert und ist ausführbar
`-n` *STRING*	die Zeichenkette in *STRING* hat nicht die Länge 0
`-z` *STRING*	die Zeichenkette in *STRING* hat die Länge 0
STRING1 `=` *STRING2*	die Zeichenketten *STRING1* und *STRING2* sind identisch
STRING1 `!=` *STRING2*	die Zeichenketten *STRING1* und *STRING2* sind nicht identisch
INTEGER1 `-eq` *INTEGER2*	die Zahlen *INTEGER1* und *INTEGER2* sind identisch

Tabelle 7: Ausgewählte Tests des Shelloperators `[` bzw. des Programmes `test` (Forts.)

Test	**Beschreibung**
INTEGER1 `-ne` *INTEGER2*	die Zahlen *INTEGER1* und *INTEGER2* sind nicht identisch
INTEGER1 `-gt` *INTEGER2*	die Zahl *INTEGER1* ist grösser *INTEGER2*
INTEGER1 `-ge` *INTEGER2*	die Zahl *INTEGER1* ist grösser oder gleich *INTEGER2*
INTEGER1 `-lt` *INTEGER2*	die Zahl *INTEGER1* ist kleiner *INTEGER2*
INTEGER1 `-le` *INTEGER2*	die Zahl *INTEGER1* ist kleiner oder gleich *INTEGER2*
AUSDRUCK1 `-a` *AUSDRUCK2*	beide Ausdrücke sind wahr, z.B. `[ $vorname = "Hans" -a $nachname = "Meier" ]`
AUSDRUCK1 `-o` *AUSDRUCK2*	einer der Ausdrücke ist wahr, z.B. `[ $vorname = "Hans" -o $vorname = "Peter" ]`

Die Funktion `eval` fungiert als Interpreter im Interpreter. Sie wertet den ihr übergebenen String als Shellskript aus. Dies wird häufig zur indirekten Adressierung und aufgrund fehlender komplexer Datenstrukturen verwendet:

```
#!/bin/sh

$benutzer="1"

$User_0_Name="Hans Meier"
$User_1_Name="Peter Maier"

eval "ausgabe=\$User_${benutzer}_Name"

echo $ausgabe
```

Dieses Skript gibt `Peter Maier` aus. Beim Aufruf von `eval` wird die Variable `$benutzer` expandiert, so dass `eval` die Zeichenkette `"ausgabe=$User_1_Name"` erhält. Das Dollarzeichen nach dem Gleichheitszeichen wird nicht als Variablenprefix gewertet, da es maskiert ist. Die Variable `$User_1_Name` wurde zuvor mit dem Wert `Peter Maier` belegt, so dass `eval` diesen String in `ausgabe` kopiert. Würde man die Variable `$benutzer` mit `0` initialisieren, so gäbe das obige Beispiel den Namen `Hans Meier` aus, bei allen anderen Werte eine leere Zeichenkette. Funktionen werden relativ einfach definiert und aufgerufen:

```
#!/bin/sh

ausgabe ()
{
    echo "hello, world"
}

ausgabe
```

Parameter werden in den Variablen `$1`, `$2`, usw. übergeben:

```
#!/bin/sh

print_name ()
{
    echo "Vorname:  $1"
    echo "Nachname: $2"
```

```
    return 0
}

print_name "Hans" "Meier"
```

Rückgabewerte von Funktionen sind auf einen numerischen Statuscode von 0 bis 255 beschränkt, der in der aufrufenden Funktion in der Variablen `$?` abgefragt werden kann. Ein Shellskript kann weitere Shellskripte einbinden und auf deren Variablen und Funktionen zugreifen:

```
#!/bin/sh

. /usr/local/etc/functions.sh
```

Das Beispiel bindet (engl. *to source*) die Datei `/usr/local/etc/functions.sh` lexikalisch an der angegebenen Stelle ein. Etwaiger Code auf der Hauptebene der eingebundenen Datei wird unmittelbar ausgeführt.

6. Auszeichnungssprachen

6.1. XML

Die *Extensible Markup Language* (XML) ist eine Metasprache. Mit ihrer Hilfe können eigene Formate zum Transport von Daten definiert werden. XML verhält sich zu einem selbst definierten Transportformat nahezu wie die *Backus-Naur-Form* (BNF) zu einer in BNF spezifizierten Programmiersprache. In der Praxis wird der Begriff *XML-Dokument* verwendet, wenn man Daten diskutiert, die in einem XML-Format transportiert werden. Mit XML können ausschliesslich hierarchische Formate definiert werden. Sie müssen genau ein Wurzelelement (*root node*) besitzen, welches beliebig viele und beliebig tief verschachtelte Kindelemente beinhalten darf. Elemente bestehen aus einem öffnenden und einem schliessenden *Tag*, die Daten und/oder weitere (Kind-)Elemente umfassen. Tags werden in spitze Klammern gefasst:

```
<buch>
```

Das gezeigte Tag öffnet das Element `buch`. Das zugehörige schliessende Tag wird per Schrägstrich gebildet:

```
</buch>
```

Enthält ein Element keine Kindelemente, kann die Folge von öffnendem und schliessendem Tag wie folgt abgekürzt werden:

```
<buch/>
```

Elementnamen sind frei wählbar, müssen jedoch mit einem Buchstaben, Unterstrich oder Doppelpunkt beginnen und mit alphanumerischen Zeichen, Punkt, Bindestrich, Unterstrich oder Doppelpunkt fortgesetzt werden. Zudem dürfen Elementnamen nicht mit `XML` in jedweder Gross-/Kleinschreibung anfangen. Tags müssen in der richtigen, LIFO-artigen Reihenfolge geschlossen werden. Folgendes Beispiel ist ungültig:

```
<buch>
<autor>
</buch>
</autor>
```

XML-Dokumente müssen mit einer sogenannten *XML-Deklaration* eingeleitet werden:

```
<?xml version="1.0"?>
```

Allgemein stellt die XML-Deklaration eine sogenannte *Processing Instruction* (PI) dar. Sie werden per `<?` eingeleitet und per `?>` abgeschlossen. Auf diese Art lassen sich dem XML-Parser Anweisungen übergeben. Entspricht ein XML-Dokument den bisher erläuterten Regel, ist es *wohlgeformt*. Attribute sind ergänzende Angaben zu Elementen, die im öffnenden Tag angegeben werden. Für ihre Bezeichnung gelten die gleichen Regeln wie für Elementnamen:

```
<?xml version="1.0"?>
<buecher>
    <buch preis="74,95 EUR">
        <titel>Internet Routing Architekturen</titel>
    </buch>
</buecher>
```

Einrückungen am Zeilenanfang sind primär kosmetischer Natur. Zu beachten ist allerdings, dass sie beim Parsen eines XML-Dokumentes nicht zwangsweise gelöscht werden. Es ist dem XML-Parser überlassen, z.B. den Zeilenvorschub und jegliche Leerzeichen zwischen `</titel>` und `</buch>` zu ignorieren. Ein Programmierer sollte dies beim Schreiben einer XML verarbeitenden Anwendung auf jeden Fall berücksichtigen. Daher ist es

auch ratsam, Elementwerte wie den gezeigten Buchtitel ohne voran- oder nachgestellte Whitespaces zu notieren. Attribut- und Elementwerte dürfen beliebige Zeichen enthalten mit folgenden Ausnahmen:

`&` Das kaufmännische Und-Zeichen muss durch `&` ersetzt werden

`<` Die öffnende spitze Klammer muss durch `<` ersetzt werden.

Ferner müssen in Attributwerten, die per doppelter Anführungsstriche umfasst sind, eben doppelte Anführungsstriche durch `>` ersetzt werden. Gleiches gilt für Attributwerte in einfachen Anführungsstrichen. Hier müssen einfache Anführungsstriche durch `&apos;` ersetzt werden. Diese vordefinierten Ersatzzeichen werden *Entities* genannt. Generell kann man jedes Zeichen durch seinen entsprechenden Unicode- oder ASCII-Wert ersetzen, entweder in dezimaler oder hexadezimaler Darstellung. Dem grossen `A` ist der ASCII-Code 65 zugeordnet. Es kann daher durch `&#65;` oder hexadezimal durch `&#x41;` ersetzt werden. Kommentare werden durch `<!--` begonnen und per `-->` geschlossen. Innerhalb von Attributwerten werden sie nicht erkannt, innerhalb eines Tags führen sie zu fehlerhaftem XML.

6.1.1. DTD

Eine *Dokumenttypdefinition* (*Document Type Definition* (DTD)) beschreibt Sprachumfang und Struktur eines XML-Formats. Ein XML-Dokument kann gegen seine DTD geprüft werden, und wird – falls es der DTD genügt – *valide* genannt. Eine DTD ist eine Textdatei, deren Aufbau an die *Erweiterte Backus-Naur-Form* (EBNF) erinnert. Sie kann entweder in ein XML-Dokument eingebettet werden, oder als externe Datei referenziert werden. In beiden Fällen muss die DTD direkt nach der XML-Deklaration eingebunden werden:

```
<?xml version="1.0"?>
<!DOCTYPE buecher SYSTEM "http://bib.fh-bielefeld.de/dtds/buecher.dtd">
<buecher>
    <buch preis="74,95 EUR">
        <titel>Internet Routing Architekturen</titel>
    </buch>
</buecher>
```

Das Wurzelelement `buecher`, alle untergeordneten Elemente und somit – weil ein wohlgeformtes XML-Dokument genau ein Wurzelelement aufweisen muss – das gesamte XML-Dokument müssen der DTD genügen, die über den *Uniform Resource Identifier* (URI) `http://bib.fh-bielefeld.de/dtds/buecher.dtd` referenziert wird. Obwohl dieser URI eine gültige URL darstellt, muss es sich nicht um eine gültige (Internet-)Adresse handeln. Die Menge aller URLs stellt eine Untermenge von URIs dar. Von einer URI fordert man lediglich, dass sie einen eindeutigen Bezeichner darstellt. Eine URL hingegen muss Protokoll und Pfad zu einer Resource wie z.B. zu einem Dokument aufweisen. In der Praxis hat es sich jedoch durchgesetzt, die URI zu einer DTD als konkrete (und öffentlich abrufbare) URL zu notieren. So kann ein Parser jederzeit die XML-Dokumente validieren, welche auf diese DTD verweisen. Eine Document Type Definition selbst ist kein XML-Dokument:

```
<!ELEMENT buecher (buch*)>
<!ELEMENT buch    (titel)>
<!ELEMENT titel   (#PCDATA)>
```

Das (Wurzel-)Element `buecher` kann eine beliebige Anzahl von Elementen des Typs `buch` enthalten. Neben dem Sternchen `*` gibt es wie bei regulären Ausdrücken als Wiederholungsoperatoren das Fragezeichen `?`, welches anzeigt, dass das vorangehende Element nicht oder nur einmal vorkommen darf, und das Pluszeichen `+`, welches ein- oder mehrmaliges Vorkommen ausdrückt. Das Element `buch` muss genau ein Element `titel` enthalten, welches nur Zeichen(-ketten) (*parsed character data*) enthalten darf. Der Typ `#PCDATA` ist vordefiniert. Mit Hilfe des senkrechten Striches `|` können in einer DTD Variationen ausgedrückt werden:

```
<!ELEMENT buecher (buch*)>
<!ELEMENT buch    (titel, (autor | autoren))>
<!ELEMENT titel   (#PCDATA)>
```

Ein Buch muss somit einen Titel und ein Element `autor` oder `autoren` enthalten. Natürlich müssen auch diese definiert werden:

```
<!ELEMENT autor   (#PCDATA)>
<!ELEMENT autoren (autor+)>
```

Das Element `autoren` ist somit eine Auflistung von einem oder mehreren Autoren, die wiederum Zeichenketten sind. Attribute wie z.B. `preis` in den obigen Beispieldokumenten werden in einer DTD separat aufgeführt:

```
<!ATTLIST buch preis CDATA #REQUIRED>
```

Die Attributliste sagt aus, dass das Element `buch` genau ein Attribut `preis` aufweisen muss (`#REQUIRED`). Die Attributwerte müssen Zeichenketten (*character data*) sein. Neben `#REQUIRED`, welches anzeigt, dass ein Attribut aufgeführt werden muss, sind folgende Angaben möglich:

`#IMPLIED` Das Attribut ist optional

`#FIXED` *<Defaultwert>* Das Attribut muss bei jeder Verwendung des Elementes angegeben werden und muss den vorgegebenen Defaultwert haben

<Defaultwert> Das Attribut ist optional. Fehlt es, wird es implizit auf den angegebenen Defaultwert gesetzt.

Als Attributtypen können die folgenden verwendet werden:

`CDATA` Der Attributwert muss eine beliebige Zeichenkette sein

`ID` Der Attributwert muss eine innerhalb des Dokumentes eindeutige Zeichenkette sein

`IDREF` Der Attributwert muss gleich dem Wert eines Attributes vom Typ `ID` sein. Es ist nicht möglich, den Namen des so referenzierten Attributes oder Elementes anzugeben. Die korrekte Verknüpfung muss also die Anwendung selbst herstellen

<Liste von selbst definierten Bezeichnern> Der Attributwert muss ein Bezeichner aus der Liste sein (vgl. `enum` in C).

Die Attributliste für das Element `buch` kann also z.B. wie folgt erweitert werden:

```
<!ATTLIST buch
    preis       CDATA       #REQUIRED
    isbn        ID          #REQUIRED
    hardcover   (yes | no)  #REQUIRED
    notiz       CDATA       #IMPLIED
    beschaedigt CDATA       "nein"
>
```

Ein im Sinne dieser DTD valides XML-Dokument wäre z.B. das folgende:

```
<?xml version="1.0"?>
<!DOCTYPE buecher SYSTEM "http://bib.fh-bielefeld.de/dtds/buecher.dtd">
<buecher>
    <buch preis="74,95 EUR" isbn="3-8272-5938-X" hardcover="yes">
        <titel>Internet Routing Architekturen</titel>
        <autoren>
            <autor>Bassam Halabi</autor>
            <autor>Danny McPherson</autor>
        </autoren>
    </buch>
</buecher>
```

Die DTD kann im XML-Dokument transportiert werden, anstatt per URI referenziert zu werden:

```
<?xml version="1.0"?>
<!DOCTYPE buecher [
    <!ELEMENT buecher (buch*)>
    <!ELEMENT buch    (titel, (autor | autoren))>
    <!ELEMENT titel   (#PCDATA)>
    <!ELEMENT autor   (#PCDATA)>
    <!ELEMENT autoren (autor+)>
    <!ATTLIST buch
        preis       CDATA      #REQUIRED
        isbn        ID         #REQUIRED
        hardcover   (yes | no) #REQUIRED
        notiz       CDATA      #IMPLIED
        beschaedigt CDATA      "nein"
    >
]>
<buecher>
<!-- ... Buecher ... -->
</buecher>
```

6.1.2. Namensräume

Obwohl mit den oben erläuterten Dokumenttypdefinitionen rigide XML-Formate festgelegt werden können, so dass sie z.B. Datenbanktabellen mit ihren Constraints nahezu vollständig abbilden könnten, werden sie in der Praxis selten angewendet. Zum einen stellt die Syntax einer DTD kein valides XML dar. Zum anderen kann ein XML-Dokument nur genau eine DTD referenzieren. So ist es z.B. nicht möglich, eine `person.dtd` zu definieren, die Angaben zu einer Person wie Name, Vorname, Emailadresse, etc. verlangt, und diese DTD bzw. das Element `person` in der o.g. Buchliste als Autor zu verwenden. Daher hat das *World Wide Web Consortium* (W3C)[16], das federführend XML und dessen Derivate entwickelt, sogenannte *Namensräume* (*namespaces*) eingeführt. Ein Namensraum wird im XML-Dokument als Prefix vor Elementnamen verwendet. Damit Namensräume eindeutig sind, werden sie über einen eindeutigen URI referenziert:

```
<?xml version="1.0"?>
<fhbielefeld:buecher
    xmlns:fhbielefeld="http://bib.fh-bielefeld.de/nspace/buecher">
<!-- ... Buecher ... -->
</fhbielefeld:buecher>
```

[16] http://www.w3.org

Das (Wurzel-)Element `buecher` entstammt dem Namespace `fhbielefeld`. Dieser ist eindeutig dem URI `http://bib.fh-bielefeld.de/nspace/buecher` zugeordnet. Es ist dabei dem XML-Parser überlassen, ob und wie er jenen Namensraum behandelt. Im Gegensatz zu DTDs verbirgt sich dahinter keine formale Vorschrift zum Validieren des XML-Dokumentes. Das oben skizzierte Beispiel, in dem das Format der Autorendaten aus einer weiteren DTD importiert werden sollte, lässt sich mit Namensräumen lösen:

```
<?xml version="1.0"?>
<fhbielefeld:buecher
    xmlns:fhbielefeld="http://bib.fh-bielefeld.de/nspace/buecher"
    xmlns:person="http://bib.fh-bielefeld.de/nspace/person">
    <fhbielefeld:buch>
        <fhbielefeld:titel>Modern Operating Systems</fhbielefeld:titel>
        <fhbielefeld:autor>
            <person:name>Tanenbaum</person:name>
            <person:vorname>Andrew</person:vorname>
            <person:titel>Professor</person:titel>
        </fhbielefeld:autor>
    </fhbielefeld:buch>
</fhbielefeld:buecher>
```

Zum einen kann ein Elementname scheinbar mehrfach verwendet werden. Zu beachten ist in obigem Beispiel jedoch, dass der Buchtitel zum Namensraum `fhbielefeld` gehört, der (akademische) Titel des Autors jedoch zu `person`. Zum anderen muss der Namespace `fhbielefeld` nicht die Elemente von `person` implementieren. Letzterer wird dadurch in unterschiedlichen XML-Formaten verwendbar. Definiert man einen *Default-Namespace*, so muss ein Namensraum nur bei Elementen angegeben werden, die nicht zum Default-Namespace gehören. Er wird durch Weglassen des Namespace-Namens zwischen `xmlns` und dem Namespace-URI definiert. Folgendes Dokument ist daher zu obigem äquivalent:

```
<?xml version="1.0"?>
<buecher xmlns="http://bib.fh-bielefeld.de/nspace/buecher"
    xmlns:person="http://bib.fh-bielefeld.de/nspace/buecher">
    <buch>
        <titel>Modern Operating Systems</titel>
        <autor>
            <person:name>Tanenbaum</person:name>
            <person:vorname>Andrew</person:vorname>
            <person:titel>Professor</person:titel>
        </autor>
    </buch>
</buecher>
```

6.1.3. XML Schema

Das W3C hat keine Möglichkeit vorgesehen, Namensräume an eine DTD zu binden. Statt dessen wurde *XML Schema* entwickelt, mit dessen Hilfe Formatvorschriften in einem XML-Format definiert werden können. XML Schema ist selbst über den Namespace `http://www.w3.org/2001/XMLSchema` definiert. Ein gültiges XML Schema hat folgenden Aufbau:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://bib.fh-bielefeld/nspace/buecher">
</xs:schema>
```

Das Attribut `targetNamespace` des Wurzelelementes `schema` gibt den Namensraum an, für den dieses XML Schema definiert wird. Folgendes Schema beschreibt ein triviales XML-Format, dessen Wurzelelement `buchtitel` nur eine Zeichenkette enthalten darf:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://bib.fh-bielefeld/nspace/buecher">
    <xs:element name="buchtitel" type="xs:string"/>
</xs:schema>
```

Neben `string` kennt XML Schema u.a. folgende *simple* Datentypen:

`boolean` Mögliche Werte sind `true` und `1` für logisch wahr, `false` und `0` für logisch falsch

`string` beliebige Zeichenkette

`integer` beliebiger Ganzzahlwert

`dateTime` Datum und Zeit im Format `YYYYMMDDTHH:MM:SS` mit

YYYY Jahr
MM Monat
DD Tag
T Trennzeichen zwischen Datum und Zeit
HH Stunde
MM Minute
SS Sekunde

Alle vordefinierten Datentypen findet man auf den Webseiten des W3C[17]. Elementen kann ein Standardwert zugewiesen werden, so dass sie im XML-Dokument nicht explizit aufgeführt werden müssen:

```
    <xs:element name="buchtitel" type="xs:string"
    default="Ein tolles Buch" />
```

Ebenso kann einem Element ein konstanter Wert zugeordnet werden:

```
    <xs:element name="buchtitel" type="xs:string"
    fixed="Fester Buchtitel" />
```

Elemente, die als Typ keinen simplen Datentyp wie `string` oder `integer` aufweist, werden *komplexe* Elemente genannt. Hierzu zählen insbesondere Elemente, die untergeordnete Kindelemente enthalten:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://bib.fh-bielefeld/nspace/buecher">
    <xs:element name="buch">
        <xs:complexType>
            <xs:all>
                <xs:element name="titel" type="xs:string" />
                <xs:element name="autor" type="xs:string" />
            </xs:all>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Per `complexType` wird ein komplexer Datentyp definiert. Das Element `xs:all` ist ein sogenannter *Indikator*. Er gibt an, wie oft und in welcher Reihenfolge die Kindelemente aufgeführt werden müssen. Es gibt drei Variationen:

`all` Alle untergeordneten Elemente dürfen im XML-Dokument maximal einmal vorkommen, und zwar in der vom XML Schema vorgegebenen Reihenfolge

[17] http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#built-in-datatypes

`choice` Jedes untergeordnete Element kann beliebig oft vorkommen

`sequence` Alle untergeordneten Elemente müssen im XML-Dokument mindestens einmal vorkommen, und zwar in der vom XML Schema vorgegebenen Reihenfolge.

Zusätzlich existieren Indikatoren, die als Attribut einer Elementdefinition verwendet werden können.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://bib.fh-bielefeld/nspace/buecher">
    <xs:element name="buch">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="titel" type="xs:string"
                    minOccurs="1" maxOccurs="1"/>
                <xs:element name="autor" type="xs:string"
                    minOccurs="1" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Somit muss das Element `titel` genau einmal vorkommen, `autor` hingegen mindestens einmal. Das vollständige XML Schema für eine Buchliste sieht wie folgt aus:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://bib.fh-bielefeld/nspace/buecher">
    <xs:element name="buecher">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="buch"
                    minOccurs="1" maxOccurs="unbounded"/>
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="titel" type="xs:string"
                                minOccurs="1" maxOccurs="1"/>
                            <xs:element name="autor" type="xs:string"
                                minOccurs="1" maxOccurs="unbounded"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Dieses Schema sollte öffentlich zugänglich abgelegt werden. z.B. unter der URL `http://bib.fh-bielefeld/nspace/buecher.xsl`. Es kann unter Verwendung des Namespaces `http://www.w3.org/2001/XMLSchema-instance` in einem XML-Dokument referenziert werden:

```
<?xml version="1.0"?>
<buecher xmlns="http://bib.fh-bielefeld.de/nspace/buecher"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://bib.fh-bielefeld.de/nspace/buecher.xsl">
    <buch>
        <titel>Modern Operating Systems</titel>
        <autor>Andrew Tanenbaum</autor>
    </buch>
</buecher>
```

Das Prefix `xsi` bezieht sich auf den Namensraum `http://www.w3.org/2001/XMLSchema-instance`. Dieser stellt das Attribut

`schemaLocation` bereit, über den das Schema eingebunden wird. XML Schema wird u.a. in SOAP (s. Kapitel 6.3) und WSDL (s. Kapitel 6.4) zur Parameterdefinition und -übergabe verwendet.

6.2. XSLT

Mit Hilfe von *Extensible Stylesheet Language Transformations* (XSLT) werden XML-Dokumente in andere Datentypen wie z.B. Textdateien umgewandelt. XSLT ist ein Teil der *Extensible Stylesheet Language* (XSL). Diese umfasst zusätzlich die sogenannten *Extensible Stylesheet Language Formatting Objects* (XSL-FO), mit denen XML-Dokumente für die Ausgabe auf einem Bildschirm, für den Druck, etc. formatiert werden. XSLT und XSL-FO verwenden zur jeweiligen Umwandlung von XML-Dokumenten sogenannte *Stylesheets*, die ebenfalls in XML notiert werden. Stylesheets für XSL-FO enthalten i.d.R. physikalische Angaben wie Papiergrösse, Abstände in Pixel, etc. Stylesheets für XSLT stellen hingegen einen Compiler für XML-Dokumente in das jeweilige Zielformat dar. Weiterer Bestandteil von XSL ist *XPath*, eine Sprache, mit der Elemente eines XML-Dokumentes ausgewählt werden können. XSL-FO und XSLT verwenden XPath. XSLT ist im Namensraum `http://www.w3.org/1999/XSL/Transform` definiert. Ein Stylesheet hat daher folgenden Aufbau:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
    <xsl:output method="text" />
</xsl:stylesheet>
```

Die Angabe der Versionsnummer im Wurzelelement `stylesheet` ist zwingend notwendig. Mit dem Element `output` teilt man dem XSLT-Parser mit, welches (Datei-)Format das Zieldokument hat. Mögliche Werte sind `xml`, `text` und `html`. Die eigentlichen Transformationsvorschriften werden mit dem Element `template` eingeleitet. Sie werden auf die Elemente des Quelldokumentes angewendet, die das Attribut `match` beschreibt:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
    <xsl:output method="text" />
    <xsl:template match="/">
hello, world
    </xsl:template>
</xsl:stylesheet>
```

Der Attributwert von `match` stellt eine Pfadangabe in XPath dar. Wie in Unix-Dateisystemen dient der Schrägstrich zur Hierarchietrennung. Ein einzelner Schrägstrich wie im Beispiel spricht jedoch nicht das Wurzelelement des Quelldokumentes an, sondern die übergeordnete Ebene, sprich das gesamte Dokument. Das Template wird also jedes XML-Dokument in den Text `hello, world` umformen. Mit `for-each` kann man über Elemente gleichen Namens und gleicher Hierarchiestufe iterieren:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
    <xsl:output method="text" />
    <xsl:template match="/">
        <xsl:for-each select="buecher/buch">
            <!-- ... ein einzelnes Buch transformieren ... -->
        </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

Mit dieser Schleife iteriert man über jedes Element namens `buch`, falls es ein Kindelement von `buecher` ist. Die Reihenfolge, in der eine Schleife abgearbeitet wird, legt man optional per `sort` innerhalb des Schleifenkörpers fest:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
    <xsl:output method="text" />
    <xsl:template match="/">
        <xsl:for-each select="buecher/buch">
            <xsl:sort select="autor" order="ascending" />
            <!-- ... ein einzelnes Buch transformieren ... -->
        </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

Die Liste aller Bücher wird somit aufsteigend nach dem Namen des Autors sortiert bearbeitet. Analog zu `ascending` gibt es `descending` für absteigende Sortierung. Über das Attribut `data-type` legt man die Art der Sortierung fest. Es kann die Werte `text` für textuelle oder `number` für Sortierung nach Zahlen haben. Den Wert eines Elements gibt man per `value-of` aus:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
    <xsl:output method="text" />
    <xsl:template match="/">
        <xsl:for-each select="buecher/buch">
            <xsl:value-of select="titel" />
        </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

Die XPath-Angaben sind i.d.R. nicht voll qualifiziert. Sie beziehen sich immer auf das aktuelle Element. So ist `titel` relativ zu `buecher/buch` auszuwerten. Daher ist es z.B. per `../element` auch möglich, übergeordnete Elemente anzusprechen. Unformatierter Text kann ohne eigene Elemente ausgegeben werden. Da jedoch Entities wie z.B. die öffnende spitze Klammer `<` durch `<` ersetzt werden, sollte unformatierter Text mit dem XSLT-Element `text` ausgegeben werden. Dieses akzeptiert das boolsche Attribut `disable-output-escaping`:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
    <xsl:output method="text" />
    <xsl:template match="/">
        <xsl:for-each select="buecher/buch">
            <xsl:value-of select="titel" />
            <xsl:text disable-output-escaping="yes">, </xsl:text>
        </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

Somit erhält man eine kommaseparierte Ausgabe aller Buchtitel. Mit `if` und `choose` existieren zwei Elemente zur Fallunterscheidung:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
    <xsl:output method="text" />
    <xsl:template match="/">
```

```
        <xsl:for-each select="buecher/buch">
            <xsl:if test="titel = 'Ein toller Buchtitel'">
                <xsl:value-of select="autor" />
            </xsl:if>
        </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

Das gezeigte Stylesheet gibt den Namen des Buchautors nur aus, wenn der Titel `Ein toller Buchtitel` lautet. Für eine Mehrfachauswahl muss `choose` verwendet werden, welches auch einen Defaultzweig anbietet:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
    <xsl:output method="text" />
    <xsl:template match="/">
        <xsl:for-each select="buecher/buch">
            <xsl:choose>
                <xsl:when test="titel = 'Ein toller Buchtitel'">
                    <xsl:value-of select="titel" />
                </xsl:when>
                <xsl:when test="titel = 'Ein brilianter Buchtitel'">
                    <xsl:value-of select="titel" />
                </xsl:when>
                <xsl:otherwise>
                    <xsl:text disable-output-escaping="yes">
                        Ein normaler Buchtitel
                    </xsl:text>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

XSLT-Stylesheets werden i.d.R. mittels eines XSLT-Parsers auf XML-Dokumente angewendet. Für visuelle Tests können alternativ ein aktueller Browser (z.B. Firefox ab Version 1.0.2, Opera ab Version 9 oder Internet Explorer ab Version 6) verwendet werden. Zunächst referenziert man das Stylesheet per Processing Instruction im XML-Dokument:

```
<?xml version="1.0"?>
<?xml-stylesheet href="buecher.xsl" type="text/xsl"?>
<buecher>
    <buch>
        <titel>Modern Operating Systems</titel>
        <autor>Andrew Tanenbaum</autor>
    </buch>
    <buch>
        <titel>Internet Routing Architekturen</titel>
        <autor>Bassam Halabi and Danny McPherson</autor>
    </buch>
</buecher>
```

Dieses XML-Dokument muss im selben lokalen Verzeichnis gespeichert sein wie das folgende Stylesheet, dessen Dateiname `buecher.xsl` lauten muss:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
    <xsl:output method="text" />
    <xsl:template match="/">
        <xsl:for-each select="buecher/buch">
            <xsl:sort select="titel" order="ascending" />
```

```
            <xsl:text disable-output-escaping="yes">Titel: </xsl:text>
            <xsl:value-of select="titel" />
            <xsl:text>&#10;</xsl:text> <!-- Zeilenvorschub -->
            <xsl:text disable-output-escaping="yes">Autor: </xsl:text>
            <xsl:value-of select="autor" />
            <xsl:text>&#10;</xsl:text> <!-- Zeilenvorschub -->
        </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

Ruft man nun das obige XML-Dokument in einem Browser auf, wird er die Bücherliste zur Darstellung nach den Regeln des XSLT-Stylesheets in ein Textdokument transformieren (s. Abbildung 10).

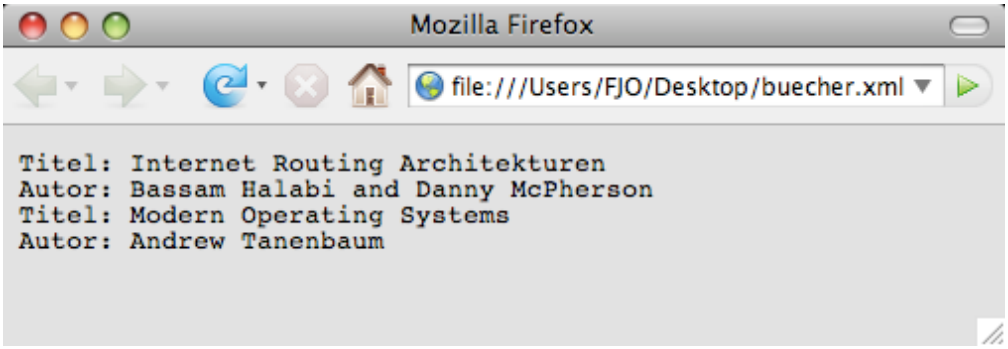

Abbildung 10: Der Webbrowser Firefox als XSLT-Parser

6.3. SOAP

Das *Simple Object Access Protocol* (SOAP) ist ein Protokoll zum entfernten Aufruf von Funktionen (*Remote Procedure Call* (RPC)). Nachrichten zwischen Client und Server werden in einem besonderen XML-Format ausgetauscht. Als Transportprotokoll wird überwiegend HTTP bzw. dessen sichere Variante HTTPS eingesetzt[18]. Generell sind alle Medien möglich, die den Transport von (XML-)Dokumenten erlauben wie z.B. Email, FTP, etc. Das Wurzelelement einer SOAP-Nachricht lautet `envelope`. Dieses muss genau ein Element `body` aufweisen, welches die vom Programmierer festgelegten Funktionsaufrufe und Parameterübergaben beinhaltet. Optional kann dem Element `body` ein per `header` begrenzter Abschnitt vorangestellt werden, der Metaangaben über die Nachricht wie z.B. Authentifizierungsparameter, eine eindeutige Nachrichten-ID, o.ä. enthalten kann. SOAP ist durch den Namespace `http://schemas.xmlsoap.org/soap/envelope/` gegeben. Eine typische SOAP-Nachricht vom Client zum Server sieht wie folgt aus:

```
<?xml version="1.0"?>
<S:envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:body>
        <ns2:CreateProject
            xmlns:ns2="http://fjo-otrs.dts-online.net/Kernel/DTSSoap">
            <User>test</User>
            <Pass>test</Pass>
            <ProjectName>Ein neues Projekt</ProjectName>
            <ProjectDebtorID>1111</ProjectDebtorID>
        </ns2:CreateProject>
    </S:body>
```

[18] Daher auch der Begriff *Webservice(s)*.

```
</S:envelope>
```

Diese Nachricht ruft die Funktion `CreateProject` mit den Argumenten `User`, `Pass`, `ProjectName` und `ProjectDebtorID` auf. War der Aufruf fehlerfrei und gibt die Funktion einen Wert zurück, so antwortet der Server ebenfalls mit einem SOAP-Body:

```
<?xml version="1.0"?>
<soap:envelope
    xmlns:namesp1="http://fjo-otrs.dts-online.net/Kernel/DTSSoap"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:body>
        <namesp1:CreateProjectResponse>
            <CreateProjectReturn
                xsi:type="xsd:string">ok</CreateProjectReturn>
        </namesp1:CreateProjectResponse>
    </soap:body>
</soap:envelope>
```

Die Funktion gibt einen String mit dem Wert `ok` zurück. Tritt ein Fehler auf, so beinhaltet der SOAP-Body lediglich ein Element namens `Fault`, das Aufschluss über das Scheitern des Funktionsaufrufes gibt (aus Platzgründen ist das Element `faultstring` hier mehrzeilig):

```
<?xml version="1.0"?>
<soap:envelope
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:body>
        <soap:Fault>
            <faultcode>soap:Server</faultcode>
            <faultstring>User test is not authorized to call method
                CreateProject from remote host 81.89.251.79 at
                /usr/local/otrs/Kernel/DTSSoap.pm line 372,
                &lt;PRODUCT&gt; line 24.
            </faultstring>
        </soap:Fault>
    </soap:body>
</soap:envelope>
```

Offenbar liegt ein Berechtigungsproblem vor. Der Fehlertext lässt vermuten, dass es sich um ein in Perl geschriebenes SOAP-Modul handelt. SOAP-Nachrichten wird ein Programmierer selten manuell erzeugen. Statt dessen setzt man entsprechende Bibliotheken ein, die entfernte Funktionsaufrufe wie herkömmliche, lokale Funktionsaufrufe ermöglichen und so jegliche Komplexität verbergen[19].

6.4. WSDL

Die *Web Services Description Language* (WSDL) ermöglicht es, automatisch aus den Signaturen der von einem SOAP-Server bereitgestellten Funktionen lokale Funktionsrümpfe (sogenannte *Stubs*) zu erzeugen. Bei schwach typisierten Programmiersprachen wie

[19] Vgl. andere RPC-Protokolle wie Corba oder DCOM

z.B. Perl werden die Funktionssignaturen aus speziell formatierten Kommentaren gewonnen. In Java hingegen könnte dies per Introspektion über das Reflection API[20] geschehen. C-Derivate könnten mittels separatem Compiler die jeweiligen statischen Headerfiles in WSDL-Dokumente transformieren. Während SOAP-Libraries die eigentlichen RPC-Aufrufe kapseln, entbindet WSDL also den Programmierer von der Aufgabe, die Stubs manuell mit den SOAP-Funktionen abzugleichen. WSDL stellt daher eine unidirektionale Kommunikation dar, in der Clients vom Server pollen. Die Funktionsbeschreibung in WSDL erfolgt in einem eigenen XML-Format. Neben dem Wurzelelement `definitions` werden die folgend erläuterten Elemente erwartet:

message Das Element `message` fasst Argumente zusammen, die einer Funktion übergegeben werden oder die eine Funktion zurückliefert. Es umfasst meist mehrere Kindelemente namens `part`, die Name und Typ des jeweiligen Argumentes beschreiben. XML Schema stellt hierzu die Datentypen bereit.

portType Das Pivotelement `portType` definiert die vom SOAP-Server bereitgestellten Funktionen. Es verknüpft die per `message` definierten Argumente mit den Funktionsnamen. Ein WSDL-Dokument enthält meist nur ein Element `portType`, welches mehrere Kindelemente namens `operation` umfasst. Diese beschreiben die jeweilige Funktionssignatur mit erwarteten Argumenten (Element `input`) und etwaigen Rückgabewerten (Element `output`).

binding Das Element `binding` beschreibt, wie Funktionsargumente in den jeweiligen SOAP-Nachrichten serialisiert werden sollen (*Bindings*). Die Auswahl der richtigen Serialisierungsart bzw. des *Binding Styles* hängt von den verwendeten SOAP- und WSDL-Bibliotheken ab, die Server und Clients verwenden. Zwischen den Kindelementen von `binding` und `portType` besteht eine 1:1-Beziehung.

services Das Element `services` ordnet die per `binding` definierten Funktionsaufrufe der URL des SOAP-Servers zu.

Ein typisches WSDL-Dokument ist wie folgt aufgebaut:

```
<?xml version="1.0"?>
<wsdl:definitions
  targetNamespace="http://fjo-otrs.dts-online.net/Kernel/DTSSoap"
  xmlns:impl="http://fjo-otrs.dts-online.net/Kernel/DTSSoap"
  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns1="http://fjo-otrs.dts-online.net/Kernel/DTSSoap">

  <wsdl:message name="CreateProjectRequest">
    <wsdl:part name="User" type="xsd:string" />
    <wsdl:part name="Pass" type="xsd:string" />
    <wsdl:part name="ProjectName" type="xsd:string" />
    <wsdl:part name="ProjectDebtorID" type="xsd:string" />
  </wsdl:message>

  <wsdl:message name="CreateProjectResponse">
    <wsdl:part name="CreateProjectReturn" type="xsd:string" />
  </wsdl:message>

  <wsdl:portType name="KernelDTSSoapHandler">
```

[20] http://java.sun.com/docs/books/tutorial/reflect/

```
    <wsdl:operation name="CreateProject"
      parameterOrder="User Pass ProjectName ProjectDebtorID">
      <wsdl:input message="impl:CreateProjectRequest"
        name="CreateProjectRequest" />
      <wsdl:output message="impl:CreateProjectResponse"
        name="CreateProjectResponse" />
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="KernelDTSSoapSoapBinding"
    type="impl:KernelDTSSoapHandler">
    <wsdlsoap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="CreateProject">
      <wsdlsoap:operation soapAction="" />
      <wsdl:input name="CreateProjectRequest">
        <wsdlsoap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://fjo-otrs.dts-online.net/Kernel/DTSSoap"
          use="literal" />
      </wsdl:input>
      <wsdl:output name="CreateProjectResponse">
        <wsdlsoap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://fjo-otrs.dts-online.net/Kernel/DTSSoap"
          use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="KernelDTSSoapHandlerService">
    <wsdl:port binding="impl:KernelDTSSoapSoapBinding"
      name="KernelDTSSoap">
      <wsdlsoap:address
        location="http://fjo-otrs.dts-online.net/soap" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Unter der URL `http://fjo-otrs.dts-online.net/soap` wird ein SOAP-Service bereitgestellt. Dieser umfasst die Funktion `CreateProject`, welche vier Zeichenketten als Parameter erwartet. Diese heissen `User`, `Pass`, `ProjectName` und `ProjectDebtorID`. Die Funktion liefert als Rückgabewert ebenfalls eine Zeichenkette. Als Serialisierungsart definiert das Element `binding` hier den Typ `RPC/literal`. Es gibt fünf Serialisierungsarten, die sich in zwei Klassen einteilen lassen: *RPC* encodierte und *Document* encodierte Binding Styles. Verwendet man letztere, so werden die Funktionsargumente in der WSDL-Definition als eigenes XML-Schema dargestellt. Dies hat den Vorteil, dass jede SOAP-Nachricht mit einem generischen XML-Parser gegen dieses Schema verifiziert werden kann. Folgend werden alle Serialisierungsarten erläutert:

RPC/encoded Die Funktionsargumente einer nach *RPC/encoded* serialisierten SOAP-Nachricht weisen nicht nur den Wert des jeweiligen Argumentes, sondern auch dessen Typ auf, z.B.:

```
<?xml version="1.0"?>
<S:envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:body>
    <ns2:CreateProject
      xmlns:ns2="http://fjo-otrs.dts-online.net/Kernel/DTSSoap"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <User xsi:type="xsd:string">test</User>
      <Pass xsi:type="xsd:string">test</Pass>
      <ProjectName
        xsi:type="xsd:string">Ein neues Projekt</ProjectName>
      <ProjectDebtorID xsi:type="xsd:string">1111</ProjectDebtorID>
    </ns2:CreateProject>
  </S:body>
</S:envelope>
```

RPC/literal SOAP-Nachrichten im Stil von *RPC/literal* verzichten auf eine Typangabe bei Funktionsargumenten:

```
<?xml version="1.0"?>
<S:envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:body>
    <ns2:CreateProject
      xmlns:ns2="http://fjo-otrs.dts-online.net/Kernel/DTSSoap">
      <User>test</User>
      <Pass>test</Pass>
      <ProjectName>Ein neues Projekt</ProjectName>
      <ProjectDebtorID>1111</ProjectDebtorID>
    </ns2:CreateProject>
  </S:body>
</S:envelope>
```

Document/literal Verwendet man die Bindungsart *Document/literal*, so wird das WSDL-Dokument um einen Schemaabschnitt erweitert. Auf dessen Elemente verweisen dann die Funktionsargumente:

```
<!-- ... -->
<wsdl:types>
  <xsd:schema
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="UserElement" xsi:type="xsd:string" />
    <xsd:element name="PassElement" xsi:type="xsd:string" />
    <xsd:element name="ProjectNameElement"
      xsi:type="xsd:string" />
    <xsd:element name="ProjectDebtorIDElement"
      xsi:type="xsd:string" />
  </xsd:schema>
</wsdl:types>
<wsdl:message name="CreateProjectRequest">
  <wsdl:part name="User" element="UserElement" />
  <wsdl:part name="Pass" element="PassElement" />
  <wsdl:part name="ProjectName" element="ProjectNameElement" />
  <wsdl:part name="ProjectDebtorID"
    element="ProjectDebtorIDElement" />
</wsdl:message>
<!-- ... -->
```

Quellen wie Butek (2005) geben an, dass SOAP-Nachrichten nach Document/literal nicht den Namen der aufgerufenen Funktion mitführen, z.B.:

```
<?xml version="1.0"?>
<S:envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:body
      xmlns:ns2="http://fjo-otrs.dts-online.net/Kernel/DTSSoap">
    <ns2:User>test</ns2:User>
    <ns2:Pass>test</ns2:Pass>
```

```
    <ns2:ProjectName>Ein neues Projekt</ns2:ProjectName>
    <ns2:ProjectDebtorID>1111</ns2:ProjectDebtorID>
  </S:body>
</S:envelope>
```

Eine derartige Nachricht wäre jedoch nicht konform mit dem SOAP-Schema, welches nur ein Kindelement im Body zulässt. Quellen wie Shohoud (2003) zeigen daher Beispielnachrichten, die nach Document/literal encodiert sind und dennoch den Funktionsnamen aufweisen.

Document/literal wrapped Im Unterschied zu *Document/literal*, welches für jedes Argument einer Funktion ein eigenes, per Schema definiertes Element verwendet, kapselt *Document/literal wrapped* die Argumente einer Funktion in einem komplexen Datentyp. Das WSDL-Dokument hat daher folgenden Aufbau:

```
<!-- ... -->
<wsdl:types>
  <xsd:schema
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="CreateProjectParameters">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="User" xsi:type="xsd:string" />
          <xsd:element name="Pass" xsi:type="xsd:string" />
          <xsd:element name="ProjectName"
            xsi:type="xsd:string" />
          <xsd:element name="ProjectDebtorID"
            xsi:type="xsd:string" />
        </xsd:sequence>
      </xsd:complexType>
    <xsd:element name="CreateProjectParameters">
  </xsd:schema>
</wsdl:types>
<wsdl:message name="CreateProjectRequest">
  <wsdl:part name="parameters"
    element="CreateProjectParameters" />
</wsdl:message>
<!-- ... -->
```

Document/encoded Die Serialisierungsart *Document/encoded* wird von keiner Anwendung eingesetzt. Eine Implementierung müsste wie bei *RPC/encoded* die Datentypen in der SOAP-Nachricht aufführen und in der WSDL-Beschreibung ein Schema definieren, welches ebenfalls alle Argumente samt Datentypen darstellt.

In der Praxis werden vorwiegend RPC/literal und Document/literal wrapped verwendet. RPC/encoded entspricht nicht den Vorgaben der *Web Services Interoperability Organization* (WS-I)[21], einem Firmenkonsortium, welches das Zusammenspiel von Web Services-Plattformen verschiedener Anbieter sicherstellen möchte.

[21] http://www.ws-i.org

7. Datenbankabfragesprachen

7.1. SQL

Die *Structured Query Language* (SQL) dient zur Daten- und Strukturmanipulation sowie zur Verwaltung einer Datenbank. Sie stellt keine Programmiersprache dar. So fehlen u.a. Möglichkeiten zur Variablendefinition und Kontrollstrukturen wie Schleifen. SQL ist semantisch an umgangssprachliches Englisch angelehnt. Anweisungen müssen mit einem Semikolon abgeschlossen werden:

```
SELECT * FROM studenten;
```

Der Befehl selektiert alle Datensätze (Zeilen) sowie alle Attribute (Spalten) aus der Tabelle `studenten`. Um unnötige Last zu vermeiden, sollten zum einen nur tatsächlich von der Anwendung benötigte Attribute selektiert werden, z.B.:

```
SELECT email FROM studenten;
```

Ausserdem sollte die Anzahl der ausgewählten Datensätze limitiert werden, sofern z.B. von vornherein feststeht, dass die Anwendung nur Studenten mit einer bestimmten Matrikelnummer verarbeiten soll:

```
SELECT email FROM studenten WHERE matrikelnr >= 200000;
```

Die Optionen der `WHERE`-Klausel stellen boolsche Ausdrücke dar und können mit `AND`, `OR` oder `NOT` (als Negation vor einem Ausdruck) kombiniert werden:

```
SELECT email FROM studenten \
    WHERE (matrikelnr >= 200000) OR (matrikelnummer = 100000);
```

Mit der Anweisung `INSERT` fügt man einer Relation Datensätze hinzu. Zu beachten ist, dass Zeichenketten im Gegensatz zu den meisten Programmiersprachen in einfache Anführungsstriche gefasst werden müssen:

```
INSERT INTO studenten (matrikelnummer, vorname, name, email) VALUES (
    203583,
    'Felix',
    'Ogris',
    'felix@fh-bielefeld.de'
);
```

Lässt man beim Einfügen eines neuen Datensatzes Attribute aus, so wird der Datensatz an ihrer Stelle mit Defaultwerten ergänzt, die der Programmierer beim Anlegen der Relation vorgegeben hat. Wurden keine Defaultwerte vorgegeben, so erhält man einen Fehler. Der Befehl `DELETE` zum Löschen von Datensätzen ähnelt der `SELECT`-Anweisung:

```
DELETE FROM studenten WHERE matrikelnummer < 100000;
```

Datensätze können mit dem Befehl `UPDATE` verändert werden. Hierbei kann man den Attributen nicht nur feste Werte zuweisen:

```
UPDATE studenten SET matrikelnummer = matrikelnummer + 1000;
```

Somit wird jede Matrikelnummer um 1000 erhöht. Vor der Änderung kann eine Selektion stattfinden:

```
UPDATE studenten SET matrikelnummer = matrikelnummer - 1000 \
    WHERE name = 'Meier';
```

Die gezeigte Anweisung ändert die Matrikelnummer nur von Studenten, die `Meier` heissen. Mit dem Befehl `CREATE TABLE` werden Relationen angelegt:

```
CREATE TABLE studenten (
    id              SERIAL,
    matrikelnummer  INTEGER,
    name            VARCHAR(100),
    vorname         VARCHAR(100),
    email           VARCHAR(100)
);
```

SQL bzw. PostgreSQL kennt eine Vielzahl von Datentypen. Tabelle 8 erläutert die wichtigsten.

Tabelle 8: Ausgewählte Datentypen in PostgreSQL

Datentyp	**Beschreibung**
SMALLINT	2 Byte grosser Integerwert im Bereich von -32768 bis 32767
INTEGER	4 Byte grosser Integerwert im Bereich von -2147483648 bis 2147483647
BIGINT	8 Byte grosser Integerwert im Bereich von -9223372036854775808 bis 9223372036854775807
SERIAL	4 Byte grosser Integerwert im Bereich von -2147483648 bis 2147483647, der beim Einfügen eines neuen Datensatzes atomar inkrementiert wird
BIGSERIAL	8 Byte grosser Integerwert im Bereich von -9223372036854775808 bis 9223372036854775807, der beim Einfügen eines neuen Datensatzes atomar inkrementiert wird
REAL	4 Byte grosser Fließkommawert mit einer Genauigkeit von 6 Stellen
DOUBLE PRECISION	8 Byte grosser Fließkommawert mit einer Genauigkeit von 15 Stellen
VARCHAR(n)	Zeichenkette mit maximaler Länge von n Zeichen
CHAR(n)	Zeichenkette mit fester Länge von n Zeichen, ungenutzte Stellen müssen mit Leerzeichen gefüllt werden
TEXT	beliebig lange Zeichenkette
BOOLEAN	boolscher Wert; logisch wahr kann durch `TRUE`, `1`, `'t'`, `'true'`, `'y'`, `'yes'`, `'1'` dargestellt werden, logisch falsch durch `FALSE`, `0`, `'f'`, `'false'`, `'n'`, `'no'`, `'0'`
TIMESTAMP WITHOUT TIME ZONE	8 Byte grosser Zeit- & Datumswert (s.a. Kapitel 4.2.3)
TIMESTAMP WITH TIME ZONE	8 Byte grosser Zeit- & Datumswert, der vor der Ausgabe in die lokalen Zeitzone umgerechnet wird

Die Typen `SERIAL` und `BIGSERIAL` verdienen besondere Beachtung. Weisst man einem derartigen Feld innerhalb einer `INSERT`-Anweisung keinen Wert zu, so wird gegenüber einer vorangegangenen Einfügeoperation automatisch der um 1 grössere Wert verwendet. Da derartige Attribute atomar inkrementiert werden, ist gewährleistet, dass z.B. das Feld `id` der Tabelle `studenten` eindeutig ist. Es ist daher zum primären Schlüssel geeignet. Problematisch bei diesen autoinkrementellen Typen können Überläufe werden. Geht man von einer Lebensdauer der Tabelle von 10 Jahren aus, so können Felder vom

Typ `SERIAL` höchstens

$$\frac{2^{31}}{10 * 365 * 86400} \frac{U}{s} \approx 7 \frac{U}{s}$$

und Felder vom Typ `BIGSERIAL` maximal

$$\frac{2^{63}}{10 * 365 * 86400} \frac{U}{s} \approx 2,9 * 10^{10} \frac{U}{s}$$

verarbeiten, bevor ein Überlauf eintritt (mit $\frac{U}{s}$: Updates pro Sekunde). Sogenannte *Constraints* stellen Bedingungen dar, die jeder Datensatz erfüllen muss. Erfüllt er diese nicht, wird er nicht in die Relation aufgenommen bzw. nicht aktualisiert. Constraints werden bei der Tabellendefinition angegeben:

```
CREATE TABLE studenten (
    id              SERIAL NOT NULL,
    matrikelnummer  INTEGER NOT NULL,
    name            VARCHAR(100) NOT NULL,
    vorname         VARCHAR(100) NOT NULL,
    email           VARCHAR(100) NOT NULL
);
```

Implizit hat jedes Attribut einen *Null-Constraint*, d.h. dem Attribut muss nicht zwingend ein Wert zugewiesen werden, damit der Datensatz gültig ist. Die *Not-Null*-Bedingungen im Beispiel verlangen hingegen, dass den Attributen immer ein Wert zugewiesen ist. Der *Unique*-Constraint erzwingt, dass der Wert eines Attributes einmalig in der gesamten Relation ist:

```
CREATE TABLE studenten (
    id              SERIAL NOT NULL,
    matrikelnummer  INTEGER NOT NULL UNIQUE,
    name            VARCHAR(100) NOT NULL,
    vorname         VARCHAR(100) NOT NULL,
    email           VARCHAR(100) NOT NULL
);
```

Somit darf jede Matrikelnummer nur ein einziges Mal vorkommen. Zu beachten ist, dass nur Attribute eine *Unique*-Bedingung erhalten sollten, die auch einen expliziten *Not-Null*-Constraint besitzen. Nur belegte, sprich mit einem Wert versehene Felder unterliegen einem *Unique*-Constraint, so dass trotz vermeintlicher Einzigartigkeit mehrere nicht wertbehaftete Felder in einer Relation vorkommen können. Sogenannte *Check*-Constraints werden als boolsche Ausdrücke formuliert:

```
CREATE TABLE studenten (
    id              SERIAL NOT NULL,
    matrikelnummer  INTEGER NOT NULL UNIQUE,
    name            VARCHAR(100) NOT NULL,
    vorname         VARCHAR(100) NOT NULL,
    email           VARCHAR(100) NOT NULL CHECK(char_length(email) > 3)
);
```

Die Funktion `char_length()` ermittelt die Zeichenlänge eines Strings, hier des Attributes `email`. Folglich müssen alle Emailadressen 4 oder mehr Zeichen aufweisen. Ferner können derartige Bedingungen nicht nur pro Attribut definiert werden, sondern auch als Tabellenconstraint:

```
CREATE TABLE studenten (
    id              SERIAL NOT NULL,
```

```
    matrikelnummer INTEGER NOT NULL UNIQUE,
    name           VARCHAR(100) NOT NULL,
    vorname        VARCHAR(100) NOT NULL,
    email          VARCHAR(100) NOT NULL CHECK(char_length(email) > 3),
    CHECK (name != vorname)
);
```

Somit wird versichert, dass der Name eines Studenten nicht gleich seinem Vornamen ist. Primärschlüssel werden mit der Option `PRIMARY KEY` definiert, Fremdschlüssel per `REFERENCES` oder `FOREIGN KEY`:

```
CREATE TABLE studenten (
    id             SERIAL NOT NULL PRIMARY KEY,
    matrikelnummer INTEGER NOT NULL UNIQUE,
    name           VARCHAR(100) NOT NULL,
    vorname        VARCHAR(100) NOT NULL,
    email          VARCHAR(100) NOT NULL CHECK(char_length(email) > 3),
    plz            INTEGER REFERENCES staedte(plz),
    CHECK (name != vorname)
);

CREATE TABLE staedte (
    plz    INTEGER NOT NULL UNIQUE PRIMARY KEY,
    stadt VARCHAR(100)
);
```

Schlüssel, die aus mehreren Attributen bestehen, müssen wie Tabellen-Constraints definiert werden. Allerdings sollte man sie nur mit Bedacht einsetzen, da sie oftmals Indiz für ein nicht normalisiertes Datenmodell sind:

```
CREATE TABLE raeume (
    gebaeude_nummer INTEGER NOT NULL,
    raum_nummer     INTEGER NOT NULL,
    sitzplaetze     INTEGER NOT NULL CHECK(sitzplaetze > 0),
    PRIMARY KEY (gebaeude_nummer, raum_nummer)
);

CREATE TABLE mitarbeiter (
    personal_nr     INTEGER NOT NULL UNIQUE PRIMARY KEY,
    name            VARCHAR(100) NOT NULL,
    gebaeude_nummer INTEGER NOT NULL,
    raum_nummer     INTEGER NOT NULL,
    FOREIGN KEY (gebaeude_nummer, raum_nummer) REFERENCES raeume
);
```

Sind die Attributnamen von Primär- und Fremdschlüssel gleichlautend, so können sie beim Referenzieren der Fremdtabelle weggelassen werden. Vergisst man bei der Definition der Fremdschlüssel den *Not-Null*-Constraint, können in die Relation Datensätze eingefügt werden, die nicht mit einem Datensatz in der Fremdtabelle verknüpft sind. Relationen werden per `DROP` gelöscht:

```
DROP TABLE studenten;
```

Wird die Tabelle jedoch noch als Fremdtabelle referenziert, kann sie nicht gelöscht werden. Relationen können per `ALTER TABLE` verändert werden. So kann z.B. ein weiteres Attribut hinzugefügt werden:

```
ALTER TABLE studenten ADD strasse VARCHAR(100);
```

Analog werden Spalten gelöscht:

```
ALTER TABLE studenten DROP strasse;
```

SQL-Befehle dienen nicht nur zum Verändern von Daten und Relationen, sondern auch, um Benutzer und Datenbanken anzulegen:

```
CREATE USER benutzer1 WITH PASSWORD 'geheim';
```

Die Anweisung legt den User `benutzer1` mit dem Passwort `geheim` an. Das Schlüsselwort `WITH` ist optional. Da Benutzerkonten intern in Relationen gespeichert werden, kommen zur Benutzermanipulation ebenfalls SQL-Anweisungen zum Einsatz:

```
ALTER USER benutzer1 PASSWORD 'geheimer';
```

Zusätzlich können einem Benutzer die Rechte zum Anlegen von Datenbanken (s.u.) oder weiteren Benutzern zugesprochen werden:

```
ALTER USER benutzer1 CREATEDB;
ALTER USER benutzer1 CREATEUSER;
```

Per `DROP USER` wird ein Login gelöscht. Den Benutzer, mit dem man gerade angemeldet ist, kann man jedoch nicht löschen. Nach dem Anlegen des Datenbankclusters enthält ein PostgreSQL-Server drei Datenbanken:

template1 Die Datenbank `template1` dient per default als Vorlage für weitere Datenbanken. Sie kann vom Administrator angepasst werden, so dass neue Datenbanken mit den veränderten Werten angelegt werden.

template0 Die Datenbank `template0` diente als Vorlage für `template1`. Sie stellt daher die "Ur-Datenbank"dar, akzeptiert standardmässig keine Verbindungen und wird nur im Notfall zur Wiederherstellung von `template1` verwendet.

postgres Die Datenbank `postgres` ist eine Beispieldatenbank, die von `template1` beim Anlegen des Datenbankclusters kopiert wurde. Sie hat den selben Status wie z.B. die Datenbank `test` eines MySQL-Servers und kann auf einem Produktivsystem gelöscht werden.

Weitere Datenbanken werden mit dem Befehl `CREATE DATABASE` angelegt:

```
CREATE DATABASE shop WITH OWNER = benutzer2;
```

Dem User `benutzer2` gehört somit die Datenbank `shop` und besitzt implizit das Recht, in dieser Datenbank Tabellen anzulegen. Auch hier ist das Füllwort `WITH` optional. Per `GRANT` und `REVOKE` können einem Benutzer Rechte zugesprochen bzw. entzogen werden:

```
GRANT SELECT ON studenten TO benutzer1;
```

Der Datenbankuser `benutzer1` erhält hiermit das Recht zum Abrufen von Datensätzen aus der Tabelle `studenten`. Neben `SELECT` können alle oben erläuterten SQL-Befehle, eine beliebige Kombination dieser Befehle oder das subsummierende Schlüsselwort `ALL` als Berechtigungsstufe vergeben werden:

```
GRANT UPDATE,DELETE ON studenten TO benutzer3;
GRANT ALL ON studenten TO benutzer4;
```

Analog setzt man `REVOKE` ein, um Rechte zu entziehen:

```
REVOKE ALL ON studenten FROM benutzer3;
REVOKE INSERT ON studenten FROM benutzer5;
```

Rechte werden nicht nur auf Tabellenebene vergeben, sondern auch auf Schemata und Datenbanken. Ein Schema kann als Container oder auch Namensraum (*namespace*) innerhalb einer Datenbank interpretiert werden. Eine Relation ist genau einem Schema zugeordnet (s. Abbildung 11). Berechtigungen zum Kreieren einer Tabelle werden daher an das jeweilige Schema geknüpft. Der Name eines Schemas wird als Prefix vor einem Tabellennamen angegeben:

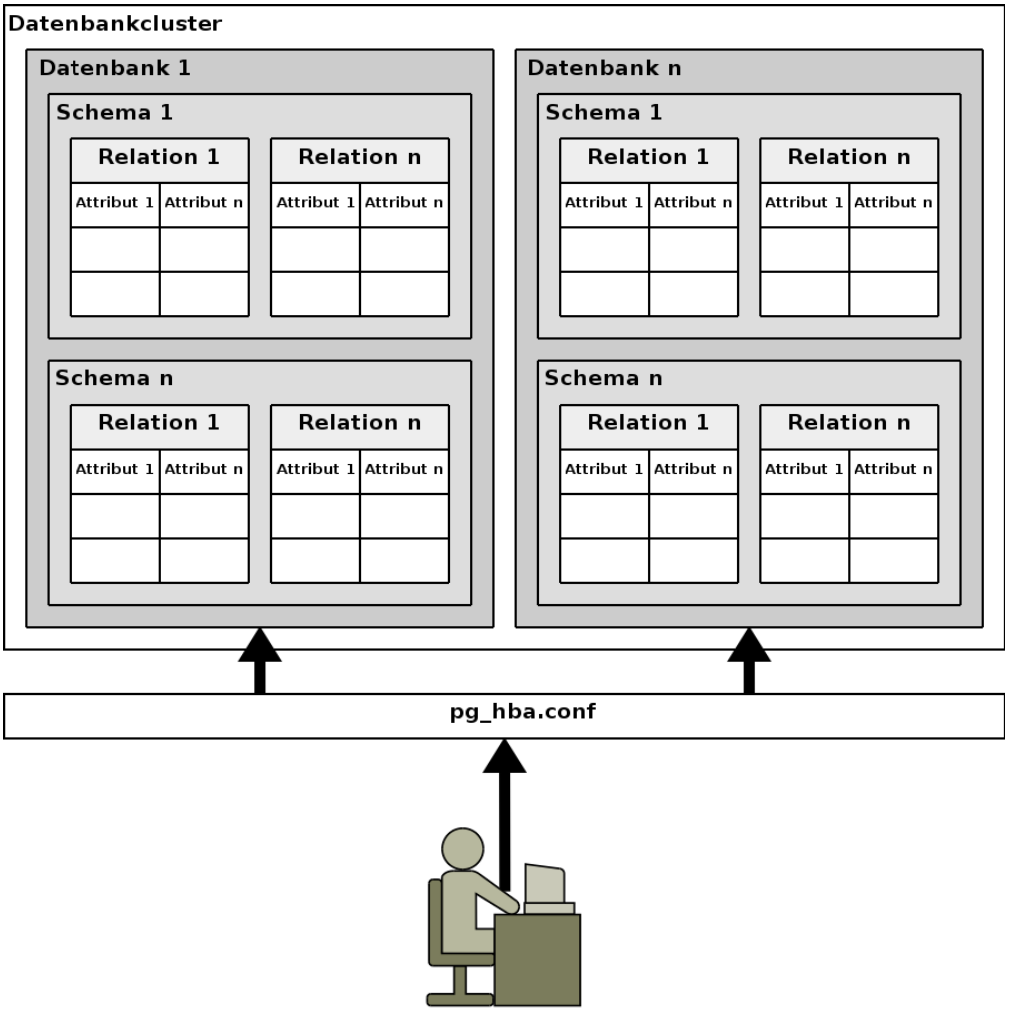

Abbildung 11: Der schematische Aufbau eines PostgreSQL-Datenbankclusters

```
SELECT * FROM schema1.studenten;
```

Lässt man den Schemanamen aus, so wird implizit das Schema `public` verwendet. Daher sind die beiden folgenden Anweisungen äquivalent:

```
DELETE FROM public.studenten WHERE matrikelnummer > 1000000;
DELETE FROM studenten WHERE matrikelnummer > 1000000;
```

Per `GRANT CREATE` wird generell das Recht zum Anlegen von Schemata oder Tabellen vergeben:

```
GRANT CREATE ON SCHEMA schema1 TO benutzer3;
```

Der User `benutzer3` erhält somit das Recht, Tabellen im Schema `schema1` anzulegen. Analog kann ihm dieses Recht wieder entzogen werden:

```
REVOKE CREATE ON SCHEMA schema1 FROM benutzer3;
```

Um Schemata anzulegen, benötigt man das `CREATE`-Recht auf Datenbankebene:

```
GRANT CREATE ON DATABASE shop TO benutzer3;
```

Der Besitzer einer Datenbank verfügt implizit über die Rechte zum Anlegen von Schemata und Relationen. Die meisten Anwendungen wie auch OTRS verlagern Berechtigungs-

hierarchien nicht in die Datenbank, sondern greifen mit nur einem Datenbankbenutzer auf Relationen zu und implementieren eigene Zugriffsrechte.

8. OTRS

8.1. Installation

Die Installation von OTRS gestaltet sich aufgrund des Portssystems von FreeBSD relativ einfach. So werden u.a. Perlmodule zum Ansprechen der Datenbank und zum Parsen und Erzeugen von Emails automatisch installiert. Abbildung 12 zeigt das Optionsmenü beim obligatorischen Aufruf von `make config install clean` im Verzeichnis `/usr/ports/devel/otrs`. Hier wurde die Unterstützung von PostgreSQL-Datenbanken

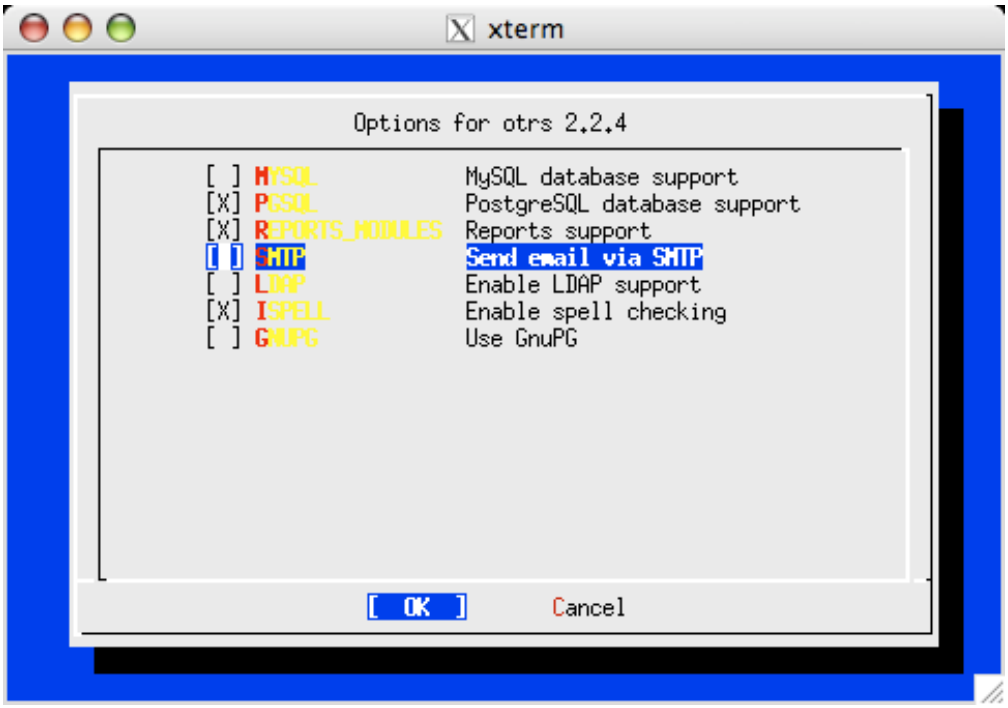

Abbildung 12: Das Optionsmenü zur Installation von OTRS

aktiviert, die von MySQL-Datenbanken deaktiviert. OTRS kann optional Emails direkt per SMTP versenden. Dies wurde ebenfalls abgewählt, da der lokale Postfix verwendet werden soll. Während der Installation wird der Systembenutzer `otrs` sowie die Gruppe `otrs` angelegt. Dies ist notwendig, da OTRS periodisch ausgeführte Programme, sogenannte *cronjobs* benötigt. Aus Sicherheitsgründen sollten diese nicht mit den Rechten des Administrators, sprich Root-Rechten laufen. Nach der Installation beinhaltet das Homeverzeichnis `/usr/local/otrs` einer OTRS-Installation 4 Unterverzeichnisse:

Kernel Hier liegen die Konfigurationsdateien, sämtliche Perlmodule und HTML-Templates für die Weboberfläche

bin Skripte zur Administration sind im Verzeichnis `bin` hinterlegt

scripts Hier sind Konfigurationsbeispiele für den Apache Webserver, diverse Testskripte sowie SQL-Anweisungen zum Anlegen einer OTRS-Datenbankinstanz gespeichert

var In den Unterverzeichnissen von `var` werden zur Laufzeit temporäre Dateien abgelegt; im Verzeichnis `var/httpd` liegen ferner Bilder bzw. Icons für die Weboberfläche

Ausserhalb des Homeverzeichnisses legt OTRS keine Dateien an (abgesehen von der Datenbank, die im Verzeichnis `/var/db/pgsql` liegt). Die Datenbank für eine OTRS-Instanz wird in 2 Schritten angelegt:

1. Als Datenbankadministrator legt man den Benutzer `otrs` an und die ihm gehörende Datenbank `otrs` an. Hierzu ruft man das Kommandozeilenprogramm `psql` mit den Parametern `postgres pgsql` auf, um sich als DB-Admin `pgsql` auf die Datenbank `postgres` zu verbinden. Anschliessend setzt man die folgenden SQL-Befehle ab:

   ```
   CREATE USER otrs PASSWORD 'geheim';
   CREATE DATABASE otrs OWNER otrs;
   ```

2. Struktur und initiale Daten werden mit 3 SQL-Skripten in die neu angelegte Datenbank geschrieben. Diese liegen im Verzeichnis `scripts/database` und müssen in der angegebenen Reihenfolge mit dem Kommandozeilenprogramm `psql` eingespielt werden:

   ```
   psql -f otrs-schema.postgresql.sql otrs otrs
   psql -f otrs-initial_insert.postgresql.sql otrs otrs
   psql -f otrs-schema-post.postgresql.sql otrs otrs
   ```

   Die Datei `otrs-schema.postgresql.sql` enthält alle Relationen der OTRS-Datenbank. Basisdaten wie z.B. der Administrator des OTRS oder Emailtemplates werden mit `otrs-initial_insert.postgresql.sql` geschrieben. Die Datenbank wird mit Constraints aus `otrs-schema-post.postgresql.sql` kompletiert.

Das Passwort des OTRS-Administrators `root@localhost` hat per default den Wert `root`. Dies sollte auf jeden Fall mit dem Skript `bin/otrs.setPassword` geändert werden:

```
$ bin/otrs.setPassword root@localhost neues_Passwort
```

8.2. Administration

Im Unterverzeichnis `bin` des OTRS-Homeverzeichnisses befinden sich Skripte zur Administration einer OTRS-Instanz, von denen die gebräuchlichsten nachfolgend beschrieben werden:

CheckDB.pl überprüft, ob die Datenbank konnektiert werden kann und ob in einer durch die Installation angelegten Systemtabelle Werte vorhanden sind

Cron.sh legt die *crontab* für den Systembenutzer der OTRS-Instanz an, mit der der Systemdienst *cron* periodisch Wartungsaufgaben ausführt. Das Skript setzt die crontab aus Textdateien im Verzeichnis `var/cron` zusammen. Dateien, deren Name auf `.dist` enden, werden nicht beachtet. Es handelt sich dabei um entsprechende Vorlagen

GenericAgent.pl wird periodisch als cronjob ausgeführt und versendet z.B. Emails im Falle von eskalierten Tickets. Im Verzeichnis `Kernel/System/GenericAgent` können hierfür selbst erstellte Perlmodule installiert werden

PendingJobs.pl wird ebenfalls als cronjob ausgeführt und setzt Tickets, die sich in einem der Stati vom Typ *PendingAuto* befinden, in den jeweils nächsten Status

PostMaster.pl nimmt Emails von der Standardeingabe entgegen und speichert sie in der Datenbank. Es sollte i.d.R. mit dem Parameter `-t 0` aufgerufen werden, damit etwaige Header in der Email nicht ausgewertet werden und so z.B. die Queue, in die die Email einsortiert wird, nicht vorgegeben werden kann

SetPermissions.sh passt die Berechtigungen aller Dateien und Verzeichnisse unterhalb des OTRS-Homeverzeichnisses an. Es sollte immer mit den folgenden 5 Parametern aufgerufen werden:

- vollständige Pfadangabe des OTRS-Homeverzeichnisses
- Name des Systembenutzers der OTRS-Instanz
- Name des Systembenutzers, mit dessen Rechten der Webserver läuft
- Name der Systemgruppe der OTRS-Instanz
- Name der Systemgruppe, mit dessen Rechten der Webserver läuft

UnlockTickets.pl entsperrt bei Aufruf mit dem Parameter `--timeout` alle Tickets, die länger als die pro Queue eingestellte Zeitspanne von einem Bearbeiter gesperrt sind, oder bei Aufruf per `--all` alle gesperrten Tickets

opm.pl dient zum Einspielen, Deinstallieren und Erzeugen von Modulpaketen

otrs.checkModules überprüft, ob alle von OTRS benötigten Perlmodule installiert sind

otrs.setPassword setzt das Passwort für einen OTRS-Benutzer (wie oben gezeigt)

8.3. Module

OTRS ist modular aufgebaut. Die Perlmodule unterhalb des Verzeichnisses `Kernel` abstrahieren

- den Datenbankzugriff
- die Funktions- oder auch *Business*-Logik
- die auf Templates basierende Webseitendarstellung
- den Zugriff auf die Weboberfläche.

Abbildung 13 zeigt den schematischen Aufbau einer OTRS-Instanz. Die Module zum Zugriff auf die Webseiten liegen im Verzeichnis `Kernel/System/Web`. Ihr Name beginnt per Konvention mit `Interface`. So stellt z.B. `InterfaceAgent.pm` den Zugriff auf die Weboberfläche für Bearbeiter zur Verfügung. Setzt man ein standardmässiges OTRS ein, so werden diese Zugriffsmodule vom Webserver über CGI-Skripte angesprochen, die in `scripts/cgi-bin` unterhalb des OTRS-Homeverzeichnisses liegen. Um den bei jedem Seitenaufruf notwendigen CGI-Prozess zu umgehen, wurde das Modul `DTSWeb.pm` entwickelt, welches beim Starten in den Webserver geladen wird. Die eigentliche Weboberfläche wird von Modulen im Verzeichnis `Kernel/Modules` aufgebaut. Sie nehmen etwaige, vom Anwender per Formular übergebene Daten an und speichern sie mit Hilfe der Businesslogik in die Datenbank. Analog erzeugen sie mit über die Businesslogik abgerufenen Werten aus der Datenbank und Webseitenvorlagen die Oberfläche. Die Templates liegen in Verzeichnissen unterhalb von `Kernel/Output/HTML`. Die Namen dieser Verzeichnisse entsprechen dem Namen des eingestellten *Theme*. Wird ein Template nicht gefunden, so wird per default im Verzeichnis `Kernel/Output/HTML/Standard` gesucht. Zusätzlich befinden sich im Verzeichnis `Kernel/Languages` Perlmodule für verschiedene Sprachen, z.B. `de.pm` für die Darstellung in deutsch. Diese Sprachsets können einfach erweitert bzw. angepasst werden, indem man dort Module wie `de_Custom.pm` oder `en_Custom.pm` hinterlegt. Das Verzeichnis `Kernel/System` stellt die Funktionslogik einer OTRS-Instanz dar. Sie greift über Module in `Kernel/System/DB` auf die Datenbank

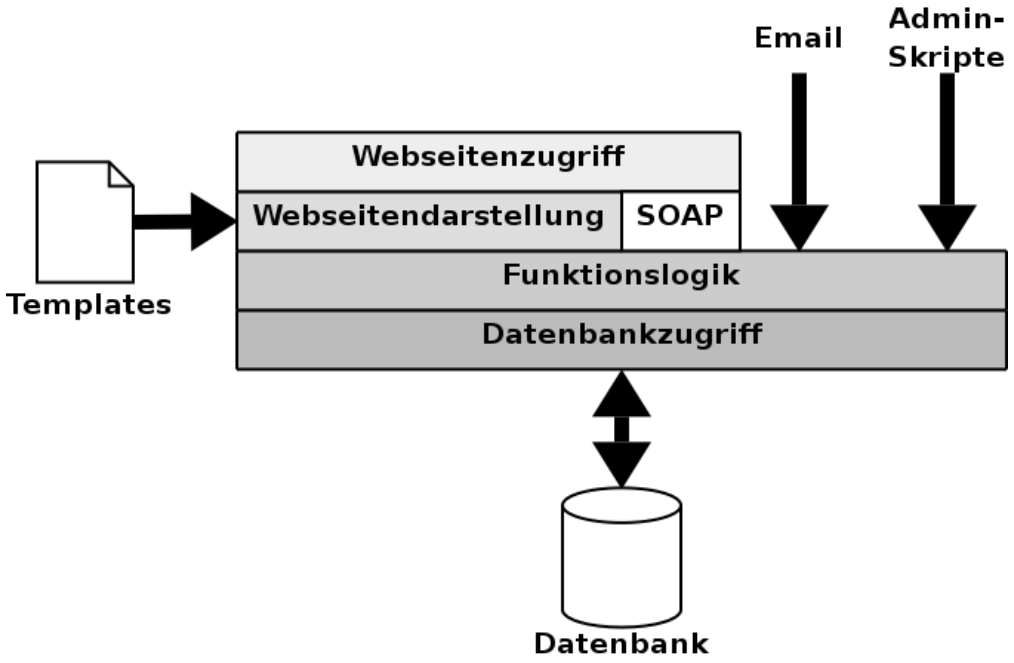

Abbildung 13: Der schematische Aufbau von OTRS

Nicht dargestellt ist der schreibende Zugriff auf Templates (s. Kapitel 9.4) und der Versand von Emails aus dem System

zu. In den Verzeichnissen `Kernel/System/Auth` und `Kernel/System/CustomerAuth` liegen Module zur Authentifizierung von Bearbeitern bzw. Kunden. Sie können gegen die Datenbank, LDAP-Verzeichnisse, *Remote Authentication Dial-In User Service* (RADIUS)-Server oder andere Webserver authentifiziert werden. OTRS verfügt über mindestens zwei Konfigurationsdateien. Nacheinander werden `Kernel/Config/Defaults.pm`, alle optionalen Perlmodule in `Kernel/Config/Files` und die Datei `Kernel/Config.pm` geladen. Dabei überschreibt bzw. ergänzt eine nachfolgende Datei die jeweils vorangegangenen. Das System kann auch über die Weboberfläche konfiguriert werden. Das mitgelieferte Frontendmodul *SysConfig* beschreibt jedoch nicht die erwähnten Perlmodule, sondern XML-Dateien, aus denen die Datei `Kernel/Config/Files/ZZZAAuto.pm` erzeugt wird. Diese wird wie oben erläutert zwischen der `Defaults.pm` und der `Config.pm` ausgewertet. Die `Defaults.pm` sollte man nicht verändern, da sie bei einem Update des OTRS ggf. geändert wird. Konfigurationsoptionen werden in der `Kernel/Config.pm` nicht durch reines Auflisten notiert, sondern als Membervariablen der Klasse `Kernel::Config` in der Methode `Load` definiert, z.B.:

```
sub Load ()
{
    my $Self = shift;

    # ...

    $Self->{'FQDN'} = 'fjo-otrs.dts-online.net';
    $Self->{'AdminEmail'} = 'fjo@dts.de';
    $Self->{'Organization'} = 'DTS Service GmbH';

    # ...
}
```

8.4. Modulprogrammierung

Die Perlmodule des OTRS stellen jeweils eine separate Klasse dar. Ausnahmen sind lediglich einige Hilfsmodule, die nur eine spezielle Aufgabe übernehmen, wie z.B. Module zum Generieren von Ticketnummern. Ein typisches Modul hat folgenden Aufbau:

```
#!/usr/bin/perl

package Kernel::System::Demomodul;

use strict;
use warnings;
use Kernel::System::MyModule;
# ... weitere Module einbinden ...

our $VERSION = "1.0";

sub new ()
{
    my $Type = shift;
    my %Param = @_;
    my $Self = {};

    bless ($Self, $Type);

    # adopt all provided objects
    foreach (keys %Param) {
        $Self->{$_} = $Param{$_};
    }

    # check needed stuff
    foreach (qw(LogObject TimeObject ConfigObject)) {
        if (!$Self->{$_}) {
            die "Got no $_!";
        }
    }

    # create additional objects
    $Self->{MyModuleObject} = Kernel::System::MyModule->new(%Param);

    return $Self;
}

# ... Memberfunktionen ...

1;
```

Neben den beiden Modulen `strict` und `warnings` müssen alle Packages bzw. Klassen eingebunden werden, von denen das gezeigte `Demomodul` Objekte instanziert. Der Konstruktor `new` segnet bzw. klassifiziert zunächst den leeren Hash, auf den die Referenz `$Self` verweist. Alle übergebenen Parameter werden in den (temporären) Hash `%Param` übernommen und dann als Membervariablen im neuen Objekt gespeichert. In der Regel enthalten die Parameter ausschliesslich Objekte und keine Werte wie Zeichenketten o.ä. So erwartet der Konstruktor im Beispiel, dass er in den Parametern `LogObject`, `TimeObject` und `ConfigObject` Instanzen der Klassen `Kernel::System::Log`, `Kernel::System::Time` respektive `Kernel::Config` erhält. In Srinivasan (1997) (S. 136 - 137) bzw. in Srinivasan (1999) (S. 147 - 148) wird diese Art der Objektübergabe *composition* bzw. *Komposition* genannt. Sie kann als einfache und lockere Variante zu einer rigiden Klassenhierarchie angesehen werden, die normalerweise in der objektorientierten

Programmierung konstruiert wird. Welche Objekte einem Konstruktor übergeben werden, kann nur durch Studium des Quellcodes von mitgelieferten Modulen und Skripten festgestellt werden. So erzeugen die Module im Verzeichnis `Kernel/System/Web`, welche für den Zugriff über die Weboberfläche verantwortlich sind, mindestens Instanzen der Klassen

- `Kernel::Config`
- `Kernel::System::Log`
- `Kernel::System::Main`
- `Kernel::System::Time`
- `Kernel::System::DB`.

Per Konvention setzt sich der Parametername aus dem letzten Teil des Klassennamens und dem String `Object` zusammen. Benötigt eine Instanz Objekte, die ihr nicht übergeben werden, so müssen sie im Konstruktor angelegt werden, wie `MyModuleObject` im Beispiel. Memberfunktionen erhalten die jeweilige Objektreferenz (`$Self`) als ersten Parameter und können so auf alle Objekte zugreifen. Scheitert ein Methodenaufruf, so wird dies entweder durch den Rückgabewert von `undef`, was durch ein parameterloses `return` geschieht, oder durch den sofortigen Abbruch mittels `die()` angezeigt. Zusätzlich kann ein Logeintrag mit der Methode `Log` aus der Klasse `Kernel::System::Log` bzw. dessen Instanz `LogObject` geschrieben werden.

8.5. Templates

Die Weboberfläche des OTRS basiert auf Templates. Diese Vorlagen liegen in Verzeichnissen unterhalb von `Kernel/Output/HTML`. Die Dateinamen müssen auf `.dtl` enden. Jedes Template stellt nur einen Teil der fertigen Oberfläche dar. So existiert eine Vorlage nur für den Kopfbereich jeder Seite, eine Vorlage für die Navigationsleiste, eine Vorlage zur Eingabe von Ticketdaten, eine Vorlage zur Anlage eines neuen Benutzers, usw. Daher bestimmt jedes Frontendmodul (im Verzeichnis `Kernel/Modules`) die äussere Struktur einer Seite selbst. Innerhalb der Templates können einzelne Bereiche markiert werden und so in der Webseite gezielt ausgeblendet oder aber mehrfach verwendet werden. Derartige Blöcke werden mit Kommentaren in HTML definiert, z.B.:

```
<!-- dtl:block:Hinweis -->
<p>Hinweis: $Data{"Warnung"}</p>
<!-- dtl:block:Hinweis -->
```

Die Klasse `Kernel::Output::HTML::Layout` stellt die Methode `Block` bereit, mit der solche Blöcke wie `dtl:block:Hinweis` aktiviert werden können und mit der Variablen wie `$Data{"Warnung"}` ein Wert zugewiesen wird:

```
$LayoutObject->Block(
    Name => "Hinweis",
    Data => {
        Warnung => "Ein Fehler ist aufgetreten."
    }
);
```

Somit wird in der Webseite ein Abschnitt mit dem entsprechenden Hinweis bzw. der Warnung angezeigt. Jedes Frontendmodul muss eine Methode `Run` definieren. Diese baut die Webseite auf und hat i.d.R. folgende Struktur:

```
sub Run ()
{
    my $Self = shift;
```

```
    my $LayoutObject = $Self->{LayoutObject};
    my $ParamObject = $Self->{ParamObject};
    my $MyModuleObject = $Self->{MyModuleObject};

    # fetch submitted form data
    my %Data = ();
    foreach (qw(Subaction ... )) {
        $Data{$_} = $ParamObject->GetParam(Param => $_) || "";
    }

    # start HTML output
    my $Output = $LayoutObject->Header(Title => "Frontendtest");
    $Output .= $LayoutObject->NavigationBar();

    # evaluate action
    if ($Data{Subaction} eq "Speichern") {
        $MyModuleObject->Schreiben(%Data);
        $Output .= $LayoutObject->Block(
            Name => "Hinweis",
            Data => {
                Warnung => "Daten wurden gespeichert."
            }
        );
    }
    else
        %Data = $MyModuleObject->Lesen(%Data);
    }

    # apply template
    $Output .= $LayoutObject->Output(
        TemplateFile => "MyTemplate",
        Data => \%Data
    );

    $Output .= $LayoutObject->Footer();

    return $Output;
}
```

Die Objekte `$LayoutObject`, `$ParamObject` und `$MyModuleObject` werden dem Konstruktor des Frontendmoduls übergeben bzw. dieser muss sie anlegen. `$ParamObject` ist eine Instanz von `Kernel::System::Web::Request` und verfügt über die Methode `GetParam`, mit der etwaige Formulardaten abgerufen werden können. Per default liest das Frontendmodul mit Hilfe eines Objektes der Businesslogik (hier: `$MyModuleObject`) Werte aus der Datenbank und stellt sie dem Anwender dar. Werden jedoch Formulardaten übergeben (angezeigt durch den Parameter `Subaction`), speichert sie das Modul in der Datenbank. Hält man die Namen von Datenbankfeldern und Formularparametern identisch, so kann der Datenhash (`%Data`) ohne Modifikation als Argument für Methoden der Businesslogik und für Layoutmethoden verwendet werden. Optionale Templateelemente wie z.B. `Hinweis` müssen vor dem finalen Aufruf der Methode `Output` per `Block` aktiviert werden.

9. Entwickelte Module

9.1. DTSTicketNumber

9.1.1. Beschreibung

Das Modul *DTSTicketNumber* erzeugt Ticketnummern in einem eigenen, mindestens neunstelligen Format. Es setzt sich zusammen aus der eindeutigen System-ID einer OTRS-Instanz, dem Jahr, dem Monat, dem Tag und einem mindestens zweistelligen Zähler, der für jedes neue Ticket inkrementiert, bei Beginn eines neuen Tages jedoch auf Null gesetzt wird. Für das 18. Ticket am 22. Dezember 2007 ergibt sich für die OTRS-Instanz mit der System-ID 3 daher die Ticketnummer `307122218`. Ein derartiges Modul muss die beiden Funktionen `TicketCreateNumber`, welche eine neue Ticketnummer als Zeichenkette liefert, und `GetTNByString` implementieren, die aus einer Betreffzeile einer Email eine etwaig vorhandene Ticketnummer extrahiert und zurückliefert. Da OTRS datenbankunabhängig ausgelegt ist, jedoch kein allgemeiner Standard zum expliziten Sperren einer Datenbank bzw. Relation existiert, greifen Module zum Generieren von Ticketnummern auf die Textdatei `var/log/TicketCounter.log` zu, um so mittels einer Dateisperre den Zähler atomar setzen zu können. Der Kern der Funktion `TicketCreateNumber` hat folgenden Aufbau:

```
1  if (!open(COUNTER, "+<$CounterLog")) {
2      # Ticketlog existiert noch nicht
3      if (!open(COUNTER, ">>$CounterLog")) {
4          # Fehler
5      }
6
7      close(COUNTER);
8
9      if (!open(COUNTER, "+<$CounterLog")) {
10         # Fehler
11     }
12 }
13
14 if (!flock(COUNTER, 2)) {
15     # Fehler
16 }
17
18 # Zähler einlesen
19 my $TicketNumber = <COUNTER>;
20
21 $TicketNumber++;
22
23 if (!truncate(COUNTER, 0)) {
24     # Fehler
25 }
26
27 if (!seek(COUNTER, 0, 0)) {
28     # Fehler
29 }
30
31 print COUNTER $TicketNumber;
32
33 close(COUNTER);
```

In Zeile 1 wird versucht, die Logdatei lesend und schreibend zu öffnen (Operator `+<`). Schlägt dies fehl, wird die Datei nur zum Beschreiben geöffnet und der Dateideskriptor an das Dateiende gesetzt (Operator `>>`). Somit ist in Zeile 9 sichergestellt, dass die Logdatei existiert und kann erneut lesend und schreibend geöffnet werden. Der Aufruf von

`flock` stellt sicher, dass folgend nur ein OTRS-Prozess auf die Datei zugreift. Nach dem Einlesen und Inkrementieren des Zählerstandes wird die Logdatei auf die Länge von null Bytes gekürzt. Da der alte Zählerstand eingelesen wurde, steht der Filedeskriptor jedoch noch auf dem Dateiende. Der Aufruf von `seek` in Zeile 27 setzt ihn an den Dateianfang. Anschliessend wird der neue Zählerstand geschrieben. Der Aufruf von `close` schliesst den Dateideskriptor und entfernt gleichzeitig die Dateisperre.

9.1.2. Konfigurationsparameter

DTSTicketNumber verwendet folgende Konfigurationsparameter:

SystemID die für jede OTRS-Instanz eindeutige numerische Kennung

Ticket::CounterLog vollständige Pfadangabe der Datei, die den Zählerstand der Tiketnummer aufnimmt

Ticket::NumberGenerator::MinCounterSize Mindestgrösse des Tageszählers

Ticket::NumberGenerator::CheckSystemID gibt an, ob beim Erkennen einer Ticketnummer die System-ID beachtet werden soll

Ticket::Hook gibt das Prefix einer Ticketnummer an

Ticket::HookDivider gibt die Zeichenkette an, die zwischen `Ticket::Hook` und der Ticketnummer steht; hat z.B. `Ticket::Hook` den Wert `Ticket` und `Ticket::HookDivider` den Wert ”`: `”, so wird für die Ticketnummer `307122218` die Zeichenkette `Ticket: 307122218` als Betreffzeile einer Email verwendet.

9.2. DTSLib

9.2.1. Beschreibung

Das Modul *DTSLib* stellt drei häufig benötigte Dateisystemmethoden in der Klasse `Kernel::System::DTSLib` bereit:

MakeDirectories erwartet den Parameter `Directories`, der eine Referenz auf ein Array von Zeichenketten darstellen muss. Die Methode sieht jede Zeichenkette als Pfad zu einem Verzeichnis an und erstellt dieses, falls es noch nicht existiert

WriteVersionedFile erwartet die Parameter `FileName` und `Data`. Die Methode schreibt die übergebenen Daten sowohl in die durch `FileName` angegebene Datei als auch in eine Sicherheitskopie, deren Name sich aus dem übergebenen Dateinamen, dem aktuellen Datum sowie Benutzer-ID und Benutzername zusammensetzt

WriteFile erwartet die Parameter `FileName` und `Data`. Die Methode überschreibt die spezifizierte Datei atomar mit den übergebenen Daten.

Der Kern der Methode `WriteFile` hat folgenden Aufbau:

```
1  my $TempFile = $Param{FileName} . ".tmp";
2
3  if (!open(FH, ">$TempFile")) {
4      # Fehler
5  }
6  if (!flock(FH, 2)) {
7      # Fehler
8  }
9  if (!seek(FH, 0, 0)) {
10     # Fehler
11 }
12
13 print FH ${$Param{Data}};
14 close(FH);
15
16 if (!rename($TempFile, $Param{FileName})) {
17     # Fehler
18 }
```

Die Daten werden zunächst in eine temporäre Datei geschrieben. Durch den Aufruf von `flock` in Zeile 6 ist gewährleistet, dass nur ein Prozess auf diese temporäre Datei zugreift. Die Funktion `seek` setzt den Deskriptor an den Dateianfang. Die Systemroutine `rename` zum Umbenennen einer Datei arbeitet atomar. Somit ist sichergestellt, dass die vorhandene Datei bei Fehlern nicht gelöscht oder nur teilweise überschrieben wird.

9.2.2. Konfigurationsparameter

Das Modul DTSLib verwendet keine Konfigurationsparameter.

9.3. DTSFreetext

9.3.1. Beschreibung

Das Modul *DTSFreetext* versetzt den Administrator einer OTRS-Instanz in die Lage, für jede Queue sogenannte *Freitexte*[22] zu definieren. Diese Freitexte stellen Schlüssel-/Wertepaare dar, die bei jedem Ticket ausgefüllt werden müssen. So lässt sich erzwingen, dass bei jedem Ticket notwendige Daten wie z.B. die Seriennummer eines Gerätes vorhanden sind. Zusätzlich lassen sich Auswahllisten definieren, aus denen beim Erstellen eines Tickets eine Option ausgewählt werden muss. Die Verwendung solcher Freitext-

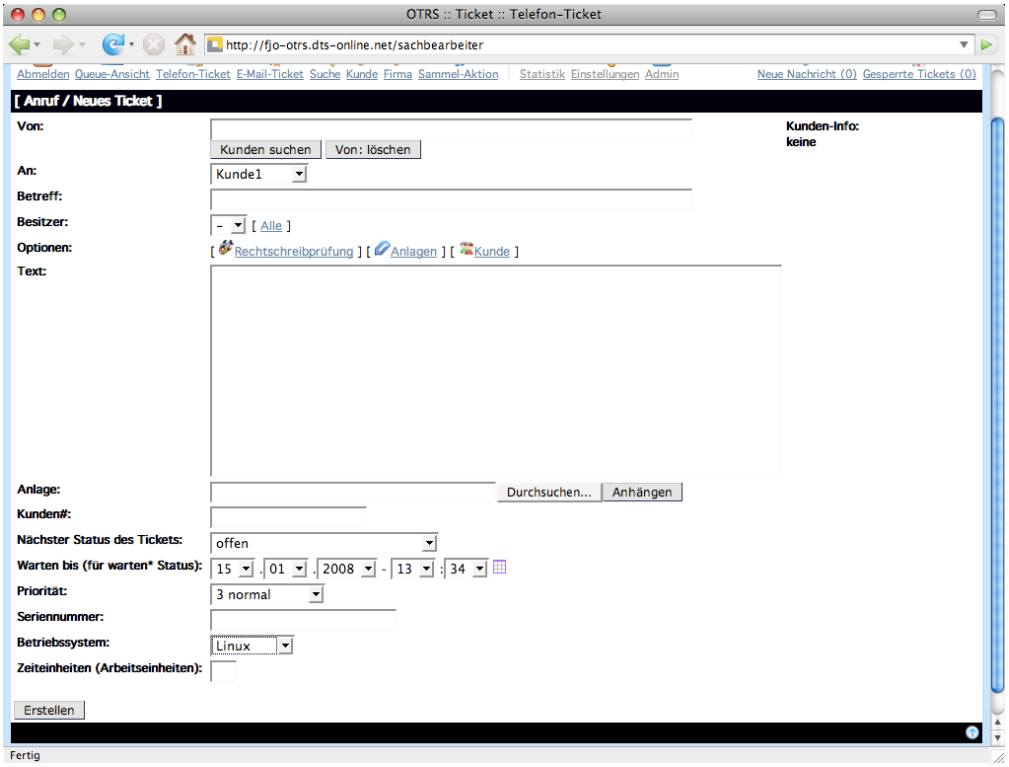

Abbildung 14: Freitextfelder beim Anlegen eines neuen Tickets

optionen ist jedoch global für eine Instanz anzusehen. OTRS bietet keine Möglichkeit, sie gezielt pro Queue ein- oder auszublenden. Daher verwendet das Modul DTSFreetext den Metawert "*(not used)*", wenn einer Queue keine expliziten Freitexte zugewiesen sind. Zudem ist auch der Typ einer Freitextoption (Eingabefeld oder Dropdown-Menü) global festgelegt und kann nicht pro Queue geändert werden. Das Modul besteht aus 5 Komponenten:

DTSFreetext.pgsql.sql beinhaltet die Relation `dts_queue2freetext`, die jeder Queue entsprechende Freitextoptionen zuordnet

DTSFreetext.pm abstrahiert den Zugriff auf die Relation `dts_queue2freetext` und stellt mit der Klasse `Kernel::System::DTSFreetext` die Methoden `FreetextList` zum Auslesen, `FreetextAdd` zum Hinzufügen und `FreetextModify` zum Aktualisieren von Freitextoptionen bereit

DTSFreetextAdmin.pm ist das Frontendmodul zum Modifizieren von Freitextoptionen

[22] Die Bezeichnung ist irreführend, wird jedoch so in der Dokumentation zu OTRS verwendet.

DTSFreetext.dtl stellt das Template für die Administrationsoberfläche dar

DTSFreetextAcl.pm ist das Kernstück dieses Moduls. OTRS ruft die in dieser Perldatei definierte Methode `Run` beim Anlegen oder Modifizieren eines Tickets auf und übergibt ihr die ID der Queue und optional die ID des Tickets. Mit diesen Werten wird die Tabelle `dts_queue2freetext` befragt und für diese Queue vorgegebene Freitextoptionen in Form einer *Access Control List* (ACL) zurückgeliefert. Mit dieser ACL überprüft OTRS, ob der Anwender die Freitextfelder ausfüllen muss.

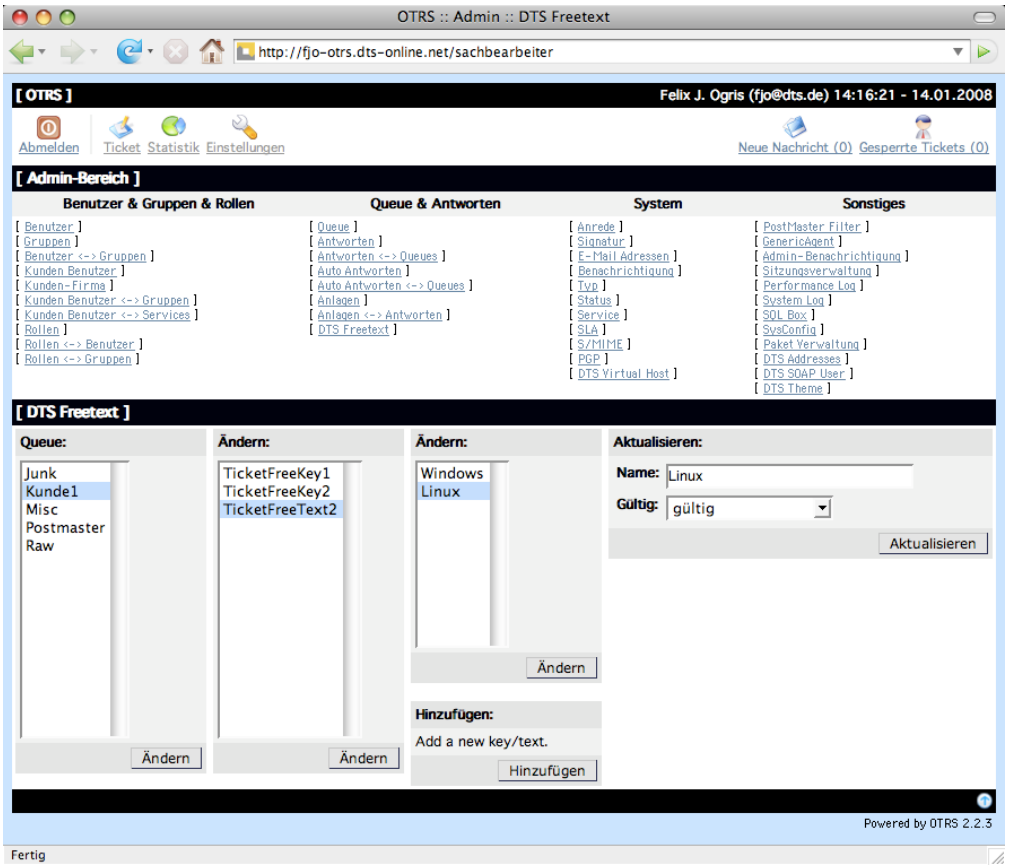

Abbildung 15: Administration der Freitextfelder

9.3.2. Konfigurationsparameter

Damit Freitextoptionen angezeigt werden, müssen diese in einer der Konfigurationsdateien wie `Kernel/Config.pm` aufgeführt werden. Um nicht bei jeder Änderung der Freitexte die gesamte Konfiguration des OTRS neu schreiben zu müssen, wird die Perlfunktion `do` verwendet. Sie bindet Dateien erst zur Laufzeit an ihrer Stelle ein:

```
$Self->{"DTSFreetext::TicketFreeText2"} = $Self->{"Home"}.
    "/var/dts_freetext/TicketFreeText2.txt";

$Self->{'TicketFreeText2'} = {
  "TicketFreeText2_(not_used)" => "TicketFreeText2_(not_used)",
  do $Self->{"DTSFreetext::TicketFreeText2"}
};
```

Der Parameter `DTSFreetext::TicketFreeText2` beinhaltet den Pfad zu einer Textdatei. Diese wird vom Modul `DTSFreetext.pm` bei jeder Änderung der Tabelle `dts_queue2freetext` neu geschrieben. Sie enthält den Rumpf eines Hashes, der Schlüssel-/Wertepaare darstellt. Die Datei wird beim Abruf des Parameters `TicketFreeText2` ausgewertet und formt so zusammen mit dem Defaulteintrag `(not used)` die möglichen Freitextoptionen. Das Modul `DTSFreetext` greift ferner auf die folgenden Konfigurationsparamter zu:

DTSFreetext::TicketFreeKey1, ... , DTSFreetext::TicketFreeKey16 müssen gültige Dateipfade darstellen und sollten den Wert `$Self->{"Home"}."/var/dts_freetext/TicketFreeKey1.txt"` usw. aufweisen. Sie beinhalten jeweils einen Hash und stellen die möglichen Namen für die erste, zweite, usw. Freitextoption dar

DTSFreetext::TicketFreeText1, ... , DTSFreetext::TicketFreeText16 müssen gültige Dateipfade darstellen und sollten den Wert `$Self->{"Home"}."/var/dts_freetext/TicketFreeText1.txt"` usw. aufweisen. Sie beinhalten jeweils einen Hash und stellen die möglichen Werte für die erste, zweite, usw. Freitextoption dar

TicketFreeKey1, ... , TicketFreeKey16 müssen anonyme Hashes darstellen. Sie beinhalten die möglichen Namen für die erste, zweite, usw. Freitextoption und müssen wie folgt definiert werden:

```
$Self->{'TicketFreeKey1'} = {
  "TicketFreeKey1 (not used)" => "TicketFreeKey1 (not used)",
  do $Self->{"DTSFreetext::TicketFreeKey1"}
};
```

TicketFreeText1, ... , TicketFreeText16 müssen anonyme Hashes darstellen. Sie beinhalten die möglichen Werte für die erste, zweite, usw. Freitextoption und müssen wie folgt definiert werden:

```
$Self->{'TicketFreeText1'} = {
  "TicketFreeText1 (not used)" => "TicketFreeText1 (not used)",
  do $Self->{"DTSFreetext::TicketFreeText1"}
};
```

Ferner muss die Datei `Kernel/Config.pm` um folgenden Eintrag ergänzt werden, damit die Administrationsseite für die Freitextoptionen im Webfrontend angezeigt wird:

```
$Self->{'Frontend::Module'}->{'DTSFreetextAdmin'} = {
    Group => [ 'admin' ],
    NavBarName => "Admin",
    NavBarModule => {
        Name => 'DTS Freetext',
        Block => 'Block2',
        Prio => 9999,
        Module => 'Kernel::Output::HTML::NavBarModuleAdmin',
    },
};
```

Ausserdem muss das Modul `DTSFreetextAcl.pm` entsprechend eingebunden werden:

```
$Self->{"Ticket::Acl::Module"} = {
    "DTSFreetextAcl" => {
       Module => "Kernel::System::Ticket::DTSFreetextAcl",
    }
};
```

9.4. DTSTheme

9.4.1. Beschreibung

Das Modul *DTSTheme* ermöglicht es dem Administrator einer OTRS-Instanz, die Standardansicht der Weboberfläche zu kopieren und diese Kopie nach seinen Wünschen anzupassen. Es handelt sich dabei um einen triviales Textfeld, mit dem die Templates geladen, bearbeitet und gespeichert werden können. Zusätzlich kann jedes dieser *Themes* mit einem eigenen *Favicon* versehen werden, welches in der Adressezeile eines Webbrowsers angezeigt wird.

9.4.2. Konfigurationsparameter

Das Modul greift auf folgende Konfigurationsparameter zu:

DTSTheme::ImagesDirectory sollte den Wert `$Self->{"Home"}."/var/httpd/htdocs/images"` aufweisen und gibt das Verzeichnis an, in dem Icons für Schaltflächen u.ä. liegen

DTSTheme::FaviconsDirectory sollte den Wert `$Self->{"Home"}."/var/httpd/htdocs/favicons"` aufweisen und gibt das Verzeichnis an, in dem die Favicons für die Weboberfläche liegen

DTSTheme::FaviconName sollte immer den Wert `favicon.ico` haben und gibt den Dateinamen eines Favicons an.

Die Administrationsoberfläche des Modules DTSTheme muss wie folgt in die Konfiguration in `Kernel/Config.pm` eingebunden werden:

```
$Self->{'Frontend::Module'}->{'DTSThemeAdmin'} = {
    Group => [ 'admin' ],
    NavBarName => "Admin",
    NavBarModule => {
        Name => 'DTS_Theme',
        Block => 'Block4',
        Prio => 9999,
        Module => 'Kernel::Output::HTML::NavBarModuleAdmin',
    },
};
```

Ferner muss der Konstruktor der Klasse `Kernel::Output::HTML::Layout` erweitert werden, damit auch Icons themebasiert angezeigt werden. Die Zeilen

```
    # define $Env{"Images"}
    $Self->{Images} = $Self->{ConfigObject}->Get('Frontend::ImagePath');
```

müssen durch

```
    # define $Env{"Images"}
    $Self->{Images} = $Self->{ConfigObject}->Get('Frontend::ImagePath'). \
        $Theme."/";
```

ersetzt werden.

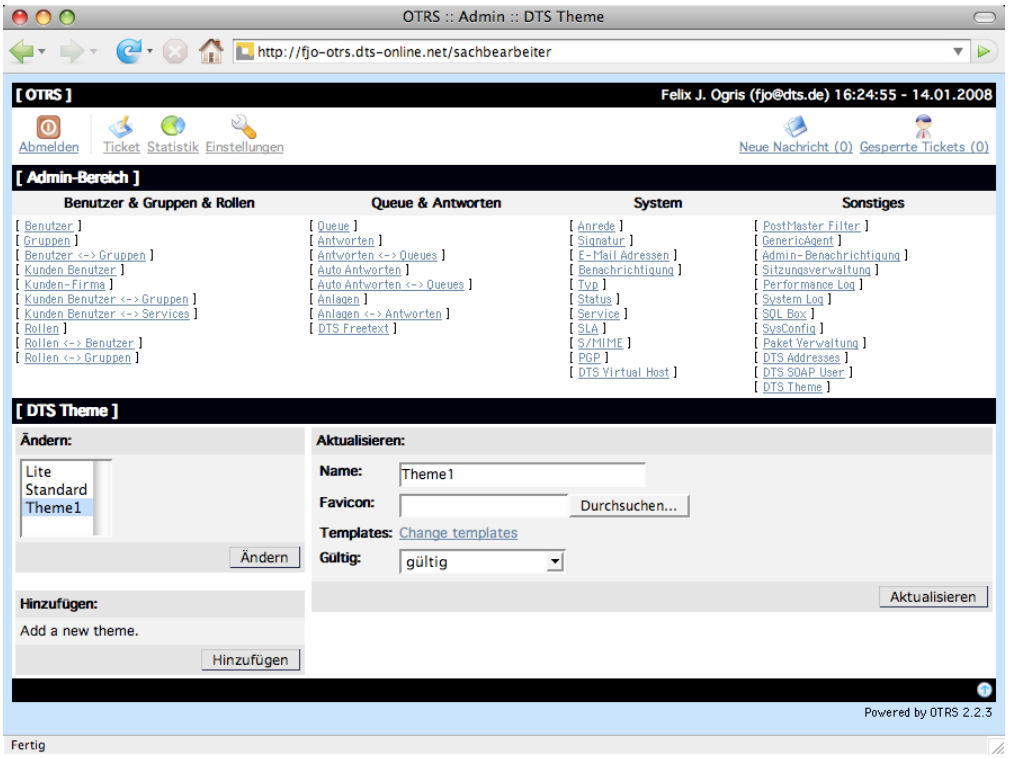

Abbildung 16: Administration der Themes

Ein solches Theme kann entweder mit dem Modul *DTSVirtualHost* als Defaulttheme für unterschiedliche Hosts eingesetzt werden oder von einem Anwender als persönliche Voreinstellung ausgewählt werden. Das Modul besteht aus vier Komponenten:

DTSTheme.pm abstrahiert den Zugriff auf Templates und stellt in der Klasse `Kernel::System::DTSTheme` die folgenden Methoden bereit:

- **WriteTemplate** zum Abspeichern eines Templates
- **ListTemplates** zum Lesen aller Templates für ein bestimmtes Theme
- **WriteFavicon** zum Abspeichern eines Favicons
- **ThemeList** zur Aufzählung aller Themes einer OTRS-Instanz
- **ThemeAdd** zum Hinzufügen eines Themes
- **ThemeModify** zum Abspeichern eines Themes.

DTSThemeAdmin.pm stellt das Frontendmodul zur Verwaltung der Themes dar

DTSTheme.dtl ist das Template für die Administrationsoberfläche (s. Abbildung 16)

DTSTemplateEditor.dtl stellt die Maske zur Bearbeitung der Templates eines Themes dar (s. Abbildung 17)

Jedes veränderte Template wird als Sicherheitskopie auf dem Server hinterlegt. Zudem ist es nicht möglich, die beiden mitgelieferten Themes *Standard* und *Lite* zu verändern.

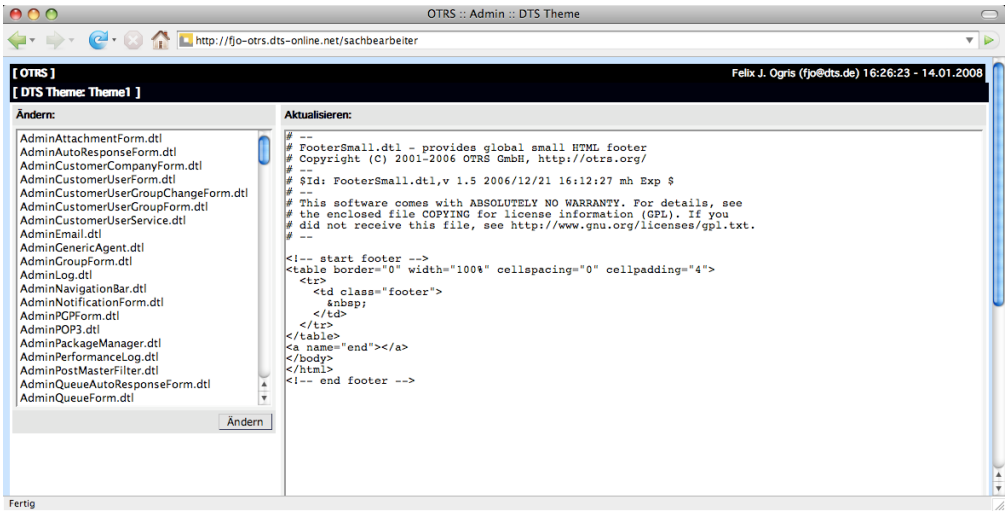

Abbildung 17: Modifikation eines Templates

9.5. DTSVirtualHost

9.5.1. Beschreibung

Mit dem Modul *DTSVirtualHost* kann eine OTRS-Instanz so eingerichtet werden, dass sie unter verschiedenen Hostnamen erreichbar ist. Dies betrifft jedoch nur die Konfiguration der OTRS-Instanz. Die Betriebsparameter des Apache Webservers o.ä. müssen durch ein weiteres Modul wie z.B. *DTSMaster* angepasst werden. DTSVirtualHost benötigt zum Betrieb das Modul *DTSTheme*, da jedem Hostnamen ein eigenes Theme zugeordnet werden kann. Das Modul besteht aus vier Komponenten:

DTSVirtualHost.pgsql.sql definiert die Relation `dts_virtual_host`, in der alle Hostnamen einer OTRS-Instanz hinterlegt sind

DTSVirtualHost.pm abstrahiert den Zugriff auf jene Relation und stellt mit der Klasse `Kernel::System::DTSVirtualHost` folgende Methoden bereit:

VirtualHostList listet alle Hosts einer OTRS-Instanz auf

VirtualHostAdd fügt einer OTRS-Instanz einen neuen Hostnamen hinzu

VirtualHostModify aktualisiert einen Host

Für jeden Host müssen neben seinem Namen folgende Attribute hinterlegt werden:

AgentUrl beschreibt die Adresse, unter der sich Bearbeiter von Tickets einloggen können

CustomerUrl beschreibt die Adresse, unter der Kunden ihre Tickets einsehen können

PublicUrl beschreibt die Adresse, unter der der öffentliche FAQ-Bereich zu finden ist

SoapUrl beschreibt die Adresse, unter der die SOAP-Schnittstelle abrufbar ist

Secure / HTTPS gibt an, ob für den Virtualhost durch das Modul *DTSMaster* ein SSL-Zertifikat erzeugt werden soll

IP Adresse Jeder per HTTPS geschützte Virtualhost muss unter einer eigenen IP Adresse abrufbar sein an. Auch bei Einsatz des Modules *DTSMaster* muss diese IP Adresse manuell im Betriebssystem konfiguriert werden und ggf. vorgeschalteten Firewalls oder Routern bekannt gemacht werden.

9.5.2. Konfigurationsparameter

Das Modul *DTSVirtualHost* greift auf zwei Konfigurationsparameter zu:

DTSVirtualHost::Host2ThemeFile gibt die Textdatei an, in der jedem Hostnamen der Instanz ein Theme zugeordnet wird, und sollte den Wert `$Self->{"Home"}."/var/httpd/host2theme.txt"` haben

DTSVirtualHost::SSLDirectory gibt das Verzeichnis an, in dem etwaige SSL-Schlüssel und -Zertifikate hinterlegt werden, und sollte den Wert `$Self->{"Home"}."/var/httpd/etc"` haben.

Analog zu *DTSFreetext* wird die Zuordnung von Hostnamen zu Themenamen als anonymer Hash in der Konfiguration erwartet. Daher kommt auch hier die Perlfunktion `do` zum Einsatz und lädt zur Laufzeit die unter dem Parameter `DTSVirtualHost::Host2ThemeFile` angegebene Textdatei:

```
$Self->{'DefaultTheme::HostBased'} = {
    do $Self->{"DTSVirtualHost::Host2ThemeFile"}
};
```

Die Konfigurationsoberfläche muss ebenfalls in die Datei `Kernel/Config.pm` eingebunden werden:

```
$Self->{'Frontend::Module'}->{'DTSVirtualHostAdmin'} = {
    Group => [ 'admin' ],
    NavBarName => "Admin",
    NavBarModule => {
        Name => 'DTS Virtual Host',
        Block => 'Block3',
        Prio => 9999,
        Module => 'Kernel::Output::HTML::NavBarModuleAdmin',
    },
};
```

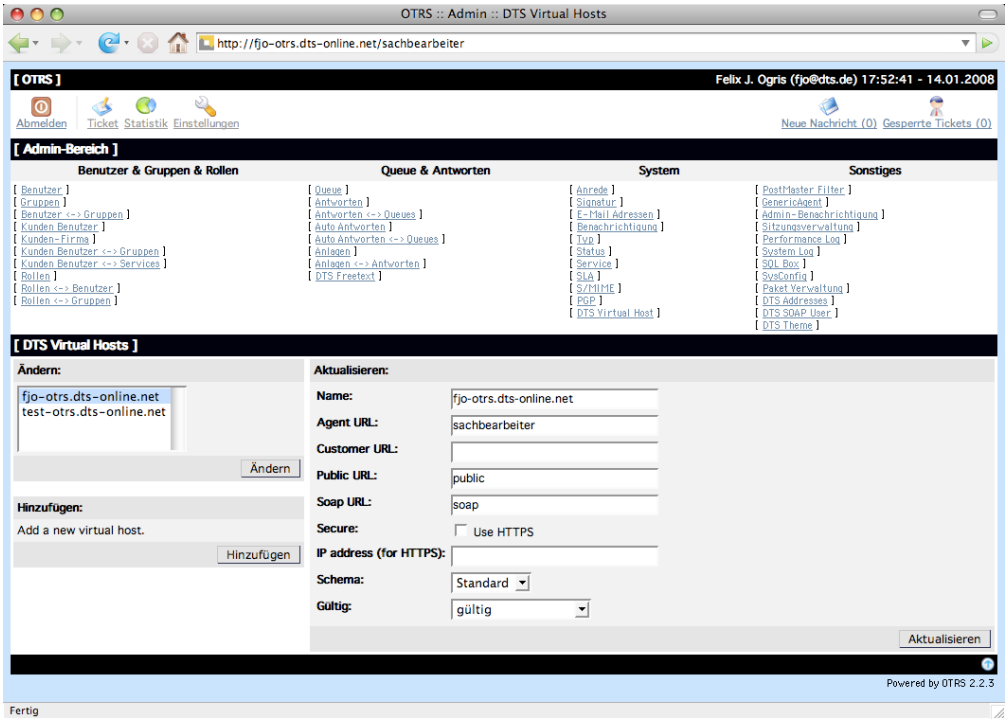

Abbildung 18: Administration der Virtualhosts

9.6. DTSMaster

9.6.1. Beschreibung

Das Paket *DTSMaster* dient zum Einrichten und Betrieb mehrerer OTRS-Instanzen auf einer gemeinsamen Plattform. Die Grundidee liegt im Einsatz eines weiteren Apache-Prozesses, der als Proxyserver betrieben wird. Er nimmt zunächst stellvertretend für alle OTRS-Instanzen Anfragen entgegen und reicht diese dann aufgrund des angefragten Hostnamens an den Webserver der jeweiligen OTRS-Instanz weiter. Anfragen an per HTTPS geschützte OTRS-Instanzen beantwortet der entsprechende Webserver hingegen direkt. DTSMaster greift nicht auf OTRS-Funktionen oder -Module zurück, da es nicht im Kontext einer OTRS-Instanz läuft. Das Paket umfasst die folgenden Komponenten:

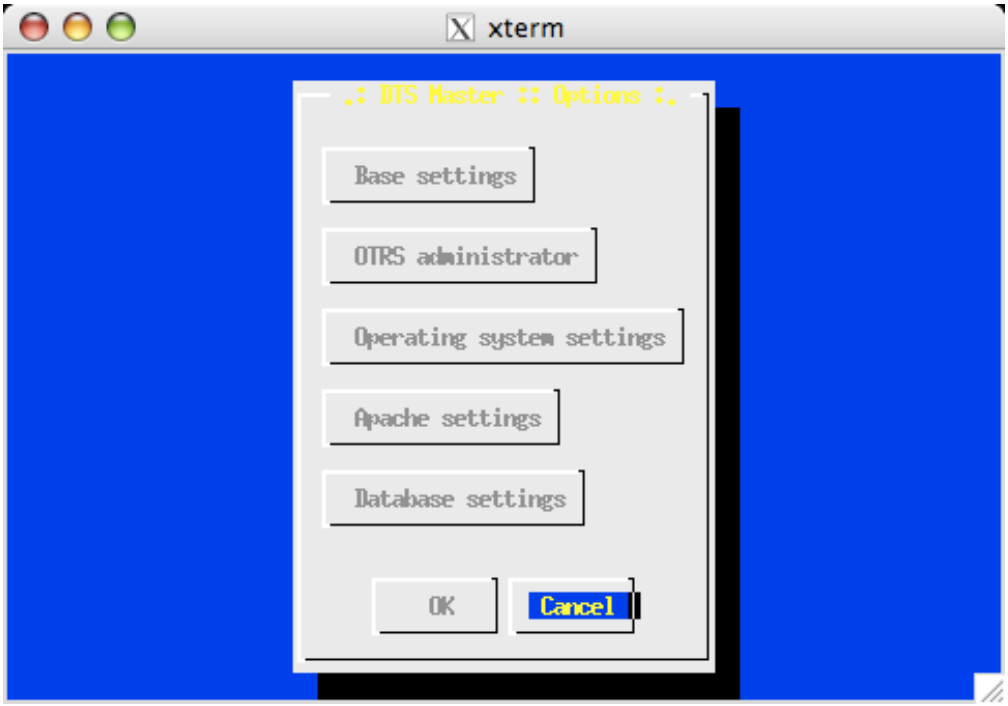

Abbildung 19: DTSMaster.pl zum Anlegen neuer OTRS-Instanzen

DTSMaster.pm ist die zentrale Bibliothek und stellt abstrahierende Funktionen zum Zugriff auf die PostgreSQL-Datenbank, zum Anlegen von Betriebssystembenutzern und -gruppen, und zum Anlegen[23] einer neuen OTRS-Instanz bereit. Da sämtliche Konfigurationen auf XSLT-Templates basieren, verwendet DTSMaster.pm die beiden Module `XML::LibXML` sowie `XML::LibXSLT`

DTSMaster.pl ist ein menügesteuertes Kommandozeilenprogramm zum Einrichten einer neuen OTRS-Instanz (s. Abbildung 19). Zur Darstellung der Menüs wird das eingebettete Package `DTSDisplay` verwendet, welches wiederum das externe Modul `Dialog` verwendet. DTSDisplay stellt einfache GUI-Elemente wie Eingabefehler oder Hinweisboxen bereit

DTSMasterCron.pl wird periodisch über den Systemdienst *cron* ausgeführt und schreibt Konfigurationsdateien für den lokalen Postfixserver und alle Apache-Instanzen (ins-

[23] Diese Funktion ist derzeit noch nicht vollständig, es fehlen das Befüllen der Datenbank und des Homeverzeichnisses der neuen Instanz mit Hilfe einer als Vorlage dienenden, ungenutzten OTRS-Installation.

besondere den Proxyserver). Ausserdem legt dieses Skript SSL-Zertifikate und -Schlüssel an. DTSMasterCron.pl greift auf DTSTheme und DTSVirtualHost von jeder OTRS-Instanz zurück. Die jeweiligen Dienste werden nur neu gestartet, wenn sich ihre Konfiguration geändert hat.

DTSMaster.sql beschreibt die Relation `dts_master`, in der alle OTRS-Instanzen einer Plattform verzeichnet sind

DTSWeb.pm stellt die Schnittstelle zwischen Webserver und OTRS dar. Es wird in jeder OTRS-Instanz verwendet, um unnötige CGI-Aufrufe zu vermeiden

dtspreload.pl wird wie DTSWeb.pm in jeder OTRS-Instanz eingesetzt, um bei Start des jeweiligen Webservers alle OTRS-Module zu laden.

Ferner beinhaltet das Paket *DTSMaster* ein angepasstes Startskript für den Apache Webserver, welches im Gegensatz zum Original das Starten mit den Rechten eines unpriviligierten Benutzers erlaubt.

9.7. DTSAddress

9.7.1. Beschreibung

Das Modul *DTSAddress* dient zur vereinfachten Weiterleitung von Tickets an andere OTRS-Instanzen. Es basiert auf den mitgelieferten Modulen `AgentTicketForward` und `AgentTicketEmail` sowie deren Templates. Diese Module verlangen jedoch, dass die Empfängeremailadresse entweder eingegeben oder aus einem Adressbuch heraus gesucht werden muss. DTSAddress bietet dem Administrator einer OTRS-Instanz jedoch die Möglichkeit, zentral ein Adressbuch zu pflegen. Dieses verknüpft die Namen von OTRS-Instanzen, an die Tickets delegiert werden sollen, mit deren Emailadressen. In

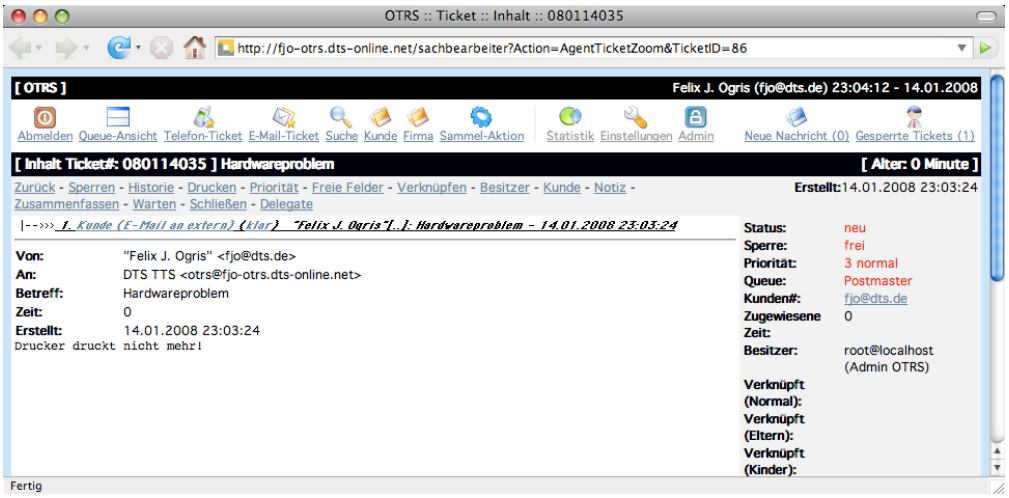

Abbildung 20: Ticketansicht mit der Schaltfläche zum Delegieren

der Ticketansicht hat ein Bearbeiter somit die Möglichkeit, die registrierten Instanzen ohne Umwege aus einem Dropdown-Menü auszuwählen. Das Modul besteht aus sechs Komponenten:

DTSAddress.pgsql.sql stellt die Relation `dts_address` bereit, die Namen von anderen OTRS-Instanzen deren Emailadressen zuordnet

DTSAddress.pm abstrahiert den Zugriff auf diese Relation und stellt mit der Klasse `Kernel::System::DTSAddress` folgende Methoden bereit:

- **AddressList** zum Auslesen von OTRS-Instanzen und deren Emailadressen
- **AddressAdd** zum Hinzufügen einer neuen OTRS-Instanz und deren Emailadresse
- **AddressModify** zum Aktualisieren einer OTRS-Instanz und deren Emailadresse

DTSAddressAdmin.pm stellt das Frontendmodul zum Administrieren von OTRS-Instanzen und deren Emailadressen dar

DTSAddress.dtl beinhaltet das Template für die Administrationsoberfläche

AgentTicketDelegate.pm stellt das Frontendmodul zur Delegation von Tickets dar. Es präsentiert dem Anwender eine Maske, in der das Zielsystem ausgewählt und der Inhalt des Tickets vor dem Absenden bearbeitet werden kann, versendet das Ticket und trägt eine entsprechende Notiz in der Tickethistorie ein

AgentTicketDelegate.dtl ist das Template für AgentTicketDelegate.pm.

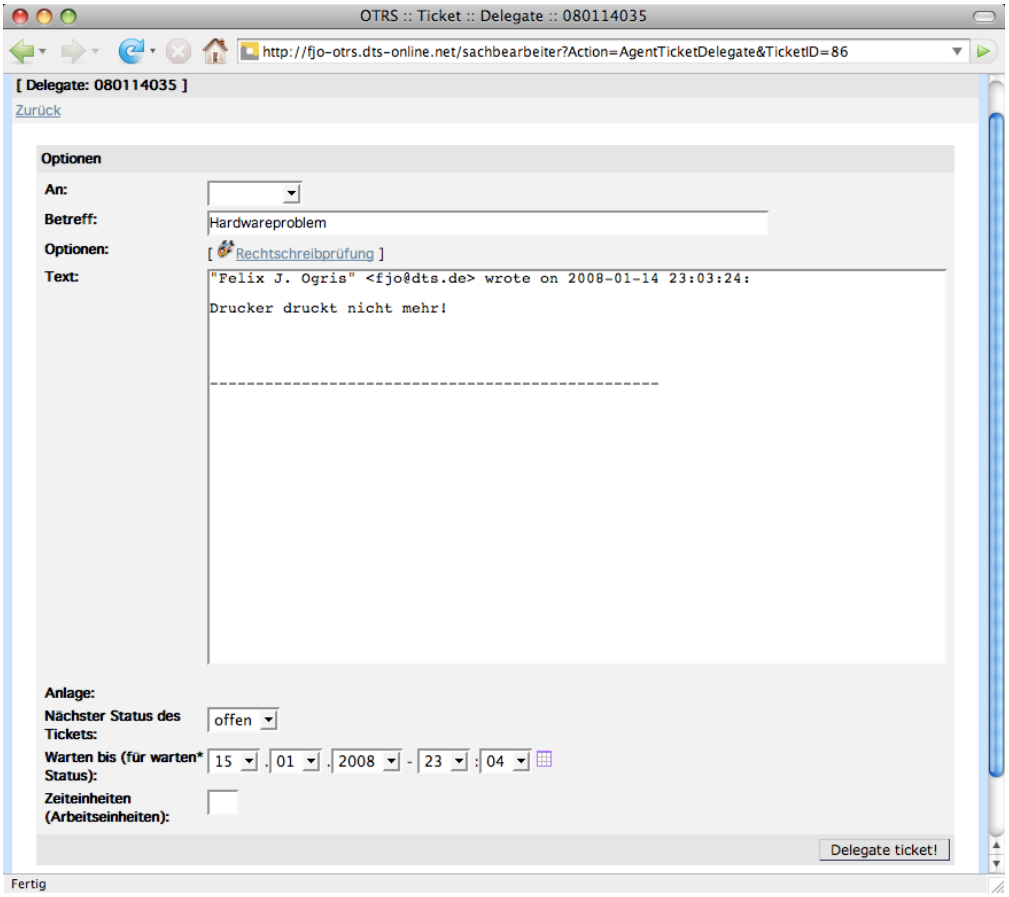

Abbildung 21: Ticketdelegation

9.7.2. Konfigurationsparameter

Das Modul *DTSAddress* greift auf einen eigenen Konfigurationshash unterhalb von `Ticket::Frontend::AgentTicketDelegate` zu. Folgende Parameter werden dabei abgefragt:

RequiredLock gibt an, ob das Ticket durch den Agenten gesperrt sein muss, um es delegieren zu können

StateDefault gibt an, welcher Ticketstatus per default ausgewählt ist.

Zusätzlich greift das Modul auf folgende Parameter zu:

Ticket::Frontend::AccountTime bestimmt, ob eine Bearbeitungszeit eingegeben werden muss

SpellChecker gibt an, ob die Rechtschreibprüfung angezeigt werden soll.

Zusätzlich muss das Administrationsfrontend `DTSAddressAdmin` in der Konfiguration verankert werden, um es über die Weboberfläche aufrufen zu können:

```
$Self->{'Frontend::Module'}->{'DTSAddressAdmin'} = {
    Group => [ 'admin' ],
    NavBarName => "Admin",
    NavBarModule => {
        Name => 'DTS Addresses',
        Block => 'Block4',
        Prio => 9999,
```

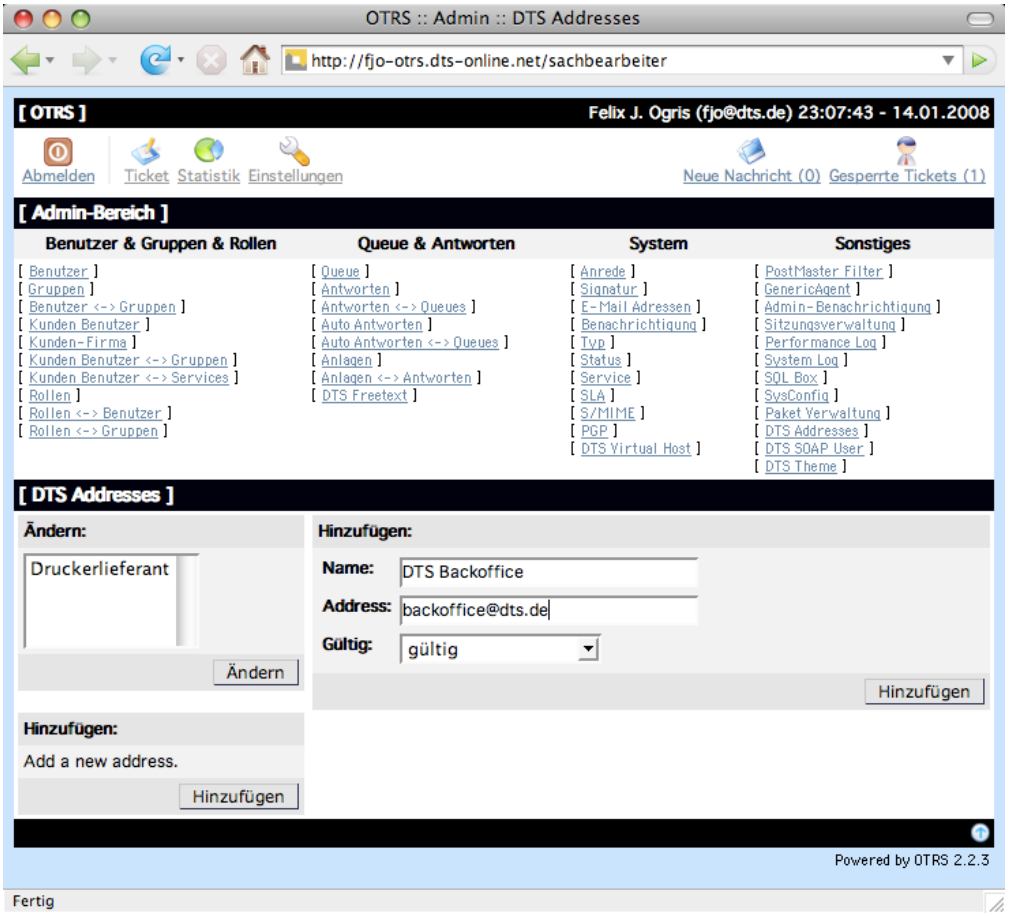

Abbildung 22: Adressbuch für Ticketdelegationen

```
        Module => 'Kernel::Output::HTML::NavBarModuleAdmin',
    },
};
```

Das Perlmodul `AgentTicketDelegate.pm` ist aufwendiger einzubinden. Zunächst muss es als generelles Frontendmodul deklariert werden:

```
$Self->{"Frontend::Module"}->{"AgentTicketDelegate"} = {
  'NavBarName' => 'Ticket',
  'Description' => 'Ticket Delegation',
  'Title' => 'Delegate'
};
```

Damit es in der Statusleiste in der Ticketansicht erscheint, muss es ferner als *MenuModule* eingebunden werden:

```
$Self->{"Ticket::Frontend::MenuModule"}->{"999-Delegate"} = {
    "Action" => "AgentTicketDelegate",
    "Module" => "Kernel::Output::HTML::TicketMenuGeneric",
    "Link" => 'Action=AgentTicketDelegate&TicketID=$QData{"TicketID"}',
    "Description" => "Delegate ticket to another OTRS instance",
    "Name" => "Delegate",
};
```

Ausserdem müssen wie für jedes Ticketmodul die möglichen Statustypen vorgegeben werden:

```
$Self->{'Ticket::Frontend::AgentTicketDelegate'}->{'StateType'} = \
    [ 'open', 'closed' ];
```

9.8. DTSSoapUser

9.8.1. Beschreibung

Das Modul *DTSSoapUser* stellt eine SOAP-Schnittstelle und eine HTTP-GET-Schnittstelle zum Anlegen von Kunden und Projekten aus anderen Buchhaltungs- und Projektsystemen wie dem *Work@Web* bereit. Die clientseitigen SOAP-Stubs können aus einer automatisch erzeugten WSDL-Beschreibung generiert werden. Zudem kann der authentifizierte Zugriff auf diese Schnittstellen auf bestimmte IP-Adressen beschränkt werden. Das Modul besteht aus sieben Komponenten:

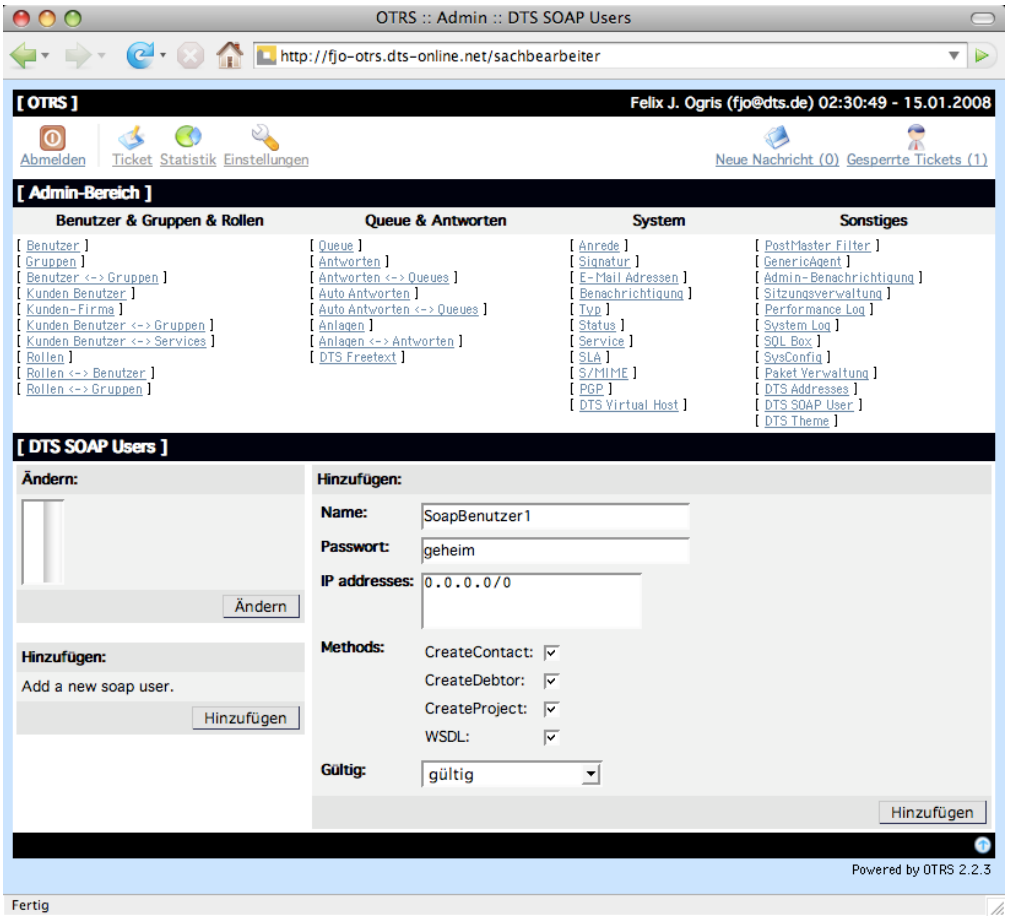

Abbildung 23: Anlage eines neuen SOAP-Benutzers

DTSSoapUser.pgsql.sql definiert 3 Relationen:

dts_soap_user führt Benutzernamen und Passwörter zur Authentifizierung an der SOAP-Schnittstelle

dts_soap_user_address referenziert die Benutzernamen aus `dts_soap_user` und verbindet sie mit Netzadressen, aus denen ein SOAP-Benutzer sich verbinden darf

dts_soap_user_method referenziert die Benutzernamen aus `dts_soap_user` und verbindet sie mit Methodennamen, die ein SOAP-Benutzer aufrufen darf.

Mit diesem Datenmodell kann ein SOAP-Benutzer auf IP-Adressbereiche und Funktionen eingeschränkt werden

DTSSoapUser.pm abstrahiert den Zugriff auf die o.g. Relationen und stellt in der Klasse `Kernel::System::DTSSoapUser` die Methoden `SoapUserList` zum Abruf

aller SOAP-Benutzer inklusive ihrer Rechte, `SoapUserAdd` zum Hinzufügen und `SoapUserModify` zum Aktualisieren eines SOAP-Benutzers bereit. Zudem wird die Methode `IsSoapUserAllowed` exportiert, die anhand der ihr übergebenen Parameter Benutzername, Passwort, SOAP-Funktion und Clientadresse überprüft, ob der SOAP-Benutzer zum Aufruf der Funktion berechtigt ist

DTSSoapUserAdmin.pm stellt das Frontendmodul zur Pflege von SOAP-Benutzern dar

DTSSoapUser.dtl ist das Template für den Administrationsbereich

DTSSoap.pm implementiert die eigentlichen SOAP-Funktionen und dient somit auch zur Erstellung der WSDL-Beschreibung

DTSWsdl.pm erzeugt einerseits die WSDL-Beschreibung für das o.g. Perlmodul `DTSSoap.pm`. Andererseits stellt es für das Frontendmodul `DTSSoapUserAdmin.pm` die Methode `MethodList` bereit, die ein Array mit allen verfügbaren SOAP-Funktionen liefert

InterfaceSoapUser.pm ist das Bindeglied zwischen Webserver und den SOAP-Funktionen in `DTSSoap.pm`. Sie definiert die Klasse `Kernel::System::Web::InterfaceSoapUser`. Bei einem HTTP-GET-Aufruf wertet die Methode `Run` selbst alle übergebenen Parameter aus und ruft dann die gewünschte Funktion in `DTSSoap.pm` auf. Ein POST-Aufruf, der einen SOAP-Request darstellt, wird von einer Instanz von `SOAP::Transport::HTTP` bearbeitet

Das Modul `DTSSoap.pm` definiert die Klasse `Kernel::DTSSoap`. Ihre Memberfunktionen stellen die nach aussen freigegebenen SOAP-Funktionen dar. Sie haben folgende Struktur:

```
=begin WSDL

_DOC Dies ist eine Testfunktion
_IN  Username $string Benutzername
_IN  Password $string Passwort
_OUT          $string Rückgabewert

=end WSDL

=cut

sub HelloWorld ()
{
    my $Self = shift;
    my $User = shift;
    my $Pass = shift;

    # Username und Passwort prüfen

    return SOAP::Data->type("string")
                     ->name("HelloWorldReturn")
                     ->value("hello, world");
}
```

Der vorangestellte POD-Block wird benötigt, um daraus eine WSDL-Beschreibung zu erzeugen. Das Flag `_DOC` zeigt dabei einen Kommentar an, der in das WSDL-Dokument zwecks Lesbarkeit übernommen werden kann. `_IN` und `_OUT` bezeichnen die der Funktion übergebenen bzw. die von ihr zurückgelieferten Datentypen, in diesem Falle einfache Zeichenketten. Der Rückgabewert der Funktion `HelloWorld` ist zwar lediglich ein String, muss aber aufwendig codiert werden, damit in Java geschriebene SOAP-Clients keinen

Fehler werfen[24]. Das resultierende SOAP-Fragment sieht in diesem Falle wie folgt aus:

```
<HelloWorldResponse>
    <HelloWorldReturn
        xsi:type="xsd:string">hello, world</HelloWorldReturn>
</HelloWorldResponse>
```

Wird hingegen die Zeichenkette direkt per `return` zurückgegeben, so ergibt sich folgendes SOAP-Teilstück:

```
<HelloWorldResponse>
    <s-gensym3
        xsi:type="xsd:string">hello, world</s-gensym3>
</HelloWorldResponse>
```

Die WSDL-Beschreibung wird on-the-fly vom Modul `DTSWsdl.pm` erzeugt. Es greift auf das Package `Pod::WSDL` zurück:

```
my $WsdlObject = Pod::WSDL->new(
    source => "Kernel::DTSSoap",
    location => "http://fjo-otrs.dts-online.net/soap",
    withDocumentation => 1,
    use => $Pod::WSDL::LITERAL_USE,
    pretty => 1
);

my $WsdlDocument = $WsdlObject->WSDL;
```

Der Konstruktor erwartet folgende Parameter:

source gibt das Modul an, für das die WSDL-Beschreibung erzeugt werden soll. Dieses Modul sollte über entsprechend formatierte POD-Abschnitte verfügen

location gibt die Adresse an, unter der die SOAP-Funktionen abgerufen werden können

withDocumentation gibt an, ob die per `_DOC` gekennzeichneten Dokumentationen der POD-Blöcke in das WSDL-Dokument übernommen werden sollen

use zeigt die Serialisierungsart an. Per default wird *RPC/encoded* verwendet, im Beispiel wird hingegen *RPC/literal* eingesetzt. Document/literal wird nicht unterstützt

pretty zeigt an, ob das WSDL-Dokument zwecks Lesbarkeit mit Whitespaces am Zeilenanfang formatiert werden soll.

Im Scalar `$WsdlDocument` steht somit die erzeugte WSDL-Beschreibung zur Verfügung.

Die Methode `Run` in der Klasse `Kernel::System::Web::InterfaceSoapUser` bzw. im Perlmodul `InterfaceSoapUser.pm` weist folgende Struktur auf:

```
sub Run ()
{
    my $RequestMethod = $ENV{REQUEST_METHOD};

    if ($RequestMethod eq "GET") {
        # Parameter "Action" auswerten, dann entsprechende Methode
        # in DTSSoap.pm aufrufen
    }
    else {
        my $SoapCGIObject = SOAP::Transport::HTTP::CGI->new();
        $SoapCGIObject->dispatch_to("Kernel::DTSSoap");
        $SoapCGIObject->handle();
    }
}
```

[24] Dies herauszufinden hat einige Tage in Anspruch genommen und konnte nur mit Hilfe eines Netzwerkanalyseprogrammes (http://www.wireshark.org) festgestellt werden.

In der Umgebungsvariablen `$ENV{REQUEST_METHOD}` wird die Art des Webseitenaufrufes mitgeteilt. Findet ein HTTP-GET-Aufruf statt, ist dort der Wert `GET` hinterlegt. In diesem Fall wertet die Funktion den Parameter `Action` aus und ruft die entsprechende Methode in `DTSSoap.pm` auf. Bei einem SOAP-Aufruf per HTTP-POST hingegen wird eine Instanz von `SOAP::Transport::HTTP::CGI` verwendet. Diese benötigt zuerst den Namen des Moduls, in dem sich die SOAP-Funktionen befinden, hier also `Kernel::DTSSoap`. Anschliessend vollzieht die Methode `handle` prinzipiell die gleichen Schritte wie der HTTP-GET-Zweig, indem die gewünschte Funktion aus der SOAP-Anfrage extrahiert und aufgerufen wird.

Durch den Einsatz von WSDL können die bereitgestellten SOAP-Funktionen relativ leicht in andere Projekte eingebunden werden (s. Abbildung 24). Anschliessend genügt

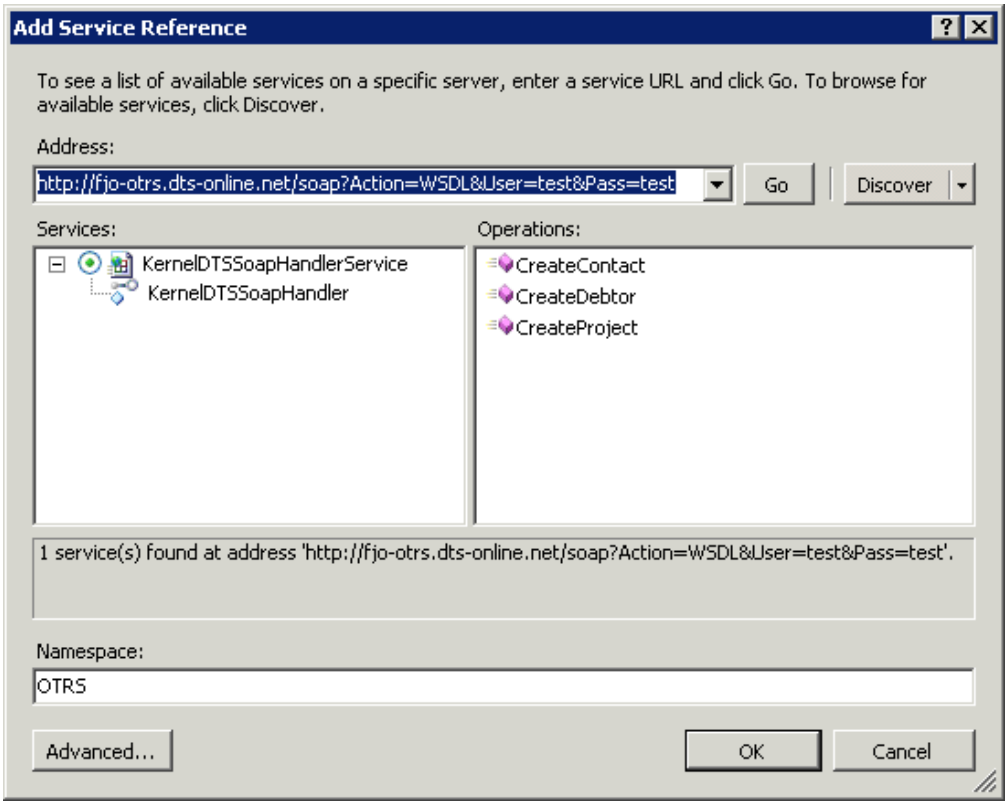

Abbildung 24: Import der SOAP-Funktionen in Microsoft Visual Studio

folgendes Minimalprogramm (hier in C#), um die Funktion `CreateProject` des OTRS aufzurufen:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SoapTest
{
    class Program
    {
        static void Main(string[] args)
        {
            const String User     = "test",
                         Pass     = "test",
```

```
                        Projekt = "Neues Projekt",
                        Kunde   = "1111";
            String Result;
            OTRS.KernelDTSSoapHandlerClient otrs;

            otrs = new OTRS.KernelDTSSoapHandlerClient();
            Result = otrs.CreateProject(User, Pass, Projekt, Kunde);

            System.Console.WriteLine("Projektnummer: " + Result);
            System.Console.ReadLine();
        }
    }
}
```

Äquivalent ist folgendes Beispiel in PHP:

```
<?
$wsdl   = "http://fjo-otrs.dts-online.net/soap?Action=WSDL&" .
          "User=test&Pass=test";
$otrs   = new SoapClient($wsdl);
$nummer = $otrs->CreateProject("test", "test", "Neues Projekt", "1111");

echo "Projektnummer: " . $nummer . "\n";
?>
```

9.8.2. Konfigurationsparameter

In der Konfiguration muss lediglich das Frontendmodul `DTSSoapUserAdmin` registriert werden:

```
$Self->{'Frontend::Module'}->{'DTSSoapUserAdmin'} = {
    Group => [ 'admin' ],
    NavBarName => "Admin",
    NavBarModule => {
        Name => 'DTS SOAP User',
        Block => 'Block4',
        Prio => 9999,
        Module => 'Kernel::Output::HTML::NavBarModuleAdmin',
    },
};
```

9.9. DTSNotifyAgentAsterisk

9.9.1. Beschreibung

Das Modul *DTSNotifyAgentAsterisk* kann einen Bearbeiter telefonisch auf ein eskaliertes Ticket hinweisen. Hierzu ist ein Asteriskserver[25] notwendig, der skriptgesteuert über sein *Asterisk Manager Interface* (AMI) Telefonate aufbauen kann. Zudem muss auf dem Asteriskserver das Programm *Festival*[26] zur Sprachsynthese installiert sein. Zum Zugriff auf das Asterisk Manager Interface kommt seitens des OTRS das Perlmodul `Asterisk::Manager` zum Einsatz. Es wird wie folgt verwendet:

```
use Asterisk::Manager;

my $AMIObject = Asterisk::Manager->new();

# set connection parameters for ami server
$AMIObject->host("asterisk.dts.de");
$AMIObject->port(5038);
$AMIObject->user("otrs");
$AMIObject->secret("geheim");

if (!$AMIObject->connect()) {
    # Fehler
}

my %AMIResult = $AMIObject->sendcommand(
    Action => "Originate",
    Channel => "CAPI/g0-9/052211011000",
    Exten => "123456",
    Priority => "1",
    Async => "0",
    Timeout => 15000,
    Context => "default",
    Variable => "TicketNumber=307122218|Text=A ticket is escalated"
);
```

Nachdem dem ein neues Objekt instanziert wurde, werden die Verbindungsparameter gesetzt. Das AMI basiert auf einer herkömmlichen TCP-Verbindung, über die Befehle im Klartext abgesetzt werden. Die Methode `sendcommand` kapselt sämtliche Protokolldetails. Sie erwartet folgende Parameter:

Action gibt an, welche Aktion ausgeführt werden soll. Der Wert `Originate` weist den Asteriskserver an, ein Gespräch zwischen zwei Teilnehmern aufzubauen

Channel stellt einen Endpunkt dieses Telefonates dar, nämlich denjenigen Teilnehmer, der angerufen wird, hier die Telefonnummer `05221-101-1000`. Diese soll per ISDN-Karte bzw. CAPI-Schnittstelle angerufen werden

Exten stellt die Nummer des Anrufenden dar. Für diesen Fall muss der Asteriskserver so konfiguriert werden, dass unter der Nummer `123456` das Programm *Festival* zu erreichen ist

Priority gibt die Priorität dieses Telefonates an

Async sollte den Wert `1` haben, wenn der Aufruf von `sendcommand` sofort zurückkehren soll, `0`, falls `sendcommand` erst zurückkehren soll, wenn das Telefonat zustande gekommen ist

[25] http://www.asterisk.org
[26] http://www.cstr.ed.ac.uk/projects/festival/

Timeout gibt die Zeit in Millisekunden an, die `sendcommand` wartet, bis das Telefonat zustanden gekommen ist

Context gibt den Asterisk-Kontext (sprich: die "Telefonie-Routingtabelle") für diese Verbindung an

Variable gibt eine Liste von per senkrechtem Strich getrennten Schlüssel-/Wertepaaren an, die im Dialplan (sprich: in der Telefonie-Routingtabelle) als Variablen zur Verfügung stehen. Diese Variablen werden durch den Dialplan dem Programm *text2wave* übergeben, welches daraus Sprachsamples erzeugt und somit dem angerufenen Bearbeiter über den Ticketstatus informiert.

Der Dialplan des Asteriskservers muss für dieses Beispiel um folgende Zeilen erweitert werden[27]:

```
exten => 123456,1,Answer()
exten => 123456,n,Set(FileBase=/tmp/otrs-${TicketNumber})
exten => 123456,n,Set(FileType=ulaw)
exten => 123456,n,Set(FileName=${FileBase}.${FileType})
exten => 123456,n,System([ -e \'${FileName}\' ] || echo \'${Text}\' |\
    text2wave -otype \'${FileType}\' -o \'${FileName}\')
exten => 123456,n,Playback(${FileBase})
exten => 123456,n,Hangup()
```

DTSNotifyAgentAsterisk basiert zu grossen Teilen auf dem mitgelieferten Modul *NotifyAgentGroupWithWritePermission*, welches jedoch Emails an Bearbeiter sendet.

9.9.2. Konfigurationsparameter

Das Modul *DTSNotifyAgentAsterisk* wertet folgende Konfigurationsparameter aus:

DTSAsterisk::AMIHostname gibt den Hostnamen des Asteriskservers an

DTSAsterisk::AMIPort gibt den Port des Asterisk Manager Interfaces an

DTSAsterisk::AMIUsername gibt den Benutzer an, der über das AMI Gespräche aufbauen darf

DTSAsterisk::AMIPassword stellt das Passwort für den Benutzer dar

DTSAsterisk::Channel gibt das Device (ISDN-Karte o.ä.) an, über das die Bearbeiter telefonisch erreicht werden können, z.B. `CAPI/g0-9/<PHONE_NUMBER>`, wobei `<PHONE_NUMBER>` durch die Rufnummer des jeweiligen Bearbeiters ersetzt wird

DTSAsterisk::Extension gibt die Durchwahl des Programmes *text2wave* aus dem Festivalpaket an

DTSAsterisk::Timeout gibt die Zeit in Millisekunden an, die maximal auf das Zustandekommen des Gespräches gewartet werden soll

DTSAsterisk::Context gibt die Telefonie-Routingtabelle an

DTSAsterisk::FestivalTextKey spezifiziert den Namen der Variablen, unter der der Dialplan den zu synthetisierenden Text erwartet

DTSAsterisk::FestivalText gibt den Text an, der synthetisiert werden soll, z.B. `Attention! Attention! Ticket number <TICKET_NUMBER> is escalated!`, wobei `<TICKET_NUMBER>` durch die jeweilige Ticketnummer ersetzt wird

DTSAsterisk::FestivalTicketNumberKey stellt den Namen der Variablen dar, unter der der Dialplan die Ticketnummer erwartet

[27] s. Ogris (2007)

A. Literatur

[Ahmed 2006] Ahmed, Tarek: *Pod::WSDL - Creates WSDL documents from (extended) pod*, Oktober 2006. – Manpage zum Perlmodul Pod::WSDL

[Almquist u. a. 2006] Almquist, Kenneth u. a.: *sh – command interpreter (shell)*, Juli 2006. – Manpage zur sh

[Biron u. Malhotra 2004] Biron, Paul V. ; Malhotra, Ashok: *XML Schema Part 2: Datatypes Second Edition*. Version: October 2004. http://www.w3.org/TR/xmlschema-2/, Abruf: 2008-01-06

[Butek 2005] Butek, Russell: *Which style of WSDL should I use?* Version: Mai 2005. http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/, Abruf: 2007-12-15

[Christiansen 2006] Christiansen, Tom: *perltoot - Tom's object-oriented tutorial for perl*, Januar 2006. – Manpage zu Perl 5.8.8

[Clark 1999] Clark, James: *XSL Transformations (XSLT)*. Version: November 1999. http://www.w3.org/TR/xslt/, Abruf: 2008-01-07

[Costales u. Allman 2002] Costales, Bryan ; Allman, Eric: *sendmail*. O'Reilly, 2002. – ISBN 1–56592–839–3

[Diverse a] Diverse: *OTRS API (HTML Developer API)*. http://dev.otrs.org/, Abruf: 2007-12-15

[Diverse b] Diverse: *Product Photos*. http://h18000.www1.hp.com/products/quickspecs/photos/photos.html, Abruf: 2007-12-15. – HP Product Bulletin

[Diverse c] Diverse: *WSDL Tutorial*. http://www.w3schools.com/wsdl/, Abruf: 2008-01-07

[Diverse d] Diverse: *XML Schema Tutorial*. http://www.w3schools.com/schema/, Abruf: 2008-01-06

[Diverse e] Diverse: *XSLT Tutorial*. http://www.w3schools.com/xsl/, Abruf: 2008-01-06

[Diverse 2005a] Diverse: *openssl - OpenSSL command line tool*, Februar 2005. – Manpage zu OpenSSL 0.9.7d

[Diverse 2005b] Diverse: *tcsh - C shell with file name completion and command line editing*, März 2005. – Manpage zur tcsh 6.14.00

[Diverse 2006a] Diverse: *perlfunc - Perl builtin functions*, Januar 2006. – Manpage zu Perl 5.8.8

[Diverse 2006b] Diverse: *perlop - Perl operators and precedence*, Januar 2006. – Manpage zu Perl 5.8.8

[Diverse 2006c] Diverse: PostgreSQL 8.2.5 Documentation. (2006). http://www.postgresql.org/docs/8.2/static/index.html, Abruf: 2008-01-01

[Diverse 2007a] Diverse: Berkeley Software Distribution. (2007), Dezember. http://de.wikipedia.org/wiki/Berkeley_Software_Distribution, Abruf: 2007-12-29

[Diverse 2007b] Diverse: Dokumentation zum Apache HTTP Server Version 2.2. (2007). http://httpd.apache.org/docs/2.2/, Abruf: 2008-01-01

[Diverse 2007c] Diverse: Extensible Markup Language. (2007), Dezember. http://de.wikipedia.org/wiki/XML, Abruf: 2007-12-17

[Diverse 2007d] Diverse: mod_perl: Documentation. (2007), Dezember. http://perl.apache.org/docs/index.html, Abruf: 2008-01-01

[Diverse 2007e] Diverse: Perl. (2007), Dezember. http://de.wikipedia.org/wiki/Perl, Abruf: 2007-12-15

[Diverse 2007f] Diverse: Unix-Shell. (2007), Dezember. http://de.wikipedia.org/wiki/Unix-Shell, Abruf: 2007-12-27

[Eckstein 2000] Eckstein, Robert: *XML - kurz & gut*. O'Reilly, 2000. – ISBN 3–89721–219–6

[Kulchenko u. a. 2006] Kulchenko, Paul ; Ray, Randy J. ; Reese, Byrne: *SOAP::Lite - Perl's Web Services Toolkit*, August 2006. – Manpage zum Perlmodul SOAP::Lite

[Éric Lévénez 2007] Lévénez Éric: *Unix History*. Version: Dezember 2007. http://www.levenez.com/unix/, Abruf: 2007-12-29

[Martin u. a. 2000] Martin, Didier ; Birbeck, Mark ; Kay, Michael ; Loesgen, Brian ; Pinnock, Jon ; Livingstone, Steven ; Stark, Peter ; Williams, Kevin ; Anderson, Richard ; Mohr, Stephen ; Baliles, David ; Peat, Bruce ; Ozu, Nikola: *Professional XML*. Wrox, 2000. – ISBN 1–861003–11–0

[Münz u. a. 2007] Münz, Stefan u. a.: *SELFHTML 8.1.2*. Version: März 2007. http://de.selfhtml.org/, Abruf: 2007-12-16

[Ogris 2007] Ogris, Felix J.: *Asterisk - ein Überblick*. Version: Januar 2007. http://www.ogris.de/docs/studienarbeit.pdf, Abruf: 2008-01-14

[Rowe u. Stonebraker 1987] Rowe, Lawrence A. ; Stonebraker, Michael R.: *The POSTGRES Data Model*. Version: September 1987. http://s2k-ftp.cs.berkeley.edu:8000/postgres/papers/ERL-M87-13.pdf, Abruf: 2008-01-01

[Schöpplein u. a. 2007a] Schöpplein, Christian ; Kammermeyer, Richard ; Rother, Stefan ; Raith, Thomas ; Steinbild, Burchard ; Mindermann, André ; Edenhofer, Martin ; Kuhn, Christopher ; Oschwald, Henning ; Hecht, Manuel ; Bakker, René ; Bauer, Bodo ; Böttcher, Hauke ; Bothe, Jens: *OTRS 2.2 - Admin Manual*. Version: 2007. http://ftp.otrs.org/pub/otrs/doc/doc-admin/2.2/en/pdf/otrs_admin_book.pdf, Abruf: 2007-12-15

[Schöpplein u. a. 2007b] Schöpplein, Christian ; Kammermeyer, Richard ; Rother, Stefan ; Raith, Thomas ; Steinbild, Burchard ; Mindermann, André ; Kuhn, Christopher ; Edenhofer, Martin: *OTRS 2.2 - Developer Manual*. Version: 2007. http://ftp.otrs.org/pub/otrs/doc/doc-developer/2.2/en/pdf/otrs_developer_book.pdf, Abruf: 2007-12-15

[Shohoud 2003] Shohoud, Yasser: *RPC/Literal and Freedom of Choice*. Version: April 2003. http://msdn2.microsoft.com/en-us/library/ms996466.aspx, Abruf: 2008-01-08

[Siever u. a. 1999] Siever, Ellen ; Spainhour, Stephen ; Patwardhan, Nathan: *Perl in a Nutshell*. O'Reilly, 1999. – ISBN 1–56592–286–7

[Srinivasan 1997] Srinivasan, Sriram: *Advanced Perl Programming*. O'Reilly, 1997. – ISBN 1–56591–220–4

[Srinivasan 1999] Srinivasan, Sriram: *Fortgeschrittene Perl-Programmierung*. O'Reilly, 1999. – ISBN 3–89721–107–6

[Thompson u. a. 2004] Thompson, Henry S. ; Beech, David ; Maloney, Murray ; Mendelsohn, Noah: *XML Schema Part 1: Structures Second Edition*. Version: October 2004. http://www.w3.org/TR/xmlschema-1/, Abruf: 2008-01-06

[Venema u. a.] Venema, Wietse u. a.: Postfix Documentation. http://www.postfix.org/documentation.html, Abruf: 2008-01-01

[Wall u. Burke 2006] Wall, Larry ; Burke, Sean M.: *perlpod - the Plain Old Documentation format*, Januar 2006. – Manpage zu Perl 5.8.8

[Wall u. a. 2000] Wall, Larry ; Christiansen, Tom ; Orwant, Jon: *Programming Perl*. O'Reilly, 2000. – ISBN 0–596–00027–8

[Weinelt] Weinelt, Jürgen: *LaTeX-Befehlsreferenz*, http://www.weinelt.de/latex/, Abruf: 2007-12-17

B. CD-ROM

Die CD-ROM enthält alle im Rahmen dieser Diplomarbeit erstellten Programme und Skripte sowie die vorliegende Arbeit im PDF-Format.